Zdravko Dovedan Han

progovorimo pythonski

Zdravko Dovedan Han

progovorimo

pythonski

Vlastita naklada

Zagreb, 2021.

Recenzenti

Boris Čulina
Davor Lauc
Aleksander Radovan
Mirko Smilevski

Lektorica

Vjera Lopina

Slog i prijelom

Zdravko Dovedan Han

Dizajn ovitka

Zdravko Dovedan Han

Nakladnik

Vlastita naklada: Zdravko Dovedan Han
Zagreb, 2021.

Tisak

PRENSA j.d.o.o
Kaštel Kambelovac

CIP zapis je dostupan u računalnom katalogu
Nacionalne i sveučilišne knjižnice u Zagrebu
pod brojem 001118605

ISBN 978-953-49798-0-8

Sadržaj

Predgovor

Python je dinamički, objektno orijentirani programski jezik opće namjene. Svrha dizajna jezika Python naglašava produktivnost programera i čitljivost koda. Python je u početku razvio Guido van Rossum. Prvo je objavljen 1991. Python je nadahnut programskim jezicima ABC, Haskell, Java, LISP, Icon i Perl.

Službeno web mjesto za programski jezik Python je python.org. Knjiga se odnosi na opis verzije 3.9 Pythona koja se razlikuje od verzija 2.x prije nje.

Python je objektno orijentirani jezik visoke razine. Njegov je prevodilac implementiran kao interpretator, s mogućnošću interaktivnog izvršavanja naredbi jezika. Interaktivnost Pythona posebno je važna u njegovom bržem i potpunijem učenju.

Dakle, pred nama je i jednostavan i moćan jezik, prilagodljiv uzrastu i predznanju, slično kao što je jezik matematike, jezik za sva vremena, besplatan s velikom bibliotekom gotovih programa, primjenljivih u mnogim područjima, od matematike, fizike i kemije do elektrotehnike, strojarstva, računarstva i, dakako, informatike, te u mnogim disciplinama kao što su teorija algoritama, struktura podataka i teorija formalnih jezika. Podesan je za rad s bazama podataka, za izradu aplikacija koje sadrže grafiku, izradu web aplikacija i obradu teksta. Nalazi svoje primjene u biologiji, medicini, cvjećarstvu itd.

Temeljne karakteristike Pythona su:

- *lagan je za učenje (interaktivni mod)*
- *može se učiti postupno (kao matematika)*
- *prilagodljiv je znanju i „uzrastu“ onih koji ga uče,*
- *moćan je (tipovi i strukture podataka, OOP, velika biblioteka standardnih i nestandardnih modula i programskih paketa)*
- *dobro je dokumentiran*
- *raširen je*
- *besplatan je*

Ima široki spektar primjene u:

- *matematici, fizici, kemiji, na svim razinama obrazovanja,*
- *obradi teksta,*
- *grafici,*
- *bazama podataka,*
- *teoriji algoritama i struktura podataka,*
- *web aplikacijama,*
- *strojnom učenju,*
- *teoriji sintaksne analize,*
- *teoriji prevođenja itd.*

Objedinjuje sve navedene paradigme programiranja:

- *strukturno programiranje*
- *objektno orijentirano programiranje*
- *logičko programiranje*
- *funkcijsko programiranje*

Python može učiniti gotovo sve: Web aplikacije, korisnička sučelja, analiza podataka, statistika ...

Od nedavno, Python se koristi kao ključni alat za znanstvene divovske skupove podataka za bilo koje industrije. Rad s velikim cijelim brojevima i dugačkim stringovima, LAMBDA funkcije, uvjetni izrazi itd! Da ne nabrajamo. Možda bi trebalo postaviti pitanje: „Gdje se Python ne bi mogao primijeniti!?“

Python je opskrbljen velikom on-line dokumentacijom koja se svakodnevno dopunjuje i proširuje, zajedno s poboljšanjem samog jezika.

Programiranje u Pythonu uvodi disciplinu “lijepog” (strukturiranog) pisanja programa jer je struktura programa uvlačenjem pojedinih blokova unutar složenih naredbi mnogo preglednija od BEGIN i END u Pascalu ili vitičastih zagrada u jeziku C i njegovim derivatima.

Koliko je Python popularan jezik, možda nam najbolje pokazuje PYPL (The PopularitY of Programming Language) indeks koji je kreiran na temelju analize koliko se često pretražuju tutorijali pojedinih programskih jezika na Googleu. U srpnju 2021. godine rezultati su prikazani u sljedećoj listi (samo prvih sedam jezika). Posljednji put je Python bio na drugom mjestu u prosincu 2017. godine (sačuvali smo tadašnju listu, v. sliku). Na prvom mjestu je od siječnja 2018. do danas!

Worldwide, Jul 2021 compared to a year ago:

Rank	Change	Language	Share	Trend
1		Python	30.32 %	-1.8 %
2		Java	17.79 %	+1.0 %
3		JavaScript	9.03 %	+1.1 %
4		C#	6.55 %	-0.2 %
5	↑	C/C++	6.02 %	+0.3 %
6	↓	PHP	5.94 %	+0.0 %
7		R	3.96 %	-0.0 %

Worldwide, Dec 2017 compared to a year ago:

Rank	Change	Language	Share	Trend
1		Java	21.5 %	-1.4 %
2		**Python**	**19.3 %**	**+5.6 %**
3		PHP	8.3 %	-1.4 %
4	↑	Javascript	7.9 %	+0.3 %
5	↓	C#	7.6 %	-0.8 %
6		C	6.3 %	-0.9 %
7		C++	6.3 %	-0.8 %

Kad se danas kaže „Python“ misli se na Python 3.x koji se počeo razvijati od inačice 3.0 2010. godine do 3.10 2021. godine. Prije toga je bila inačica Pythona 2.7.x koja je paralelno razvijana (bilo je još uvijek dosta korisnika) i došla do 2.7.18 (vjerojatno posljednje) publicirane 20.4.2020. godine.

„progovorimo pythonski“

Još jedna knjiga o Pythonu!? I to s pomalo provokativnim naslovom „progovorimo pythonski“! Pa zar ne postoji već nekoliko stotina knjiga koje se bave Pythonom i programiranjem u njemu?

U pravu ste, a možda i niste! Ako pogledate sve te knjige, naći ćete dosta sličnosti među njima. Ne, nije nam namjera potcjenjivati sve te napore mnogih autora koji su se oduševili Pythonom i pokušali to prenijeti široj populaciji. I autor ovih redaka je među njima. Ali, za razliku od ostalih djela, u većini knjiga nedostaje sintaksa i semantika, tajni programiranja, primjera programa...

Osim toga, naša je namjera kroz učenje Pythona istodobno razvijati poseban odnos prema programiranju kao posebnoj disciplini, učenjem pravila pisanja (sintaksa), pravila „lijepog i preglednog“ pisanja, te razumijevanjem semantike (značenja) naredbi, izraza, primitivnih i složenih tipova podataka.

O formatu teksta

Programi pisani u Pythonu su „pjesma“, a ne „proza“. Želimo reći da nisu široki. Zbog toga bi uobičajeni format zauzeo mnogo slabo popunjenih stranica, pa smo odlučili pisati u dva stupca. Time je dobiven kompaktan i pregledan tekst.

Struktura teksta

„Hello world“, tako počinje većina knjiga koje opisuju jezike za programiranje! Samo vas pozdravljamo, a pristup učenju će se donekle razlikovati od uobičajenih: počet ćemo s interaktivnim modom Pythona i brojčanim podacima, a kasnije ćemo se baviti i obradom teksta. Pretpostavka je da se proučavanjem teksta paralelno radi na računalu.

Knjiga je podijeljena na poglavlja i podpoglavlja. Označili smo ih posebnom veličinom i vrstama slova. Općenito se struktura teksta može prikazati kao:

n. POGLAVLJE

Podpoglavlje

SEKCIJE PODPOGLAVLJA

GOVORIMO PYTHONSKI

P R O G R A M I

Strukturom teksta određen je opći sadržaj poglavlja: uvod (motivacija), sintaksa i semantika naredbi, govorimo pythonski („male tajne programiranja“) i programi. Time je poglavlje podijeljeno na dva dijela: dio koji se odnosi na opis sintakse i semantike naredbe, zajedno s jednostavnim primjerima za ilustraciju, i dio koji je posvećen programiranju, gdje se u dijelu „govorimo pythonski“ analiziraju mogućnosti primjene prethodno opisanih tipova podataka i naredbi i uvede neke metode, programi i načela programiranja, a u dijelu „programi“ se na primjeru algoritama ili rješenju poznatih problema matematike, fizike itd. sumiraju dotad uvedene naredbe u kontekstu programa.

Kako koristiti knjigu?

Ovisno o uzrastu i predznanju. Karakteristika je Pythona da se može učiti kao i matematika: od lakšeg prema težem. U prvom prolasku uče se osnove, koje će biti dovoljne za rješavanje jednostavnih i „srednje teških“ problema. Težište nije na što većem broju programa, u kojima samo „promjenimo neke brojeve“, već na rješavanju nekoliko problema na više načina, ovisno o naučenim naredbama, tipovima i strukturama podataka!

U drugom prolasku prelazi se na napredno znanje jezika. Dakako, nije zabranjeno da odmah učite i jedno i drugo, ali je preporuka da se ide postupno, kao što, na primjer, učimo govoriti neki prirodni jezik.

Kompletan opis Pythona podijeljen je u Osnove, 14 glavnih poglavlja i završno poglavlje 15 u kojem je sumirano sve ono što je prikazano u pethodnih 14 poglavlja i prikazani neki radovi u kojima je primijenjen Python. Knjiga sadrži preko 300 programa iz teorije i prakse programiranja i još nebrojeno primjera (vježbi) koji dopunjuju pojedine teme.

Struktura izloženog gradiva, opis pojedinih tipova i sintaksnih kategorija koje se odnose na njih, koncipirana je tako da se uvijek počinje s numeričkim (cjelobrojnim i realnim) tipovima. Uvođenje pojedinih naredbi, tipova i struktura podataka je postupno, od jednostavnih do složenih.

Editor i Shell

Python je sustav koji sadrži i dio za pisanje programa ("editor") koji interaktivno provjerava jesu li uparene zagrade, na primjer. Izlazni ekran (Shell) služi za interaktivno pisanje naredbi i za prikaz rezultata izvršenja programa u kojem je moguće dodatno provjeriti vrijednosti pojedinih varijabli programa. Sve to znatno ubrzava učenje Pythona.

Kome je knjiga namijenjena?

Python je jezik osnovne i srednje škole, visokih učilišta, doktorskih studija, jezik informatičara, matematičara, kemičara, biologa, tehničara, inženjera, psihologa, sociologa, liječnika, jezik laika, jezik za cjeloživotno učenje, jezik „rekreativaca“ i sanjara, ali i snažan alat za potporu u izradi profesionalnih aplikacija! Svatko će od vas uzeti onaj dio Pythona kojem ste u danom času dorasli.

Python je sa svojom semantikom sigurno jedinstven jezik za programiranje koji potiče maštu i kreativnost. Ogromne su mogućnosti u primjeni njegovih logičkih izraza, uvjetnih izraza, LAMBDA funkcija, struktura podataka, posebno lista i mapa (rječnika). Primjeri programa dani u ovoj knjizi sve to ilustriraju u dovoljnoj mjeri.

Python objedinjuje sve paradigme programiranja, od konvencijalnog do funkcijskog, dinamičkog i, prije svega, objektnog programiranja. Sa svojim standardnim modulima i velikim brojem razvijenih nestandarnih modula i programskih paketa dovoljan je za većinu kolegija na veleučilištima i fakultetima:

- *Uvod u programiranje*
- *Objektno programiranje*
- *Algoritmi i strukture podataka*
- *Numeričko programiranje*

Knjiga je namijenjena i onima (od 15 do 99 godina!) koji se žele samostalno upustiti u učenje Pythona i primijeniti ga u raznim djelatnostima, od tehničarske i inžinjerske prakse, do primjene u obradi teksta.

Neka vam ova knjiga uz mnoštvo primjera programa, bude samo poticaj za daljnje učenje jer „znati" programirati u Pythonu znači neprekidno istraživati i povezivati se s mnogim dobrim ljudima koji svoje uratke stalno publiciraju na webu.

Napomenimo da ova je knjiga pisana s prekidima nekoliko godina. Počelo se s inačicom 2.7.3. Ako ste početnik, odlučite se za 3.9 (opisane u ovoj knjizi) ili 3.10. Ako ste radili u 2.7.x inačici i polako je napuštate, bitne razlike su:

- `print` *i* `exec` *su funkcije, a ne naredbe.*
- *Unos podataka je samo s* `input` *(ne postoji* `raw_input`*).*
- *Ne mogu se uspoređivati podaci različitog tipa, niti sortirati heterogene liste.*
- *Ne postoji cijeli broj tipa* `long`*, samo* `int` *(koji je kao bivši* `long`*).*
- *Kosa crta „/" je uvijek realno dijeljenje, bez obzira na tip operanada. „//" se može koristiti za „staro" cjelobrojno dijeljenje.*
- *Upotreba svih alfa znakova Unicode standardizacije u svim imenima. Na primjer, legalna su imena Čiča, Površina, α, π, Σ itd.*
- *ne postoji relacija <>, samo !=*

Ostale razlike su u imenima nekih modula i njihovom sadržaju. Kao i uvijek, s F1 se može dobiti kompletna on-line dokumentacija.U njima je sintaksa Pythona opisana regularnim izrazima.

Autor duguje posebnu zahvalnost ***mr. sc. Mirku Smilevskom*** *koji je pratio stvaranje ove knjige i svojim sugestijama i ispravkama pogrešaka pomogao da tekst bude što prihvatljiviji.*

Na kraju, Autor će vam biti zahvalan ako pažljivo pročitate ovu knjigu, prihvatite barem jedan njezin djelić i primijenite u svojoj praksi. Možete mu se slobodno javiti, sa sugestijama ili eventualnim pohvalama, na e-mail adresu:

zdovedan@hotmail.com

Zagreb, srpnja 2021.

A u t o r

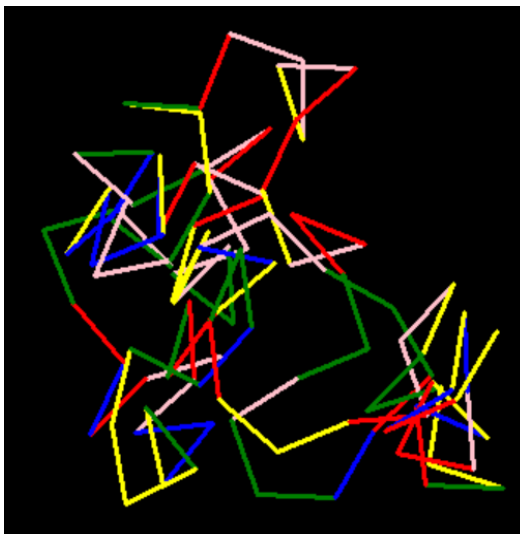
0.

OSNOVE

Izučavanje jezika za programiranje, algoritama, tipova i struktura podataka te programiranja zahtijeva strog, formalni pristup. Također pretpostavlja solidno predznanje iz matematike, posebno diskretne matematike: teorije skupova, grafova i matematičke logike. Sve to, veoma sažeto, dano je u ovom uvodnom dijelu.

škrabotina.py

```
# "škrabotina u 2D"
from turtle import *
from random import randint, choice
bgcolor ("black"); pensize (3)
def crta (n, d):
  tracer (0)
  for x in range (n):
    r = rt (randint (0, 360))
    l = lt (randint (0, 360))
    color (choice(["blue", "red",
           "green", "yellow", "pink"]))
    choice ([r, l]);  fd (d)
crta (100, 50)
```



Matematičke osnove

SKUPOVI

Skup intuitivno shvaćamo kao kolekciju elemenata (ili članova) koji posjeduju izvjesna svojstva. Elemente skupa pišemo između vitičastih zagrada, odvojene zarezom. Na primjer, skup brojki možemo napisati kao *Brojke* = {0,1,2,3,4,5,6,7,8,9}.

Ako je α element skupa *S*, piše se $\alpha \in S$ i čita "α je element skupa *S*", a ako nije, piše se $\alpha \notin S$ i čita "α nije element skupa *S*". Na primjer, $5 \in Brojke$, pet $\notin$ *Brojke*.

Elementi skupa mogu biti jedinke, koje predstavljaju same sebe, ili neki drugi skupovi. Prikazuje se samo jedno pojavljivanje nekog elementa u skupu. Redoslijed pisanja elemenata skupa nije bitan.

Definira se i prazan skup, skup koji ne sadrži nijedan element. Označavat ćemo ga s $\varnothing$.

Podskupovi

Do pojma podskupa dolazi se promatranjem dijela nekog skupa. Kaže se da je *B* podskup skupa *A* ako je svaki element *x* iz *B* ujedno i element skupa *A*. U tom slučaju piše se

$$B \subseteq A$$

Na primjer, skup *P*={2,4,6,8} podskup je skupa *Brojke*, tj. $P \subseteq Brojke$. Piše se

$$B \subset A$$

i kaže da je *B* pravi podskup skupa *A* ako u *A* postoji najmanje jedan element koji nije u *B*. Ako se želi istaknuti da *B*, u krajnjem slučaju, može biti cijeli *A*, piše se $B \subseteq A$.

Zadavanje skupova

Skup se S smatra zadanim ako je nedvosmisleno rečeno, objašnjeno ili specificirano, što su elementi tog skupa. U nekim se slučajevima elementi skupa jednostavno navedu između vitičastih zagrada. Na primjer:

$$S = \{1,3,a,d\}$$

Operacije sa skupovima

Postoji nekoliko operacija sa skupovima koje se mogu koristiti pri izgrađivanju novih skupova. Neka su A i B skupovi. Unija od A i B, napisana kao $A \cup B$, jest skup koji sadrži sve elemente skupa A zajedno sa svim elementima skupa B:

$$A \cup B = \{x \mid x \in A \text{ ili } x \in B\}$$

Presjek skupova A i B, $A \cap B$, skup je elemenata sadržanih i u A i u B:

$$A \cap B = \{x \mid x \in A \text{ i } x \in B\}$$

Razlika skupova A i B, $A \setminus B$, skup je elemenata koji pripadaju skupu A, a nisu u B:

$$A \setminus B = \{x \mid x \in A \text{ i } x \notin B\}$$

RELACIJE

Izravni ili Kartezijev produkt dvaju skupova A i B je skup:

$$A \times B = \{(a,b) \mid a \in A, b \in B\}$$

Element tako nastalog skupa, (a,b), naziva se uređeni par. Na primjer, ako je A={a,b}, B={0,1}, Kartezijev produkt $A \times B$ jest skup

$$\{(a,0), (a,1), (b,0), (b,1)\}$$

Prva komponenta bilo kojeg para mora biti iz A, druga iz B. Zbog toga (0,a) nije element skupa $A \times B$.

Relacija, označimo je s ρ, jest bilo koji podskup Kartezijevog produkta skupova. Na primjer, ako je N={1,2}, M={0,1,2,3,4,5}, relacija ρ, $\rho \subseteq N \times M$, može biti

$$\rho = \{(1,0), (1,1), (1,2), (1,3), (2,0), (2,1), (2,2)\}$$

Relacija ρ u ovom primjeru označuje svojstvo parova (x,y), $x \in N$, $y \in M$, da im je zbroj manji ili jednak 4. Isto svojstvo može se napisati kao $x \rho y$ što se čita "x je u relaciji ρ sa y" ili "između x i y postoji relacija ρ".

FUNKCIJE

Funkcija (preslikavanje, transformacija) *f* iz skupa A u skup B jest relacija iz A u B takva da, ako su (a,b) i (a,c) u *f*, vrijedi b=c.

Ako je (a,b) u *f* često se piše $b = f(a)$. Kaže se da je $f(a)$ definirano ako postoji b u B tako da je (a,b) u *f*. Ako je $f(a)$ definirano za sve a iz A, kaže se da je *f* potpuno preslikavanje sa A u B. Ako to nije ispunjeno, *f* je djelomično preslikavanje iz A u B. U oba slučaja piše se

$$f: A \to B$$

i kaže da je A domena, a B kodomena funkcije *f*.

MATEMATIČKA LOGIKA

Dobro poznavanje elemenata matematičke logike osnovni je uvjet shvaćanja problema programiranja. Ovo podpoglavlje predstavlja kratki repetitorij matematičke logike.

Algebra sudova osnova je drugih dijelova matematičke logike, prijeko potrebna za njihovo razumijevanje. Osnovni pojmovi, objekt ili elementarni sud, najčešće se objašnjavaju primjerima:

"Broj 100 djeljiv je s 4"
"Danas je lijepo vrijeme"
"Štef nema djevojku"
"Mjesec je veći od Zemlje"
"17 je prost broj"
"Darko je stariji od brata Igora"

Svi elementarni sudovi moraju imati jedno i samo jedno svojstvo:

"biti istinit" ili "biti lažan"

Na primjer, sud "7 je veće od 5" je istinit, a sud "8 je prost broj" nije istinit. U daljnjem tekstu elementarne sudove označivat ćemo malim slovima *a*, *b*, *c*, itd.

Za primjene u programiranju posebno su važni elementarni sudovi nazvani relacijski izrazi. Općenito relacijski izraz sadrži dva izraza istog tipa (na primjer, dva aritmetička izraza) između kojih je napisana jedna od relacija:

= jednako < manje > veće
≠ različito ≤ manje ili jednako ≥ veće ili jednako

Na primjer, relacijski izraz (elementarni sud) "9>5" čitat ćemo "9 je veće od 5".

Od elementarnih sudova moguće je, uz pomoć nekoliko logičkih operacija, graditi složene sudove. Jasno je da će i složeni sud biti istinit ili lažan. U daljnjem tekstu govorit će se o njegovoj "vrijednosti istinitosti", koja će biti označena s T za istinito, i s F za lažno.

Vrijednost istinitosti složenog suda ovisi o istinitosti sudova od kojih je složeni sud izgrađen. Postoji jedna unarna i nekoliko binarnih logičkih operacija (logičkih veznika ili konektiva). Ovdje su opisane samo one osnovne; negacija, kao unarna logička operacija, te dvije binarne logičke operacije:

- konjunkcija
- disjunkcija

Negacija je unarna logička operacija i ujedno najjednostavnija operacija algebre sudova. U prirodnom jeziku odgovara joj približno riječ "ne". Ako negaciju označimo s "$\neg a$", njezino djelovanje na sud a dano je u sljedećoj tablici:

a	$\neg a$
F	T
T	F

Ako je a neki sud, na primjer "5 je djelitelj od 7", $\neg a$ novi je sud "5 nije djelitelj od 7". Sud $\neg$a čita se "ne a" ili "non a".

Ako su a i b sudovi, onda je $a \wedge b$ novi složeni sud ili konjunkcija sudova a i b. Znak "$\wedge$" čita se "i" ili "et". Djelovanje operacije konjunkcije definirano je sljedećom tablicom:

a	b	$a \wedge b$
F	F	F
F	T	F
T	F	F
T	T	T

Treba zapamtiti da će složeni sud $a \wedge b$ biti istinit onda i samo onda ako su i a i b istiniti.

Operacija disjunkcije najčešće se označuje sa "$\vee$" i čita "ili". Treba napomenuti da veznik "ili" u mnogim jezicima ima dva različita značenja. U jednom slučaju radi se o tzv. "isključnom", u drugom o "neisključnom" vezniku "ili". Razlika među njima pokazana je u sljedećem primjeru:

Ako su a i b dva lažna suda, lažan je i složeni sud $a \vee b$. Ako je a istinito, a b lažno, ili a lažno, a b istinito, istinit je i složeni sud $a \vee b$. Što je, međutim, s vrijednošću istinitosti suda $a \vee b$ ako su i a i b istiniti? U prvom slučaju, ako se složena izjava smatra istinitom, govori se u neisključnoj disjunkciji, u drugom o isključnoj disjunkciji.

U matematičkoj logici operacija disjunkcije odgovara neisključnom vezniku "ili". Iz prethodnih razmatranja slijedi definicija: disjunkcija sudova a i b, napisana kao $a \vee b$, složen je sud koji je lažan onda i samo onda ako su i a i b lažni. Iz te definicije slijedi tablica:

a	b	$a \vee b$
F	F	F
F	T	T
T	F	T
T	T	T

Logički izrazi

Uporabom navedenih logičkih operacija i uvođenjem zagrada mogu se, kao i u algebri, graditi razni složeni sudovi ili logički izrazi. Na primjer:

$(a \vee b) \wedge c \quad (a \wedge b) \vee d \quad \neg a \vee b$

Logičke varijable koje se pojavljuju u izrazima mogu biti elementarni sudovi, na primjer "x<6", ili nosioci logičkih vrijednosti dobivenih kao rezultat prethodnog izračunavanja (nekog logičkog izraza). Također se može pojaviti logička konstanta F, čija je vrijednost istinitosti uvijek "laž", i logička konstanta T čija je vrijednost istinitosti uvijek "istina". Logički izrazi koji sadrže samo operacije negacije, konjunkcije i disjunkcije, te zagrade, nazivaju se Booleove formule. Za Booleove formule vrijede sljedeći zakoni:

Zakon komutacije: $x \vee y \equiv y \vee x$ $x \wedge y \equiv y \wedge x$	*Zakon asocijacije*: $x \vee (y \vee z) \equiv (x \vee y) \vee z$ $x \wedge (y \wedge z) \equiv (x \wedge y) \wedge z$
Zakon idempotentnosti: $x \vee x \equiv x$ $x \wedge x \equiv x$	*Zakon distribucije*: $x \vee (y \wedge z) \equiv (x \vee y) \wedge (x \vee z)$ $x \wedge (y \vee z) \equiv (x \wedge y) \vee (x \wedge z)$
De Morganov zakon: $\neg (x \vee y) \equiv \neg x \wedge \neg y$ $\neg (x \wedge y) \equiv \neg x \vee \neg y$	*Zakon dvostruke negacije*: $\neg \neg x \equiv x$

Posebno je česta uporaba de Morganovog zakona u programiranju.

Jezici za programiranje

Kompjuter može uraditi samo ono za što mu je netko dao instrukcije (program) - niz logičkih i aritmetičkih operacija napisanih jezikom kompjutera. Kompjuter takve instrukcije izvršava brzo i gotovo nepogrešivo, upravo onako kako su zadane.

Kompjuter može izvršiti samo mali broj veoma jednostavnih operacija. Na primjer, oduzimanje, množenje i dijeljenje svodi na operacije zbrajanja i pomicanja znamenki. Kompjuter „razumije" i može izvršiti samo instrukcije strojnog jezika. S razvojem kompjutera usavršavao se i jezik za komuniciranje (programiranje), pa razlikujemo četiri generacije:

1) Prva generacija - strojni jezici
2) Druga generacija - simbolički (asemblerski) jezici
3) Treća generacija - jezici za programiranje visoke razine
4) Četvrta generacija - jezici četvrte generacije (jezici krajnjih korisnika)

Generacije jezika za programiranje ne treba poistovjećivati s generacijama kompjutera. Na primjer, danas su u upotrebi kompjuteri četvrte generacije, na kojima nalazimo sve četiri generacije jezika za programiranje.

Osim dane podjele jezika za programiranje, postoje i druge. Prema jednoj od njih, na primjer, svi jezici za programiranje dijele se na:

- proceduralne
- neproceduralne

a prema drugoj, postoje slijedeće kategorije jezika:

- proceduralni
- objektno orijentirani
- funkcionalni
- logički

Radi izbjegavanja takvih i sličnih problema, razvijeni su jezici visoke razine. U osnovi, jezici visoke razine omogućuju programeru da piše algoritme u prirodnijoj notaciji u kojoj se ne treba baviti mnogim detaljima vezanim za neki specifični kompjuter. Na primjer, neusporedivo je ugodnije pisati A=B+C nego niz asemblerskih instrukcija.

Danas je u široj uporabi petnaestak jezika visoke razine. To su, napisani alfabetski: Ada, APL, BASIC, Quick BASIC, C, C++, COBOL, Delphi, FORTRAN, Java, JavaScript, LISP, Pascal, PHP, Python itd. Razlikuju se po svom stupnju bliskosti matematičkome ili prirodnim jezicima, s jedne strane, i strojnom jeziku, s druge strane. Također se razlikuju po vrsti problema čijem su rješavanju najbolje prilagođeni.

DEFINIRANJE JEZIKA ZA PROGRAMIRANJE

Jezici za programiranje daleko su jednostavniji od prirodnih (npr. hrvatskoga, engleskoga, francuskoga, talijanskoga, itd). Osim toga, „rečenice" jezika za programiranje mogu se opisati strogim pravilima, bez

izuzetaka i s jedinstvenim značenjem. Na žalost, u mnogim knjigama o jezicima za programiranje i školama programiranja jezik se nastoji prikazati isključivo primjerima, a onome tko ga uči preostaje da sam zaključi i izvede opća pravila za pisanje naredbi. S druge strane, još uvijek ne postoji „recept" koji bi upućivao na prave puteve definiranja i učenja jezika za programiranje. Međutim, iskustvo pokazuje da se najveći učinci postižu ako se pri učenju jezika istaknu tri stvari: leksička struktura, sintaksna struktura i semantika jezika.

Leksička struktura

Definirati leksičku strukturu nekog jezika znači definirati alfabet i rječnik. Alfabet je skup svih znakova koji se koriste u pisanju. To su slova, znamenke, operacije, te drugi znakovi.

Rječnik je skup riječi (simbola) definiranih nad alfabetom. Riječ (ili simbol) je niz znakova iz alfabeta koji se može promatrati kao jedinstvena, nedjeljiva cjelina. Na primjer, neka je dan niz A-B*C. Sastoji se od pet znakova koji mogu biti grupirani na nekoliko načina. Može se smatrati da niz A-B čini jednu riječ (u jeziku COBOL to bi bilo ime varijable). Ili se A-B može promatrati kao varijabla A minus varijabla B (kao što je u FORTRANu i nekim drugim jezicima). Što je od ovog korektno, ovisi o definiciji leksičke strukture jezika. Njome će biti propisano koje riječi treba tretirati kao imena, a koje kao operatore.

Radi boljega sagledavanja leksičke strukture nekoga jezika uobičajeno je rječnik podijeliti u klase riječi (simbola) koje imaju zajednička svojstva: brojeve, imena, rezervirane riječi, imena funkcija, ostale (posebne) simbole itd.

U opisu leksičke strukture često ćemo koristiti notaciju regularnih izraza Pythona. Često ćemo je kombinirati s proširenom Backus-Naurovom formom (ENBF) i/ili sintaksnim dijagramima. Na primjer, *slovo* se u EBNF-u može napisati kao

slovo : *malo_slovo* | *veliko_slovo*

Reći ćemo da je *slovo* ime sintaksne kategorije koju treba dalje definirati. Ovdje je definirana preko dvije nove sintaksne katogorije: *malo_slovo* ili *veliko_slovo*. Meta simbol „|" čitamo „ili". Dalje se malo_slovo može definirati regularnim izrazom:

malo_slovo : (a | b | c | ... | x | y | z)

što čitamo "malo slovo je "a" ili "b" ili "c" ili ... ili "x" ili "y" ili "z". Napomena: Potpuna definicija malog slova engleske abecede podrazumijeva da se mora napisati svih 26 slova. Dani regularni izraz ekvivalentan je izrazu:

[abc...xyz]

što čitamo na isti način kao u prethodnoj notaciji. Dakle, uglate zagrade naznačuju da se bira jedan od navedenih znakova (i u ovom slučaju moraju biti napisani svi znakovi). Ako znakovi čine uređeni niz prema ASCII uređenju, kao što je slučaj u ovom primjeru, može se rabiti skraćena notacija, pa se malo i veliko slovo može definirati sa:

malo_slovo : [a-z] ***veliko_slovo*** : [A-Z]

Dodajmo i definiciju brojke:

brojka : [0-9]

Ako je regularni izraz napisan kao **(...)?** ili **[]?**, značenje je znaka "**?**" da se mogu izostaviti znakovi navedeni unutar okruglih ili uglatih zagrada ili izabrati samo jedanput. Ako je iza zagrada napisano "**+**", **()+** ili **[]+**, znakovi se moraju birati jedanput ili više puta. I, ako je izaz zagrada napisano "*****", znakovi mogu biti birani nijedanput, jedanput ili više puta. Na primjer, regularni izraz

[-+]? [1-9] [0-9]*

generira nizove: -1 +1 1 +9 10 -707

Znak "–" ili "+" može biti izostavljen ili napisan jedanput, jedna brojka od "1" do "9" mora biti napisana, a potom brojke od "0" do "9" mogu biti izostavljene, napisane jedanput ili više puta.

Sintaksna struktura

Sintaksna (ili sintaktička) struktura jezika utvrđuje grupiranje leksičkih konstrukata u šire strukture, nazvane *sintaksne kategorije*. Pravila koja određuju pripada li niz simbola jeziku ili ne nazivaju se sintaksa jezika.

Danas je u uporabi nekoliko notacija ili načina prikazivanja sintakse nastalih iz Backus-Naurove forme (BNF), prvi puta objavljene 1963. godine u opisu jezika ALGOL 60. Upravo od pojave Pascala popularni su *sintaksni dijagrami*. Ukratko, sintaksni dijagrami sadrže sljedeće simbole:

Simbol jezika. Ono što je upisano unutar ovog simbola dio je naredbe jezika opisane dijagramom, odnosno definira samo sebe. Na primjer:

Općenito ime ili konstrukt koji je već definiran ili će biti definiran naknadno. Ono što je upisano u pravokutniku nije element jezika, već služi za opis strukture naredbe jezika. Drugim riječima, to je sintaksna kategorija. Na primjer:

Pokazuje mogući tok kretanja kroz dijagram (usmjeren strelicom).

Ako u tekstu upućujemo na sintaksnu kategoriju (ono što je upisano u pravokutniku), pisat ćemo je izmedu znakova "<" i ">". Na primjer:

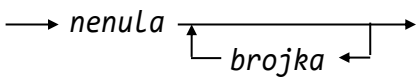
```
<naredba>  <ime>  <broj>  <izraz>
```

Notacija sintaksnih dijagrama primijenjena u opisu sintakse Pythona može biti pojednostavljena:

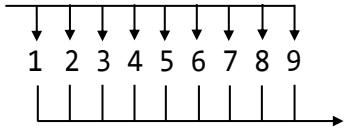
- sintaksne kategorije bit će pisane nakošenim malim slovima
- ako je ime sintaksne kategorije sačinjeno od više riječi, riječi će biti razdvojene donjom crtom (podcrtom)
- simboli jezika bit će pisani normalnim slovima ili nizovima znakova u boji ili ne

Sada se postavlja pitanje: kako „čitati" sintaksne dijagrame? Odgovor je jednostavan: treba krenuti slijeva i prolazeći kroz dijagram definiranim putovima (pazeći na njihovo usmjerenje) doći do izlaza. Izlaz je strelica iza koje nema više nijednog simbola. Pri tome treba zapisivati sadržaj simbola dijagrama: simbole jezika izravno prepisivati, a one koji predstavljaju sintaksne kategorije napisati između znakova "<" i ">", potom ih zamijeniti njihovom definicijom (njihovim dijagramom). Postupak treba ponavljati sve dok se ne dobije niz koji je sačinjen samo od simbola iz jezika (rječnika). Na primjer, definirajmo primjenom sintaksnih dijagrama pravilo pisanja prirodnih brojeva:

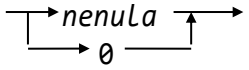
prirodni_broj:



nenula:



brojka:



Pravilo pisanja prirodnih brojeva sadrži dva simbola koji nisu u jeziku. Sve tri definicije u potpunosti opisuju tvorbu prirodnih brojeva. Na primjer, krenuvši od definicije prirodnog broja prvo nailazimo na *nenula* ili <nenula>. Poslije toga može se završiti ili krenuti putem preko *brojka*. Pretpostavimo da smo završili. Međutim, *nenula* nije element jezika, pa ga moramo zamijeniti njegovom definicijom. Ako pogledamo definiciju sintaksne kategorije *nenula*, zaključit ćemo da moramo izabrati jednu brojku od 1 do 9. Na primjer, izaberimo brojku 8. Dakle, umjesto *nenula* možemo napisati 8. To je ujedno i prirodni broj. Evo još jednog primjera primjene danih pravila:

nenula	*brojka*	*brojka*
9	*brojka*	*brojka*
9	0	*brojka*
9	0	1

Uočimo da dijagram kojim je definiran *prirodni_broj* sadrži petlju koja osigurava da generiramo potpuni (beskonačni) skup zapisa beskonačnog skupa prirodnih brojeva. Također primijetimo da je bilo neophodno uvesti strukturu *nenula* koja je osigurala da prirodni broj ne smije biti 0, niti smije početi nulom.

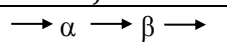
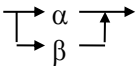
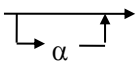
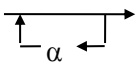
Već iz ovog jednostavnog primjera mogu se uočiti prednosti primjene sintaksnih dijagrama, odnosno definiranje pravila pisanja jezika, u njegovom učenju. Navedimo samo najbitnije:

1) Kompaktnim konačnim pravilom - sintaksnim dijagramom - moguće je opisati veliki skup kombinacija u generiranju dane naredbe.
2) Pravila predočena dijagramima brže se pamte i uče, jer većina ljudi bolje pamti vizualno.
3) Promatranjem svih naredbi jezika moguće je uočiti dijelove koji su jednaki. Uvođenjem posebnih imena za takve dijelove znatno se pojednostavljuje učenje jezika.

Međutim, sintaksni dijagrami neće uvijek biti dovoljni za potpuni opis svih naredbi. Naime, pojedine dodatne (kontekstne) uvjete koji moraju biti ispunjeni prilikom pisanja nekih naredbi nije moguće opisati dijagramom. Na primjer, u Pythonu izraz `A+B` napisan je korektno ako su `A` i `B` varijable istog primitivnog tipa, brojevi, dva stringa ili dvije liste.

Osim toga, postojat će situacije gdje je moguć opis sintaksnim dijagramom, ali bi bio prekompliciran, kao na primjer da ime može sadržavati od jednog do *n* znakova. I u jednom i u drugom slučaju davat ćemo dodatna pravila riječima. Na primjer, ako želimo definirati pravilo za tvorbu prirodnih brojeva, od `1` do `9999999`, onda ćemo uz dano pravilo reći da ono vrijedi uz uvjet da se *`brojka`* smije upotrijebiti do `6` puta ili ćemo u sintaksnom dijagramu označiti maksimalni broj prolazaka određenim putem.

Česta je uporaba još jedne notacije koju ćemo također rabiti u našem opisu sintakse Pythona. Sintaksne kategorije i riječi jezika pišu se jednako kao i u pojednostavljenom sintaksnom dijagramu. Ako su α i β dvije sintaksne kategorije, u sljedećoj je tablici opisano značenje te notacije uz pomoć sintaksnih dijagrama:

Pravilo	*Opis*	*Značenje*
α β	slijed	
α | β	alternativa (α ili β)	
[α]	α izostavljeno ili napisano jedanput	
{α}	α izostavljeno ili napisano jedanput, dvaput, triput, ...	

Evo pravila pisanja prirodnih brojeva u ovoj notaciji:

```
prirodni_broj: nenula { brojka }
nenula:        1| 2| 3| 4| 5| 6| 7| 8| 9
brojka:        nenula | 0
```

Algoritmi

Algoritam je postupak ili pravilo za sustavno rješavanje određene vrste problema. Sastoji se od opisa konačnog skupa koraka. Svaki od njih sadrži jednu ili više izjava, a svaka izjava jednu ili više operacija.

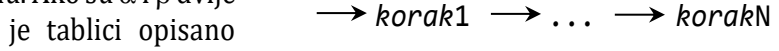
Za prikazivanje algoritma potrebno je usvojiti određenu notaciju - tekstualnu, grafičku, skupom formula, kombiniranu ili neku drugu. No, radi implementacije algoritma na kompjuteru, potrebno je opisati ga u nekom, za tu svrhu odabranom, jeziku za programiranje. Upravo je Python kao stvoren za to!

OSNOVNI ALGORITAMSKI KONSTRUKTI

Oblikovanje algoritma zahtijeva poznavanje nekoliko algoritamskih konstrukata. Tri su osnovna konstrukta, čijom se kompozicijom može oblikovati algoritam kao rješenje zadanog problema: slijed (sekvenca), grananje (selekcija) i ponavljanje (iteracija).

Slijed

Slijed, odnosno sekvenca, jest skup jednog ili više koraka algoritma koji se odvijaju sekvencijalno - redno, u nizu - jedan za drugim. Broj je koraka proizvoljan. Svaki korak može biti skup od jedne ili više izjava, a može predstavljati i cijeli algoritamski konstrukt - sekvencu, selekciju ili iteraciju. Shematski, sekvenca se može prikazati kao



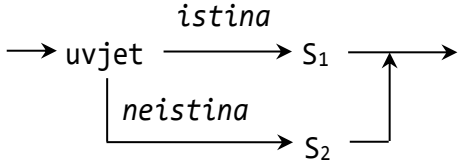
Na primjer, algoritam

1) Unesi `N` vrijednosti i zbroji ih
2) Izračunaj prosječnu vrijednost
3) Ispiši zbroj i prosječnu vrijednost

jest sekvenca triju koraka koji se odvijaju jedan za drugim, a svaki može biti algoritamski konstrukt.

Grananje

Često je u algoritmima potrebno odlučiti koju od dviju ili više sekvenci treba riješiti s obzirom na postavljeni uvjet. Jednostavniji oblik grananja jest selekcija, izbor jedne od dviju sekvenci, ovisno o tome je li postavljeni uvjet istinit, kad bi se izvela prva sekvenca (S_1), ili nije istinit, kad bi se izvela druga (S_2). To je poznati IF-THEN-ELSE konstrukt ili naredba za odabir u većini jezika za programiranje visoke razine, a shematski se može prikazati kao



Selekcija može imati jednu granu „praznu“, tj. može biti bez *ELSE grane*. Neki jezici, pa tako i Python, imaju prošireno značenje grananja u kojem može biti jedna ili više *ELIF grana* sa svojim uvjetima i naredbama koje će biti izvršene ako se redom dođe do tog uvjeta (svi prethodni nisu istiniti). Na kraju je eventualno *ELSE grana* koja se izvršava ako nije ispunjen nijedan uvjet u *ELIF granama*.

Ponavljanje

Ponavljanje, odnosno iteracija, jest sekvenca koraka algoritma koja se odvija izvjestan broj puta, sve dok je postavljeni uvjet ispunjen (okončava se kad uvjet više nije ispunjen – kad postane neistinit), ili sve dok određeni uvjet nije ispunjen (okončava se kad je uvjet ispunjen – postao je istinit). Prvi od iterativnih konstrukata poznat je pod imenom DO-WHILE, drugi kao DO-UNTIL (ili REPEAT-UNTIL).

U DO-WHILE konstruktu (ili, kako programeri kažu „WHILE petlja“) slijed koraka algoritma *S* zaklonjen je i bit će izvršen samo ako je uvjet ispunjen (istinit). Drugim riječima, sekvenca *S* neće biti izvršena, odnosno, bit će izvršena jedanput ili više puta, ovisno o ispunjenju uvjeta. Tada je pretpostavka da će postavljeni uvjet poslije konačnog broja koraka biti ispunjen. U suprotnom ćemo imati beskonačnu petlju!

Neki jezici, Python također, imaju i *FOR petlju* u kojoj se iteracija, izvršavanje sekvence, izvodi zadani broj puta.

REKURZIJE

Rekurzija je, općenito, svojstvo objekta da se definira i pomoću samoga sebe (ili, da sudjeluje u definiciji samog sebe).

U matematici, rekurzija je takav algoritam, odnosno notacija ili pravilo opisa objekta, u čijim se definicijama kao parametar ili argument pojavljuju (i) sami objekti. Tako, na primjer, funkcija faktorijela nenegativnog cijelog broja definirana je rekurzivno sljedećim pravilom:

$$0! = 1$$
$$n! = n \times (n-1)!, \qquad n>0$$

U definiciji funkcije $n!$ javlja se rekurzivno, kao parametar, ista funkcija, ali s drugom vrijednošću argumenta.

Rekurzija u matematici omogućuje kompaktno definiranje proizvoljno velikog skupa vrijednosti objekata minimalnim skupom izjava. Na primjer, skupom izjava:

1) 1 je prirodni broj,
2) ako je n, $n \geq 1$, prirodni broj, $n+1$ također je prirodni broj

rekurzivno je definirano jedno svojstvo prirodnih brojeva.

Pri izračunavanju vrijednosti elemenata rekurzivno definiranih objekata, svaka pojava objekta zamjenjuje se njezinom definicijom. Na primjer, faktorijel broja 3 rekurzivnim algoritmom izračunava se kao

$$3! = 3\times(3-1)! = 3\times2!$$

a $2!$ se, istim algoritmom, izračunava kao

$$2! = 2\times(2-1)! = 2\times1!$$

Dalje je

$$1! = 1\times(1-1)! = 1\times0!$$

Iz definicije faktorijela je $0! = 1$, pa se konačno dobije:

$$3! = 3\times2! = 3\times2\times1! = 3\times2\times1\times0! = 3\times2\times1\times1 = 6$$

Rekurzivni, kao i svaki drugi algoritam, mora biti karakteriziran sljedećim svojstvima:

1) da je opisan konačnim brojem koraka,
2) da izračunavanje vrijednosti okonča nakon konačnog broja izračunavanja.

Dok je prvo svojstvo relativno jednostavno ostvariti, za ostvarenje drugog svojstva potrebno je uvesti tzv. uvjet okončanja. Ako se rekurzivni algoritam promatra kao funkcija jednog ili više argumenata, kažemo da će izračunavanje vrijednosti po tom algoritmu okončati ako se pri svakom pozivu rekurzivne funkcije bar jedna od izračunatih vrijednosti „približi“ uvjetu okončanja. Tako u primjeru izračunavanja faktorijela broja n, što smo mogli napisati kao

$$f(0) = 1$$
$$f(n) = n \times f(n-1), \quad n\geq1$$

uvjet okončanja je `f(0)`. U svakom se koraku vrijednost argumenta umanjuje za `1` i tako se približava uvjetu okončanja.

U jezicima za programiranje rekurzivni algoritmi implementirani su kao potprogrami (procedure i funkcijski potprogrami) radi mogućnosti rekurzivnog pozivanja. Međutim, rijetki su jezici za programiranje visoke razine u kojima je dopušteno opisivanje i izračunavanje rekurzivnim algoritmima.

OD ALGORITMA DO PROGRAMA

Algoritam za rješenje nekog problema jest niz koraka koji moraju biti izvedeni da bi se dobilo rješenje problema na temelju informacija dobivenih iz specifikacije (opisa) problema.

Dakle, kompjuterski program u kojem se takav algoritam implementira mora prihvatiti i upamtiti dane informacije (podatke) nad kojima će biti izvršen niz operacija, u određenom broju koraka, da bi problem bio riješen. Iskustvo pokazuje da se najbolji učinak u rješavanju nekog problema postiže kad izabrana metoda rješavanja i program prate strukturu problema.

Na primjer, ako se za rješenje nekoga problema primijeni silazna strategija razvoja, i program treba biti tako napisan. To znači da se najprije u glavnom programu definira temeljna struktura rješavanja s nizom poziva potprograma koji, dalje, predstavljaju rješenje problema na sljedećoj (nižoj) razini. Istodobno se uvode potrebni tipovi podataka, konstante i varijable.

Prevodioci

Evolucija jezika za programiranje, od asemblerskih jezika do jezika visoke razine, uvela je potrebu za posebnim programima – prevodiocima. Kako kompjuter neposredno izvršava instrukcije u svom strojnom jeziku, neophodno je prevesti programe u taj jezik.

VRSTE PREVODILACA

Prevodilac je program koji instrukcije izvornog jezika prevodi u program izražen ciljnim jezikom. Ako izvorni jezik pripada klasi jezika visoke razine, kao što su to na primjer Pascal ili C, a ciljni je asemblerski ili strojni jezik, prevodilac se naziva kompilator. Izvršavanje programa pisanog u jeziku visoke razine u osnovi je dvodijelni proces. Izvorni program najprije se kompilira, tj. prevede u ciljni program, potonji se zatim smjesti u memoriju i izvršava. Budući da postoji nekoliko vrsta izvornih i ciljnih jezika, razlikuje se nekoliko vrsta prevodilaca.

Asembler je prevodilac (a ne jezik, kako se najčešće misli!) koji prevodi program pisan u asemblerskom jeziku u strojni jezik. Neki prevodioci, a takav je i prevodilac Pythona, transformiraju program pisan u izvornom jeziku u pojednostavljeni jezik, nazvan "međukod", koji se može direktno izvršiti koristeći program zvan interpretator.

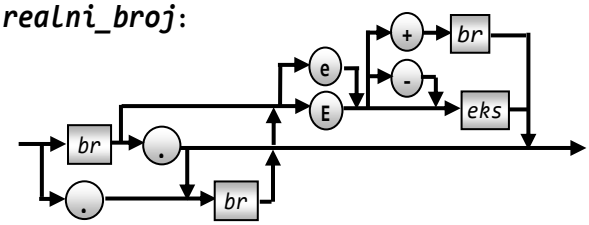
Termin predprocesor katkad se upotrebljava za prevodioce koji prihvaćaju programe pisane u jednom jeziku visoke razine i prevode ga u ekvivalentan program u drugom jeziku visoke razine.

1.

UVOD U PYTHON

Počet ćemo „govoriti pythonski“ u njegovom interaktivnom ili Shell modu koji će nam pomoći da, uvodeći neformalno brojeve, izraze, stringove, standardne funkcije i procedure, već poslije pola sata učenja rješavamo neke brojčane probleme. Podrazumijeva se da sve vježbe i programe izvršavate na svom računalu.

realni_broj:



br: [0-9]⁺ ***eks***: ([0-2][0-9][0-9] | 30[0-8])

```
>>> Fibonaccijev niz (2)
>>> FIB = """
Fn = "t = b; b = a+b; a = t;"
n = int (input ('n = '))
I = int (input ('Ispis svih članova ' +
        '(1); ili n-tog (0) '))
a = 0; b = 1; exec ("print (b,
                    end = ' ')" *I)
x = Fn +" print (b, end = ' '); " *I
exec (x *(n-2))
I = 0; exec (Fn); print (b) """
>>> exec (FIB)
n = 20
Ispis svih članova (1); ili n-tog (0) 0
6765
>>> exec (FIB)
n = 20
Ispis svih članova (1); ili n-tog (0) 1
1 1 2 3 5 8 13 21 34 55 89 144 233 377
610 987 1597 2584 4181 6765
```

```
>>> Interesantni izrazi
>>> \
_a = "B = B*10 +1; print (B, "   + \
     "'x', B, '=', B**2); ";       \
_b = "B = B*10 +i; print (B, "   + \
     "'x 9 +', i+1, '=', "        + \
     "B*9 +i+1); i = i +1; ";      \
_c = "B = B*10 +i; print (B, "   + \
     "'x 8 +', i, '=', B*8 +i);" + \
     "i = i +1; ";                 \
_d = "B = B*10 +i; print (B, "   + \
     "'x 9 +', i-2, '=', "        + \
     "B*9 +i-2); i = i -1; "
```

```
>>> B = 0;  i = 1; exec (9 *_b)

1 x 9 + 2 = 11
12 x 9 + 3 = 111
123 x 9 + 4 = 1111
1234 x 9 + 5 = 11111
12345 x 9 + 6 = 111111
123456 x 9 + 7 = 1111111
1234567 x 9 + 8 = 11111111
12345678 x 9 + 9 = 111111111
123456789 x 9 + 10 = 1111111111
```

Instaliranje Pythona

Python je osmislio **Guido van Rossum** krajem osamdesetih godina prošloga stoljeća. Bio je instaliran kao nasljednik programskog jezika ABC. Van Rossum je dalje bio glavni autor u nadgradnji Pythona, od inačice 2.0 objavljene 16.10.2000. koja je sadržavala značajne dogradnje, pa do inačice 3.0 objavljene 3.12.2008. godine. Poslije duljeg perioda testiranja mnoge su karakteristike te inačice vraćene u inačice 2.6 i 2.7.

Danas se paralelno razvijaju dvije inačice Pythona: 2.7.x i 3.x.y. U trenutku pisanja ovoga teksta aktualne su bile inačice **2.7.18** i **3.9.0**. Bez obzira na još uvijek zastupljenost "staroga" Pythona, inačice 2.7.x", mi smo se odlučili za inačicu 3.9.0. jer će se nadgradnje inačice 2.7.x uskoro ugasiti, a na inačici 3.x.y ne. Možete je instalirati na svoje računalo s adrese

http://www.python.org/.

Instalacija će sadržavati i dokumentaciju Pythona.

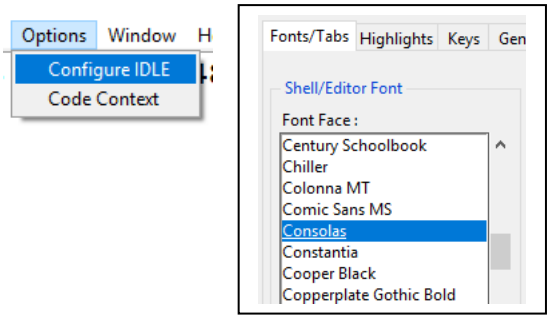
Interaktivni (Shell) môd

Ljuska (engl. *shell* - ljuska, školjka) je termin koji se često koristi i često je pogrešno shvaćen. U računalnoj se znanosti općenito doživljava kao dio softvera koji pruža sučelje za korisnika na neki drugi softver ili operacijski sustav. Dakle, ljuska može biti sučelje između operacijskog sustava i njegovih usluga. Često je to sinonim za sučelje izvedeno kao CLI (command-line interface) ili `GUI` (graphical user interface). Poznate su, na primjer, ljuske `Bourne-Shell` (`Linux` i `Unix`), `C-Shell` ili `Bach shell`. U većini se operacijskih sustava ljuske koriste u tzv. interaktivnom modu rada. Tako je i u Pythonu. Poslije instalacije Pythona i njegova poziva imali biste sljedeće:

```
Python 3.9.0 (tags/v3.9.0:9cf6752, Oct 5
2020, 15:34:40) [MSC v.1927 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or
"license()" for more information.
>>>
```

Ovo je primjer poziva Pythona instaliranoga na Windows platformi. Moguće su instalacije i na drugim platformama (operacijskim sustavima), kao što su `Linux`, `Unix` ili `Mac OS X`.

Prije nego što nastavimo, promijenimo temeljne postavke uređivača teksta (editora):



U inicijalnim je postavkama vrsta pisma (font) *Courier New*. Preporučujemo da radite u fontu *Consolas* jer je jedini font u kojem se brojka nula i slovo O razlikuju, `0` i `O`. U svakom slučaju, treba koristiti fiksne fontove, fontove u kojima su svi znakovi jednake širine.

U interaktivnom se modu mogu upisivati i odmah izvršavati sve naredbe Pythona, ali se ne pamte. U radnoj se memoriji pamte samo definirane funkcije i procedure i vrijednosti definiranih varijabli. Poslije prvog poziva Pythona radna memorija je prazna. Simbol `>>>` oznaka je „komandne linije". Sa „|" je označena pozicija kursora u njoj od koje se unose „komande".

Da bismo što prije „progovorili pythonski" u većini ćemo poglavlja s `>>> P.B Opis` posebno naznačiti dijelove teksta koji istodobno predstavljaju vježbu, podrazumijevajući rad na kompjuteru, interaktivni unos naredbi i vaš angažman u analizi rezultata (odziva) Pythona. Počinjemo s prvom vježbom.

>>> 1.1 prve riječi

```
>>> 1
1
>>> 101
101
>>>
12345678999923498765432100000155999000
12345678999923498765432100000155999000
```

Poslije <Enter> svi su brojevi „prepisani" u plavoj boji. Vidimo da možemo pisati brojeve s puno znamenki (brojki), praktički bez ograničenja duljine! Sve mora biti napisano bez vodećih razmaka, od prve kolone. Ako napišemo:

```
>>>  123
SyntaxError: unexpected indent
```

Dobili bismo „crveni karton“ s porukom da smo načinili „sintaksnu pogrešku“, a vidimo da je broj 123 pravilno napisan! Poruka se odnosi na nedopušteno uvlačenje, jer, karakteristika je Pythona da se tekst mora unositi od početka linije.

```
>>> 1 0
SyntaxError: invalid syntax
```

Ulazni niz `1  0` ne može biti prihvaćen kao broj (niti bilo što drugo). Ispis takve pogreške ne upućuje preciznije na njezin sadržaj. Ako je to trebao biti broj `10`, ili bilo koji višeznamenkasti broj, mora biti napisan kompaktno, bez razmaka.

Komentari

Komentar započinje znakom „#”. Tekst iza tog znaka se ne izvršava, već nam služi za dodatna objašnjenja. Prikazan je crvenom bojom (prema osnovnim postavkama konfiguracije Pythona). Dopušteno je pisati ga bilo gdje u liniji i iza izvršne naredbe.

››› 1.2 komentari

```
>>> # Ovo je komentar
>>>     # i ovo je komentar, ne mora biti napisan otpočetka
>>> 98765432100099999999999 # veliki broj
98765432100099999999999
>>> # Cijeli brojevi su bez ograničenja
>>>
```

Cijeli brojevi

Cijeli broj može biti dekadski, binarni, oktalni i heksadecimalni. Ime cjelobrojnog tipa (klase) je „int”. Zasad dajemo pravilo pisanja dekadskih cijelih brojeva:

dekadski_broj: `[0]`$^{+}$ | `[1-9][0-9]`*

Znak | čitamo "ili", `[0]`$^{+}$, ima značenje `0` | `00` | `000` | ... Notacija `[1-9]` upućuje na izbor brojke

`1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9`

a `[0-9]` bilo koje brojke od `0` do 9. Zvjezdica ima značenje da znakovi unutar uglatih zagrada mogu biti izostavljeni ili izabrani više puta, a plus izabrani barem jedanput. Dakle, dano pravilo čitamo: „Dekadski (cijeli) broj je `0` ili više nula, ili mora početi s 1, 2, ... ili 9 iza čega slijedi neograničeni broj znamenki od `0` do 9“. Pravilu pisanja treba dodati uvjet da se broj piše kompaktno, bez razmaka između brojki.

››› 1.3 Cijeli dekadski brojevi

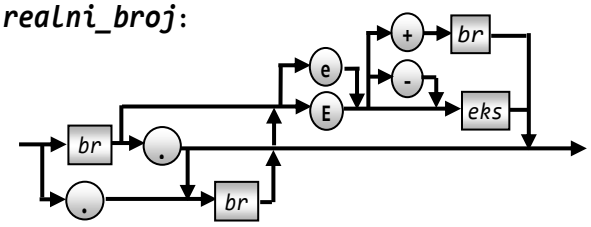
```
>>> 0                    0
>>> 0000000              0
>>> 01
>>> 01SyntaxError: invalid token
>>> 1 0 0 1
SyntaxError: invalid syntax
>>> 1001                 1001
>>> 98765432101234567
98765432101234567
>>> 12345678909876543217532498573498573
23498572348957483957934875943875934 8
12345678909876543217532498573498573498
57234895748395793487594387593488
```

Realni brojevi

Osim cijelobrojnih vrijednosti Python ima (kao i svi jezici za programiranje) i realne vrijednosti. Ime realnog tipa (klase) je „float”. Za razliku od cijelih brojeva, za koje možemo reći da su jednaki cijelim brojevima u matematici, realni su brojevi u Pythonu aproksimacija realnih brojeva u matematici. Ograničeni su donjom i gornjom granicom domene, s jedne strane, i preciznošću decimalnog dijela, s druge strane.

Realni brojevi mogu biti napisani u normalnom (uobičajenom) i eksponencijalnom obliku. Potpuno pravilo pisanja dano je u sljedećem sintaksnom dijagramu:

realni_broj:



br: `[0-9]`$^{+}$ ***eks***: (`[0-2][0-9][0-9]` | `30[0-8]`)

››› 1.4 realni brojevi

```
>>> 1.0     1.0     >>> 1.        1.0
>>> 0.5     0.5     >>> .5        0.5
>>> .123    0.123   >>> 2.        2.0
>>> 12.50   12.5    >>> 3.14E00   3.14
>>> 1e-6    1e-06
>>> # veliki realni brojevi:
>>> 999999999999999.0
999999999999999.0
>>> 9999999999999991.0  # rezultat +1:
9999999999999992.0
```

```
>>> # aproksimacija zadanog broja
>>> 9999999999999998.0
9999999999999998.0
>>> # maksimalni realni broj
>>> 9999999999999998.9
9999999999999998.0
>>> 9999999999999999.0  1e+16
>>> # aproksimacija
>>> 1.797e308           1.797e+308
>>> 1.7977e308          inf
>>> # "beskonačan"
>>> 1.79769e308         1.79769e+308
>>> 1e-323              1e-323
>>> 1e-324              0.0
>>> type (1e-323)       <class 'float'>
```

Vidimo da se brojevi koji sadrže više od petnaest znamenki cijelog dijela interpretiraju s približnim vrijednostima. Značenje brojeva napisanih (ili interpretiranih) u eksponencijalnom obliku, *m*E±*n*, je

$m \times 10^{\pm n}$.

Najveći realni broj prikazan u eksponencijalnom obliku je `1.797e308`. Za veću vrijednost dojavljuje se `inf` što ima značenje beskonačnog broja. Najmanji realni broj je `1e-323` (`0`. pa `322` nule i na kraju `1`).

Brojčani izrazi

Brojčani izrazi su sintaksne kategorije koje sadrže operande, operatore i zagrade. Ako s ***br_izraz*** označimo brojčani izraz, slijede pravila pisanja:

```
br_izraz : predznak operand |
           br_izraz operator br_izraz |
           ( br_izraz )
predznak : [+-]*
operand  : cijeli_broj | realni_broj |
           br_funkcija
operator : + | - |* | / | // | % | **
```

OPERACIJE S CIJELIM I REALNIM BROJEVIMA

Predznak je unarna operacija, što znači da nije dio napisanog broja! Vidimo da može biti napisan više (neograničen) broj puta. Rezultirajući predznak, ako je napisan više puta, jednak je „-“, ako je napisan neparni broj minusa, „+” ako je napisan parni broj minusa. Predznak „+” se izostavlja u interpretaciji (prikazu vrijednosti) cijelog ili realnog broja.

››› 1.5 predznak

```
>>> - - - 10              -10
>>> --+214748364802222   214748364802222
>>> -++----0x100          -256
```

Osim predznaka, kao unarne operacije, nad cijelim i relnim brojevima definirane su binarne operacije. Ako su *x* i *y* dva operanda, simboli i značenja operatora su:

```
+  zbrajanje, x+y        -  oduzimanje, x-y
*   množenje, x*y        /  realno dijeljenje, x/y
//  cjelobrojno dijeljenje, x//y
**  potenciranje, x**y = x^y
%   ostatak cjelobrojnog dijeljenja, x%y
```

Značenje operacija zbrajanja, oduzimanja, množenja i dijeljenja jednako je kao u matematici.

Pisanje izraza u interaktivnom modu ima značenje izračunavanja i ispisa rezultata. Ako izraz sadrži sintaksne pogreške, bit će dojavljene. Također će biti dojavljene i eventualne semantičke pogreške, kao na primjer nedopušteno dijeljenje s nulom.

››› 1.6 zbrajanje i oduzimanje

```
>>> # zbrajanje      >>> # oduzimanje
>>> 12   + 5   17    >>> 12   - 5    7
>>> 12   + 5. 17.0   >>> 12   - 5.   7.0
>>> 12.  + 5   17.0  >>> 12.  - 5    7.0
>>> 12.0 + 5.0 17.0  >>> 12.0 - 5.0  7.0
>>> 1+2+
SyntaxError: invalid syntax
```

››› 1.7 množenje

```
>>> # cjelobrojno množenje:
>>> -15 *12    -180   >>> -15 *-12   180
>>> 10 *---- 2   20
>>> 9776766757655564445
*787656500077564453221
770073388640965966409994537202455332734 5
>>> # realno množenje
>>> 9776766757655564445
*787656500077564453221.0
7.700733886409659e+39
>>> 9776766757655564 *1.0
9776766757655564.0
>>> 9776766757655564 *2.0
1.955353351531113e+16
```

››› 1.8 realno dijeljenje

```
>>> 12 /4     3.0   >>> 12 /16       0.75
>>> 111 /11          10.090909090909092
>>> 1/-33          -0.030303030303030304
>>> 12.55/.2356      53.26825127334465
```

```
>>> 12345/0.00001            1234500000.0
>>> 12345678901234567.25
1.2345678901234568e+16
>>> 10 /0
... ZeroDivisionError: division by zero
```

Vidimo da se u realnom dijeljenju može dobiti približan rezultat.

CJELOBROJNO DIJELJENJE

Cjelobrojno dijeljenje, *a*//*b*, dvaju brojčanih operanada *a* i *b* jednako je najvećoj cijeloj vrijednosti dobivene realnim dijeljenjem s istim operandima, *a*/*b*. Najveće cijelo od *x* se u matematici označuje kao $\lfloor x \rfloor$, a to je najveći cijeli broj koji nije veći od *x*. Još se naziva i funkcija *pod* (eng. *floor*).

Ako je *x* cijeli broj, ili realan s decimalnim dijelom koji je jednak `0.0`, onda je $\lfloor x \rfloor = x$. Na primjer, $\lfloor$`2.0`$\rfloor$ jednako je `2.0` ili $\lfloor$`-2`$\rfloor$ je `-2`.

Ako je *x* realan broj i *x*>`0`, onda je $\lfloor x \rfloor$ jednako cijelom dijelu od x, odnosno cijelom dijelu od *x*-`1`, ako je *x*<`0`. Na primjer, $\lfloor$`2.99`$\rfloor$ jednako je `2.0`, a $\lfloor$`-2.99`$\rfloor$ je `-3.0`.

››› 1.9 cjelobrojno dijeljenje

```
>>> 12 // 4    3      >>> -12 // 4     -3
>>> 12.0 // 4  3.0    >>> 12 // 5      2
>>> -12 // 5   -3     >>> 12 // 5.0    2.0
>>> 12 // 6    2      >>> -12 // 6     -2
>>> -12  // 6.0      -2.0
```

OSTATAK CJELOBROJNOG DIJELJENJA

Ostatak cjelobrojnog dijeljenja brojeva *x* i *y*, *x*%*y*, što čitamo "*x* modulo *y*", može se definirati preko drugih cjelobrojnih operacija kao

x % *y* = *x* - (*x* // *y*) * *y*

Ovako definirana operacija *x* modulo *y* nadilazi značenje te operacije u matematici, jer operandi mogu biti cjelobrojni i realni. Iz dane definicije ostatka cjelobrojnog dijeljenja slijede dva posebna slučaja:

x % *y* = *x* ako je x < y i *x* % *x* = 0

››› 1.10 ostatak cjelobrojnog dijeljenja

```
>>> 12  % 4     0      >>> -12  % 4     0
>>> -12.0 % 4   0.0    >>> 12  % 5      2
>>> -12  % 5    3      >>> -12 % 5.0    2.0
>>> 12  % 6     0      >>> -12  % 6     0
>>> -12   % 6.0 0.0    >>> 12. % 7      5.0
>>> -12. % 7    2.0    >>> -12.  % 7    2.0
```

Ako je x realan broj, x>0, ostatak cjelobrojnog dijeljenja s 1, *x* % 1, *bit će jednak decimalnom dijelu od x.* □

POTENCIRANJE

Operacija potenciranja broja *x* brojem *y*, x^y, piše se *x****y* i ima značenje kao u matematici.

››› 1.11 potenciranje

```
>>> 2 **3     8        >>> 2 **-3      0.125
>>> -2 **3    -8       >>> (-2) **3    -8
>>> 2 **-2    0.25     >>> 0 **125     0
>>> 125**0    1        >>> 0 **0       1
>>> 2 **0.5   1.4142135623730951
```

U matematici je potenciranje 0^0 *neodređeno. U Pythonu je* 0**0 *jednako* 1. □

TIP BROJČANOG IZRAZA

S obzirom na to da brojčani izraz općenito sadrži cjelobrojne i realne brojeve (ili vrijednosti) uvodi se pojam tipa izraza. Tip izraza određen je tipom njegove vrijednosti. Na primjer, ako je „+" operacija zbrajanja, zbrajanje dvaju cijelih brojeva imat će za rezultat cijeli broj, a ako je jedan operand realni broj, rezultat će biti realni broj. Posebno, ako je vrijednost takvog izraza cjelobrojna, tada je izraz „cjelobrojni" (tipa „`int`"), a ako je realna, izraz je „realni" (tipa „`float`").

Ponovimo, tip vrijednosti dobivene kao rezultat izračunavanja binarnog izraza s operandima *x* i *y* ovisan je o njihovom tipu. Pravilo je: vrijednost je cjelobrojna samo u slučaju da su oba operanda cjelobrojna, inače je realna. Cjelobrojni će izraz biti tipa „`int`" ako su svi operandi tipa „`int`" i ako je rezultat izračunavanja tipa „`int`". Izuzetak je operacija realnog dijeljenja, operator /, koja uvijek daje rezultat realnog tipa, bez obzira na tip operanada.

Ako s *c* označimo cjelobrojni, a s *r* realni tip, u sljedećoj su tablici dani tipovi izračunate vrijednosti dobivene izvršenjem pojedinih binarnih operacija.

x	*y*	*x* [+-*] *y*	*x* / *y*	*x* // *y*	*x* % *y*	*x* ** *y*
c	*c*	*c*	*r*	*c*	*c*	*c* ako je *y*>0 *r* ako je *y*<0
c	*r*	*r*	*r*	*r*	*r*	*r*
r	*c*	*r*	*r*	*r*	*r*	*r*
r	*r*	*r*	*r*	*r*	*r*	*r*

Postoji standardna funkcija `type()` koja vraća tip argumenta. Preciznije, klasu kojoj pripada, o čemu će biti riječi u narednom poglavlju. U većini jezika za programiranje, pa tako i u Pythonu `2.x`, radilo se o tipovima pa ćemo zadržati to tradicionalno ime.

››› 1.12 tip izraza

```
>>> type (12 +4)        <class 'int'>
>>> type (12 /4)        <class 'float'>
>>> type (12 -4)        <class 'int'>
>>> type (1 //-33)      <class 'int'>
>>> type (12 *4)        <class 'int'>
>>> type (2**10)        <class 'int'>
>>> type (1 + 2.)       <class 'float'>
>>> type (2**(-3))      <class 'float'>
>>> type (10 / 20)      <class 'float'>
>>> type (11. *5.5)     <class 'float'>
```

PRIORITET IZVRŠAVANJA OPERACIJA

Iz pravila pisanja brojčanih izraza slijedi da je samo jedan operand, s predznakom ili bez njega, brojčani izraz. Ali, najčešće ćemo pisati izraze koji sadrže više operacija i zagrada. Tada će se izračunavanje izvoditi kao i u matematici, slijeva nadesno, vodeći računa o prioritetu izvršavanja pojedinih operacija, kao što slijedi:

1) izraz u zagradi
2) funkcija
3) potenciranje
4) predznak (unarna operacija "-" ili "+")
5) množenje, realno dijeljenje, cjelobrojno dijeljenje, ostatak dijeljenja
6) zbrajanje i oduzimanje

››› 1.13 izračunavanje izraza

```
>>> 1 +2*3      7   >>> (1 +2)*3      9
>>> -2 **3      -8  >>> (1 +2)*(2+4)  18
>>> 2 **3 **4
              2417851639229258349412352
```

Primijetiti da je rezultat izračunavanja izraza `2 **3 **4` dobiven kao:

```
>>> 2 **(3**4)
2417851639229258349412352
```

a ne kao:

```
>>> (2 **3) **4                    4096
```

jer je `2 **3 **4` jednako 2^{3^4}

Zadatak 1.1

Izračunati faktorijel broja 20.

Faktorijel broja n, $n!$, jednak je produktu brojeva 1 do n:

$$n! = 1 \times 2 \times 3 \times \ldots \times n$$

››› 1.14 20 faktorijel

```
>>> 1 *2 *3 *4 *5 *6 *7 *8 *9 *10 *11
*12 *13 *14 *15 *16 *17 *18 *19 *20
2432902008176640000
```

STANDARDNE BROJČANE FUNKCIJE

Python ima nekoliko desetaka standardnih ili „ugrađenih" (*built-in*) funkcija koje su uvijek dostupne. Njihov popis dan je u Pythonovoj dokumentaciji, u standardnoj Pythonovoj biblioteci (*F1* → Contents → The Python Standard Library → Built-in Functions). Ovdje izdvajamo samo brojčane funkcije – funkcije čija je domena i kodomena cijeli ili realni broj. Općenito su argumenti izrazi odgovarajućeg tipa.

Ime	*Argumenti*	*Tip*	*Opis*
`abs`	(*x*)	*x*	apsolutna vrijednost od *x*, $\|x\|$; tip je jednak tipu argumenta
`float`	(*x*)	`r`	pretvorba *x* u realni broj; `float()` vraća `0.0`
`int`	(*x*)	*c*	cijeli dio vrijednosti od *x*; `int()` vraća `0`
`pow`	(*x*,*y*)	*c*,*r*	potenciranje, x^y ; `pow (0, 0)` vraća `1`
`round`	(x) (x, c)	*r*	zaokruženi broj *x* zaokruženi broj *x* na *c* decimala

r - realni tip; *c* - cjelobrojni tip;
x,*y* - brojčani tip (realni ili cjelobrojni)

››› 1.15 brojčane funkcije

```
>>> abs (-1)       1    >>> abs (-1.5)  1.5
>>> int ()         0    >>> float ()    0.0
>>> int (1.5)      1    >>> int (1.99)    1
>>> int (-1.999)   -1   >>> float(10)  10.0
>>> round(1.499) 1.0    >>> round(1.5)  2.0
>>> pow (0,10)     0    >>> pow (2,-2) 0.25
>>> pow (-2,10)    1024 >>> pow (10, 0)   1
>>> pow (1,-1)     1.0  >>> pow (0, 0)    1
>>> pow (2, 0.5)  1.4142135623730951
>>> pow (2, pow (3, 3))  134217728
```

„Komandna linija“

U nedostatku boljeg prijevoda, s „komandna linija“ nazvali smo prostor poslije **>>>** u kojem smo unosili brojčane izraze. Proširimo značenje unosa sa:

unos : *izraz* { [,;] *izraz* }

››› 1.16 komandna linija

```
>>> 1, 2, 3                (1, 2, 3)
>>> 1; 2, 3
1
(2, 3)
>>> 1+2,1-2, 1*2,1/2       (3, -1, 2, 0.5)
>>> 1+2; 1-2; 1*2; 1/2
3
-1
2
0.5
>>> 1; 2, 3; 4
1
(2, 3)
4
```

NASTAVAK KOMANDNE LINIJE

Znak \ poslije [,;+*/%-] | // | ** i poslije zareza u pozivu funkcije, ako funkcija ima više od jednog argumenta, ima značenje nastavka komandne linije prelaskom u sljedeći red.

››› 1.17 nastavak komandne linije

```
>>> 1; \
    2, \
    3
1
(2, 3)
>>> 12 +      \
        13 * 14              194
>>> round (1.5555,
                  3 )        1.556
>>> 100\
1
SyntaxError: invalid syntax
>>> (  100
      +200
      +300 ) *5              3000
>>> (  100
      +200
      +300 ) *5 + 10 * ( 11 **2
                +12 **2)     5650
```

Python provjerava jesu li uparene zagrade! Ako nisu, očekuje se nastavak pisanja izraza ili liste argumenata poziva funkcije.

Brojčane varijable

Pretpostavimo da treba riješiti sljedeći zadatak:

Zadatak 1.2

Izračunati i ispisati opseg i površinu kruga polumjera jednakog 6.25.

Ako je *r* polumjer, formule su $2r\pi$, za opseg, i $r^2\pi$ za površinu, pa je rješenje kao što slijedi.

››› 1.18 opseg i površina kruga

```
>>> 2 *6.25 *3.14159  # Opseg :
39.269875
>>> 6.25**2 *3.14159  # Površina :
122.71835937499999
```

Nedostatak je ovoga rješenja što smo na dva mjesta morali napisati vrijednost polumjera i broja π. Osim što smo mogli pogriješiti u pisanju tih vrijednosti, izrazi nisu pregledni jer ne predočuju navedene formule. Naravno, moguće je uvesti simbolička imena ili varijable koje nam daju preglednije rješenje.

››› 1.19 opseg i površina kruga (2)

```
>>> # dvije naredbe odvojene s ;
>>> r = 6.25;  Pi = 3.14159
>>> r, Pi              (6.25, 3.14159)
>>> 2 *r *Pi           39.269875
>>> r**2 *Pi           122.71835937499999
```

JEDNOSTAVNO PRIDRUŽIVANJE

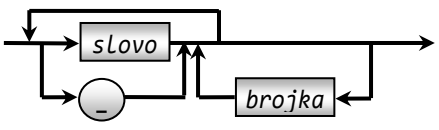
U prethodnoj smo vježbi sa `r=6.25` i `Pi=3.14159` napisali dvije naredbe za jednostavno pridruživanje. Pravilo pisanja jednostavnog pridruživanja dano je sa:

jednostavno_pr : *ime* = *izraz*
izraz : *br_izraz*

Imena

Imena su klase riječi koje se, u notaciji sintaksnih dijagrama, pišu prema sljedećem pravilu:

ime:



ime je sačinjeno od samo jednog slova ili znaka „_", odnosno, to su riječi koje počinju slovom ili znakom „_", a potom slijedi ***slovo***, znak „_" ili ***brojka***. Karakteristika je Pythona, od inačice `3.0`, da može biti bilo koje veliko ili malo slovo *Unicode* standardizacije, o čemu će biti više riječi u šestom poglavlju. Početni je cilj bio da *Unicode* sadrži pisma svakog pojedinog prirodnog jezika. Na primjer, abecede hrvatskog, engleskog, francuskog, grčkog ili ruskog jezika. Evo nekoliko primjera imena:

```
i X A1 _2 π α Δ1 ε0 Površina Jojo
Jacques Mérci _ Толстой od_α_do_ω Σn
```

Danom pravilu tvorbe imena treba dodati sljedeće:

1) Imena sastavljena samo od slova ne smiju biti iz skupa rezerviranih riječi (definirane su u drugom poglavlju). Na primjer, `if`, `for`, `True` i `continue` su rezervirane riječi.
2) Imena su "case sensitive", što znači da su dva imena napisana samo malim slovima i istim takvim samo velikim slovima, različita. Na primjer, `A` i `a` su dva različita imena. Ili, `Print`, `while` i `WHILE`, `jaune` i `Jaune`, također su različita imena. Riječ `while` je rezervirana pa se ne može tvoriti ime, dok su `Print` i `WHILE` prihvatljivi kao imena.
3) Standardna imena (imena standardnih funkcija i procedura) nisu rezervirana pa se mogu rabiti kao imena, ali se to ne preporučuje, jer tada standardna imena gube svoje osnovno značenje što može prouzročiti logičke pogreške u programu.
4) Maksimalna duljina imena nije ograničena, odnosno, ograničena je veličinom radne memorije! Na primjer,

   ```
   Δaaaaa98765aaaΔaaaaaaa___aaaaahjsh
   addkUIZUIZIUZIaaaaŠĆbcdwwwqqq
   ```

   je ime. Postavlja se pitanje čemu bi služilo tako dugo i složeno ime?! U svakom slučaju, pri izboru imena trudimo se da nas podsjeća na ono što treba predstavljati, vodeći pritom računa da ne pretjerujemo u duljini imena, jer se time smanjuje čitljivost.

Varijable

Ako naredbu za pridruživanje napišemo kao

v = i

gdje je ***v*** ime, a ***i*** izraz, značenje se može prikazati s

v ← vrijednost (i)

ili riječima, imenu ***v*** bit će pridružena vrijednost izraza ***i*** i njegov tip. Time će ime poprimiti svojstvo varijable tipa jednakog tipu izraza. Zasad bi to bio cjelobrojni ili realni tip. Prethodna vrijednost i tip varijable ***v*** (ako je postojala) bit će izgubljena.

Napomenimo da je dano značenje vrijedilo u inačicama `2.x` i `2.x.y`. U sljedećem je poglavlju dano „pravo" značenje od inačice `3.0`, ali dano tumačenje zadovoljava potrebe za razumijevanjem pojma varijable u našim uvodnim razmatranjima. Tako bi, na primjer, izvršenjem jednostavnih pridruživanja, `r=6.25` i `Pi=3.14159` iz vježbe **`>>>1.19`**, bile definirane dvije varijable u dijelu radne memorije, s imenima `r` i `Pi`. Varijabli s imenom `r` pridružena je realna vrijednost `6.25`, a varijabli s imenom `Pi` također realna vrijednost `3.14159`.

Sada se i pravilo pisanja operanda u brojčanom izrazu može proširiti na:

operand : *broj* | *br_varijabla*
br_varijabla : *cjelobrojna_varijabla* | *realna_varijabla*

Provjerimo vrijednosti i tipove varijabli `r` i `Pi`:

```
>>> r       6.25    >>> Pi       3.14159
>>> type(r)         <class 'float'>
>>> type(Pi)        <class 'float'>
```

Izračunavanjem izraza koji sadrži varijable prvo će varijable biti zamijenjene svojom vrijednošću iz radne memorije.

Za razliku od nekih jezika za programiranje, varijable u Pythonu se ne deklariraju prije uporabe, već naredbom za pridruživanje. Drugo, tip varijable mijenja se promjenom tipa izraza čija će joj vrijednost biti pridružena.

`>>>` `1.20 promjena tipa varijable`

```
>>> a = 55;    type (a) <class 'int'>
>>> a = 125.0; type (a) <class 'float'>
```

Početnici često simbol dodjeljivanja, „=", poistovjećuju s izjednačivanjem vrijednosti, pa naredbu `x=y` čitaju "`x` je jednako `y`". Zato bi, na primjer, na upit kolika je vrijednost varijable `y` poslije izvršenja niza naredbi:

```
>>> x = 100; y = x; x = 200
```

možda odgovorili 200! Ako se značenje naredbe za pridruživanje odmah pravilno shvati, da je varijabli pridružena vrijednost izraza, tj. prvo se izračunava izraz pa se dobivena vrijednost pridruži varijabli, ne bi bilo problema.

U našem primjeru prvom naredbom za dodjeljivanje varijabli x bila bi pridružena vrijednost 100, potom bi vrijednost izraza u drugoj naredbi, a to je tekuća vrijednost varijable x, bila pridružena varijabli y i na kraju bi vrijednost 200 bila pridružena varijabli x. Sada je jasno da to nije imalo utjecaja na sadržaj varijable y, pa je njezina vrijednost ostala nepromijenjena.

Drugi primjer pridruživanja koji najčešće zbunjuje početnike jeste pridruživanje koje u izrazu sadrži ime varijable kojoj se vrijednost izraza pridružuje. Na primjer, ako je varijabli i bila pridružena vrijednost 10,

```
>>> i = 10
```

što će se dogoditi poslije izvršenja naredbe i = i+5?

Zapamtimo! Vrijednosti varijabli u izrazu se ne mijenjaju, na njih se referira, a to znači da će njihove tekuće vrijednosti zamijeniti simboličko ime u izrazu.

Ovo je ujedno i objašnjenje. Varijabla i u izrazu bit će zamijenjena svojoj „starom" vrijednošću. Poslije evaluiranja izraza i+5 dobivena vrijednost bit će pridružena varijabli i. „Nova" vrijednost varijable i bit će 15. U trećem ćemo poglavlju za ovakva pridruživanja uvesti posebne operande.

Brisanje varijable

Komandom DEL može se izbrisati jedna ili više varijabli. Piše se prema pravilu:

komanda_DEL : del *ime_varijable*
{, *ime_varijable* }

Riječ del pripada skupu rezerviranih riječi Pythona. Izvršenjem *komande DEL* mogu se brisati samo definirane varijable. U suprotnom se dojavljuje pogreška.

›››1.21 brisanje varijable

```
>>> a = 5;  b = 6; a          5
>>> del a
>>> a
NameError: name 'a' is not defined
```

Sadržaj se radne memorije može resetirati s **Ctrl_F6**.

Globalna varijabla „_"

Postoji globalna varijabla s imenom „_" (podcrta) kojoj se automatski pridružuje posljednja izračunata vrijednost. Pozivom interaktivnog moda ili poslije resetiranja je nepoznata.

```
>>> _
NameError: name '_' is not defined
```

Poslije izvršenja bilo koje naredbe kojom se prikazuje rezultat ista se vrijednost automatski pridružuje varijabli _. Na primjer, poslije:

```
>>> a = 5
>>> _
...
NameError: name '_' is not defined
```

varijabla _ je nepoznata. Ali, poslije:

```
>>> a**3      125      i     >>> _    125
```

vidimo da joj je pridružena posljednja vrijednost. Ako imamo svoju varijablu s imenom „_", globalna varijabla _ bit će predefinirana. Na primjer:

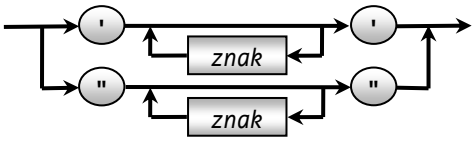
```
>>> _ = 101
>>> _          101
>>> 2**8       256
>>> _          101
```

Brisanjem varijable _ ili resetiranjem (Ctrl_F6), vraća se prvobitno značenje globalne varijable _.

Znakovni nizovi

Znakovni niz ili string posebna je struktura podataka u Pythonu. Ime joj je str. Detaljno smo je opisali u šestom poglavlju. Ovdje dajemo pravila pisanja stringova i njihovu uporabu u dodatnim opisima brojčanih rezultata. Pišu se prema sljedećim pravilima:

znakovni_niz:



Vidimo da postoje dva načina pisanja znakovnih nizova: počinju i završavaju polunavodnikom ili počinju i završavaju navodnikom. Polunavodnik (ili navodnik) označuje početak i kraj niza i ne smatra se njegovim dijelom. Na primjer, napišimo:

```
>>> "Ovo je interaktivni ('Shell') mod
Pythona"
```

To je string i prikazan je u zelenoj boji (inicijalna postavka editora Pythona). Poslije <Enter> bilo bi ispisano:

```
"Ovo je interaktivni ('Shell') mod Pythona"
>>>
```

Dakle, odgovor je „prepisani" ulazni string, prikazan u plavoj boji. Ovo je istodobno bio primjer znakovnog niza koji sadrži drugi znakovni niz. Da smo napisali "Shell":

```
>>> "Ovo je interaktivni ("Shell") mod Pythona"
SyntaxError: invalid syntax
```

bila bi dojavljena sintaksna pogreška.

Duljina znakovnog niza jest broj znakova koji ga čine (bez polunavodnika ili navodnika koji označuju početak i kraj niza). Niz '' (ili "") je prazan znakovni niz ili prazan string. Duljina mu je jednaka 0.

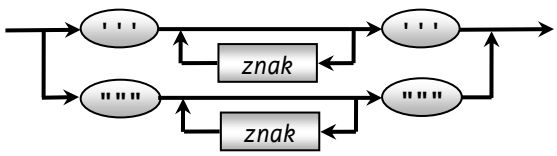
››› 1.22 znakovni nizovi

```
>>> "1234321"                    '1234321'
>>> 'Ovo je niz znakova'
'Ovo je niz znakova'
>>> "C'est la vie, mon amour!"
"C'est la vie, mon amour!"
>>> "I'm a student."    "I'm a student."
>>>'Površina: \nP ='    'Površina: \nP ='
```

TEKST

Nizovi znakova moraju biti napisani u jednoj liniji, bez nastavljanja. Python ima posebnu vrstu nizova znakova – tekst, koji može biti napisan u više redova, bez ograničenja duljine. Piše se prema pravilu:

tekst:



››› 1.23 tekst

```
>>> """
Prvi red teksta.
Drugi red teksta."""
'\nPrvi red teksta.\nDrugi red teksta.'
```

Tekst je ispisan kao „normalni" string, uz prikaz kontrolnog znaka za prelazak u novi red (mjesto gdje je pritisnuto <Enter>).

FUNKCIJE chr() i ord()

Znak je string duljine jednake 1. To je bilo koji znak izabrane kodne stranice uz dodatak razmaka (blanka). Mi ćemo koristiti kodnu stranicu 1250 koja sadrži hrvatska slova.

Funkcija chr(*i*), gdje je *i* cijeli broj od 0 do 1114111, vraća znak čiji je *Unicode* jednak *i*. Kontrolni znakovi imaju kôd od 0 do 31, a ostali znakovi od 32 (praznina ili blank) sve do dane gornje granice. Na primjer, pogledajmo što vraća funkcija chr() za neke kodove:

››› 1.24 chr()

```
>>> chr (65)   'A'    >>> chr (66)    'B'
>>> chr (48)   '0'    >>> chr (97)    'a'
>>> chr (98)   'b'    >>> chr (49)    '1'
>>> chr (352)  'Š'    >>> chr (381)   'Ž'
>>> chr (960)  'π'    >>> chr (353)   'š'
>>> chr (382)  'ž'    >>> chr (8364)  '€'
```

Funkcija ord(*Ch*), gdje je *Ch* znak, inverzna je funkcija funkciji chr() i vraća *Unicode* argumenta.

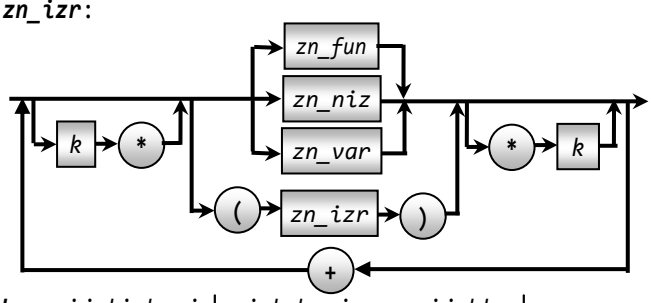
››› 1.25 ord()

```
>>> ord ('A')  65   >>> ord ('B')    66
>>> ord ('0')  48   >>> ord ('a')    97
>>> ord ('1')  48   >>> ord ('Š')    352
>>> ord ('Ž')  381  >>> ord ('π')    960
>>> ord ('ž')  382  >>> ord (' ')    32
```

ZNAKOVNI IZRAZI

Znakovni izrazi će biti izrazi koji kao rezultat izvršenja operacija daju znakovni niz (string). Nepotpuno pravilo pisanja je:

zn_izr:



k : *cijeli_broj* | *cjelobrojna_varijabla* | (*cjelobrojni_izraz*)

k ima značenje multipliciranja stringa. Ako je ***k*** ≤ **0**, rezultat je prazan string. Znak „+" je operacija nastavljanja (konkatenacije ili dopisivanja) niza znakova. Može biti izostavljena između dva znakovna niza. Na primjer:

```
>>> '1'  "2"    'abc'          '12abc'
```

›››1.26 znakovni izrazi

```
>>> '='*10                    '=========='
>>> 'ha, '*5 +'...'
'ha, ha, ha, ha, ha, ...'
>>> '1' +"2" +'3'          '123'
>>> '-'*3 +2*'*'*2 +3*'-'   '---****---'
```

ZNAKOVNE VARIJABLE

Znakovni niz (string) dobiven evaluiranjem znakovnog izraza može biti pridružen imenu u jednostavnom pridruživanju, pa proširujemo značenje izraza sa:

izraz : *zn_izr*

Ime će imati svojstvo „znakovna varijabla“. U sintaksnom dijagramu označena je s *zn_var*.

›››1.27 znakovni izrazi (2)

```
>>> ha = 'ha, '*5 +'...'; ha
'ha, ha, ha, ha, ha, ...'
>>> a = '='; b = '*'; a*10 + b*10 +a*10
'==========**********=========='
>>> (a + b) *5                  '=*=*=*=*=*='
>>> (b + a) *5                  '*=*=*=*=*=*'
>>> (2*a +3*b) *3          '==***==***==***'
>>> P = 'Python'; v = '3.9.1';  P +' ' +v
'Python 3.9.1'
```

STANDARDNE BROJČANE I ZNAKOVNE FUNKCIJE

Proširujemo značenje standardnih brojčanih funkcija `int()` i `float()` koje mogu imati string (dobiven evaluiranjem znakovnog izraza) kao argument.

`eval`	(*s*)	`i`,*f*	izračunava (evaluira) brojčani izraz sadržan u nepraznom stringu *s*; dojavljuje pogrešku ako izraz nije sintaksno korektan.
`float`	(*s*)	*f*	pretvorba stringa *s* u realni broj
`int`	(*s*)	`i`	pretvorba stringa *s* u cijeli broj; dojavljuje se pogreška ako se *s* ne može interpretirati kao cijeli broj
`len`	(*s*)	`i`	duljina (broj znakova) stringa. `len ('')` i `len ("")` vraćaju `0`
`str`	(*x*) (*s*)	`s`	pretvorba broja *x* u string vraća *s*; `str()` vraća prazan string.

f - realni tip; `i` - cjelobrojni tip; *x* – brojčani izraz; *s* – string (rezultat izračunavanja znakovnog izraza)

›››1.28 standardne brojčane funkcije (2)

```
>>> int ('1' +'2')                    12
>>> int ('1.99')
ValueError: invalid literal for int()
with base 10: '1.99'
>>> int ('1+2')
ValueError: invalid literal for int()
with base 10: '1+2'
>>> int ('1 0')
ValueError: invalid literal for int()
with base 10: '1 0'
>>> float ('1' +'2')                  12.0
>>> len ('0123456789')                10
>>> len ("'")                          2
>>> eval ('2 +2**2 +2**3 +2**4')      30
>>> eval (str (round (1.55, 2)) )     1.55
>>> round (1.555, 2)                  1.55
# Pythonova pogreška zaokruživanja!
>>> eval ('round (1/321, 5)')    0.00312
>>> a = 10; b = 20
>>> eval ('(a + b) * 3')          90
>>> eval ('10 +(20 +30)**4')      6250010
>>> eval ('ab*10')
NameError: name 'ab' is not defined
>>> eval ('10 +20 +30)*4')
    10 +20 +30)*4
               ^
SyntaxError: invalid syntax
```

›››1.29 str()

```
>>> str (1.45)                     '1.45
>>> i = 125; str (i)               '125'
>>> str (1 2)
SyntaxError: invalid syntax
>>> str (2 *5.5)                   '11.0'
>>> str (11**5)                   '161051'
>>> str ('Python')                'Python'
>>> str (int ('1001'))            '1001'
```

Naredba za ispis - print()

U interaktivnom se modu rezultati izračunavanja napisanih brojčanih izraza ispisuju u sljedećem redu, bez posebnog zahtjeva za to. Postoji posebna funkcija `print()` koja to čini. Iz tradicionalnih razloga smo je nazvali *naredba za ispis* ili *naredba PRINT*, jer je do inačice `3.0` to bila. Piše se prema jednostavnom pravilu:

naredba_PRINT : `print (` [*izraz* {, *izraz* }
[, (*end* [, *sep*] | *sep* [, *end*])])
izraz : *br_izraz* | *zn_niz*
end : end = *string*
sep : sep = *string*

Značenje *naredbe PRINT* jest izračunavanje i ispis vrijednosti niza izraza. Tipovi izraza mogu biti različiti. Zasad su to brojčani izrazi (cjelobrojni i realni) i

stringovi. Cijeli brojevi se ispisuju u dekadskom obliku, realni s decimalnim dijelom i stringovi bez navodnika. Iz pravila pisanja naredbe za ispis mogu postojati sljedeći slučajevi:

1) `print ()`
   Ispis praznog reda.
2) `print ( izraz { , izraz } )`
   Ispis niza vrijednosti dobivenih izračunavanjem izraza ili stringa, u jednom redu, odvojenih razmakom. Poslije posljednjeg ispisa prelazi se u novi red.
3) `print (izraz { , izraz } , end = string)`
   Ispis kao pod (2). Poslije posljednjeg ispisa dodaje se string i ostaje u istom redu. To će biti mjesto od kojeg će početi novi ispis (ako ga ima).
4) `print (izraz { , izraz } , sep = string)`
   Ispis separatora (zadanog stringa) između parova izraza i na kraju prelazak u novi red.
5) `print (izraz { , izraz } , sep = s1, end = s2 )` ili
   `print (izraz { , izraz } , end = s2, sep = s1 )`
   Ispis separatora (stringa *s1*) između parova izraza i stringa *s2* poslije posljednjeg. To će biti mjesto od kojeg će početi novi ispis (ako ga ima).

››› 1.30 tekst (2)

```
>>> print ("""
Prvi red teksta.
Drugi red teksta.
Treći red teksta. """)
Prvi red teksta.
Drugi red teksta.
Treći red teksta.
```

Ispis kontrolne sekvence `'\n'` imao je značenje „prijeđi na početak novoga reda".

››› 1.31 print()

```
>>> print ()       # ispis praznog reda

>>> print (1001)                    1001
>>> print (1, 2, 3)                1 2 3
>>> print (1, 2,
       3)  # unos u više redova (bez \)
1 2 3
>>> print (1); print (2)  # ispis u
prvom, pa u drugom redu
1
2
>>> print ((10 +20)*30)          900
>>> print ("C'est la vie, mon amour!")
C'est la vie, mon amour!
>>> print ('Površina je',12.5*42,'m2')
Površina je 525.0 m2
>>> print (1, 2, 3, sep = '***')
1***2***3
>>> print (1, 2, 3, sep = '')          123
```

Zadatak 1.3

Izračunati i ispisati oplošje i volumen valjka polumjera osnovice 12.565 i visine 14.838.

››› 1.32 Oplošje i volumen valjka

```
>>> Pi = 3.14159
>>> # r-polumjer osnovice; h-visina
>>> r = 12.565; h = 14.838
>>> # površina osnovice valjka
>>> P = r**2 *Pi
>>> # ispis rezultata
>>> print ('Oplošje valjka =',
           2 *P +(2*r *Pi) *h); \
    print ('Volumen valjka =', P *h)
Oplošje valjka = 2163.41633805
Volumen valjka = 7359.52624631
```

KONTROLNI STRINGOVI

Postoje dva kontrolna stringa (ili „kontrolne sekvence") koji imaju posebno značenje u naredbi za ispis. To su `'\n'` i tabulator `'\t'`. Nailaskom na `'\n'` u stringu koji se ispisuje naredbom za ispis, prijeći će se na početak novog reda i nastaviti s ispisom preostalog dijela stringa, a `'\t'` ima značenje tabulatora u tekućem redu i nastavak ispisa.

››› 1.33 '\n' i '\t'

```
>>> print (1, 2, 3, sep = '\t')
1       2       3
>>> print ('\n')  # dva prazna reda
>>> print ('\n'*29)
>>> # "izbrisano 30 redova ekrana"
❶>>> print ('.\t.\t.\t.')
.       .       .       .       .
>>> #   8       16      24      32
❷>>> print ('0\t8\t16\t24\t32')
0       8       16      24      32
❸>>> print ('0\t8\t16\t2412345\t32')
0       8       16      2412345 32
❹>>> print ('0\t8\t16\t24123456\t40')
0       8       16      24123456
40
❺>>> print ('jedan \t\t\t\tpet' +
    '\n\tdva\t\tčetiri \n\t\ttri')
jedan                           pet
        dva             četiri
                tri
```

Možemo zamisliti da su stupci označeni brojevima od 0 nadalje i da je 0-ti stupac 0-ti tabulator. Izvršenjem naredbe ❶ bit će ispisane točke na početnim pozicijama tabulatora. Ispisali smo ih naredbom ❷. U naredbi ❸ poslije pozicije 24 ispisano je još 5 znakova i 32 koji označuje poziciju sljedećeg tabulatora. U naredbi ❹ ispisano je 6 znakova poslije oznake pozicije 24, pa je sljedeći tabulator pomaknut za osam mjesta i počinje od pozicije 40. U naredbi ❺ niz znakova 'jedan pet dva četiri tri' s umetnutim kontrolnim znakovima tabulatora i prelaska u novi red ispisan je u tri reda.

Funkcija input()

U nekim se jezicima za programiranje vrijednost varijabli može pridružiti naredbom za unos (na primjer, naredbom READ u Pascalu, ili naredbom INPUT u BASIC-u). U Pythonu to nije naredba već funkcija - input(), a to znači da je input() dio izraza. Pravilo pisanja je:

```
funkcija_INPUT : input ( [poruka ] )
  poruka       : zn_izr
```

Značenje funkcije input() je prekid evaluiranja izraza, ispis poruke (ako je ima) i čekanje da se unese string (bez oznake početka i kraja stringa). Najčešće će to biti eksplicitno zadane vrijednosti, ali mogu biti izrazi koji sadrže prethodno definirane varijable. Funkcija input() vraća string kao rezultat.

```
>>> r = input ('Zadaj polumjer kruga ')
Zadaj polumjer kruga 15
>>> r                '15'
```

Poruka (string) služi za opis ili komentar onoga što treba upisati. Ako je ime r polumjer kruga, koristimo funkciju eval():

```
>>> r = eval  (
        input ("Zadaj polumjer kruga "))
Zadaj polumjer kruga 13.5
```

U sljedećem smo unosu napisali izraz, s varijablama r i Pi, koji je predstavljao površinu kruga polumjera r i čija je vrijednost pridružena varijabli P:

```
>>> Pi = 3.14159; P = eval (
   input ('Upiši formulu za površinu '))
Upiši formulu za površinu r**2 *Pi
>>> r, P              (13.5, 572.5547775)
```

Ili, ako ne želimo pamtiti vrijednost polumjera:

```
>>> P = round ( eval (input ("Zadaj
polumjer kruga ") )**2 *Pi, 4 )
Zadaj polumjer kruga 25
>>> P                            1963.4937
```

S obzirom na to da je input() funkcija, može biti napisana i kao dio izraza u naredbi za ispis. Na primjer, u pretvorbi stopa u metre:

```
>>> print ( round (0.3048 * eval (input
('unesi koliko stopa ')), 2), 'm')
unesi koliko stopa 36             10.97 m
```

Poruka može biti string dobiven izračunavanjem (evaluiranjem) znakovnog izraza, kao što je pokazano u sljedećoj vježbi:

››› 1.34 poruka

```
>>> A = input ('Auto? '); W = eval (
        input ('Snaga motora [kW] auta '
        +A +'? ')); \
 print ("Snaga motora auta " +A
        +" je " +str (W) +"[kw] -> "
        +str (round (1.36 *W)) +'[ks]')
Auto? Octavia RS
Snaga motora [kW] auta Octavia RS? 135
Snaga motora auta Octavia RS je 135[kw] -
> 184[ks]
```

Procedura exec()

Procedura exec() nam omogućuje interpretiranje stringa kao niza naredbi koji se, ako je sintaksno korektan, može izvršiti. Piše se prema pravilu:

```
procedura_EXEC : exec ( naredbe )
  naredbe      : zn_izr | tekst
```

Na primjer:

```
>>> exec ("a = 10; b = 20; "
          "print ( a*b )" )          200
>>> a, b                          (10, 20)
```

Vidimo da je string "a = 10; b = 20; print (a*b)" bio interpretiran kao da smo napisali:

```
>>> a = 10; b = 20; print (a*b)
```

Pogledajmo još jedan primjer:

```
>>> Naredbe = "print ( eval (input
('Upiši brojčani izraz ')) ); "
>>> # mora biti ; na kraju
>>> exec (Naredbe *2)
Upiši brojčani izraz 1/10 +1/20 +1/30
0.18333333333333335
```

```
Upiši brojčani izraz 123**5   28153056843
```

Dvaput su bile izvršene naredbe sadržane u znakovnoj varijabli `Naredbe`. Zato je `Naredbe` imala „`;`“ na kraju.

S obzirom na to da procedura **`exec`**`()` interpretira (izvršava) ulazni string (ulaznu liniju), naredbe sadržane u njemu moraju početi bez vodećih razmaka. U protivnom bi bila dojavljena pogreška. Na primjer:

```
>>> exec (" a = 10; b = 20; "
          "print ( a*b ) " )
a = 10; b = 20; print ( a*b )
^
IndentationError: unexpected indent
```

TEKST KAO PROGRAM

Tekst može sadržavati naredbe napisane u više redova (linija), pridružene tekstualnoj varijabli. Na primjer, niz naredbi iz vježbe **`>>>1.34`** :

›››1.35 tekst kao program

```
>>> Auto = """
A = input ('Auto? '); W = \
eval (input ('Snaga motora[kW] auta' +A
             +'? ')); \
print ("Snaga motora auta " +A +" je "
       +str (W) +"[kw] -> "
       +str (round (1.36 *W)) +'[ks]')"""
```

Program će se izvršiti sa:

```
>>> exec (Auto)
Auto? Golf V
Snaga motora [kW] auta Golf V? 66
Snaga motora auta Golf V je 66[kw] ->
90[ks]
```

Tekstualna varijabla `Auto` sadrži samo jednu liniju s nastavcima u više redova. Naredbe ili nizovi naredbi mogu biti napisani u više redova, kao što je pokazano u sljedećoj vježbi. Svaki red mora početi bez vodećih razmaka.

›››1.36 tekst kao program (2)

```
>>> Auto2 = """
A = input ('Auto? ')
W = eval (input (
  'Snaga motora [kW] auta ' +A +'? '))
print ("Snaga motora auta " +A +" je "
       +str (W) +"[kw] -> "
       +str (round (1.36 *W)) +'[ks]' )
"""
```

*Kontekstni aspekt pisanja teksta kao programa jest da ne smije sadržavati eksplicitno napisan kontrolni string `'\n'`, koji ima značenje kraja reda. Ako postoji potreba za tim, u naredbi **`print`**`()`, može se pridružiti nekom imenu izvan teksta, na primjer `NL='\n'`.*

GOVORIMO PYTHONSKI

Svako će poglavlje sadržavati poseban dio pod nazivom „Govorimo pythonski” u kojem ćemo analizirati mogućnosti primjene prethodno opisanih tipova podataka i naredbi, te ponekad prikazati njihovo „skriveno“ značenje kao „trikove“ ili „male tajne“ programiranja. Također će biti uvedene neke metode i načela programiranja.

Ovdje, poslije uvođenja brojčanog tipa podataka, znakova i stringova, definiranja brojčanog i znakovnog izraza, uvođenja varijabli i naredbe za jednostavno pridruživanje, naredbi (standardnih procedura) za ispis, izvršavanje programa sadržanih u stringu ili tekstu, te funkcije za unos podataka, pokazat ćemo njihovu primjenu u rješavanju jednostavnih problema.

Pokazat ćemo kako realizirati ponavljanje izvršavanja naredbi (bez uporabe tzv. *FOR* i *WHILE petlji*, koje ćemo uvesti u petom poglavlju). Posebno treba obratiti pozornost na „snagu“ procedure **`exec`**`()`.

DECIMALNI DIO REALNOG BROJA

Često trebamo ekstrahirati decimalni dio realnog broja. Ako je `X` realni broj i `Y` decimalni dio od `X`, to možemo uraditi na dva načina:

```
1)  Y = X - int (X)      2) Y = X % 1
```

Na primjer:

```
>>> X = 12.75; print (X -int(X), '\n',
                      X %1, sep = '')
0.75
0.75
```

DRUGI I TREĆI KORIJEN

Drugi i treći korijen nekog broja možemo dobiti potenciranjem s `1/2` i `1/3` (`1/4` za četvrti korijen itd).

Pritom treba paziti na prioritet izvšavanja operacija. Na primjer, ako napišemo:

```
>>> 2 **1/2 # = (2**1)/2                1.0
```

dobiven je pogrešan rezultat jer je izraz bio interpretiran kao `(2**1)/2`. Dakle, potenciju `1/2` treba napisati u zagradi ili `0.5`.

```
>>> 2 **(1/2); 2 **0.5
1.4142135623730951
1.4142135623730951
```

IMENA

Znamo da ime varijable može sadržavati bilo koje slovo abecede jednog ili više jezika. Na primjer, kombinaciju hrvatske (engleske), njemačke, francuske i grčke abecede:

```
Štef  Groß  Merci  Garçon  α_do_ω
```

Kako izabrati „pravo ime“ varijable? Pravilo bi bilo da treba birati imena koja podsjećaju na svoju namjenu. Na primjer, ako smo brzinu označili s `X`:

```
>>> X = eval (input ('Zadaj brzinu '))
# X - brzina
```

izborom imena `Brzina` ili `V`, kao što je uobičajeno u fizici, komentar bi bio suvišan! Početnici u programiranju skloni su za imena koristiti vlastito ime, na primjer:

```
Viktor  Viktor_  Viktor125  Viktor_5
 Darko_05_05_1980  Igor_26_04_1984
```

Treba izbjegavati predugačka imena jer se tada smanjuje čitljivost programa, „poezija“ se pretvara u „prozu“. Na primjer:

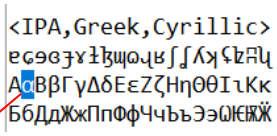
```
PDV_iznos_tarifa_1  Iznos_bruto_dohotka
```

Poneki programeri biraju „duhovita” imena varijabli i konstanti. Možda je u tome najviše pretjerao Jean-Pierre, ljubitelj mačaka i svega što je francusko. U svojim programima koristi isključivo svoje ime i imena:

```
Jeanette   Jaune   Jacques   Jacqueline
Jojo   Jean   Georges   Jean_Pierre
```

I tu priči nije kraj, ali na ovom mjestu nadilazi naše potrebe.

U poglavlju „ZNAKOVI I ZNAKOVNI NIZOVI“ detaljno su opisane kodne stranice. Ovdje dajemo primjer kako ispisati grčka slova, jer mišljenja smo da ih treba rabiti kao imena varijabli u rješavanju mnogih problema matematike, fizike, kemije itd. Na primjer, grčka slova α, β, γ i π u matematici, η, μ i Ω u fizici.
Da bismo ispisali abecedu (alfabet) velikih i malih grčkih slova najprije izaberimo Options pa Configure IDLE. U desnom dijelu prozora bit će prikazana dijelovi alfabeta nekoliko pisama. Prekopirajmo α za argument funkcije `ord()`. Kôd je 945. Kopirajmo `A` (nemojte upisati `A` jer je to slovo engleske abecede, iako izgleda jednako kao grčko, a kôd mu je 65!) za argument funkcije `ord()`. Kôd je 913.




```
>>> ord ('α')                          945
>>> ord ('A') # veliko slovo α         913
```

Sada nije problem napisati niz naredbi koje će ispisati mala i velika slova grčke abecede ili „alfabeta“.

››› Grčka slova

```
>>> znak = ("i = i+1; "
            " print (chr(i), end = ' ');")
>>> mala_grčka = ('i = 944; ' +
                  'exec (znak *25)' )
>>> velika_grčka = ('i = 912; ' +
                    'exec (znak *25)')
>>> exec (mala_grčka); print (); \
    exec (velika_grčka)
α β γ δ ε ζ η θ ι κ λ μ ν ξ ο π ρ ς σ τ
υ φ χ ψ ω
Α Β Γ Δ Ε Ζ Η Θ Ι Κ Λ Μ Ν Ξ Ο Π Ρ ? Σ Τ
Υ Φ Χ Ψ Ω
```

Možemo u formulama gdje se pojavljuje broj π umjesto imena `pi` ili `Pi` rabiti grčko slovo π, pridruživši mu prije toga vrijednost, `π = 3.1415926`.

LOTO 7/35 I EUROJACKPOT 5/50 + 2/10

Zadatak 1.4

*Izračunati broj kombinacija 7 od 35 brojeva igri **LOTO 7/35** i 5 od 50 i 2 od 10 brojeva u igri **EUROJACKPOTa**.*

Ako treba pogoditi m od n brojeva, broj kombinacija k jednak je:

$$k = \binom{n}{m} = \frac{n \times (n-1) \times \ldots \times (n-m+1)}{m!}$$

pa je broj kombinacija u igri Loto i Eurojackpot dan formulama:

$$Loto = \binom{35}{7} \quad Eurojackpot = \binom{50}{5} * \binom{10}{2}$$

››› LOTO 7/35 i EUROJACKPOT 5/50 + 2/10

```
>>> LOTO_EURO = """
n = 35; m = 1; i = n; j = 1
n_m = ("i = i-1; n = n *i; j = j+1;"
       "m = m *j; ")
exec (n_m *6); Loto = n //m *2
n = 50; m = 1; i = n; j = 1
exec (n_m *4); Euro = n //m *10*9 //2
print ('Broj kombinacija:')
print ('  Loto ', Loto);
print ('  Euro ', Euro); """
>>> exec (LOTO_EURO)
Broj kombinacija:
  Loto  6724520
  Euro  95344200
```

Program može biti dopunjen izračunom vjerojatnosti dobitka za uplatu jedne kombinacije, što vam ostavljamo za vježbu.

TREĆI KUT TROKUTA

Zadatak 1.5

Za zadana dva kuta trokuta, α i β u obliku s.m, gdje su s stupnjevi, m minute, izračunati treći kut γ.

Znamo da je zbroj kuteva trokuta jednak 180 stupnjeva, pa ćemo kut γ dobiti oduzevši od 180 zbroj kuteva α i β :

```
γ = 180 -(α + β)
```

Problem je što se kutevi α i β ne zadaju kao decimalni brojevi, već se decimalni broj „čita“ kao minute. U rješenju koje slijedi problem ćemo riješiti pretvarajući kuteve u minute. A je kut α, a B kut β u minutama. Ako je M=60, izračunava se treći kut C u minutama prema formuli:

```
C = 180*M -(A + B)
```

Kut u obliku s.m dobit će se formulom:

```
γ = C //M +(C % M)/100
```

››› Treći kut trokuta (1)

```
>>> Treći_kut = """
print ('Treći kut trokuta')
α = eval (input ('α = '))
β = eval (input ('β = ')); M = 60
s = int (α); m = round (100 *(α - s))
A = s*M +m  # A - kut α u minutama
s = int (β); m = round (100 *(β - s))
B = s*M +m  # B - kut β u minutama
C = 180*M -(A + B)
γ = C //M +(C % M)/100
print ('γ =', γ) """
>>> exec (Treći_kut)
Treći kut trokuta
α = 55.55
β = 70.07    γ = 53.58
```

Pazite, decimalni dijelovi kuteva predstavljaju minute i mogu biti u intervalu od 0.0 do 0.59. Zbog toga je zbroj u našem primjeru:

```
>>> α +β +γ    179.2
```

Ako kut x ima s stupnjeva i m minuta danih kao realni broj s.m, pretvorba minuta u decimalni broj je:

```
x % 1 /0.6
```

pa se x pretvoren u decimalni broj X može dobiti prema formuli:

```
X = int(x) +(x % 1)/0.6
```

Rabeći tu formulu sada možemo provjeriti zbroj kuteva:

```
>>> Σ = ( int(α) +int(β) +int(γ)
          +(α%1 +β%1 +γ%1) /0.6 )
>>> Σ,round(Σ)  (179.99999999999997, 180)
```

Ovdje se prvi put susrećemo s numeričkim problemom koji se odnosi na interpretaciju realnih brojeva. Pogreška je 3e-14 ($3x10^{-14}$):

```
>>> Σ +3e-14          180.0
```

Evo druge inačice rješenja problema uz pretvaranje kuteve i u decimalne brojeve.

››› Treći kut trokuta (2)

```
>>> Treći_kut_2 = """
print ('Treći kut trokuta'); m = 0.60
α = eval (input ('α = '))
β = eval (input ('β = '))
A = int (α) +(α % 1/m)
B = int (β) +(β % 1/m)
C = 180 -(A + B)
γ = int (C) +round ((C % 1)*m, 2)
print ('γ =', γ) """
>>> exec (Treći_kut_2)
Treći kut trokuta
α = 55.55
β = 70.07    γ = 53.58
```

Provjera korektnosti programa svodi se na zbrajanje decimalnih interpretacija A, B i C kuteva α, β i γ:

```
>>> A+B+C        180.0
```

FAKTORIJEL

Možda je izračunavanje faktorijela brojeva od 1 do n najbolji primjer za pokazivanje kako ponavljati izvršavanje niza naredbi. Najprije napišimo niz naredbi za ispis faktorijela brojeva od 1 do 5:

››› Faktorijel (1)

```
>>> i = 0; F = 1; T = '\t'; n = 5
>>> # početne vrijednosti
>>> # ponoviti niz naredbi n puta:
>>> i = i +1; F = F *i; print (i, T, F);\
    i = i +1; F = F *i; print (i, T, F);\
    i = i +1; F = F *i; print (i, T, F);\
    i = i +1; F = F *i; print (i, T, F);\
    i = i +1; F = F *i; print (i, T, F)
# n-ti put
1        1
2        2
3        6
4        24
5        120
```

Nedostatak ovakvog rješenja je n-puta ponavljanje niza naredbi. Kako bi to izgledalo da je n=100? Nadiđimo to uvođenjem znakovne varijable fakt koja će sadržavati niz naredbi koje se ponavljaju, s obveznim znakom ';' na kraju:

```
>>> fakt = ("i = i +1; F = F *i;" +
            "print (i, T, F); ")
```

Ponavljanje toga niza naredbi n-puta bit će fakt*n, što nas dovodi do sljedeće inačice:

››› Faktorijel (2)

```
>>> i = 0; F = 1; T = '\t'; n = 10; \
    fakt = ("i = i +1; F = F *i; "
            "print (i, T, F); " ); \
    print (); exec (fakt *n)
1        1
2        2
3        6
4        24
5        120
6        720
7        5040
8        40320
9        362880
10       3628800
```

Sada nije problem ispisati faktorijele do velikog broja n, na primjer n=100! Nedostatak je što za promjenu broja moramo ponoviti unos odlaskom iznad i s <Enter> prenijeti niz naredbi i poslije prijenosa treba unijeti novu vrijednost za n. Da bismo i to nadišli i da bismo mogli zadati vrijednost broja n izvana, uvodimo tekstualnu varijablu Fakt u sljedećoj inačici:

››› Faktorijel (3)

```
>>> Fakt = """
n = int (input (
'Računam faktorijele do broja? '))
F = 1; i = 0; T = '\t'
fakt = ("i = i +1; F = F *i; "
            "print (i, T, F); " )
print (); exec (fakt *n) """
>>> exec (Fakt)
Računam faktorijele do broja? 12

1    1
2    2
3    6
4    24
5    120
6    720
7    5040
8    40320
9    362880
10   3628800
11   39916800
12   479001600
```

››› Faktorijel (4)

```
>>> FAKT = """
n = int (input (
'Računam faktorijele do broja? '))
I = int (input (
'Ispis svih faktorijela brojeva do n'
 + ' (1) ili samo n-tog (0) '))
F = 1; i = 0; T = '\t'
fakt = 'i = i +1;  F = F *i; '
x    = fakt +" print (i, T, F); " *I
print ()
exec ( x *(n-1) )
exec ( fakt );  print (i, T, F);
print () """
>>> exec (FAKT *5)
Računam faktorijele do broja? 6
Ispis svih faktorijela brojeva do n (1)
ili samo n-tog (0) 1

1        1
2        2
3        6
```

```
4       24
5       120
6       720

Računam faktorijele do broja? 20
Ispis svih faktorijela brojeva do n (1)
ili samo n-tog (0) 0

20      2432902008176640000


Računam faktorijele do broja? 40
Ispis svih faktorijela brojeva do n (1)
ili samo n-tog (0) 0

40
815915283247897734345611269596115894272000000000
Računam faktorijele do broja? 70
Ispis svih faktorijela brojeva do n (1)
ili samo n-tog (0) 0

70
11978571669969891796072783721689098736458938142546425857555362864628009582789845319680000000000000000

Računam faktorijele do broja? 100
Ispis svih faktorijela brojeva do n (1)
ili samo n-tog (0) 0

100
933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000000
```

„CAJGER NA CAJGERU“

Zadatak 1.6

Koliko će biti točno sati, minuta i sekundi kad se na analognom satu poklope kazaljke poslije 20 sati?

Autor je ove knjige dvadesetak godina davao ovaj zadatak na ispitu iz programiranja (BASIC, FORTRAN, Pascal, Python) i, vjerovali ili ne, nitko ga nije riješio, pa ga je odlučio objaviti, [Dov1995]. Saznao je, zahvaljujući internetu, da je to postao „svjetski problem”. Evo što se moglo pročitati na jednom forumu susjedne države 2006. godine:

[**atlas** @ 08.01.2006. 16:00] @
hvala 8

ljudi necete vjerovat citav dan sam radio ovaj zadatak sa kazaljkama (kad ce se poklopiti)

Ovako sam krenuo: Kada se kazaljke(od sata i minuta) poklope to znaci da opisuju isti ugao pocev od 0... kazaljka za sate opiše 6° dok prede 1h a kazaljka za minute opise 1,2° dok prede 1min
tako i sekundara..... kazaljka za sekunde vuce kazaljku za minute a ova onu za sate...
pa sam napravio dvije petlje(za minute i sekunde) od 1 to 60 a prije toga treba unijeti "s" cijeli broj sati pa je samo sati
h:=s+prva petlja/60+druga petlja/360
samo minuta min:=prva petlja+druga petlja/60

uslov poklapanja h*6=min*1,2

dakle sve je to u redu.....ali najgore je sad dolazi jer se radi sa realnim brojevima
pa kada puno preciziram onda dobijem da se nikad u potpunosti kazaljke ne poklope
(sto i mislim da je tacno----kada bi oznacili mikrostepene na satu)

ali kada idem na priblizne vrijednosti onda mi program ispise da se kazaljke poklapaju citavu

minutu dakle svake secunde u min...
to je npr.. za SAT=3 poklapaju se od 3h :15min : 0sec do 3h : 15min 59sec
dakle dobijem 60 rezultata..... TO JE ONA ĆIZA KOJA TRAJE 1MIN

al sad kada idem na zaokruzivanje na 2 decimale problem je jos veci ali se dobije rezultat
puno vise varijabli,transformisanje iz integer u string i obratno,,kod je prevelik itd...
dakle previse.....
necete vjerovat tona hartije oko mene slike grafovi satovi pa sam odustao i od prihvatanja rjesenja mada mislim da sam u pravu ali kako ja uvijek nadem tezu varijantu nadam se da ce mi neko dodat ideju...

... imam kod od nekog dr. Zdravko Dovedan al ne razumem **nek mi neko objasni zasto ide kod ovako** zasto *60/11....

Da, zašto `60/11`? Vrlo jednostavno, jer se kazaljke od `0.00` do `12.00` sati poklope `11` puta, pa je razmak između dva poklapanja jednak `60/11`! Na primjer, kad se kazaljke poklope poslije `1` (ili `13`) to će biti `60/11` minuta, poslije `2` (ili `14`) `2*60/11` itd.

"cajger na cajgeru" (1)

```
>>> T = eval (input ("Zadaj sat "))
Zadaj sat 20
>>> S = T % 12      # S je od 0 do 11
>>> m = S *60/11.0 # m minuta poslije T
>>> M = int (m)     # M minute (cijeli)
>>> s = round ((m-M)*60, 2) # s sekunde
>>> print (T, ':', M, ':', s,
           sep = '' )          20:43:38.18
```

Ako želimo ponoviti naredbe za neki drugi sat, morali bismo redom ponoviti sve naredbe. Bolje je napisati ih u jednoj liniji, više redova:

"cajger na cajgeru" (2)

```
>>> T = eval (input ("\nZadaj sat "));\
    S = T % 12; m = S *60/11.0; \
    M = int (m); s = round (
        (m-M)*60, 2);  print(); \
    print (T, ':', M, ':', s, sep = '')

Zadaj sat 18                    18:32:43.64
```

Sada, ako želimo ponoviti izvršavanje naredbi, treba se vratiti na kraj naredbi i s <Enter> ih kopirati u tekuću liniju. No, možemo naredbe pretvoriti u tekst i pridružiti ga varijabli, na primjer CAJGER:

››› "cajger na cajgeru" (3)

```
>>> CAJGER = """
T = eval (input ("Zadaj sat "));  \
    S = T % 12; m = S *60/11.0;   \
    M = int (m); s = round (
                     (m-M)*60, 2);\
    print (T, ':', M, ':', s,
           sep = ''); print () """
>>> exec (CAJGER *2)
Zadaj sat 15    15:16:21.82
Zadaj sat 16    16:21:49.09
```

FIBONACCIJEV NIZ

U matematici su Fibonaccijevi brojevi, označeni s F_n brojevi Fibonaccijevog niza dobiveni tako da je svaki član jednak zbroju prethodna dva,

https://en.wikipedia.org/wiki/Fibonacci_number.

Prva dva člana su $F_1 = 1$, $F_2 = 1$, potom $F_3 = F_1+F_2 = 2$ itd. Općenito se n-ti član izračunava rekurzivnom formulom:

$$F_1 = 1,\ F_2 = 1$$
$$F_n = F_{n-2} + F_{n-1}$$

Zasad ne znamo definirati rekurzivne funkcije pa ćemo pokazati kako riješiti problem iteracijom.

››› Fibonaccijev niz (1)

```
>>> fib = """
print (a+b, end = ' ')
t = b; b = a+b; a = t """
>>> print (fib) # Ispis programa
print (a+b, end = ' ')
t = b; b = a+b; a = t
>>> a = 0; b = 1; print (); print (b,
  end = ' '); n = 20; exec (fib *(n-1))

1 1 2 3 5 8 13 21 34 55 89 144 233 377
610 987 1597 2584 4181 6765
```

Niz naredbi

```
t = b; b = a+b; a = t
```

primjer je kako prethodnu vrijednost varijable b pridružiti kao novu vrijednost varijable a. Za to se rabi pomoćna varijabla t.

››› Fibonaccijev niz (2)

```
>>> FIB = """
Fn = "t = b; b = a+b; a = t;"
n = int (input ('n = '))
I = int (input ('Ispis svih članova ' +
          '(1); ili n-tog (0) '))
a = 0; b = 1; exec ("print (b,
                    end = ' ')" *I)
x = Fn +" print (b, end = ' '); " *I
exec (x *(n-2))
I = 0; exec (Fn); print (b) """
>>> exec (FIB)
n = 20
Ispis svih članova (1); ili n-tog (0) 0
6765
>>> exec (FIB)
n = 20
Ispis svih članova (1); ili n-tog (0) 1
1 1 2 3 5 8 13 21 34 55 89 144 233 377
610 987 1597 2584 4181 6765
```

INTERESANTNI IZRAZI

Na Facebooku „kruže“ interesantni izrazi, a odnosi se na dobivanje interesantnih rezultata njihovim izračunavanjem.

a)
```
1 x 1 = 1
11 x 11 = 121
...
111111111 x 111111111
= 12345678987654321
```

b)
```
1 x 9 + 2 = 11
12 x 9 + 3 = 111
...
123456789 x 9 + 10 =
1111111111
```

c)
```
1 x 8 + 1 = 9
12 x 8 + 2 = 98
...
123456789 x 8 + 9 =
987654321
```

d)
```
9 x 9 + 7 = 88
98 x 9 + 6 = 888
...
987654321 x 9 + -1 =
8888888888
```

Analizirajući dane izraze možemo definirati sljedeće nizove naredbi za njihovo generiranje:

››› Interesantni izrazi

```
>>> \
_a = "B = B*10 +1; print (B, "    + \
     "'x', B, '=', B**2); ";        \
_b = "B = B*10 +i; print (B, "    + \
     "'x 9 +', i+1, '=', "         + \
     "B*9 +i+1); i = i +1; ";       \
_c = "B = B*10 +i; print (B, "    + \
     "'x 8 +', i, '=', B*8 +i);" + \
     "i = i +1; ";                  \
_d = "B = B*10 +i; print (B, "    + \
     "'x 9 +', i-2, '=', "         + \
     "B*9 +i-2); i = i -1; "
```

Ispišimo izraze _a i _b:

```
>>> B = 0; exec (9 *_a)
  1 x 1 = 1
  11 x 11 = 121
  111 x 111 = 12321
  1111 x 1111 = 1234321
  11111 x 11111 = 123454321
  111111 x 111111 = 12345654321
  1111111 x 1111111 = 1234567654321
  11111111 x 11111111 = 123456787654321
  111111111 x 111111111 = 12345678987654321

  >>> B = 0;  i = 1; exec (9 *_b)
  1 x 9 + 2 = 11
  12 x 9 + 3 = 111
  123 x 9 + 4 = 1111
  1234 x 9 + 5 = 11111
  12345 x 9 + 6 = 111111
  123456 x 9 + 7 = 1111111
  1234567 x 9 + 8 = 11111111
  12345678 x 9 + 9 = 111111111
  123456789 x 9 + 10 = 1111111111
```

Definirajmo „glavni program“ u kojem će se izabrati izraz za ispisivanje:

```
  >>> ISPIS = """
  a = 0; b = 1; c = 1; d = 9
  x = input (
  'Ispisujem izraz a, b, c ili d ')
  y = '_' +x
  exec ("B = 0; i = eval (x); "
        +"exec (9 *eval (y)) ") """
```

```
>>> exec (ISPIS)

Ispisujem izraz a, b, c ili d c
1 x 8 + 1 = 9
12 x 8 + 2 = 98
123 x 8 + 3 = 987
1234 x 8 + 4 = 9876
12345 x 8 + 5 = 98765
123456 x 8 + 6 = 987654
1234567 x 8 + 7 = 9876543
12345678 x 8 + 8 = 98765432
123456789 x 8 + 9 = 987654321

>>> exec (ISPIS)

Ispisujem izraz a, b, c ili d d
9 x 9 + 7 = 88
98 x 9 + 6 = 888
987 x 9 + 5 = 8888
9876 x 9 + 4 = 88888
98765 x 9 + 3 = 888888
987654 x 9 + 2 = 8888888
9876543 x 9 + 1 = 88888888
98765432 x 9 + 0 = 888888888
987654321 x 9 + -1 = 8888888888

>>>                  # KRAJ POGLAVLJA #
```

2.

PROGRAMSKI MÔD

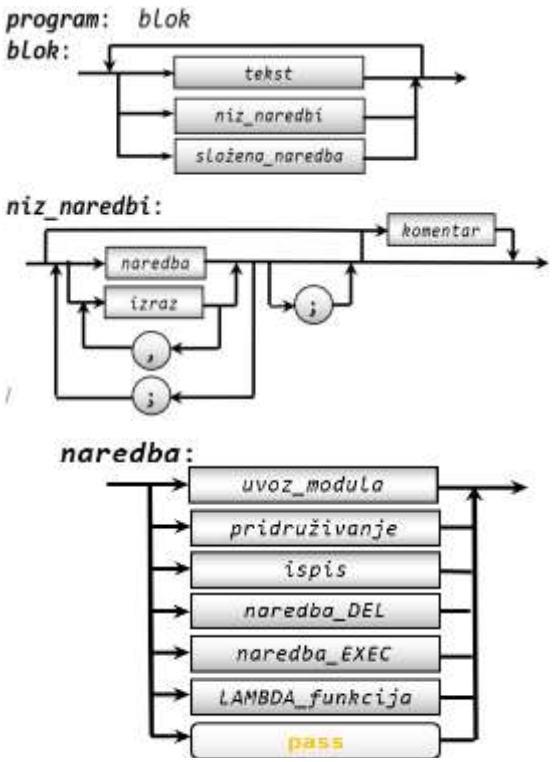
U prethodnom smo poglavlju rabeći interaktivni ili Shell mod Pythona neformalno uveli brojeve, izraze, stringove, standardne funkcije i procedure. Počeli smo pisati i izvršavati tekstove kao programe, rješavajući neke jednostavne probleme.

U interaktivnom smo modu pisali i odmah izvršavali izraze ili naredbe. Naredbe nisu bile upamćene. Ako bismo ih trebali ponoviti, ili ne, trebalo ih je ponovo napisati ili otići u liniju u kojoj je zadnji put bila napisana i s Enter je prenijeti u tekuću liniju.

U ovom poglavlju uvodimo programski môd, tekstualne datoteke koje će sadržavati programe (skripte) i koji će se moći izvršavati onda kad to želimo. Uvođenjem programskog moda nećemo se odreći interaktivnog moda.

No, za potpuno razumijevanje svakog jezika za programiranje, pa tako i Pythona, neophodno je znati njegovu osnovnu strukturu. Čine je leksička struktura, pravila pisanja naredbi (sintaksa) i značenje naredbi (semantika).

Detaljno su opisani brojčani izrazi, standardne funkcije i standardni moduli `math` *i* `random` *koji sadrže biblioteke brojčanih funkcija. Prošireno je značenje varijable i opisana naredba za višestruko, konkurentno i operatorsko pridruživanje. Sve je prikazano u interaktivnom modu, pa se podrazumijeva da ćete najbrže „progovoriti pythonski" ako usporedo s čitanjem i unosite dane primjere na svom računalu. Poseban dio pod nazivom* PROGRAMI *sadrži primjere pisanja programa u rješavanju nekih jednostavnih problema.*



Zboj.py

```
# Zbroj znamenki prirodnog broja
N     = input (
'Zadaj veliki prirodni broj ')
n     = int (N);  k = len (N); S = 0
Zbroj = ("n, b = divmod (n, 10); " +
         "S += b; ")
exec  (Zbroj *k)
print ('Broj', N, 'ima', k,
       'znamenki. Zbroj znamenki =', S)
>>>
Zadaj veliki prirodni broj
99999888888881231312357659999
Broj 99999888888881231312357659999 ima
28 znamenki. Zbroj znamenki = 175
```

Osnovna struktura Pythona

U praksi se često pokušava naučiti neki jezik za programiranje kroz primjere, bez davanja pravila pisanja. Čak i za jednostavne primjere, kao što je na primjer e-mail adresa, pravila se prenose „s koljena na koljeno“, u tramvaju ili kafiću. Dajući svoju e-mail adresu često ćemo čuti „Piši malim slovima!“ što nije istina. e-mail adresa može se pisati malim i/ili velikim slovima, a važno je da su to slova engleske abecede i da adresa ne može sadržavati neke znakove interpunkcije.

No, za potpuno razumijevanje svakog jezika za programiranje, pa tako i Pythona, neophodno je znati njegovu osnovnu strukturu. Čine je leksička struktura, pravila pisanja naredbi (sintaksa) i značenje naredbi (semantika).

LEKSIČKA STRUKTURA

Definirati leksičku strukturu Pythona znači definirati njegov alfabet ili abecedu, rječnik i leksička pravila.

Alfabet

Alfabet je skup znakova. Znak jest jedinstvena, nedjeljiva cjelina, kao što su, na primjer, slova, znamenke, +, -, *, /, (,), [,] itd. Drugim riječima, to je ono što je označeno na tipkovnici (uključujući i razmak ili "blank"), što se interpretira kao znak na ekranu ili drugom izlaznom mediju, uz dodatak kontrolnih znakova. Znakovni tip čini skup znakovnih vrijednosti. Napominjemo, da je ovo naša definicija, jer znak nije definiran kao posebni tip podataka u Pythonu. To je string duljine 1.

Godine 1968. standardizirani su kodovi znakova koji su se rabili na kompjuterima. Uvedena je ASCII tablica (American Standard Code for Information Interchange). Sadržavala je 128 znakova (brojke, slova, znakove interpunkcije itd), s kodovima od 0 do 127. Na primjer, znaku (slovu) 'A' bio je dodijeljen kôd 65, znaku 'a' 97 itd. U tom su skupu znakova bila samo velika i mala slova engleskog alfabeta, što znači da nije bilo slova s naglascima, kao što su 'ä', 'é', 'è', 'ö' ili 'ü', pa jezici koji su u svojem alfabetu imali te znakove nisu mogli rabiti ASCII tablicu. Tada su proizvođači kompjutera (IBM, Univac, CDC, DEC itd.) za naručitelje njihovih strojeva iz Francuske, Njemačke ili iz jugoistočne Europe zamjenjivali „nepotrebne“ znakove '@', '[', '\', ']', '^', '`', '{', '|', '}' i '~' s posebnim slovima, na primjer u Hrvatskoj, sa 'Ž', 'Š', 'Đ', 'Ć', 'Č', 'š', 'đ', 'ć' i 'č'.

Pojavom osobnih računala početkom 80-tih godina prošloga stoljeća, koja su bila 8-bitna, ASCII tablica je proširena s dodatnim znakovima koji su imali kodove od 128 do 255. U taj su se dio dodavali slova s naglascima, grafički znakovi i drugi znakovi, kao što su neka grčka slova. Tada je uveden i pojam „kodnih stranica“, na primjer, 437 (DOS), 850 (Latin-1) i 852 (Latin-2), koje su sadržavale različite skupove znakova. Pojavom Windowsa uvedene su još neke kodne stranice, na primjer 1252 (MS Windows – Latin 1) i 1250 (MS Windows – Latin 2).

Uvođenjem kodnih stranica različiti su strojevi imali različite kodove, što je dovelo do problema razmjene podataka. Osim toga, bilo je premalo mjesta da se paralelno rabe dva pisma, na primjer latinica u zapadnoj Europi i ćirilica u nekim drugim zemljama Europe.

Rješenje se problema naziralo u drugom dijelu 80-tih godina, pojavom 16-bitnih procesora koji su imali na raspolaganju 2^{16}=65536, umjesto 2^{8}=256, različitih vrijednosti. Uvedena je „Unicode” standardizacija. Početni je cilj bio da *Unicode* sadrži pisma svakog pojedinog ljudskog jezika. Pokazalo se da čak 16 bita nije dovoljno da zadovolji taj cilj, pa sada moderna *Unicod*e specifikacija koristi širi spektar kodova, od 0 do 1,114,111 (0x10ffff heksadecimalno).

Unicode standard opisuje kako su znakovi predstavljeni kodnim točkama. Kodna točka je cjelobrojna vrijednost, obično prikazana kao heksadecimalni broj. Unicode standard sadrži mnogo tablica znakova i njihovih kodnih točaka. Na primjer, Windows-1250 je kodna stranica u Microsoft Windowsima koja se koristi za pisanje tekstova u jezicima istočne Europe (poljski, češki slovački, ..., hrvatski). Alfabet Pythona čine sljedeći skupovi znakova:

- mala slova iz Unicode skupa znakova:
  a b … y z … č ć đ š ž … α β γ … д ж …
- velika slova iz Unicode skupa znakova:
  A B … Y Z … Č Ć Đ Š Ž … Γ Δ … Д Ж …
- brojke: 0 1 2 3 4 5 6 7 8 9
- posebni znakovi:
  ! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [\ ] ^ _ ` { | } ~ j J

Vidimo da su posebni znakovi i slova `j` i `J` koji označuju imaginarne brojeve. U opisu leksičke strukture često ćemo koristiti notaciju regularnih izraza Pythona. Ponekad ćemo je kombinirati s proširenom Backus-Naurovom formom (ENBF) i/ili sintaksnim dijagramima.

Rječnik

Nad alfabetom Pythona definirane su sljedeće klase riječi ili simbola:

- cijeli brojevi
- realni brojevi
- imaginarni brojevi
- znakovni nizovi
- rezervirane riječi
- standardna imena
- imena
- posebni simboli

Realne brojeve, znakovne nizove (stringove) i imena definirali smo u prethodnom poglavlju.

Cijeli brojevi

Cijeli broj može biti dekadski, binarni, oktalni i heksadecimalni. U prethodnom smo poglavlju pisali samo dekadske brojeve. Potpuna su pravila pisanja cijelih brojeva definirana sa:

```
cijeli_broj : dekadski_broj   |
              binarni_broj    |
              oktalni_broj    |
              heksadecimalni_broj
dekadski_broj      : [0]+ | [1-9] [0-9]*
binarni_broj       : 0[bB] [01]+
oktalni_broj       : 0[oO] [0-7]+
heksadecimalni_broj : 0[xX] [0-9a-fA-F]+
```

Ako binarni, oktalni ili heksadecimalni broj (bez prefiksa) općenito napišemo kao

$$b_n b_{n-1} \ldots b_1 b_0$$

gdje su b_i, i=0, 1, … , n, znamenke binarnih, oktalnih ili heksadecimalnih brojeva, u kojima je značenje heksadecimalnih znamenki `a` ili `A` do `f` ili `F`:

```
[aA]  [bB]  [cC]  [dD]  [eE]  [fF]
 ↓     ↓     ↓     ↓     ↓     ↓
 10    11    12    13    14    15
```

značenje binarnih, oktalnih i heksadecimalnih brojeva jest cjelobrojna vrijednost, dekadski broj, dobiven izračunavanjem izraza:

$$b_n \times B^n + b_{n-1} \times B^{n-1} + \ldots + b_1 \times B^1 + b_0$$

gdje je `B` jednako 2 za binarne, 8 za oktalne ili 16 za heksadecimalne brojeve. Evo nekoliko primjera binarnih, oktalnih i heksadecimalnih cijelih brojeva:

```
>>> 0b10101010101010101010101  592405
>>> 0B01111111111111111111111  4194303
>>> 0o10          8   >>> 0o77          63
>>> 0O77777777    16777215
>>> 0x1           1   >>> 0X10          16
>>> 0xA           10  >>> 0x53          83
>>> 0Xabcdefedcba       11806311374010
>>> 0XFFFFFF          16777215
>>> 0XaBCdEfEDCbA       11806311374010
>>> 0xabcdef          11259375
>>> 0xq       SyntaxError: invalid token
>>> 0o8       SyntaxError: invalid token
```

Imaginarni brojevi

Imaginarni broj je riječ koja se piše prema pravilu:

```
imaginarni_broj  :
  ( dekadski_broj | realni_broj ) [jJ]
```

Primjeri imaginarnih brojeva:

```
1j 1J 0.j .0J 3.14J 12.55j 0.01j 101j
```

Rezervirane (ključne) riječi

Rezervirane riječi, ili kako ih se ponekad naziva „ključne riječi“, imaju unaprijed definirano ili „rezervirano" značenje. Pojavljuju se kao dijelovi naredbi Pythona i nije ih dopušteno rabiti u druge svrhe. Evo potpunog skupa rezerviranih riječi Pythona napisanih alfabetski:

```
and  as  assert  break  class  continue
def  del  elif  else  except  False
finally for  from  global  if  import
in  is  lambda  None  nonlocal  not  or
pass  raise return  True  try  while
with  yield
```

U osnovnim postavkama Pythonovog editora rezervirane su riječi obojene narančasto. Većina rezerviranih riječi dio su naredbi Pythona. Riječi `False` i `True` su logičke vrijednosti, `and`, `not` i `or` su logički operatori, a `in` i `is` relacije.

Standardna imena

Klasa riječi nazvana standardna imena odnosi se na skup riječi koje imaju unaprijed definirano značenje. Označuju imena standardnih tipova, konstanti, varijabli, procedura i funkcija. Za razliku od rezerviranih riječi, smije im se promijeniti značenje (tj. mogu se izabrati kao imena u programu), ali se to ne preporučuje, jer im se tada gubi osnovno značenje. U osnovnim postavkama Pythonovog editora standardna su imena obojena ljubičasto. Primjeri standardnih imena:

```
abs bin chr complex dict divmod eval exec
float hex input int len list max min ord
print round set str tuple
```

Posebni simboli

Posebni simboli (riječi) su posebni znakovi ili kompozicije od dva ili tri posebna znaka. Posebni znakovi dani su u opisu alfabeta. Evo svih posebnih simbola sačinjenih od dva ili tri znaka:

```
!=  *=  <<  <<=  ^=  """  +=  <=  __  %=
==  -=  &=  >=  ...  >>  '''  >>=  //
//=  /=  |=  **  **=  f'  r'  r"
```

Leksička pravila

U tvorbi riječi Pythona vrijede sljedeća pravila:

1) Sve se riječi pišu kompaktno, bez razmaka (blankova). Jedino se u nizu znakova razmak smatra dijelom riječi.
2) Tekst može biti napisan u više redova.
3) Rezerviranje riječi, imena i standardna imena su „case sensitive", što znači da su dvije riječi napisane samo malim slovima i istim takvim samo velikim slovima, različite.

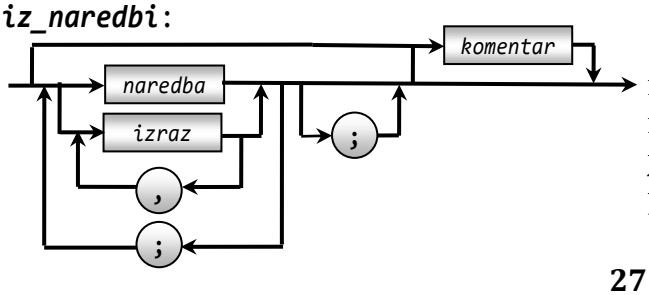
SINTAKSNA STRUKTURA

Osnovna sintaksna struktura Pythona dana je sljedećim pravilima:

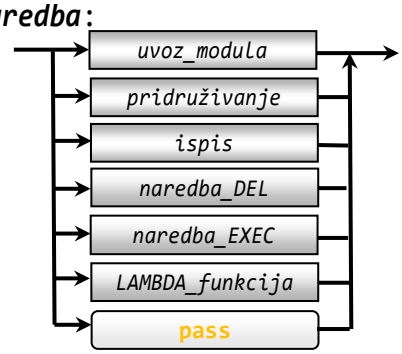
program: *blok*

blok:

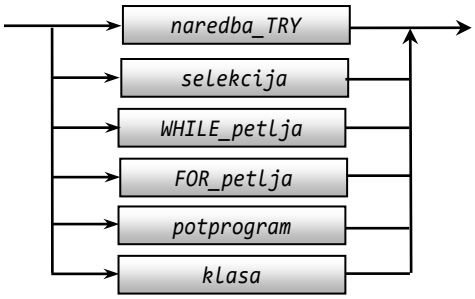
niz_naredbi:



naredba:



složena_naredba:



Dodatna sintaksna pravila

U svim pravilima pisanja naredbi Pythona pojavljuje se razmak (blank) između određenih klasa riječi. Međutim, unošenjem razmaka pravila bi postala složenija i nepreglednija. Zbog toga ćemo pravilima pisanja dodati:

1) Između parova riječi mora stajati barem jedan razmak, posebni znak ili poseban simbol.
2) Sve se primitivne naredbe, nizovi naredbi i počeci složenih naredbi pišu u jednoj liniji koja može biti nastavljena u novom redu pišući \ na kraju reda. Jedino se podaci određenih struktura podataka, koje se pišu između zagrada `[]`, `{}` ili `()`, izrazi napisani unutar zagrada i argumenti u pozivu funkcija smiju pisati u više redova bez oznake nastavka.
3) Počeci pisanja naredbi bloka moraju biti u istoj koloni (stupcu). Naredbe osnovnog bloka programa imaju razinu 0 i pišu se od prve kolone.

Programiranje u Pythonu uvodi disciplinu „lijepog" (strukturiranog) pisanja programa. Naredbe jednog bloka pomaknute su udesno („uvučene") za jedno ili više mjesta i poravnate.

To u početku zbunjuje one koji su programirali u nekom drugom jeziku, na primjer u Pascalu u kojem se naredbe bloka pišu između riječi BEGIN i END, ili u jeziku C i njegovim derivatima gdje se naredbe bloka pišu unutar vitičastih zagrada.

SEMANTIKA PYTHONA

Programiranje ima značenje obrade podataka. Možda je to Niclaus Wirth (tvorac Pascala), najbolje definirao formulom:

program = podaci + algoritam

Podaci

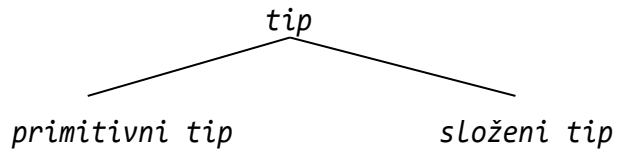
Python je u potpunosti objektno orijentirani (usmjereni) jezik za programiranje. Od inačice `3.0` više ne govorimo o „tipovima podataka“ već o „klasama“, pa podaci u Pythonovom programu predstavljaju objekte iz pojedinih klasa, koje mogu biti:

- ugrađene, standardne. Objekte tih klasa prihvaćamo kao dio Pythona. Kažemo da su to standardni objekti,
- iz biblioteka proširenja (standardnih modula) i
- stvorenu u aplikaciji od strane programera.

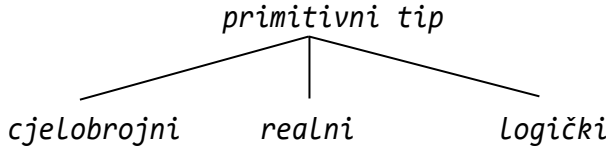
Dakle, imamo različite „vrste" objekata za različite vrste podataka. Pythonove ugrađene klase podataka iz tradicionalnih razloga nazivat ćemo „tipovima podataka“. Zadržana su samo imena tipova koja sada predstavljaju imena pojedinih klasa podataka. Na primjer, prije su imena „`int`“ ili „`str`“ predstavljala tipove podataka „`int`“ i „`str`“, a sada su to imena klasa „`int`“ i „`str`“. O klasama će biti riječi mnogo kasnije u ovoj knjizi. Dotad, da bismo mogli razumjeti opis pojedinih tipova (klasa) podataka, dovoljno je znati da klase sadrže atribute i funkcije (metode), a da su pojedine vrijednosti „objekti“ ili „instance“ tih klasa, nad kojima su definirani atributi i metode (funkcije i procedure).

Tip podatka jest skup vrijednosti koje imaju neke zajedničke karakteristike. Najznačajnija od tih karakteristika jest skup operacija koje su definirane nad vrijednostima određenog tipa.

Python ima nekoliko proširenja standardnih tipova podataka u odnosu na neke druge jezike za programiranje. Na najvišoj razini, tip podataka u Pythonu može biti prikazan kao



Primitivni (ili skalarni) tipovi podataka u Pythonu su:



Složeni tipovi podataka su:

kompleksni string n-torka lista
skup rječnik

Osim ove podjele, tipove podataka možemo grupirati i u određene skupine. Na primjer, cjelobrojni, realni i kompleksni tip podataka pripadaju brojčanim podacima, a znakovni niz, *n*-torke i liste sekvencijalnim tipovima podataka, a zajedno sa skupom i rječnikom čine tzv. „kolekcije“ ili „zbirke“ podataka.

Algoritam

Svaka naredba, koja je dio algoritma, ima svoju semantiku ili značenje koje se odnositi na efekt njezina izvršenja na računalu. To je tzv. „operativna semantika”. Pod semantikom programa podrazumijevat ćemo kompoziciju značenja naredbi od kojih je program sastavljen. Naredbe Pythona možemo pisati u interaktivnom i programskom modu.

U interaktivnom modu se izrazi i naredbe pišu i odmah izvršavaju. To je posebno važno pri učenju novih naredbi jer im možemo bolje i brže naučiti značenje.

U programskom modu naredbe se pamte kao tekstualna datoteka i izvršavaju po potrebi. Poslije svakog izvršavanja programa automatski se prelazi u interaktivni môd gdje su prikazani rezultati izvršavanja programa (ako ih ima). Karakteristika je Pythona da završetkom izvršavanja programa, ispisa ili pamćenja rezultata na sekundarnoj memoriji, u radnoj se memoriji pamte sve globalne varijable uvedene u programu. Prelazak u programski môd je unosom novog programa (treba otvoriti novi prozor: `File` → `New File` ili `Ctrl_N`) ili pozivom postojećeg (`File` → `Open` ili `Ctrl_O`).

Programi se mogu pisati i u drugim editorima teksta (Notepadu, npr.), čemu su često skloni “stari programeri”, osobe koje su već programirale u nekim drugima jezicima za programiranje. Preporučujemo da se tako ne radi iz tri razloga: Pythonov editor (Shell) obojat će pojedine vrste riječi, čime se dobiva pregledniji program, uvlači tekst blokova pri pisanju programa i, treće, možda najbitnije, program se može izravno izvršiti čime se prelazi u interaktivni mod. □

Osim navedenog, Pythonov editor kontrolira djelomično sintaksu prilikom pisanja zagrada. Na primjer, poslije **`print`**) uz zvučni signal će biti dojavljena pogreška i neće biti dopušten nastavak pisanja, bez

obzira u kojem smo modu. Ne dopušta pisanje zatvorene zagrade prije otvorene niti dopušta prelazak u novu liniju, sve dok broj zatvorenih zagrada ne bude jednak otvorenim.

Ako je bila otvorena zagrada, u izrazu ili funkciji, poslije Enter prijeći će se u novi red i linija neće biti završena, tj. bit će generiran automatski nastavak linije. To smo često koristili u prehodnom poglavlju i koristit ćemo u cijeloj knjizi, prije svega zbog suženog prikaza programa. Ponekad ćemo iz tog razloga uvesti otvorenu zagradu na početku izraza i zatvorenu na kraju.

Moduli

Python dopušta grupiranje potprograma (procedura i funkcija) u zasebne cjeline - module. To su kompilacijske cjeline koje se kao biblioteka potprograma i globalnih imena bilo kojega značenja (tipovi, konstante ili varijable) mogu uključiti u potpunosti ili djelomično u bilo koji program. Moduli mogu biti standardni (pripadaju Pythonovoj biblioteci programa) ili definirani od strane korisnika.

UVOZ MODULA

Rekavši da su moduli zasebne kompilacijske cjeline želimo istaknuti da se pišu posebno tj. da nisu dio programske cjeline koja sadrži glavni program, odnosno, nisu dio radne memorije u interaktivnom modu. Da bismo ih uključili ili „uvezli" u svoje radno okruženje, koristimo naredbu za uvoz modula. Piše se prema pravilu:

```
uvoz_modula : import ime_modula
                    { , ime_modula }
ime_modula  : ime [ as ime ]
```

Ako je uključena opcija `as ime`, originalno ime modula može se preimenovati.

Naredba DIR

Pregled sadržaja modula (lista imena podataka, atributa i funkcija - metoda) može se dobiti izvršavanjem *naredbe DIR*:

```
naredba_DIR : dir ( ime_modula )
```

Naredba HELP

Opis svih funkcija i podataka modula mogu se prikazati *naredbom HELP*:

```
naredba_HELP :
  help ( ime_modula | . ( ime_p_f ) )
```

```
ime_p_f      : ime_podatka | ime_funkcije
```

Uporaba podataka i funkcija modula ostvaruje se pisanjem:

```
ime_modula . ( ime_p_f ([ parametri ]) )
```

```
>>> help (print)
...
print(...)
    print(value, ..., sep=' ', end='\n',
file=sys.stdout, flush=False)
```

Naredba FROM

Ako modul uvezemo *naredbom FROM*:

```
naredba_FROM : from ime_modula import
                  (* | ime_p_f {, ime_p_f })
  ime_p_f    :  ime [ as ime ]
```

Ako je napisano `import *`, tada u pozivu njegovih svojstava i metoda (funkcija) izostavljamo ime modula i točku. I u *naredbi HELP* izostavlja se ime modula i točka:

```
help (ime_p_f )
```

Naredbom `help()`, pišući ime standardne funkcije kao argument, dobit ćemo opis njezina značenja. Na primjer:

```
>>> help (abs)
abs(...)
    abs(number) -> number
```

Sve posebne znakove i posebne simbole možete pogledati ako poslije

```
>>> help ()     ...
```

napišete:

```
help> symbols
...
!=   *=  <<  ^   ...   **=  <   ]
```

Rezervirane riječi možemo dobiti ako napišemo:

```
help> keywords
...
False  break   for      not    None
class  from    or       True   continue
global pass    if       raise  and
del    import  return   as     elif
in     try     assert   else   is
while  async   except   lambda with
await  finally nonlocal yield
```

Ispisana je lista rezerviranih riječi, ali nije uređena alfabetski. Prekid prikaza je sa `quit`.

PROMJENA IMENA MODULA ILI NJEGOVIH METODA

Ako želimo, možemo promijeniti ime uvezenog modula. To se postiže sa:

```
novo_ime_modula : import ime_modula as
                         drugo_ime
  drugo_ime     : ime
```

Ime metode modula možemo promijeniti pridruživanjem:

```
novo_ime_metode = [ ime_modula . ]
                  ime_metode
```

BRISANJE MODULA

Modul možemo izbrisati (izbaciti iz radne memorije) *naredbom DEL*:

```
del ime_modula { , ime_modula }
```

PRIMJER PROGRAMA

Na kraju ovoga podpoglavlja evo jednog primjera kompletnog programa. To je program za prevođenje arapskih brojeva u rimske i obrnuto (v. 9. poglavlje).

ARA.py

```
try :
  from pickle import *
  f   = open ( 'ARA.dat', 'rb')
  ARA = load (f);  f. close()
except :
  RB = lambda x, y = '', z = '' : (
       ('', x, 2*x, 3*x) if x == 'M'
       else ('', x, 2*x, 3*x, x+y, y,
             y+x, y+2*x, y+3*x, x+z) )
  M = RB ('M'); C = RB ('C', 'D', 'M')
  X = RB ('X', 'L', 'C')
  I = RB ('I', 'V', 'X')
  ARA = {}
  for i in range (1, 4000) :
    s = [int (c) for c in "%04d" % i]
    r = (M [s[0]] +C [s[1]] +
         X [s[2]] +I [s[3]] )
    a = str (i); ARA[a], ARA[r] = r, a
    f =  open ('ARA.dat', 'wb')
    dump (ARA, f); f. close()
a = input ('Upiši rimski ili arapski '
           'broj ').upper()
if a in ARA : print ( a,'-->', ARA[a])
else        : print ( a,
  'nije rimski broj' if a. isalpha()
  else 'arapski broj izvan domene'
    if a.isdigit() else
      'nije rimski niti arapski broj')
```

Dali smo ovaj program ne da biste ga razumjeli, pogotovo ako ste početnici u programiranju, niti da biste se čudili jednostavnosti rješenja problema prevođenja, ako ste programirali u nekim drugim jezicima, već da biste „stekli osjećaj“ kako izgleda leksička i sintaksna struktura programa napisanog u Pythonu.

Prepoznajemo cijele brojeve, znakove i stringove (obojeno zeleno), imena, imena standardnih funkcija (obojena ljubičasto), rezervirane riječi (obojene narančasto), ostale znakove i posebne simbole.

Programski môd

U interaktivnom smo modu pisali i odmah izvršavali izraze ili naredbe. Naredbe nisu bile upamćene. Ako bismo ih trebali ponovno, trebalo ih je ponovo napisati ili otići u liniju u kojoj je zadnji put bila napisana i s <Enter> je prenijeti u tekuću liniju.

Sada uvodimo programski môd, tekstualne datoteke koje će sadržavati programe (skripte) i koji će se moći izvršavati onda kad to želimo.

STRUKTURA I IZVRŠAVANJE PROGRAMA

Tekst programa se piše od prve kolone, bez vodećih razmaka (uvlačenja). Za razliku od drugih jezika za programiranje, u kojima uvlačenje teksta u programima služi samo za čitanje, u Pythonu je to posebno važno i obvezno u pisanju blokova složenih naredbi. Na primjer, napišimo program koji će izračunati opseg i površinu kruga polumjera r. Prvo s **Ctrl_N** otvaramo novi prozor u kojem ćemo napisati program:

```
# Krug.py
# OPSEG I POVRŠINA KRUGA
r = float (input(
           'Upiši polumjer kruga '))
π = 3.1415926
# ime π možemo "posuditi" iz Shell moda
# ili sa >>> chr(960) -> 'π'
O = round (2*r  *π, 2)
P = round (r**2 *π, 2)
O, P
```

Pokrenimo program, Run pa Run Module ili F5. Tražit će se ime datoteke u kojoj će program biti upamćen. Ime ne smije sadržavati sljedeće znakove:

```
< > / \ | " : ? *
```

Upišite ime, bez ekstenzije. Na primjer, **Krug**. Python će sam dodati ekstenziju „py“. Potom će biti ispisano:

```
>>>
====== RESTART: C:/Python39/krug.py =====
Upiši polumjer kruga 10.5
>>>
```

Prvo je resetirana radna memorija i ispisana putanja programske datoteke (njezina lokacija na vašem računalu). Potom je ispisana poruka za unos polumjera kruga. Unijeli smo **10.5** i nije se ništa dogodilo. Vrijednost izraza (varijabli) O i P nije ispisana, jer u programskom modu ispis je moguć samo naredbom za ispis. Vrijednosti su izračunate i nalaze se u radnoj memoriji, što možemo i provjeriti:

```
>>> O 65.97    >>> P 346.36    >>> r 10.5
```

Ako se sada vratimo u program i umjesto O, P napišemo:

```
print ('Opseg =', O, ' Površina =', P)
```

poslije zahtjeva za izvršenjem programa, **F5**, prvo bismo bili upozoreni da izvorni kôd mora biti spremljen (jer je bilo izmjena). Pritiskom na OK program bi bio spremljen pod ranije pridruženim imenom.

```
>>>
Upiši polumjer kruga 10.5
Opseg = 65.97  Površina = 346.36
```

Sadržaj se radne memorije može resetirati s **Ctrl_F6**.

Formatirani stringovi

Naredbom za ispis ispisivali smo niz vrijednosti odvojenih razmakom ili tabulatorom. Na primjer, ako smo trebali ispisati tablicu brojeva 1 do 10, njihovih kvadrata i drugih korijena, napisali bismo tekst kao program i izvršili ga:

››› Tablica

```
>>> Tablica = """T = '\t'
print ('i', T, 'i**2', T, 'i**0.5');
print ('-' *23);  i = 0;  n = 10
red = "i = i+1; print (i, T, i**2, T,
round (i**0.5, 4)); "
exec (red *n) """

>>> print (); exec (Tablica)
i       i**2    i**0.5
-----------------------
1       1       1.0
2       4       1.4142
3       9       1.7321
4       16      2.0
5       25      2.2361
6       36      2.4495
7       49      2.6458
8       64      2.8284
9       81      3.0
10      100     3.1623
```

Formatiranim stringom ili obrascem, kojeg ćemo ovdje opisati, moguće je generirati stringove u željenom obliku (formatu). Rekli smo „generirati" jer obrazac je string koji na određenim mjestima sadrži sekvence - formate koji će biti zamijenjeni u stringu podacima različitog tipa. Na primjer, poslije:

```
>>> Ime = input ("Tvoje ime? ")
Tvoje ime? Mirko
```

možemo napisati:

```
>>> God = int (input ("Koliko ti je "
                      "godina, %s? " % Ime))
Koliko ti je godina, Mirko? 35
```

Ovdje poruka

```
"Koliko ti je godina, %s? "
```

u unosu sadrži operator formata, **%**, koji se odnosi na string, **s**. Operator formata piše se iza stringa, a slijede ga argumenti, izrazi odgovarajućeg tipa čije će vrijednosti, redom kako su napisane, zamijeniti oznaku tipa u formatu, opet redom, slijeva nadesno.

U našem je primjeru samo jedan format, s oznakom tipa vrijednosti **s** – string, i naveden je samo jedan izraz (varijabla Ime) čija će vrijednost, string "Mirko", zamijeniti sekvencu **%s**. Ostali će znakovi stringa biti prepisani, pa je generiran string:

```
"Koliko ti je godina, Mirko? "
```

Sada možemo napisati:

```
>>> print ("%s ima %d godina!" % (Ime,
God))                Mirko ima 35 godina!
```

String "%s ima %d godina!" sadrži dva formata, **%s** i **%d**. **d** je oznaka cjelobrojne vrijednosti. Moraju se navesti dva argumenta. Napisani su između zagrada i odvojeni zarezom. Vrijednost prvog argumenta zamijenit će sekvencu **%s**, a drugog sekvencu **%d**, pa je rezultirajući string:

```
"Mirko ima 35 godina!"
```

Obrazac se piše prema pravilu:

obrazac : " *format* { *format* } " %
(*izraz* | (*izraz* {, *izraz*}))
format : % *simbol duljina tip*
simbol : [-+0#]?
duljina : [*m* | *m.n* | *.n*]?
m : [1-9] [0-9]*
n : [0-9]+
tip : (c| [diu]| [eEfFgG]| [oxX]| s)

SEMANTIKA

Ako format napišemo općenito kao ***%sdt***, gdje su: ***s*** - simbol, ***d*** - duljina (širina, broj znakova) prikaza i ***t*** – tip, značenje je dano u sljedećim tablicama:

Tip

t	*Značenje*	*>>> Primjeri*

d
i
u
	Generira cijeli broj. Argument može biti dekadski, binarni, oktalni ili heksadecimalni broj, s predznakom ili bez njega. Ako je argument realna vrijednost, prevodi se u cjelobrojnu (odbacuje mu se decimalni dio, funkcija `int()`)	`"%d" % 12345678  '1234567'`
		`"%i" % 12345678  '12345678'`
		`"%u" % 12345678  '12345678'`
		`"%d" % -12345678.999`
		`'-12345678'`
		`"%i" % 0xFFF  '4095'`
		`"%d" % 0b1111  '15'`
		`"%d" % 0017  '15'`
		`"%d" % 0xF  '15'`

o	Generira oktalni broj (bez prefiksa `0o`) za cjelobrojni ili realni argument.	`"%o" % 12345678  '57060516'`
		`"%o" % 0b1111  '17'`
		`"%o" % 99.999  '143'`

x	Generira heksadecimalni broj (bez prefiksa `0x`), s malim slovima heksadecimalnih znamenki, za cjelobrojni ili realni argument.	`"%x" % 999999  'f423f'`
		`"%x" % 9**16`
		`'6954fe21e3e81'`

X	Generira heksadecimalni broj (bez prefiksa `0X`), s velikim slovima heksadecimalnih znamenki, za cjelobrojni ili realni argument.	`"%X" % 999999  'F423F'`
		`"%X" % 9**16`
		`'6954FE21E3E81'`

e	Generira eksponencijalni prikaz realnog broja u formatu	`"%e" % 123  '1.230000e+02'`
		`"%e" % 1234  '1.234000e+03'`
		`"%e" % 123456`
		`'1.234560e+05'`

t	*Značenje*	*>>> Primjeri*

	[1-9].*bbbbbb*e[+-]*bb*
	gdje je *b* brojka [0-9].
		`"%e" % 1234567`
		`'1.234567e+06'`
		`"%e" % 12345678`
		`'1.234568e+07'`
		`"%e" % 987654321`
		`'9.876543e+08'`

E	Generira eksponencijalni prikaz realnog broja u formatu
	[1-9].*bbbbbb*E[+-]*bb*
	gdje je *b* brojka [0-9].
		`"%E" % 123  '1.230000E+02'`
		`"%E" % 1234  '1.234000E+03'`
		`"%E" % 12345  '1.234500E+04'`
		`"%E" %12345678`
		`'1.234568E+07'`
		`"%E" % 987654321`
		`'9.876543E+08'`

f F	Generira realni broj sa 6 decimalnih mjesta.	`"%f" % 100  '100.000000'` `"%f" % 99.99  '99.990000'`

c	Generira znak. Ako je argument cijeli broj, generira znak kojem je taj broj Unicod.	`"%c" % 9  '\t'`
		`"%c" % 10  '\n'`
		`"%c" % 65  'A'`
		`"%c" % 960  'π'`
		`"%c" % 'a'  'a'`

s	String. Ako argument nije string, konvertira ga u string.	`"%s" % 'Python'  'Python'`
		`"%s" % 12345  '12345'`
		`"%s" % 12.55  '12.55'`

Duljina

d	*Značenje*	*>>> Primjeri*

m	Cjelobrojna vrijednost ili string bit će generirani u polju duljine (širine) ***m***, s vodećim razmacima, ako joj je duljina manja od ***m*** . Ako je duljina podatka veća od ***m***, bit će generirana u polju duljine jednake duljini podatka. Ako je tip podatka jednak **f**, bit će generiran decimalni dio duljine 6.	`"%10d" % 123  '  123'`
		`"%10f" % 5  '  5.000000'`
		`"%10s" % 'Python'`
		`'  Python'`
		`"%10x" % 999999`
		`'  f423f'`
		`"%10x" % 9**16`
		`'6954fe21e3e81'`

m.n	Realna će vrijednost biti generirana u polju duljine (širine) ***m***, u koje je uključena decimalna točka i ***n*** znakova decimalnog dijela, s vodećim razmacima, ako nije	`"%12.4f" % 2**0.5`
		`'  1.4142'`
		`"%12.4f" % 300**0.5`
		`'  17.3205'`
		`"%12.2s" % 'abcdef'`
		`'  ab'`
		`"%12.3s" % 'abcdef'`
		`'  abc'`

d	*Značenje*	*>>> Primjeri*

	došlo do preteka zadane duljine ***m***. Ako je tip podatka cjelobrojan (**d**, **i** ili **u**), bit će generiran podatak duljine ***m***, a ako je string, bit će generirano prvih ***n*** znakova u polju duljine ***m***.	`"%12.4s" % 'abcdef'`
		`'  abcd'`
		`"%12.5s" % 'abcdef'`
		`'  abcde'`
		`"%12.6s" % 'abcdef'`
		`'  abcdef'`
		`"%12.12s"% 'abcdef'`
		`'  abcdef'`
		`"%12.2i" % 5555`
		`'  5555'`

.n	Vrijednost će biti generirana s ***n*** decimalnih znamenki.	`"%.4f" % 2**0.5  '1.4142'`
		`"%.4f" % 300**0.5'17.3205'`
		`"%.2i" % 555  '555'`
		`"%.2e" % -100  '-1.00e+02'`

Simbol

s	*Značenje*	*>>> Primjeri* 12345678901234

#	Generiranje prefiksa `'0O'`, za oktalnu vrijednost, `'0x'` ili `'0X'`, za heksadecimalnu vrijednost.	`"%#o" % 100  '0o144'`
		`"%#x" % 100  '0x64'`
		`"%#x" % 10000000 '0x989680'`
		`"%#x" % 999999999`
		`'0x3b9ac9ff'`
		`"%#X" % 999999999`
		`'0X3B9AC9FF'`

0	Prefiks generiranog brojčanog podatka bit će popunjen nulama.	`"%010d" % 123.99`
		`'0000000123'`
		`"%010.2f" % 678`
		`'0000678.00'`
		`"%010s" %'abc' '  abc'`

-	Generirana vrijednost pozicionirana je od početka polja, s razmacima kao sufiksom.	`"%-10d" % 123  '123  '`
		`"%-10.2f" % 100`
		`'100.00  '`
		`"%10s"  % 'Python'`
		`'Python  '`

+	Predznak ('+' ili '-') prethodit će generiranoj brojčanoj vrijednosti.	`"%+d"  % 123  '+123'`
		`"%+10d" % 123  '  +123'`
		`"%+d"  % -123  '-123'`
		`"%+10.2f" %100 '  +100.00'`

Zadatak 2.1

Ako je brzina vjetra m/s koliko je to km/h?

Brzina vjetra

```
>>> m_s  = eval (input ('Zadajte '
           'brzinu vjetra m/s ')); \
km_h = m_s *0.001 /(1/3600);       \
m_km = "%0.2f m/s = %0.2f km/h";   \
print (m_km % (m_s, km_h))
Zadajte brzinu vjetra m/s 10
10.00 m/s = 36.00 km/h
```

Ovdje znakovna varijabla `m_km` sadrži dva formata, mjesta označena s %: Prvi će biti zamijenjen s `m_s`, a drugi s `km_h`.

Sada se vratimo ispisu tablice iz vježbe **>>>2.1** i napišimo program **Tablica.py**, dan u nastavku.

Tablica.py

```
form   = "%2d  %4d %8.4f";
Naslov = " i  i**2   i**0.5 "
print (); print (Naslov);
print ('-' *len(Naslov))
i = 0; n = 10
red = ("i = i+1; print ( form % (i, "
       " i**2, round (i**0.5, 4))); ")
exec (red *n)
```

S obzirom na to da su naredbe programa izvršne, proceduru **exec**() rabit ćemo samo za ponavljanje izvršavanja niza naredbi sadržanih u varijabli `red`. Format ispisa redova tablice sadržan je u znakovnoj varijabli `form`. Simbol naznačuje da je izvršen program, ispisan njegov „listing“, i prešlo se u interaktivni mod.

```
>>>
 i  i**2   i**0.5
------------------
 1     1   1.0000
 2     4   1.4142
 3     9   1.7321
 4    16   2.0000
 5    25   2.2361
 6    36   2.4495
 7    49   2.6458
 8    64   2.8284
 9    81   3.0000
10   100   3.1623
```

Moduli brojčanih funkcija

Osim standardnih funkcija postoje i funkcije koje se nalaze u posebnim „bibliotekama” ili modulima. Ovdje ćemo opisati takva dva standardna modula: **math** i **random**.

MODUL math

Ovaj modul omogućuje pristup matematičkim funkcijama definiranim po C standardu (kao u jeziku C). U sljedećoj vježbi uvezimo modul `math` i pogledajmo njegov sadržaj.

››› 2.1 math

```
>>> import math;  dir (math)
[..., 'acos', 'acosh', 'asin', 'asinh',
'atan', 'atan2', 'atanh', 'ceil',
'copysign', 'cos', 'cosh', 'degrees', 'e',
'erf', 'erfc', 'exp', 'expm1', 'fabs',
'factorial', 'floor', 'fmod', 'frexp',
'fsum', 'gamma', 'hypot', 'isinf',
'isnan', 'ldexp', 'lgamma', 'log',
'log10', 'log1p', 'modf', 'pi', 'pow',
'radians', 'sin', 'sinh', 'sqrt', 'tan',
'tanh', 'trunc']
```

Ovdje, već po nazivu, prepoznajemo neke od trigonometrijskih, hiperbolnih, eksponencijalnih i logaritamskih funkcija. Tu su i funkcije za izračunavanje drugog korijena, potenciranje i faktorijel. Prepoznajemo i konstante `e` i `pi`. *Naredbom HELP*,

```
help ( ime_modula )
```

ispisuje se opis svih funkcija i podataka navedenog modula.

››› 2.2 help()

```
>>> help (math)
Help on built-in module math:
NAME
    math
...
    sqrt(...)
        sqrt(x)
        Return the square root of x.
...
DATA
    e = 2.718281828459045
    pi = 3.141592653589793
```

Pozivanje neke funkcije modula `math` („metode" u terminologiji OOP) ili konstante („atributa" - podatka) je sa:

```
math . ime_funkcije (...)
```

```
>>> math.pi              3.141592653589793
>>> math.sin(1)        0.8414709848078965
```

Da modul `math` nije bio uvezen, bila bi dojavljena pogreška:

```
NameError: name 'math' is not defined
```

Izbor funkcija i podataka iz math

U sljedećoj smo tablici dali pregled izabranih funkcija i podataka modula `math` koje ćemo koristiti u ovoj knjizi.

Ime	*Argument*	*Tip*	*Opis i primjer poziva*
ceil	*x*	*r*	Najmanji cijeli broj veći ili jednak *x* `ceil (3.99)` 4.0 `ceil (-3.99)` -3.0
cos	*x*	*r*	Kosinus od *x* `cos (0)` 1.0 `cos (pi/4)` 0.70710678118655
e	-	*r*	Konstanta `e, e=2.718281828459045` `e` 2.718281828459045
degrees	*x*	*r*	Konverzija radijana u stupnjeve `degrees (1)` 57.29577951308232 `degress (pi/4)` 45.0
exp	x	*r*	e^x, `e = 2.718281828459045` `exp(e)` 15.15426
factorial	*b*	*c*	Faktorijel od *b*, *b*≥0 `factorial (0)` 1 `factorial (15)` 1307674368000
floor	x	*r*	Najveći cijeli broj manji ili jednak *x* `floor (3.99)` 3.0 `floor (-3.99)` -4.0
log	x [, b] *x*>0 *b*>0	*r*	Prirodni logaritam (po bazi `e` Logaritam po bazi b (`log(x)/log(b)`) `log (e)` 1.0 `log (10)` 2.302585092994 `log (10, 3)` 2.095903274289 `log (10, 10)` 1.0
log10	x *x*>0	*r*	Logaritam po bazi 10, `= log(x,10)` `log10 (10)` 1.0 `log10 (e)` 0.434294481903 `log10 (0)` math domain error
pi	-	*r*	Konstanta π, `pi = 3.1415926535898` `pi` 3.1415926535898
radians	*x*	*r*	Konverzija stupnjeva u radijane `radians (30)` 0.523598775598299 `radians (90)` 1.570796326794897
sin	*x*	*r*	Sinus od *x* `sin (0)` 0.0 `sin (pi/4)` 0.70710678118655
sqrt	*x*	*r*	Drugi korijen iz *x*, *x*≥0 `sqrt (2)` 1.4142135623731 `sqrt (121)` 11.0 `sqrt (-1)` math domain error
tan	*x*	*r*	Tangens od *x* `tan (pi/4)` 0.9999999999999999

r - realni tip; *c* - cjelobrojni tip; *x*, *y* - brojčani izraz (realni ili cjelobrojni); *b* – cijeli broj

MODUL random

Ovaj modul sadrži generatore pseudoslučajnih brojeva različitih distribucija.

2.3 random

```
>>> import random;  dir (random)
..., '_acos', '_ceil', '_cos', '_e',
'_exp', '_hashlib', '_hexlify',
'_inst', '_log', '_pi', '_random',
'_sin', '_sqrt', ..., 'randint',
'random', ...
```

Prepoznajemo neke funkcije i konstante iz modula `math`, s prefiksom '_'.

```
>>> random._pi          3.141592653589793
>>> random._sqrt (2) 1.4142135623730951
```

Za naše je primjene zasad dovoljno opisati dvije funkcije: `random()` i `randint(a,b)`. Funkcija **`random`**`()` generira realni pseudoslučajni broj uniformne distribucije na intervalu `[0,1>`, a **`randint`**`(a,b)` cijeli pseudoslučajni broj uniformne distribucije na intervalu $[a,b]$, $a \leq b$.

2.4 Funkcije random() i randint()

```
>>> from random import random, randint
>>> random()          0.6953876353157467
>>> randint (0, 1)    0
>>> random()          0.1401259322213750
>>> randint (0, 1)    1
>>> random()          0.2567628508606699
>>> randint (1, 39)   26
>>> randint(100,999) 136
```

Primjeri generiranih slučajnih brojeva u ovoj vježbi razlikovat će se od brojeva koji će biti generirani pozivajući `random()` i `randint()` s istim parametrima na vašem računalu.

PROMJENA IMENA FUNKCIJA I PROCEDURA

Dopuštena je promjena imena funkcija. Zapravo, to nije „promjena“ imena nego definiranje vlastitog imena („nadimka“) nekoj funkciji ili metodi. Izvorno ime i dalje ostaje poznato.

2.5 Promjena imena funkcije

```
>>> from random import random as rnd, \
                                randint
>>> rnd()             0.1908133624454189
>>> random()
NameError: name 'random' is not defined
>>> Irnd = randint
>>> randint(1,5), Irnd (1, 5)    (4, 1)
>>> import math; f = math.factorial
>>> f (30)
265252859812191058636308480000000
```

Ime `random` je pri uvozu modula preimenovano u `rnd`, a imenu `randint` je pridruženo ime („nadimak“) `Irnd` s jednakim značenjem. Ime `randint` i dalje je poznato. U drugom smo primjeru imenu `math.factorial` pridružili ime `f`.

Dopuštena je promjena imena standardnih funkcija i procedura. Zapravo, to nije „promjena“ imena nego definiranje vlastitog imena („nadimka“) nekoj funkciji ili metodi, jer izvorno ime i dalje ostaje poznato.

2.6 Promjena imena standardne funkcije

```
>>> abs          <built-in function abs>
>>> a = abs; a(-55), abs(-55)  (55, 55)
>>> type (a)
<class 'builtin_function_or_method'>
>>> type (abs)
<class 'builtin_function_or_method'>
```

Otpočetka smo funkciju **`print`**`()` nazvali „naredba“, a da to od inačice 3.0 Pythona više nije. U prethodnim je inačicama to bila naredba, rezervirana riječ iza koje se, bez zagrada, pisao niz izraza. Sada bi je pravilnije bilo nazvati „standardnom funkcijom“. Ako napišemo:

```
>>> print     <built-in function print>
```

vidimo da je **`print`** ugrađena funkcija, pa je dopušteno imenovati je. Isto vrijedi i za **`exec`**`()`.

2.7 Promjena imena print i exec

```
>>> Ispiši = print
>>> Ispiši (2*5)                    10
>>> print  (2*5)                    10
>>> exec       <built-in function exec>
>>> Izvrši = exec
>>> Izvrši ("a = 10; b = 20; "
            "print (a*b)")          200
>>> exec   ("a = 10; b = 20; "
            "print (a*b)")          200
```

Varijable (2)

U prvom smo poglavlju uveli pojam varijable s „tradicionalnim značenjem“. Ako je ***v = i*** jednostavno pridruživanje, gdje je ***v*** ime, a ***i*** izraz, značenje smo prikazali s ***v ← vrijednost (i).*** Ili, riječima, imenu ***v*** bit će pridružena vrijednost izraza ***i*** i njegov tip. Takvo značenje vrijedilo u inačicama `2.`*x* i `2.`*x*`.`*y*.

Od inačice 3.*x* značenje varijabli potpuno se razlikuje. Znak "=" u naredbi za jednostavno pridruživanje jest operator pridruživanja. Ako jednostavnu naredbu za pridruživanje napišemo kao

i **=** ***b***

gdje je ***i*** *varijabla* (*ime*), a ***b*** *izraz*, semantika jednostavne naredbe za pridruživanje može se prikazati sa

Id ← **id** (*vrijednost* (***b***))

Dakle, prvo je izračunata vrijednost izraza čime je generiran (instanciran) objekt odgovarajuće klase (tipa) i pridružena mu je identifikacija ***Id***. Standardna funkcija `id()` vraća tu adresu. Potom je ta identifikacija, ***Id***, pridružena imenu ***i***, odnosno, ime varijable pokazuje na memorijsku lokaciju ***Id***, na objekt s identifikacijom ***Id***. Na primjer:

```
>>> a = 30
>>> id (30)                    1352036272
>>> id (a)                     1352036272
>>> type (30)                <class 'int'>
>>> type (a)                 <class 'int'>
```

Vrijednosti identifikatora na vašem računalu ovisna je o zauzeću radne memorije i neće biti jednaka kao u danim primjerima. □

Vidimo da je identifikator objekta, cjelobrojne vrijednosti 30, jednak identifikatoru objekta, varijabli s imenom a. Opet ćemo iz „tradicionalnih" razloga reći da je imenu a pridružena vrijednost 30 cjelobrojnog tipa.

```
>>> b = 30; id (b)             1352036272
```

vidimo da i ime b pokazuje na isti objekt.

REFERIRANJE NA OBJEKT

Poseban je slučaj pisanja naredbe za jednostavno pridruživanje ako izraz sadrži samo jednu varijablu. Na primjer, ako napišemo:

```
>>> c = a
>>> c; id (a); id (c); type (c)
30
1352036272
1352036272
<class 'int'>
```

Drugi primjer pridruživanja koji najčešće zbunjuje početnike jeste pridruživanje koje u izrazu sadrži ime varijable kojoj se vrijednost izraza pridružuje. Na primjer, ako je varijabli i bila pridružena vrijednost 10,

```
>>> i = 10; id (i)
>>> print (id (i), i) 1673686320 10
```

što će se dogoditi poslije izvršenja naredbe i=i+1? Pogledajmo:

```
>>> i = i +1; id (i)
>>> print (id (i), i) 1673686352 11
```

Ovo je ujedno i objašnjenje. Varijabla i u izrazu bit će zamijenjena svojom „starom" vrijednošću, objektom na koji referira. Poslije evaluiranja izraza i+1 dobivena vrijednost (objekt) imat će novu identifikaciju na koju će referirati ime i. Tradicionalno, reći ćemo da je „nova" vrijednost varijable i jednaka 11.

Naredbe za pridruživanje

Varijabla se definira naredbom za pridruživanje. Postoje četiri načina, četiri naredbe za pridruživanje vrijednosti:

pridruživanje : *jednostavno_pr* |
višestruko_pr | *konkurentno_pr* |
operatorsko_pr

Opisali smo jednostavno pridruživanje. U nastavku dajemo opis preostala tri načina pridruživanja.

VIŠESTRUKO PRIDRUŽIVANJE

Višestruko pridruživanje dobiva se proširenjem pravila pisanja jednostavnog pridruživanja:

višestruko_pr : *ime* { = *ime* } = *izraz*

SEMANTIKA

Ako naredbu za višestruko pridruživanje napišemo kao

$v_1 = v_2 = \ldots = v_n = i$

gdje su v_1 do v_n imena varijabli, a ***i*** izraz, semantika se naredbe za višestruko pridruživanje može prikazati sa:

Id ← **id** (*vrijednost* (***i***))
$v_1 \rightarrow Id \quad v_2 \rightarrow Id \ \ldots \ v_n \rightarrow Id$

Prvo je izračunata vrijednost izraza čime je generiran (instanciran) objekt odgovarajuće klase (tipa) i pridružena mu je identifikacija ***Id***. Standardna funkcija

id() vraća tu adresu. Potom je ta identifikacija, ***Id***, pridružena redom imenima v_1 do v_n, odnosno, ta imena pokazuju na memorijsku lokaciju ***Id***, na objekt s identifikacijom ***Id***. Na primjer:

››› 2.8 Višestruko pridruživanje

```
>>> x0 = y0 = z0 = 1
>>> print (x0, y0, z0)                1 1 1
```

KONKURENTNO PRIDRUŽIVANJE

Pravilo pisanja konkurentnog pridruživanja je:

konkurentno_pr : *ime, unutarnje_pr,izraz*
unutarnje_pr : *ime = izraz |*
konkurentno_pr

Koristili smo rekurzivnu definiciju koja nam kaže da broj imena odvojenih zarezom mora biti jednak broju izraza napisanih poslije znaka pridruživanja.

SEMANTIKA

Ako naredbu za konkurentno pridruživanje napišemo kao

$v_1, v_2, \ldots v_n = i_1, i_2, \ldots i_n$

gdje su v_1 do v_n imena varijabli, a i_1 do i_n izrazi, semantika se ove naredbe može prikazati sa:

$Id_1 \leftarrow$ id(*vrijednost* (i_1)); $Id_2 \leftarrow$ id(*vrijednost* (i_2)); ... ;
$Id_n \leftarrow$ id(*vrijednost* (i_n))

$v_1 \rightarrow Id_1 \quad v_2 \rightarrow Id_2 \quad \ldots v_n \rightarrow Id_n$

Prvo se izračunavaju izrazi, od i_1 do i_n. Njihove se vrijednosti instanciraju kao objekti odgovarajuće klase (tipa) i pridružuje im se identifikacije, od Id_1 do Id_n. Standardna funkcija id() vraća te adrese. Potom se te identifikacije pridružuju redom imenima v_1 do v_n, odnosno, ta imena pokazuju na objekte s memorijskim lokacijama od Id_1 do Id_n.

››› 2.9 Konkurentno pridruživanje

```
>>> x0, y0, z0 = 1, 2, 3
>>> x, y, z = x0, y0, z0
>>> print (x, y, z)                   1 2 3
```

Ako broj izraza nije jednak broju varijabli dojavljuje se pogreška:

```
>>> p, q, r = 1, 2
ValueError: need more than 2 values...
>>> p, q, r = 1, 2, 3, 4
ValueError: too many values to unpack
```

Početnici često simbol dodjeljivanja, „=", poistovjećuju s izjednačivanjem vrijednosti, pa naredbu x=y čitaju „x je jednako y". Zato bi, na primjer, na upit kolika je vrijednost varijable y poslije izvršenja niza naredbi:

```
>>> x = 100; y = x; x = 200
```

možda odgovorili 200! Ako se značenje naredbe za pridruživanje odmah pravilno shvati, da je varijabli pridružena vrijednost izraza, tj. prvo se izračunava izraz pa se dobivena vrijednost pridruži varijabli, ne bi bilo problema. U našem primjeru prvom naredbom za dodjeljivanje varijabli x bila bi pridružena vrijednost 100, potom bi vrijednost izraza u drugoj naredbi, a to je tekuća vrijednost varijable x, bila pridružena varijabli y (preciznije, ime y bi referiralo na objekt 100) i na kraju bi vrijednost 200 bila pridružena varijabli x. Sada je jasno da to nije imalo utjecaja na sadržaj varijable y, pa je njezina vrijednost ostala nepromijenjena.

Konkurentno se pridruživanje može iskoristiti za jednostavnu razmjenu vrijednosti para varijabli. U nekim se jezicima za to mora rabiti pomoćna varijabla.

››› 2.10 Razmjena vrijednosti

```
>>> a, b = eval (input('a? ')), \
           eval (input ('b? '))
a? 12
b? 66
>>> print (a, b)                      12 66
>>> a, b = b, a;  print (a, b)        66 12
```

Konkurentno pridruživanje funkcijom input()

U prethodnoj smo vježbi za pridruživanje vrijednosti varijablama a i b rabili funkcije input(), za svaku varijablu posebno. Poseban je slučaj konkurentnog pridruživanja ako se koristi samo jedna funkcija input(). Tada pri unosu mora biti jednak broj izraza, odvojenih zarezom, koliko je navedeno varijabli na lijevoj strani. Ako se unose brojčane vrijednosti, obvezno ih moramo prevesti funkcijom eval(). Ako ulazni niz sadrži znakovne vrijednosti (stringove), moraju biti napisani između navodnika.

››› 2.11 Funkcija input()

```
>>> a, b, c = eval (input ('Zadaj '
                  'stranice trokuta '))
Zadaj stranice trokuta 10, 12, 15
>>> a, b, c                    (10, 12, 15)
>>> x = 2;  X, Y, Z = eval (input
              ('Unesi tri vrijednosti '))
Unesi tri vrijednosti round (x**0.5,
                        2), x**2, x**3
```

```
>>> X, Y, Z                    (1.41, 4, 8)
>>> Ime, Prezime, God = eval (input (
"Unesi ime, prezime i godinu rođenja"))
Unesi ime, prezime i godinu rođenja
"Georges", "Moustaki", 1934
>>> print (Ime, Prezime, God)
Georges Moustaki 1934
```

Funkcija divmod()

Standardna funkcija divmod(*a*,*b*) izračunava *a*//*b* i *a*%*b* i vraća rezultat izračunavanja kao par vrijednosti:

a // b, a % b

Na primjer:

```
>>> x, y = divmod (12, 7)
>>> x   1      >>> y   5
```

OPERATORSKO PRIDRUŽIVANJE

Sintaksu i semantiku jednostavnog pridruživanje opisali smo u prethodnom poglavlju. Slično kao u nekim drugim jezicima, u Pythonu možemo pisati naredbu za pridruživanje koja će sadržavati operaciju, a izvršit će se nad poznatom varijablom, kao prvi operand, koristeći vrijednost izraza kao drugi operand.

operatorsko_pr : *varijabla operator_pr izraz*
operator_pr : += | -= | *= | /= | //= | %= | **=

Vidimo da su operatori simboli dobiveni pisanjem brojčanih operacija ispred znaka =. Na primjer, pravilno je napisano:

```
i += 1   x -= 1.5   a *= b**2   Q %= 5
C0 /= 2.45
```

SEMANTIKA

Ako naredbu za operatorsko pridruživanje napišemo kao ***v bo= i*** gdje je ***v*** varijabla (mora biti prethodno definirana), ***bo*** brojčana operacija i ***i*** izraz, semantika se može prikazati s

v ← *vrijednost*(***v bo*** *vrijednost*(***i***))

ili riječima, izračunat će se izraz ***i***, pa ***v bo i*** i potom će dobivena vrijednost biti pridružena varijabli ***v.*** Iz svega toga zaključujemo da je operatorsko pridruživanje:

v bo= i

semantički ekvivalentno jednostavnom pridruživanju:

v = v bo i

iz čega slijedi uvjet da varijabla ***v*** mora biti prethodno definirana. Drugi uvjet koji mora biti ispunjen jest da tip izraza i tip varijable moraju biti usklađeni.

››› 2.12 Operatorsko pridruživanje

```
>>> a = 1; b = 2; c = 3; d = 4
>>> a, b, c, d                  (1, 2, 3, 4)
>>> a += 10; b -= 2 #a = a+10; b = b-2
>>> c *= 10; d /= 2 #c = c*10; d = d/2
>>> a, b, c, d                (11, 0, 30, 2)
>>> a **= 3; a       #a = a**3       1331
>>> x = 10; x *= 2 +10; x             120
>>> # x = x*(2 +10), a ne x = x*2+10!
```

STANDARDNE ZNAKOVNE FUNKCIJE

Osim znakovnih funkcija str() i eval() definiranih u prvom poglavlju, u sljedećoj su tablici dane znakovne funkcije koje broj, kao rezultat izračunavanja brojčanog izraza, prevode u string.

Funkcija	*Opis*	*Primjeri* ›››
bin(*c*)	konverzija cijelog broja u binarni kao string	bin(16384) '0b100000000000000' bin(1001) '0b1111101001' bin(0b1010101 + 0b111) '0b1011100'
oct(*c*)	konverzija cijelog broja u oktalni kao string	oct(16384) '0o40000' oct(1001) '0o1751'
hex(*c*)	konverzija cijelog broja u heksadecimalni kao string	hex(16384) '0x4000' hex(1001) '0x3e9'

c - cjelobrojni izraz

››› 2.13 bin(), oct() i hex()

```
>>> n = 123456789; x = 0b111; y = 0b1000
>>> bin (n); oct (n); hex (n)
'0b111010110111100110100010101'
'0o726746425'
'0x75bcd15'
>>> bin (x**2 +y**2)          '0b1110001'
```

LAMBDA funkcija

Osim standardnih funkcija i funkcija sadržanih u pojedinim modulima, moguće je definirati vlastitu funkciju. U većini jezika za programiranje to je funkcijski potprogram. Ima ga i Python, o čemu će biti riječi u posebnom poglavlju. Ovdje ćemo opisati tzv. *LAMBDA funkciju* koju ćemo često koristiti u našim programima. Pravilo pisanja je:

```
lambda_fun : ime_lambda_fun = lambda
lambda : lambda [parametar {,
         parametar }] : izraz {, izraz }
parametar : ime [= izraz ]
```

Poziv će lambda funkcije biti iz izraza, prema pravilu:

```
poziv_fun   : ime_lambda_fun (
              [argument {, argument} ] )
argument    : [ime = ]izraz
```

Ime parametra u definiciji lambda funkcije je formalno (nije poznato izvan definicije funkcije) i može biti jednako imenu funkcije. Bit će zamijenjeno stvarnom vrijednošću argumenta pri pozivu.

2.14 LAMBDA funkcije

```
>>> # drugi korijen iz x
>>> Sqrt = lambda x : x **0.5 # Poziv:
>>> Sqrt(10)          3.1622776601683795
>>> x
... NameError: name 'x' is not defined
>>> # kvadratna funkcija:
>>> f = lambda a,b,c,x : a*x**2 +b*x -c
>>> f(1, 2, 3,  -3)                    0
>>> y = lambda x, y : x –y
>>> y (10,5) # x=10; y=5               5
>>> y (y=10, x=5)
>>> # Poziv s pridruženim vrijednostima
>>> # argumentima. Nije bitan redoslijed!
>>> y (y=12, 1)
... SyntaxError: non-keyword arg after
keyword arg
>>> y (5, y=3) # x=5                   2
>>> y (1, y(10,5))
>>> # x=1; argument je izraz!          -4
>>> Y = lambda x=10, y=20 : x +y
>>> # Parametri imaju inicijalne
>>> # vrijednosti
>>> Y() # x=10; y=20                   30
>>> Y(30, 12) # x=30; y=12             42
>>> Y(,2)
SyntaxError: invalid syntax
>>> Y(55)  # x=55; y=20                75
>>> a, b = 3, 12.5
>>> d = eval (input ('izraz '))
izraz a**2 +b**2
>>> d                                 165.25
>>> Input = lambda s : eval (input (s))
>>> a, b, c = Input (
              'Zadaj stranice trokuta ')

Zadaj stranice trokuta 10,11,12
>>>a, b, c                  (10, 11, 12)
```

GOVORIMO PYTHONSKI

Dosad smo naučili brojčane tipova podataka i gotovo sve primitivne naredbe, tako da možemo pisati nizove naredbi za rješavanje jednostavnih problema danih u ovom dijelu. Najčešće će nedostatak takvih rješenja biti što neće uvijek „raditi”, tj. zasad ne znamo način kako ispitati domenu ulaznih podataka, pa će se dogoditi da će u tim slučajevima biti dojavljena pogreška.

Naredbu za dodjeljivanje koristit ćemo najčešće kad treba zapamtiti rezultat izračunavanja nekog podizraza koji se pojavljuje na dva ili više mjesta, ili za reduciranje složenijih izraza.

Funkcija `input()` za unos podataka koristit će se za pridruživanje vrijednosti varijablama iz njihove domene definiranosti, koje su u trenutku pisanja programa nebitne i „nepoznanice". Takve su, na primjer, vrijednosti polumjera kruga i visine u programu za izračunavanje oplošja i volumena valjka, jačina struje i veličina napona u programu za izračunavanje snage električne žarulje itd. Izvršenje funkcije `input()` prekida rad čekajući na utipkavanje ulaznih vrijedosti. Da bi se znalo što treba upisati, obavezno koristite komentar u funkciji `input()`.

RAZMJENA VRIJEDNOSTI DVIJU VARIJABLI

Nekoć se umijećem programiranja smatralo ako ste u nekom od jezika za programiranje mogli napisati niz naredbi za razmjenu vrijednosti dviju varijabli bez uporabe treće, pomoćne varijable. Evo rješenja:

```
>>> a, b = 10, 20; a, b          (10, 20)
>>> a += b; b = a -b; a = a -b
>>> a, b                         (20, 10)
```

Ograničenje primjene takvog rješenja je što vrijedi za podatke istog tipa. Konkurentno pridruživanje u Pythonu sigurno je bolje rješenje i vrijedi za različite tipove podataka:

```
>>> a,b = 10,'drugi'; print (a, b)
10 drugi
>>> a,b = b,a; print (a, b)     drugi 10
```

NUMERIKA

Vidjeli smo da cijeli brojevi u Pythonu mogu biti precizno prikazani s velikim brojem znamenki. Međutim, treba stalno imati na umu da su realne vrijednosti aproksimacija stvarnih vrijednosti. Autor se ove knjige prvi put susreo s problemima numerike na računalima kad je profesor pokazao da je 2+2 jednako 3.999999, a ne 4! Bilo je to prije više od 49 godina na jednom mini računalu, u jeziku BASIC. Nije bio problem u samom jeziku već u tadašnjim mogućnostima računala koje je imalo samo 8KB radne memorije!

No, jesu li se stvari promijenile? Izgleda da nisu. Na primjer, iz učitanog realnog broja T, u kojem cijeli dio predstavlja minute, a decimalni pomnožen sa 100 sekunde, želimo ekstrahirati minute i sekunde. Ako su M minute i S sekunde, formule su jednostavne:

```
M = int (T);  S = int (100 *(T –M))
```

Testirajmo:

```
>>> T = eval (input ('Upiši trajanje'
           ' događaja u obliku M.SS '))
Upiši trajanje događaja u obliku M.SS
2.16
>>> M = int (T);  S = int (100 *(T-M))
>>> M, S                          (2, 16)
>>> T = eval (input ('Upiši trajanje'
           ' događaja u obliku M.SS '))
Upiši trajanje događaja u obliku M.SS
1.16
>>> M = int (T);  S = int (100 *(T-M))
>>> M, S                         (1, 15)
```

U oba unosa smo imali jednak decimalni dio, 0.16. U prvom slučaju je konverzija u 16 sekundi korektna, a u drugom u 15 sekundi pogrešna?! Zašto? Gdje je pogreška? Provjerimo izračunavanje izraza

```
int (100 *(T –M))
```

po koracima:

```
>>> T-M              0.15999999999999992
>>> 100 *0.15999999999999992
15.999999999999993
>>> int (15.999999999999993)          15
```

Dakle, nije „kriva" funkcija int(), već argument. Da bismo to nadišli, umjesto int(*x*), gdje je *x* realna vrijednost (realni izraz), trebamo pisati

```
int (round (x))
```

OPERACIJE S VREMENIMA

Česte su operacije s vremenima. Na primjer, treba izračunati trajanje nekog događaja u satima i minutama ako su zadana vremena njegova početka i kraja. U tom se slučaju vremena zadaju s po dvije varijable. Jednoj se pridružuju sati, a drugoj minuti. Možda je bolje da se vremena prikazuju kao decimalni broj gdje cijeli dio decimalnog broja predstavlja sate, a decimalni minute. Dakako, decimalni dio mora biti u intervalu 0.00 do 0.59. Ako nam trebaju desetinke, stotinke ili tisućinke sekunde, dodat ćemo još jedno, dva ili tri decimalna mjesta. Na primjer, 11.5999 bi imalo značenje 11 sati, 59 minuta i 99 stotinki. Bez obzira kako se zadaju vrijednosti minuta, mnogi problemi koji se odnose na rad s vremenima najlakše će biti riješeni na jedan od dva načina:

1) Pretvorba operanada u minute (sekunde), izračunavanje i pretvorba rezultata (cijeli broj) u sate, minute i sekunde.
2) Pretvorba operanada u decimalne vrijednosti, izračunavanje i pretvorba rezultata (realni broj) u sate, minute i sekunde.

Slijedi primjer računanja trajanja nekog događaja:

```
>>> Događaj = """
T1, T2 = eval(input (
'Upiši vrijeme početka i završetka '
'događaja ss.mm, ss.mm '))
m      = 60
S1, M1 = int (T1), int (
         round (100 *(T1 % 1)))
S2, M2 = int (T2), int (
         round (100 *(T2 % 1)))
D      = abs (S1*m +M1 -(S2*m +M2))
print ('Događaj je trajao %d sati'
' i %d minuta' % (divmod (D, m)))"""
>>> exec (Događaj)
Upiši vrijeme početka i završetka
događaja ss.mm, ss.mm 7.30, 16.16
Događaj je trajao 8 sati i 46 minuta
```

N-TI ČLAN FIBONACCIJEVOG NIZA (2)

Već smo *naredbu EXEC* rabili u rješavanju nekih problema kad imamo ponavljanje niza naredbi. Na primjer, ako želimo izračunati n-ti član Fibonaccijevog niza. Rješenje je u jednoj liniji koda! Pogledajmo:

```
>>> n = int ( eval ((input ('n-ti član'
' Fibonaccijevog niza, n = ')) ) ); \
    a = b = 1; exec ("a, b = b, a+b;"
                     *(n -2)); print (b)
n-ti član Fibonaccijevog niza, n = 100
354224848179261915075
```

Da bismo testirali rad programa za više ulaznih vrijednosti, možemo definirati tekst `Fib`:

```
>>> Fib = """
n = int ( eval ((input ('n-ti član '
'Fibonaccijevog niza, n = ')) ) )
a = b = 1; exec ("a, b = b, a+b;"
                 *(n -2)); print (b)
"""
>>> exec (Fib *3)
n-ti član Fibonaccijevog niza, n = 10
55
n-ti član Fibonaccijevog niza, n = 20
6765
n-ti član Fibonaccijevog niza, n = 30
832040
n-ti član Fibonaccijevog niza, n = 1000
43466557686937456435688527675040625802
54660517371780402481729089536555417949
05189040387984007925516929592259308032
26347752096896232398733224711616429964
09065331879382989696499285160037044761
37795166849228875
```

LAMBDA FUNKCIJE

LAMBDA funkcija se može pozvati eksplicitno. Na primjer:

```
>>> print ((lambda x : x**2)(10))   100
```

Tada se piše između zagrada. Argumenti su napisani u nastavku. Zasad pisanje takvih funkcija nema posebnog smisla. Bolje je imenovati funkciju.

VLASTITI MODUL

Često ćemo u našim programima imati funkcije, procedure i konstante iz standardnih modula. Osim toga, u gotovo svim programima učitavamo brojčane podatke ili nizove brojčanih podataka pa smo ih morali evaluirati funkcijom `eval()`:

```
eval (input ())
```

Definirajmo vlastitu (LAMBDA) funkciju `Input()` koja će sadržavati tu konverziju:

```
>>> Input = lambda s = '' : \
            eval (input (s))
>>> a = Input()
123
>>> a                                  123
>>> x, y = Input ('Koordinate? ')
Koordinate? 1, 3
>>> x, y                             (1, 3)
```

Sada je pravo vrijeme da definiramo vlastiti modul kojeg ćemo u svakom poglavlju proširivati novim funkcijama i procedurama. Ime modula je istodobno i ime datoteke. Mora zadovoljavati pravilo definiranja imena u Pythonu! Mi smo izabrali ime **`Moj_modul`**. Evo početnog sadržaja:

Moj_modul.py

```
# STANDARDNI MODULI
from math   import *
from random import *

# GRČKA SLOVA
# α β γ δ ε ζ η θ ι κ λ μ ν ξ π ρ σ τ φ
# χ ψ ω
# Γ Δ Θ Λ Ξ Π Σ Φ Ψ Ω

# KONSTANTE
NL  = '\n'; TAB = '\t'; π   = pi

# FUNKCIJE
Input = lambda s : eval (input (s))
```

Izvršimo ga (`F5`). Ako u nekom programu trebamo `Moj_modul`, uvest ćemo ga na početku *naredbom FROM*:

```
from Moj_modul import *
```

Ako u razvoju programa trebamo grčka slova, kopirat ćemo ih "ručno" iz modula.

P R O G R A M I

Odsad će svako poglavlje sadržavati odjeljak PROGRAMI u kojem će biti dani primjeri programskih rješenja u rješavanju mnogih problema matematike, fizike, kemije, obrade teksta, prevođenja itd. primje-

njujući dotad uvedene naredbe, tipove i strukture podataka. Neki će programi biti „usavršavani“ u narednim poglavljima, kad budu uvedene složene naredbe i složeni tipovi podataka.

UDALJENOST DVIJU TOČAKA U RAVNINI

Ako su (x_1, y_1) i (x_2, y_2) koordinate dviju točaka u ravnini, njihova se udaljenost može izračunati prema Pitagorinom poučku:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

pa se odmah može napisati program:

Udaljenost_dviju_točaka.py

```
from Moj_modul import *
d = lambda x1, y1, x2, y2: round (sqrt
    ((x2 -x1)**2 +(y2 -y1)**2), 2)
Ax, Ay = Input('Koordinate točke A: ')
Bx, By = Input('Koordinate točke B: ')
AB = d (Ax,Ay, Bx,By);print ("d =", AB)
```

```
>>>
Koordinate točke A: -3, -3
Koordinate točke B: 2, 1
d = 6.4
```

BINARNA ARITMETIKA

U sljedećem je programu pokazano kako se mogu definirati *LAMBDA funkcije* za zbrajanje, oduzimanje, množenje i cjelobrojno dijeljenje dvaju binarnih brojeva.

Binarna_aritmetika.py

```
# BINARNA_ARITMETIKA
ADD = lambda x, y : bin (x + y)
SUB = lambda x, y : bin (x - y)
MPY = lambda x, y : bin (x * y)
DIV = lambda x, y : bin (x // y)
x, y = eval (input ('Zadaj dva binarna'
           ' broja odvojena zarezom '))

print ('x =', bin(x), 'y =', bin(y));
print ('x + y  =', ADD(x,y))
print ('x - y  =', SUB(x,y))
print ('x * y  =', MPY(x,y))
print ('x // y =', DIV(x,y))
```

```
>>>
Zadaj dva binarna broja odvojena zarezom
0b11111111111111, 0b11111111
x = 0b11111111111111 y = 0b11111111
x + y  = 0b100000011111110
x - y  = 0b11111100000000
x * y  = 0b11111110111111100000001
x // y = 0b1000000
```

TABLICA

Evo još jednog primjera kako na sadašnjoj razini učenja Pythona možemo ispisati tablicu oktalnih i heksadecimalnih brojeva u zadanom intervalu.

Tablica_2.py

```
# OKTALNI I HEKSADECIMALNI BROJI
od, do = eval (input (
             'Ispis tablice od, do '))
print ()
print ('  n  oct(n)  hex(n)')
print ('-------------------')
i    = od
form = "%3d  %-6o  %-5x"
n    = ("print (form % "
        "(i, i, i)); i += 1; ")
exec (n *(do -od +1))
```

```
>>>
Ispis tablice od, do 110, 120

  n  oct(n)  hex(n)
-------------------
110  156     6e
111  157     6f
112  160     70
113  161     71
114  162     72
115  163     73
116  164     74
117  165     75
118  166     76
119  167     77
120  170     78
```

VRIJEDNOSTI FUNKCIJE NA INTERVALU [a, b]

Treba učitati funkciju $f(x)$, interval domene $[a, b]$ i broj točaka n na njemu za koje ćemo izračunati vrijednosti funkcije i prikazati u tablici. Upisana funkcija može sadržavati standardne funkcije, funkcije iz modula `math` i operacije. Neovisna varijabla će biti `x`. Na primjer:

`(25 -x**2)**0.5`	polukružnica definirana na intervalu `[-5, 5]`
`sin(x) +cos(x)`	trigonometrijske funkcije
`-x**2 +3*x -6`	kvadratna funkcija

Funkcija.py

```
# Definiranje funkcije fx i
# izračunavanje na intervalu [a, b]

from Moj_modul import Input
from math      import *
f    = lambda x : eval (fx)
fx   = input ('Upiši f(x) =  ')
a, b = Input ('Interval [a, b] ')
n    = Input (
       'Koliko vrijednosti na danom '
       'intervalu? ')
d    = round ((b -a) /n, 2)
red  = ("print ('%5.2f %8.4f'"
        " % (x, f(x))); x += d;  ")
x    = a; exec (red *n)
x    = b; exec (red)
```

```
>>>
Upiši f(x) =  x**2 +2*x +3
Interval [a, b] -4, 4
Koliko vrijednosti na danom intervalu?
10
-4.00  11.0000
-3.20   6.8400
-2.40   3.9600
-1.60   2.3600
-0.80   2.0400
-0.00   3.0000
 0.80   5.2400
 1.60   8.7600
 2.40  13.5600
 3.20  19.6400
 4.00  27.0000
```

Upisali smo kvadratnu funkciju (parabolu). Oni koji znaju derivacije lako će zaključiti da je minimum za `x = -1` iz čega slijedi da dana kvadratna funkcija nema nul-točaka.

```
>>> f(-1)    2
```

POVRŠINA TROKUTA

Ako trebamo izračunati površinu trokuta stranica *a*, *b* i *c* primjenom Heronove formule:

$$P = \sqrt{s(s-a)(s-b)(s-c)} \quad s = \frac{a+b+c}{2}$$

Još nemamo dovoljno znanja Pythona da bismo to riješili jer ne možemo provjeriti je li definiran trokut sa zadanim stranicama. Na primjer,

```
>>> a, b, c = 1, 3, 5
>>> s = (a +b +c) /2
>>> from math import sqrt
>>> P = sqrt (s *(s-a) *(s-b) *(s -c))
ValueError: math domain error
```

jer je vrijednost izraza iz koje se računa drugi korijen negativna:

```
>>> s *(s-a) *(s-b) *(s -c)    -11.8125
```

Ali, ako zadamo koordinate vrhova trokuta, `A`, `B` i `C`, problem je rješiv. Stranice a, b i c dobit ćemo izračunavanjem udaljenosti između vrhova, `B` i `C` za `a`, `A` i `C` za `b` i `A` i `B` za `c`. Evo programa:

Površina_trokuta.py

```
# Izračunavanje površine trokuta sa
# zadanim koordinatama vrhova A, B i C
from Moj_modul import *
sqr = lambda x : x**2
d   = lambda x1,y1, x2, y2 : round (
    sqrt (sqr(x1 -x2) +sqr(y1-y2)), 2)
Ax, Ay = Input ('Koordinate vrha A: ')
Bx, By = Input ('Koordinate vrha B: ')
Cx, Cy = Input ('Koordinate vrha C: ')
a = d (Bx,By, Cx,Cy)
b = d (Ax,Ay, Cx,Cy)
c = d (Ax,Ay, Bx,By)
s = (a +b +c) /2
P = sqrt (s *(s-a) *(s-b) *(s-c))
print ("P =", P)
```

```
>>>
Koordinate vrha A: 0, 0
Koordinate vrha B: 0, 3
Koordinate vrha C: 4, 0
P = 6.0
```

RASTUĆI NIZ BROJEVA

Zadane brojeve treba ispisati u rastućem nizu. Ako su zadana dva broja:

```
>>> a, b = eval (input (
                  'Zadaj dva broja '))
Zadaj dva broja 55, 11
>>> a, b                          (55, 11)
>>> a, b = min (a, b), max (a, b)
>>> a, b                          (11, 55)
```

problem je jednostavan. Ali, za tri broja problem se, na ovoj razini znanja Pythona, čini nerješiv. Ipak, koristeći konkuretno pridruživanje, postoji rješenje.

Rastući_niz_brojeva.py

```
# Uređenje triju brojčanih vrijednosti
u rastućem nizu
from random import randint as rnd
od, do  = 100, 999
a, b, c = rnd (od, do), rnd (od, do), \
          rnd (od, do)
print ('Ulazne  vrijednosti: ', a,b,c)
a, b = min (a, b), max (a, b)
a, c = min (a, c), max (a, c)
b, c = min (b, c), max (b, c)
print ('Izlazne vrijednosti: ', a,b,c)
```

```
>>>
Ulazne  vrijednosti:  929 373 264
Izlazne vrijednosti:  264 373 929
```

Zbroj znamenki prirodnog broja

Treba zbrojiti znamenke (brojke) velikog prirodnog broja. Možda će oni koji su programirali u nekim drugim jezicima pomisliti da je to nemoguće riješiti s onim što smo dosad naučili. Jer, nismo još uveli logički tip podataka niti neke složene naredbe u kojima bi to bilo jednostavno riješiti. A rješenje je ipak moguće! Ako je n zadani broj, dijelit ćemo ga s 10 i zbrajat ćemo ostatke:

```
S = 0
n, b = divmod (n, 10); S += b
```

Postupak treba ponoviti nad novom vrijednošću broja n sve dok je n>0. Ali kako znati kad će n biti jednako 0 da bismo prekinuti postupak? Jednostavno, postupak treba ponoviti k puta gdje k duljina broja n.

Zboj.py

```
# Zbroj znamenki prirodnog broja
N     = input (
'Zadaj veliki prirodni broj ')
n     = int (N);  k = len (N); S = 0
Zbroj = ("n, b = divmod (n, 10); " +
         "S += b; ")
exec  (Zbroj *k)
print ('Broj', N, 'ima', k,
       'znamenki. Zbroj znamenki =', S)
```

```
>>>
Zadaj veliki prirodni broj
999998888888812313123576599 9
Broj 999998888888812313123576599 9 ima
28 znamenki. Zbroj znamenki = 175
```

Plaćanje računa s najmanjih brojem apoena

Dani iznos računa C u kunama bez lipa treba platiti s najmanjim brojem apoena. Apoeni su 1000, 500, 200, 100, 50, 20, 10, 5, 2 i 1. Postupak ćemo započeti s iznosom c = C dijeleći ga s najvećim apoenom, 1000, i pamteći rezultat dijeljenja u varijabli koja će u svom imenu imati prefix '_'. Ostatak dijeljenja bit će nova vrijednost varijable c. Postupak treba ponoviti pamteći rezultate dijeljenja u varijablama koje će imati ime apoena s prefiksom '_' sve do dijeljenja s 2. Tada je ostatak dijeljenja _1. Broj novčanica za pojedine apoene bit će prikazan u formatu "%4d"*10. Novčanice kojih nije bilo u iznosu računa imat će 0 u ispisu.

Plaćanje_računa.py

```
# PLAĆANJE RAČUNA S NAJMANJIH BROJEM
# APOENA
d = divmod; C = c = int (eval (input (
    'Iznos računa, u kunama ')))
_1000,c = d (c,1000)
_500, c = d (c, 500)
_200, c = d (c, 200)
_100, c = d (c, 100)
_50,  c = d (c, 50); _20, c = d (c, 20)
_10,  c = d (c, 10); _5,  c = d (c,  5)
_2,   c = d (c,  2); _1     = c
X  = ("%4d"*10 %
(_1000,_500,_200,_100,_50,_20,_10,
 _5,_2,_1))
kn = ("%4d"*10 % (1000, 500, 200,
      100, 50, 20, 10, 5, 2, 1))
print ("Iznos %d kn platiti sa:" % C)
print (X);  print ("   x"*10)
print (kn)
```

```
>>>
Iznos računa, u kunama 125987
Iznos 125987 kn platiti sa:
 125   1   2   0   1   1   1   1   1   0
   x   x   x   x   x   x   x   x   x   x
1000 500 200 100  50  20  10   5   2   1
```

Kosi hitac

Napišimo program koji će izračunavati parametre kosoga hica. Iz fizike je poznat problem kosog hica.

Materijalna točka „izbacuje" se nekom početnom brzinom v_0 i kutom α_0. Treba odrediti parametre kosog hica: vrijeme leta T, domet D i maksimalnu visinu leta H. To je dano sljedećim formulama:

$$V_y = V_0 \sin\alpha_0 \quad V_x = V_0 \cos\alpha_0 \quad T = \frac{2V_y}{g} \quad D = TV_x$$

$$H = \frac{V_y^2}{2g}$$

gdje je $g=9.81$ ubrzanje Zemljine teže, V_x je horizontalna, a V_y vertikalna komponenta početne brzine. Evo rješenja:

Kosi_hitac.py

```
# Proračun parametara kosog hica
from Moj_modul import *
g = 9.81;   'ubrzanje zemljine teže'

""" varijable:
V, Vx, Vy  - početna brzina i
             projekcije brzine na
             os x i y
α          - početni kut [stup.]
Tm, D, H   - vrijeme leta, domet i
visina leta, [m]
R          - početni kut α u radijanima
"""
V, α   = Input ('Upišite početnu '
         'brzinu, m/s i kut u st. ')
R      = radians (α)
Vx, Vy = V *cos(R), V *sin(R)
Tm     = 2 *Vy /g
D      = Tm *Vx
H      = Vy**2 /(2*g)

Prikaz = """
Tmax  = %10.2f sec
Domet = %10.2f m
Hmax  = %10.2f m
"""
print (Prikaz % (round (Tm,2),
       round (D,2), round (H,2)))
```

```
>>>
Upišite početnu brzinu, m/s i kut u st.
250, 45

Tmax  =      36.04 sec
Domet =    6371.05 m
Hmax  =    1592.76 m
```

Nalazimo se u interaktivnom modu. Radna memorija sadrži sva imena varijabli i njihove vrijednosti. Na primjer:

```
>>> V, α    # početna brzina i kut
(250, 45)
>>> Vx, Vy  # komponte početne brzine
(176.7766952966369, 176.7766952966369)
```

Ako bismo sada htjeli ponoviti izvršavanje programa, morali bismo prijeći u programski môd (kliknuti na prozor programa). Preporučujemo da ekran podijelite na dva dijela i time lakše prelazite iz jednog u drugi môd, posebno u fazi pisanja i testiranja programa.

3.

KOMPLEKSNI I LOGIČKI TIPOVI PODATAKA

Vi koji ste tehničkog i prirodnog usmjerenja, matematičari, fizičari, strojari ili elektrotehničari, obavezno proučite ovo poglavlje jer je jedna od posebnih karakteristika Pythona da ima kompleksni tip podataka, a to je moćan alat za rješavanje mnogih problema iz navedenih područja.

Poslije uvodnih razmatranja, u prvom smo dijelu poglavlja definirali imaginarne brojeve kao dio kompleksnih brojeva. Proširili smo značenje brojčanih izraza, varijabli i funkcija. U dijelu PROGRAMI pokazali smo kako se kompleksni brojevi mogu rabiti u rješavanju nekoliko problema iz matematike i fizike.

Još je preostalo da opišemo logički tip podataka, kao posljednji primitivni tip, te logičke varijable i izraze. Opisali smo složene izraze, nazvali smo ih "uvjetni izrazi", u kojima se na temelju postavljenih uvjeta može izabrati izraz koji će biti evaluiran.

Može se slobodno reći da je ovladavanje logičkim tipom podataka, logičkim i uvjetnim izrazima, posebno važno za programiranje u Pythonu, jer se logički izrazi pojavljuju u složenim naredbama. Osim logičkog tipa, Python podržava i bitovne Booleove operacije na cijelim brojevima.

pH.py

```
PH = """
pH =  eval( input(
'Unesite pH vrijednost (od 0 do 14) '))
Ok =  0 <= pH <= 14
print( 'IZVAN DOMENE!' *(not Ok) )
exec( PH *(not Ok) ) """
exec (PH)
k, a = ' kiselo', ' alkalno'
print( 'ultra'    +k if pH <   3.5 else
       'izuzetno' +k if pH <=  4.4 else
       'vrlo'     +k if pH <=  5.0 else
       'jako'     +k if pH <=  5.5 else
       'umjereno' +k if pH <=  6.0 else
       'blago'    +k if pH <=  6.5 else
       'NEUTRALNO'   if pH <=  7.3 else
       'blago'    +a if pH <=  7.8 else
       'umjereno' +a if pH <=  8.4 else
       'jako'     +a if pH <=  9.0 else
       'vrlo'     +a )
```

```
>>>
Unesite pH vrijednost (od 0 do 14) 14.5
IZVAN DOMENE!
Unesite pH vrijednost (od 0 do 14) 5.5

jako kiselo
```

```
x       sin(x)  -1                  0                   1
                ----------------------------------------
0.0000  0.0000                      *
0.3142  0.3090                      |     *
0.6283  0.5878                      |           *
0.9425  0.8090                      |                 *
1.2566  0.9511                      |                    *
1.5708  1.0000                      |                     *
1.8850  0.9511                      |                    *
2.1991  0.8090                      |                 *
2.5133  0.5878                      |           *
2.8274  0.3090                      |     *
3.1416  0.0000                      *
3.4558 -0.3090                  *   |
3.7699 -0.5878              *       |
4.0841 -0.8090          *           |
4.3982 -0.9511       *              |
4.7124 -1.0000     *                |
5.0265 -0.9511       *              |
5.3407 -0.8090          *           |
5.6549 -0.5878              *       |
5.9690 -0.3090                  *   |
6.2832 -0.0000                      *
```

Kompleksni tip

Pretpostavimo da trebamo riješiti sljedeći zadatak

Zadatak 3.1

Izračunati nule kvadratne funkcije $f(x) = ax^2 + bx + c$, $a \neq 0$.

Rješenja se dobivaju iz formule

$$x_{1,2} = \frac{-b \pm \sqrt{D}}{2a}$$

gdje je $D = b^2 - 4ac$ i naziva se diskriminanta. Napišimo program:

Nul_točke_1.py

```
from Moj_modul import *
a,b,c = Input( 'Zadaj koef. kv. jed. ' )
D     = b**2 -4*a*c;  _2a   = 2*a
a1    = -b/_2a;  a2 = sqrt (D)/_2a
x1    = a1 +a2;  x2 = a1 -a2
print( "x1 =", x1, "x2 =", x2 )
```

```
>>>
Zadaj koef. kv. jed. 1, -2, -3
x1 = 3.0 x2 = -1.0
>>>
Zadaj koef. kv. jed. 1, 2, -3
x1 = 1.0 x2 = -3.0
>>>
Zadaj koef. kv. jed. 1, 2, 3
    a1  = -b/_2a;  a2 = sqrt (D)/_2a
ValueError: math domain error
```

Ako je $D \geq 0$, nule kvadratne funkcije su realne. U suprotnom su konjugirano kompleksne. Provjerimo D u posljednjem primjeru:

```
>>> D            -8
```

pa je dojavljena pogreška jer domena funkcije `sqrt (D)` mora biti veća ili jednaka nuli. No, pogledajmo što će se dogoditi ako umjesto

```
a2 = sqrt (D)/_2a
```

napišemo

```
a2 = D**0.5/_2a
```

jer je potencija `0.5` drugi korijen.

Nul_točke_2.py

```
from Moj_modul import *
a,b,c = Input( 'Zadaj koef. kv. jed. ' )
D     = b**2 -4*a*c;  _2a = 2*a
a1  = -b/_2a;  a2 = D**0.5/_2a
x1  = a1 +a2;  x2 = a1 -a2
print( "x1 =", x1, "x2 =", x2 )
```

Ponovimo posljednji unos:

```
>>>
Zadaj koef. kv. jed. 1, 2, 3
x1 = (-
0.99999999999999+1.4142135623730951j) x2
= (-1-1.4142135623730951j)
```

Dobili smo dvije konjugirano kompleksne vrijednosti! `1.4142135623730951j` je imaginarni broj. Ima sufiks `j`. Napomenimo da je u inačicama Python `2.x.y` potenciranje negativne vrijednosti s `0.5` bilo nedopušteno.

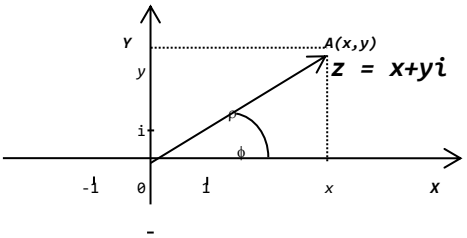
Prije nego što se upustimo u detaljan opis kompleksnih brojeva u Pythonu, prisjetimo se njihove definicije u matematici.

KOMPLEKSNI BROJEVI U MATEMATICI

Poznato nam je iz matematike da osim cijelih i realnih brojeva, koji se grafički prikazuju na osi *X*, imamo i kompleksne brojeve, koji se grafički prikazuju kao točka u ravnini.

Kompleksni brojevi su izrazi oblika ***x+yi***, gdje su ***x*** i ***y*** realni brojevi, a **i** imaginarna jedinica (drugi korijen iz -1). U kompleksnom broju **z=*x*+*y*i** broj ***x*** naziva se realni dio, ***x=Re(z)***, a broj ***y*** je imaginarni dio, ***y=Im(z)***. Kompleksan broj čiji je realni dio jednak nuli naziva se imaginarni broj. Realni brojevi predstavljaju poseban slučaj kompleksnih brojeva (kad je imaginarni dio jednak nuli).

Iako se kompleksnim brojevima ne izražavaju količine, kao što je to slučaj s realnim brojevima, njihovo je uvođenje korisno u rješavanju problema sastavljenih u terminima realnih brojeva, na primjer, problema o prolazu struje kroz vodič, o profilu krila aviona itd. Ništa manje važna nije ni primjena kompleksnih brojeva na čisto matematičke probleme. Povijesno, kompleksni su brojevi uvedeni radi rješavanja kvadratne jednadžbe.



Često se kompleksni brojevi predstavljaju vektorima u kompleksnoj ravnini (slika). Geometrijski smisao

brojeva x, y, ρ i ϕ vidi se na crtežu. Duljina vektora ρ (udaljenost od ishodišta) je modul ili magnituda kompleksnog broja. Ako je **z=x+yi** kompleksni broj, modul od **z** predstavlja njegovu apsolutnu vrijednost i može se dobiti po Pitagorinom poučku:

$$\rho = |z| = \sqrt{x^2 + y^2}$$

Konjugirano kompleksni broj broja **z = x +yi** je broj **z = x -yi**. Za potenciranje imaginarne jedinice vrijedi:

$$i^n = i^{n \bmod 4} = i^k = \begin{cases} 1 & \text{za } k = 0 \\ i & \text{za } k = 1 \\ -1 & \text{za } k = 2 \\ -i & \text{za } k = 1 \end{cases}$$

Ako su $z_1=x_1+y_1i$ i $z_2=x_2+y_2i$ dva kompleksna broja, zbrajanje, množenje i dijeljenje kompleksnih brojeva definira se formulama:

$$\begin{aligned} z_1+z_2 &= (x_1+y_1i) + (x_2+y_2i) \\ &= (x_1+x_2) + (y_1+y_2)i \\ z_1-z_2 &= (x_1+y_1i)(x_2+y_2i) \\ &= (x_1x_2-y_1y_2) + (x_1y_2+x_2y_1)i \\ z_1/z_2 &= (x_1+y_1i)/(x_2+y_2i) \\ &= (x_1x_2+y_1y_2)/(x_2^2+y_2^2) \\ &\quad +(x_2y_1-x_1y_2)i/(x_2^2+y_2^2) \text{ za } z_2 \neq 0 \end{aligned}$$

IMAGINARNI BROJEVI

Python je jedan od rijetkih jezika za programiranje koji ima imaginarne brojeve. Pišu se prema pravilu:

```
imaginarni_broj  :
  ( cijeli_broj | realni_broj ) [jJ]
```

Imaginarna jedinica je `1j`. Piše se kao **`1j`** ili **`1J`**.

››› 3.1 Imaginarni brojevi

```
>>> 1j        1j      >>> 1J      1j
>>> 10.5j     10.5j   >>> 6.0J    6j
>>> 0.0J      0j
```

IMAGINARNI IZRAZI I VARIJABLE

Imaginarni se broj može pridružiti nekom imenu koje će poprimiti svojstvo „imaginarna varijabla”.

```
>>> i = 1j;  _5i = 5J
>>> i           1j
>>> _5i         5j
```

Imaginarni se broj može dobiti i kao rezultat izračunavanja imaginarnog izraza koji se piše prema pravilu:

```
imaginarni_izraz   : imaginarni_operand
                     { [+-]+ imaginarni_operand }

imaginarni_operand: { br_op * } imag
                    { op br_op } | imag ** potencija
  br_op  : broj | brojčana_varijabla |
           ( br_izraz )
  op     : * | /
  imag   : imaginarni_broj |
           imaginarna_varijabla
  potencija : [13579] | [1-9][0-9]*[13579]
```

Potencija su cijeli neparni brojevi koji generiraju `1j` ili `-1j`.

››› 3.2 Imaginarni izrazi

```
>>> 1j +2j +3j              6j
>>> 1j -5j                  -4j
>>> 2 *1j                   2j
>>> 1J *6.0                 6j
>>> a = 2.5**2; a *1j       6.25j
>>> 2*3 *1j *5*6            180j
>>> i = 1j; 2*i +3*i -10*i  -5j
>>> (2.5*6 +13.5*8) *1j     123j
```

KOMPLEKSNI BROJEVI

Ako napišemo:

```
>>> type (1j)    <class 'complex'>
```

vidimo da su imaginarni brojevi tipa `complex`, odnosno, pripadaju klasi `complex`. Kompleksni su brojevi (ili kompleksne vrijednosti) u Pythonu podaci koji se sastoje od dvije komponente:

```
kompleksni_broj   :
  ( realni_dio [+-] imaginarni_dio )
  realni_dio      : realni_broj |
                    cijeli_broj
  imaginarni_dio  : imaginarni_broj
```

Prikazuje se u zagradi, prvo realni, potom imaginarni dio.

››› 3.3 Prikaz kompleksnog broja

```
>>> 0j +1   (1+0j)   >>>1j**0     (1+0j)
>>> 1j**1   1j       >>> 1+1j     (1+1j)
>>> 1j+1    (1+1j)   >>> (2j-2)   (-2+2j)
>>> 1j**2   (-1+0j)  >>> 1j**3    (-0-1j)
>>> 1j**4   (1+0j)   >>> 1j**5        1j
>>> 1j**6   (-1+0j)  >>> 1j**7    (-0-1j)
```

KOMPLEKSNE VARIJABLE

Naredba za jednostavno pridruživanje vrijednosti kompleksnoj varijabli piše se uobičajeno, samo na mjestu ***izraz*** stoji kompleksni izraz:

ime = kompleksni_izraz

Kompleksni broj je najjednostavniji kompleksni izraz. Njegova će vrijednost biti pridružena navedenom imenu. Ime će poprimiti svojstvo „kompleksna varijabla". Kompleksne se varijable mogu pojavljivati kao operandi u kompleksnim izrazima.

››› 3.4 Kompleksne varijable

```
>>> z1 = 1 +2j;  z1        (1+2j)
>>> z2 = 2 -5j;  z2        (2-5j)
```

››› 3.5 Izrazi s kompleksnim varijablama

```
>>> z1 = 1 +2j;  z2 = 2 +5j
>>> z1 +z2                      (3+7j)
>>> z1 -z2                      (-1-3j)
>>> z1 *z2                      (-8+9j)
>>> z1 /z2          (0.41379310344827586-
                     0.03448275862068965j)
>>> z1**2 +z2**2                (-24+24j)
>>> z2 **z1         (-.07680159841181108-
                     0.4921547083056663j)
>>> (1+z1) *(2+z2)              (-2+18j)
>>> z1 % 2
TypeError: can't mod complex numbers.
>>> z2 % 2                     5j
>>> divmod (z1, 2)             (0j, (1+2j))
>>> divmod (z2, 2)             ((1+0j), 5j)
>>> z1 /2                      (0.5+1j)
>>> z2 /2                       (1+2.5j)
>>> z1//2
TypeError: can't take floor of complex
number.
>>> A = 1 +0j;  B = 3 +4j
>>> A +B                      (4+4j)
>>> A -B                      (-2-4j)
>>> A *B                      (3+4j)
>>> A /B                      (0.12-0.16j)
```

REALNI I IMAGINARNI DIO KOMPLEKSNOG BROJA

Ako je ***z*** kompleksni izaz, sa ***z***`.real` i ***z***`.imag` mogu se dobiti realni i imaginarni dijelovi kompleksne vrijednosti dobivene evaluiranjem izraza ***z***.

››› 3.6 Realni i imaginarni dio kompleksnog broja

```
>>> 3*4 +5j.real          12.0
>>> 1+2+3.real
SyntaxError: invalid syntax
>>> (1+1j)*1j.real        0j
# = (1+1j)*0.0 = 0.0 +0j = 0j
>>> ((1+1j)*1j).real      -1.0
>>> A = 1 +0j;  B = 3 +4j
>>> A.real, A.imag        (1.0, 0.0)
>>> B.real, B.imag        (3.0, 4.0)
>>> A+B                   (4+4j)
>>> (A+B). real           4.0
>>> (A+B) .imag           4.0
>>> Re, Im = lambda z : \
              z.real, lambda z : z.imag
>>> c = 2 -1j
>>> Re(c), Im(c)          (2.0, -1.0)
```

FUNKCIJE NAD KOMPLEKSNIM BROJEVIMA

Python ima nekoliko desetaka standardnih ili „ugrađenih" (*built-in*) funkcija koje su uvijek dostupne. Njihov popis dan je Pythonovoj dokumentaciji, u standardnoj Pythonovoj biblioteci (*F1 → The Python Standard Library → 2. Built-in Functions*). Ovdje izdvajamo samo brojčane funkcije – funkcije čija je domena i kodomena cijeli ili realni broj.

Ime	*Argumenti*	*Tip*	*Opis*
abs	(x)	x	apsolutna vrijednost od x, $\|x\|$; tip je jednak tipu argumenta
pow	(x,y)	c,r	potenciranje, x^y

r - realni tip; c - cjelobrojni tip; x,y - brojčani tip (realni ili cjelobrojni)

KOMPLEKSNI IZRAZI

Osim eksplicitno napisanog kompleksnog broja, kompleksna se vrijednost može dobiti evaluiranjem kompleksnog izraza.

Kompleksni izraz je brojčani izraz koji sadrži najmanje jedan imaginarni operand. Izračunavanjem takvog izraza posebno će se zbrojiti realne vrijednosti, koje će činiti realni dio kompleksne vrijednosti, a posebno imaginarne, koje će činiti njegov imaginarni dio.

››› 3.7 Kompleksni izrazi

```
>>> 1.5 +2j*3 +(2 +3j)*3      (7.5+15j)
>>> 20 *1j**70                (-20+0j)
>>> type (20 *1j**70)  <type 'complex'>
>>> type ((2+2j)**2)  <class 'complex'>
>>> 0B111+5.5 -((1+2j)**2)    (15.5-4j)
>>> type (0B111 +5.5 -((1+2j)**2))
<class 'complex'>
```

Vidimo da je broj komplesnog tipa iako mu je imaginarni dio jednak `0`.

Operacije zbrajanja, oduzimanja, množenja i dijeljenja kompleksnih brojeva izvršavaju se kao što smo opisali u uvodnim razmatranjima.

››› 3.8 Operacije s kompleksnim brojevima

```
>>> (1 +2j) + (2 -5j)       (3-3j)
>>> (1 +2j) - (2 -5j)       (-1+7j)
>>> (1 +2j) * (2 -5j)       (12-1j)
>>> (1 +2j) / (2 -5j)
(-
0.27586206896551724+0.31034482758620
69j
)
```

KOMPLEKSNE FUNKCIJE

Kompleksni izrazi mogu sadržavati i kompleksne funkcije kao operande. Postoje samo dvije standardne kompleksne funkcije: `complex()` i `conjugate()`.

Funkcija `complex()`

Funkcija `complex()` piše se prema pravilu:

```
complex ( [(realni_dio [,
imaginarni_dio ] | string )] )
  realni_dio      : br_izraz
  imaginarni_dio  : br_izraz
```

Ako s ***r*** označimo realni dio, s ***i*** imaginarni i sa ***s*** string (koji se može interpretirati kao cijeli, realni ili kompleksni broj), postoje četiri slučaja sa značenjem:

```
complex ()       vraća 0j
complex (r)      vraća r +0j
complex (r, i)   vraća r + i*1j
complex (s)      vraća r + i*1j
```

››› 3.9 Funkcija complex()

```
>>> complex()                  0j
>>> complex (10)               (10+0j)
>>> complex ('123')            (123+0j)
>>> a = 2+2j
>>> complex (a) # = 2+2j +0j   (2+2j)
>>> complex (a, a)
>>> # = 2+2j +(2+2j)*1j        4j
>>> complex ('1 +1j')
ValueError: complex() arg is a
malformed string
>>> # ne smije biti razmak
>>> z = complex # preimenovanje
>>> z (1, 2)                   (1+2j)
>>> c = z(2, -3);  c           (2-3j)
>>> x = 10; y = -5
>>> Z = c (x, y);  Z           (10-5j)
>>> # ili, drugi način:
>>> Z = x +y*1j;   Z           (10-5j)
```

Funkcija `conjugate()`

Funkcija `conjugate()` piše se prema pravilu:

```
kompleksni_izraz . conjugate()
```

Vidimo da nema argumenata, već se prvo piše kompleksni izraz čija će vrijednost biti konjugirana.

››› 3.10 Funkcija conjugate()

```
>>> b = 2.55 +3.5j;  b      (2.55+3.5j)
>>> b. conjugate()          (2.55-3.5j)
>>> b # b je nepromijenjeno (2.55+3.5j)
>>> 2+3j. conjugate()           (2-3j)
>>> 1+2+3+5j-2j.conjugate()     (6+7j)
>>> (1+2+3+5j-2j).conjugate()   (6-3j)
```

MAGNITUDA (modul) KOMPLEKSNOG BROJA

Magnituda (modul) kompleksnog broja dobiva se pozivom funkcije `abs()` ako je argument kompleksni broj, odnosno, kompleksni izraz:

```
magnituda : abs ( kompleksni_izraz )
```

››› 3.11 Udaljenost točke od ishodišta

```
>>> Ax, Ay = eval( input( 'točka A '))
točka A 1, 2
>>> A = complex (Ax, Ay)
>>> round (abs (A), 2)          2.24
```

››› 3.12 Udaljenost između dviju točaka

```
>>> Ax,Ay = eval( input( 'točka A '));\
    Bx,By = eval( input( 'točka B '))
točka A 1, 2
točka B -1, -2
```

```
>>> A = complex (Ax, Ay); \
    B = complex (Bx, By)
>>> round (abs (A -B), 2)          4.47
```

Modul cmath

cmath je standardni modul s funkcijama za rad s kompleksnim brojevima. Sadrži trigonometrijske, logaritamske i eksponencijalnu funkciju s jednakim imenom kao u modulu math, ali su im argumenti i rezultati izračunavanja kompleksni brojevi.

```
>>> import cmath
>>> dir (cmath)
['__doc__', '__name__', '__package__',
'acos', 'acosh', 'asin', 'asinh',
'atan', 'atanh', 'cos', 'cosh', 'e',
'exp', 'isinf', 'isnan', 'log',
'log10', 'phase', 'pi', 'polar',
'rect', 'sin', 'sinh', 'sqrt', 'tan',
'tanh']
```

Za naše su primjene najvažnije ove tri navedene funkcije, a među njima posebno `cmath.sqrt()`, drugi korijen, koja „radi" i za negativni argument (znamo da takva funkcija, `math.sqrt()`, nije definirana za negativnu vrijednost argumenta).

```
>>> help(cmath)
...
FUNCTIONS
...
    polar(...)
        polar(z) -> r: float, phi:
float
        Convert a complex from
rectangular coordinates to polar
        coordinates. r is the distance
from 0 and phi the phase angle.
   rect(...)
        rect(r, phi) -> z: complex
        Convert from polar coordinates
to rectangular
        coordinates.
...
    sqrt(...)
        sqrt(x)
        Return the square root of x.
```

››› 3.13 Funkcija cmath.sqrt()

```
>>> import cmath, math
>>> csqrt = cmath.sqrt; sqrt = math.sqrt
>>> sqrt(-1)
ValueError: math domain error
>>> csqrt(-1)                      1j
>>> csqrt(-16)                     4j
>>> csqrt(16)                      (4+0j)
>>> from cmath import *
>>> sqrt(-12)      3.4641016151377544j
```

BROJČANI IZRAZI (2)

Prvo proširimo značenja nekih sintaksnih kategorija iz prvobitne definicije brojčanih izraza u prvom poglavlju:

br_vrijednost : *cijeli_broj* |
realni_broj | *kompleksni_broj*

Sada se i pravilo pisanja brojčane varijable kao operanda u brojčanim izrazima može proširiti na:

br_varijabla : *cjelobrojna_varijabla*|
realna_varijabla| *kompleksna_varijabla*

Tip brojčanog izraza (2)

Upamtimo da je tip izraza određen tipom njegove vrijednosti, odnosno klasom objekta dobivenog evaluiranjem (izračunavanjem) izraza. Na primjer, ako je „+" operacija zbrajanja, zbrajanje dvaju cijelih brojeva imat će za rezultat cijeli broj, a ako je jedan operand realni broj, rezultat će biti realni broj.

Posebno, ako je vrijednost takvog izraza cjelobrojna, tada je izraz „cjelobrojni" (tipa „int"), a ako je realna, izraz je "realni" (tipa „float"). To vrijedi ako brojčani izraz ne sadrži kompleksne brojeve niti kompleksne varijable. Ako ih sadrži, izraz je kompleksnog tipa.

Evaluiranje izraza koji sadrži kompleksne brojeve je da se posebno zbroje realne (cjelobrojne) vrijednosti i posebno imaginarne, koje imaju **j** ili **J** kao sufiks, slično kao u algebarskim izrazima. Ti će zbrojevi biti realni i imaginarni dio rezultirajućeg kompleksnog broja.

››› 3.14 Kompleksni izrazi

```
>>> 1 +2*3 +5j -2.0**3 +4**2 -1j
(15+4j)
>>> (1 +2*3 -2.0**3 +4**2) +(5j -1j)
(15+4j)
>>> 5 *complex (2, -2) -5*complex (4,-2)
(-10+0j)
>>> type (0j)          <class 'complex'>
>>> 12.0**3 +12.0**2 +12 +0J
(1884+0j)
```

Logički tip

Logički tip podataka sadrži samo dvije vrijednosti: `False` i `True`. Od inačice 3.0 Pythona to su rezervirane riječi:

log_vrijednost: `False` | `True`

Značenje logičkih vrijednosti je, kao i u svim drugim jezicima za programiranje, „neistina" (`False`) i "istina" (`True`).

```
>>> False     False     >>> True     True
```

Logički tip podataka jest klasa s imenom `bool`.

```
>>> type (False)      <class 'bool'>
>>> type (True)       <class 'bool'>
```

Nad logičkim vrijednostima definirana je unarna operacija negacija, `not`, i dvije binarne operacije, disjunkcija („ili"), `or` i konjunkcija („i"), `and`:

`not`	logička negacija
`and`	konjunkcija
`or`	disjunkcija (neisključujuća)

Značenje logičkih operacija isto je kao u matematici: logička negacija mijenja vrijednost istinitosti, disjunkcija vraća vrijednost `True` u svim slučajevima osim ako su oba operanda jednaka `False`, konjunkcija vraća `True` samo ako su oba operanda jednaka `True`.

››› 3.15 logičke operacije

```
>>> not False                  True
>>> not True                   False
>>> False or False             False
>>> False and False            False
>>> False or True              True
>>> False and True             False
>>> True  or False             True
>>> True  and False            False
>>> True  or True              True
>>> True  and True             True
```

FUNKCIJA `bool()`

Ime logičkog tipa, `bool`, istovremeno je i ime logičke funkcije, `bool()`, koja može biti napisana bez argumenta ili s argumentom, izrazom bilo kojeg tipa:

```
bool ( izraz )
```

Funkcija `bool()` vraća rezultat `False` ako je napisana bez argumenta ili ako je vrijednost izraza jednaka `0`, `0.0`, `''`, `0j` ili `False`. Inače, vraća `True`.

››› 3.16 Funkcija `bool()`

```
>>> bool ()                    False
>>> bool (True)                True
>>> bool (False)               False
>>> bool (-10)                 True
>>> bool (0)                   False
>>> bool (10)                  True
>>> bool (0.0)                 False
>>> bool (123456789)           True
>>> bool (0j)                  False
>>> bool (1-1j)                True
```

LOGIČKE VARIJABLE

Ako sintaksnu kategoriju ***izraz*** u pravilu pisanja jednostavnog pridruživanja

ime = *izraz*

proširimo s

izraz : *logički_izraz*

onda će navedeno ime poprimiti svojstvo "logička varijabla" i bit će joj pridružena vrijednost izračunavanja logičkog izraza, preciznije, ime će referirati na identifikator logičke vrijednosti. Najjednostavniji oblik logičkog izraza jest logička vrijednost. Na primjer:

```
>>> F = False; T = True
>>> print (F, T)              False True
>>> id (False)                1347154176
>>> id (True)                 1347154144
>>> a = False; id (a)         1347154176
>>> b = True; id (b)          1347154144
```

Logičke vrijednosti zapravo su logičke konstante pa imaju samo dvije identifikacije! □

LOGIČKI IZRAZI

Globalna sintaksa logičkog izraza definirana je sljedećim pravilima:

logički_izraz : *log_operand*
{ *log_operacija log_operand* } | `not` *logički_izraz* | (*logički_izraz*)
log_operand : `False` | `True` | *log_varijabla* | *log_funkcija* | *relacijski_izraz*
log_operacija : `or` | `and`

Relacijski izraz

Relacijski izrazi važna su podklasa logičkih izraza. Pišu se prema pravilu:

```
relacijski_izraz : izraz relacija izraz
                   { relacija izraz }
relacija         : <| <=| ==| !=| >=| >
```

SEMANTIKA

Značenje relacija jest sljedeće:

<	"manje od"	<=	"manje ili jednako"
=	"jednako"	!=	"različito"
>=	"veće ili jednako"	>	"veće"

Konteksni aspekt jest: mogu se uspoređivati vrijednosti istoga tipa (brojevi, logičke vrijednosti i stringovi) ili brojevi s logičkim vrijednostima. Za uspoređivanje dva broja lako zaključujemo u kojoj su relaciji. Ali, ako se uspoređuje broj i logička vrijednost, rezultat nije očigledan. Ako je ***b*** broj i ***r*** relacija, evo pravila kad će vrijednost binarnog relacijskog izraza biti jednaka `True`:

1) `False` ***r*** `True`
   samo za ***r*** jednako `<`, `<=` ili `!=`
2) Logičke vrijednosti koje se uspoređuju s brojčanim vrijednostima interpretiraju se `False` kao `0` i `True` kao `1`:

   b `>  True` za ***b*** `>  1`
   b `== True` za ***b*** `== 1` ili *b* `== 1.0`
   b `<  True` za ***b*** `<  1`

3) ***b*** `>  False` za ***b*** `>  0`
   b `== False` za ***b*** `== 0` ili ***b*** `== 0.0`
   b `<  False` za ***b*** `<  0`

4) ***b r b***

➥ *Mogu se uspoređivati samo podaci istog tipa (klase) osim kompleksnih vrijednosti.*

S uspoređivanjem dvaju stringova zasad se nemojte zamarati. Objasnit ćemo ih u šestom poglavlju.

››› 3.17 Jednostavni relacijski izrazi

```
>>> 1 < 2                       True
>>> 1 == 1.0                    True
>>> 100 > 10**2                 False
>>> 1+2+3+4+5 == 15             True
>>> a = 3; b = 3.0; a == b      True
>>> False <= True               True
>>> False != True               True
>>> 1.5 > True                  True
>>> 1 == True                   True
>>> True < 12                   True
>>> -1 < False                  True
>>> 0 =  False
SyntaxError: can't assign to literal
>>> 0 == False                  True
>>> False > -5                  True
>>> 5 < '6'
TypeError:  '>'  not  supported  between
instances of 'int' and 'str'
>>> True < '5'
TypeError:  '<'  not  supported  between
instances of 'bool' and 'str'
>>>
```

Karakteristika je sintakse relacijskih izraza u Pythonu da mogu sadržavati više od jedne relacije. Ako relaciju shvatimo kao operaciju, prioritet njezinog izvršenja niži je od svih aritmetičkih operacija. Zbog toga je suvišna uporaba zagrada ispred i iza relacije. Ako relacijski izraz napišemo kao

$$i_1\ r_1\ i_2\ r_2\ i_3\ \ldots\ i_n\ r_n\ i_{n+1}$$

gdje su i_j, j=1,...,n+1, izrazi, a r_j, j=1,...,n, relacije, značenje kompletnog relacijskog izraza jest:

1) Ako je n=1, vrijednost relacijskog izraza

   $$i_1\ r_1\ i_2$$

   jednaka je `True`, ako vrijedi relacija, inače je `False`.

2) Za n>1 prvo se računa binarna relacija

   $$i_j\ r_j\ i_{j+1}$$

   za j=1. Ako je rezultat **`False`**, to je ujedno i vrijednost ukupnog izraza. Inače, postupak izračunavanja se nastavlja za j=2, ..., n sve dok je vrijednost podizraza

   $$i_j\ r_j\ i_{j+1}$$

   jednaka **`True`**, što je ujedno i konačni rezultat izračunavanja, odnosno, postupak se prekida i vraća **`False`** kao rezultat izračunavanja.

››› 3.18 Složeni relacijski izrazi

```
>>> 1 < 2 < 3                   True
>>> 1 <  2  > 5                 False
>>> 10 < 20 < 30 <= 10          False
>>> x = 5; y = 10.5; z = 5
```

```
>>> 0 <= x <= y                    True
>>> 0 < x**2 <= y*z                True
>>> 0 == False == 0.0              True
>>> 1 < '2' > 5
TypeError: '<' not supported between
instances of 'int' and 'str'
```

Logičke operacije

U drugom obliku pisanja logičkih izraza pojavljuju se dvije sintaksne strukture, logički operand i logička operacija. Evo njihovih pravila pisanja:

```
log_operand      : log_vrijednost |
  log_varijabla | not log_operand |
  log_funkcija | ( logički_izraz )
log_operacija    : and | or
```

Sljedeća je vježba prikaz logičkih operacija na pregledniji način od onoga u vježbi **›››3.15.**

››› 3.19 Logičke operacije (2)

```
>>> F = False;  T = True
>>> # not
>>> not F          True
>>> not T          False
>>> # or
>>> F or F         False
>>> T or F         True
>>> F or T         True
>>> T or T         True
>>> # and
>>> F and F        False
>>> T and F        False
>>> F and T        False
>>> T and T        True
```

Iz sintakse logičkih izraza slijedi da oni općenito mogu sadržati aritmetičke operacije (kao dio brojčanih izraza u relacijskim izrazima), relacije, logičke operacije i zagrade. Konačna vrijednost bit će dobivena poslije izračunavanja niza podizraza pri čemu će se operacije izvršavati slijeva nadesno prema sljedećem prioritetu:

(1) izraz u zagradi
(2) brojčana funkcija (u brojčanom izrazu)
(3) predznak "-" (u brojčanom izrazu)
(4) *, /, //, %, +, - (u brojčanom izrazu)
(5) relacije: <, <=, ==, !=, >=, >
(6) not
(7) and
(8) or

Primijetiti da relacije imaju veći prioritet od logičkih operacija, pa se relacijski izrazi ne moraju pisati između zagrada, kao u nekim drugim jezicima za programiranje.

››› 3.20 Logički izrazi

```
>>> F = False;  T = True
>>> F or 2 > -2 and 10 < 9          False
>>> x = 6.5; 0 <= x and x <= 10     True
```

Sada možemo dati potpunu semantiku relacijskog izraza koji sadrži više od jedne relacije:

$i_1\ r_1\ i_2\ r_2\ i_3\ r_3\ i_4\ \ldots\ i_n\ r_n\ i_{n+1}$

Jednaka je izračunavanju logičkog izraza:

$i_1\ r_1\ i_2$ and $i_2\ r_2\ i_3$ and $i_3\ r_3\ i_4$ and ... and $i_n\ r_n\ i_{n+1}$

Poslije ovoga možemo se vratiti naredbi za pridruživanje logičkih vrijednosti i proširiti joj značenje pišući potpune logičke izraze na njezinoj desnoj strani. Značenje će biti: vrijednost logičkog izraza bit će pridružena imenu (logičkoj varijabli) navedenoj na lijevoj strani naredbe.

Zadatak 3.2

Je li 1900. godina bila prijestupna?

Većina će odgovoriti da jeste jer „znamo" da je godina prijestupna ako je djeljiva s 4, a 1900 % 4 jednako je nula! Međutim, prema Gregorijanskom kalendaru koji je uveden 1582. godine, od 1583. godine „prijestupna je svaka godina djeljiva s 4, a nije djeljiva sa 100, ili je djeljiva s 400".

›››3.21 Prijestupna godina

```
# Prijestupna.py
In = lambda   : eval (input (
     'Upiši godinu >= 1583 '))
ok = lambda g : g >= 1583
pg = lambda g : ok (g) and (
   g %400 == 0 or
   g %4   == 0 and g %100 != 0)
Prijestupna = """
G = In()
print (G, "je valjana godina?", ok(G))
print ("     prijestupna", pg(G)) """

>>> exec (Prijestupna *2)
Upiši godinu >= 1583 1900
1900 je valjana godina? True
     prijestupna False
Upiši godinu >= 1583 1952
1952 je valjana godina? True
     prijestupna True
```

Brojčane operacije s logičkim vrijednostima

U Pythonu je logički tip podataka implementiran kao podklasa cjelobrojnog tipa. To znači da se logičke vrijednosti (`False` i `True`), logičke varijable ili logički izrazi u zagradi, mogu pojaviti kao operandi u brojčanim izrazima, kao argumenti u brojčanim funkcijama ili kao multiplikatori stringa. Bit će konvertirani u `0` (`False`) ili `1` (`True`).

››› 3.22 Brojčane operacije s logičkim vrijednostima

```
>>> F, T = False, True; F +T, 1.55 *T
(1, 1.55)
>>> abs (-T), int (T), float (T)
(1, 1, 1.0)
>>> PG = True; PG * "prijestupna"
'prijestupna'
>>> PG = F; (not PG) * "nije prijestupna"
'nije prijestupna'
>>> import math as m; m.cos (False)   1.0
```

››› 3.23 Ispitivanje brojčane vrijednosti

```
# Brojčane_vr.py
B = """
x = float(input("Zadaj x: ")); V = x>0
J = x == 0; M = x < 0; print(x, "je",
"veće od 0" *V or "jednako 0" *J or
"manje od 0" *M )"""
>>> exec (B *3)
Zadaj x: -678      -678.0 je manje od 0
Zadaj x: 0         0.0 je jednako 0
Zadaj x: 9.99      9.99 je veće od 0
```

Zadatak 3.3

Usluga parkiranja u jednom danu naplaćuje se 15 kn po satu za prva tri sata, poslije toga još po 3 kn za svaki započeti sat. Izračunati cijenu usluge parkiranja ako se vrijeme dolaska i odlaska unese kao realni broj S.MM.

››› 3.24 Usluga parkiranja

```
>>> Parking = """
T1, T2 = eval (input(
'Vrijeme dolaska i odlaska u formatu '
'S.MM? '))
T  = int (round (abs (T2 -T1) +0.491))
kn = 15*T*(T <= 3) \
    +(15*3 +3*(T -3)) *(T > 3)
print (
"Platiti %d kn (%d sati parkiranja)"
 % (kn, T)) """
>>> exec (Parking)
Vrijeme dolaska i odlaska u formatu S.MM?
9.05, 14.06
Platiti 54 kn (6 sati parkiranja)
```

Bitovne Booleove operacije na cijelim brojevima

Osim logičkog tipa, Python podržava i bitovne Booleove operacije na cijelim brojevima. To znači da Python tretira svaki odgovarajući par bitova unutar dva prirodna broja (zdesna ulijevo) kao Booleove vrijednosti i primjenjuje odgovarajuću operaciju na njih. Ove operacije uključuju bitovni and (&), or (|), i isključujući „ili" (^), kao i operatore pomaka bitova za pomicanje bitnih uzoraka lijevo (<<) ili desno (>>). U sljedećoj je tablici dan pregled bitovnih operacija.

Operacija	Rezultat
`x | y`	bitovni *„ili"*
`x ^ y`	bitovni isključujući „ili"
`x & y`	bitovni „i"
`x << n`	*x* se pomiče n bitova u lijevo
`x >> n`	*x* se pomiče n bitova u desno

››› 3.25 Bitovne operacije

```
>>> x = 0; y = 0; x | y, x ^ y, x & y
(0, 0, 0)
>>> x = 0; y = 1; x | y, x ^ y, x & y
(1, 1, 0)
>>> x = 1; y = 0; x | y, x ^ y, x & y
(1, 1, 0)
>>> x = 1; y = 1; x | y, x ^ y, x & y
(1, 0, 1)
>>> form = "%10s"
>>> x = 10; y = 20; form % bin (x); \
    form % bin (y); form % bin (x | y)
'    0b1010'
'   0b10100'
'   0b11110'
>>> x = 10; y = 20; form % bin (x); \
    form % bin (y); form % bin (x ^ y)
'    0b1010'
'   0b10100'
'   0b11110'
>>> x = 10; y = 20; form % bin (x); \
    form % bin (y); form % bin (x & y)
'    0b1010'
'   0b10100'
'       0b0'
```

```
>>> form % bin (x); form % bin (
    x << 1); form % bin (x >> 1)
'    0b1010'
'   0b10100'
'     0b101'
```

Uvjetni izrazi

Karakteristika je Pythona da ima posebnu vrstu izraza – uvjetne izraze. To je složena sintaksna kategorija koja sadrži najmanje dva izraza koji će biti izvršeni ovisno o postavljenim uvjetima. Na primjer:

››› 3.26 Uvjetni izrazi

```
>>> # 1.
>>> x = eval ( input (
'Zadaj x u intervalu [-5, 5] ') )
Zadaj x u intervalu [-5, 5] -2
>>> ((25 -x **2)**0.5
     if abs(x) <= 5
     else "x izvan domene!" )
4.58257569495584
>>> # 2.
>>> x = eval ( input (
'Zadaj x u intervalu [-5, 5] ') )
Zadaj x u intervalu [-5, 5] 6
>>> ((25 -x **2)**0.5
     if abs(x) <= 5
     else "x izvan domene!" )
'x izvan domene!'
```

U prvom je primjeru, za x=-2, bio ispunjen uvjet abs(x) <= 5, pa je vraćen rezultat izračunavanja izraza (25 -x **2)**0.5. U drugom primjeru, za x=6, isti uvjet nije bio ispunjen, pa je vraćen rezultat iza riječi else.

Pravilo pisanja uvjetnih izraza je:

uvjetni_izraz : *izraz* if *uvjet*
else *izraz* { if *uvjet* else *izraz* }

uvjet : *izraz*

SEMANTIKA

Ako uvjetni izraz općenito napišemo kao

I_0 if U_0 else I_1 if U_1 else ... I_{n-1} if U_{n-1} else I_n

gdje su I_i izrazi, a U_i pridruženi uvjeti, redom se izračunavaju uvjeti bool(U_i), za $i=0,...,n-1$. Nailaskom na prvi istinit uvjet, i, generira (instancira) se objekt odgovarajuće klase (tipa) dobiven izračunavanjem izraza I_i. Ako nijedan uvjet nije istinit, generirat će se objekt dobiven izračunavanjem izraza I_n. Slijedi nekoliko primjera.

››› 3.27 Primjeri evaluiranja uvjetnih izraza

```
>>> x = -5; x**0.5 if x>=0 else 'x<0'
'x<0'
>>> x =  5; x**0.5 if x>=0 else 'x<0'
2.23606797749979
>>> ('x<0' if x<0 else 'x==0'
      if x==0 else 'x>0' )
'x>0'
>>> x = eval( input( 'x = ? ')); \
x**0.5 if x >= 0 else print (
                  'x je manji od 0' )
x = ? 12           3.4641016151377544
>>> x = eval( input( 'x = ? ')); \
x**0.5 if x >= 0 else print (
                  'x je manji od 0' )
x = ? -5          'x je manji od 0'
```

››› 3.28 Usporedba broja s nulom

```
>>> Usporedba = """
x = eval (input ('Zadaj neki broj '))
y = ('manje od 0' if x < 0 else
'jednako 0' if x == 0 else
'veće od 0' )
print (x, y) """
>>> exec (Usporedba *3)
Zadaj neki broj -10       -10 manje od 0
Zadaj neki broj 0.0        0.0 jednako 0
Zadaj neki broj 125.5    125.5 veće od 0
```

S obzirom na to da je značenje uvjetnih izraza izbor izraza koji će biti evaluiran, može se pisati na svim mjestima gdje se pojavljuje *izraz*, u pridruživanju, naredbi za ispis ili u definiciji *LAMBDA funkcije*.

››› 3.29 Usluga parkiranja (2)

```
>>> Parking2 = """
T1, T2 = eval (input ("Vrijeme dolaska i
odlaska u formatu S.MM? "))
T  = int (round (abs (T2 -T1) +0.491))
kn = 15 *T if T <= 3 else 15*3 +3*(T -3)
print ("Platiti %d kn "
"(%d sati parkiranja)" % (kn, T)) """
>>> exec (Parking2)
Vrijeme dolaska i odlaska u formatu S.MM?
9.05, 14.06
Platiti 54 kn (6 sati parkiranja)
```

Ova se inačica razlikuje od one dane u vježbi ›››3.26, u načinu izračuna ukupnog iznosa kn:

```
# >>> 3.26:
>>> kn = 15*T*(T <= 3) \
        +(15*3 +3*(T -3))*(T > 3)
```

```
# >>> 3.29:
>>> kn = 15*T if T<=3 else 15*3+3*(T -3)
```

GOVORIMO PYTHONSKI

Dosad smo naučili brojčane tipova podataka i gotovo sve primitivne naredbe, tako da možemo pisati nizove naredbi za rješavanje jednostavnih problema danih u ovom dijelu. Najčešće će nedostatak takvih rješenja biti što neće uvijek „raditi", tj. zasad ne znamo način kako ispitati domenu ulaznih podataka, pa će se dogoditi da će u tim slučajevima biti dojavljena pogreška. Realni i cijeli brojevi su također kompleksni brojevi. Na primjer:

```
>>> complex (10)                (10+0j)
>>> A = complex (1, 1);  A       (1+1j)
>>> C = 10; C.real, C.imag      (10, 0)
```

Kompleksni se brojevi mogu koristiti u rješavanju problema matematike, fizike, elektrotehnike itd.

NUL-TOČKE KVADRATNE JEDNADŽBE (2)

U sljedećem programu računamo nule kvadratne jednadžbe zadanih koeficijenata `a`, `b` i `c`, uz pretpostavku da je `a`≠`0`.

›››3.30 Kvadratna jednadžba

```
>>> KJ = """
a, b, c = eval ( input (
  'Zadaj koef. kv. jed. ') )
D = b**2 -4*a*c
f = lambda x : a*x**2 +b*x +c
Z = lambda z : complex(round (z.real, 4),
                       round (z.imag, 4))
_2a = 2*a; a1 = -b/_2a; a2 = D**0.5/_2a
x1  = Z (a1 +a2); x2 = Z (a1 -a2)
print ( "x1 =", x1, "x2 =", x2 ) """
>>> exec (KJ)
Zadaj koef. kv. jed. 1, 2, 3
x1 = (-1+1.4142j)
x2 = (-1-1.4142j)
>>> f(x1)
(-4.440892098500626e-16+0j)
>>> f(x2)
(-4.440892098500626e-16+0j)
>>> exec (KJ)
Zadaj koef. kv. jed. -1, 2, 3
x1 = (-1+0j) x2 = (3+0j)
>>> f(x1)     0j      >>> f(x2)      0j
```

EULEROVA FORMULA

Eulerova formula je formula u matematičkoj analizi kompleksne varijable. Definira odnos između trigonometrijskih funkcija i kompleksne eksponencijalne funkcije.

https://en.wikipedia.org/wiki/Euler%27s_formula

Ako je *x* realni broj, vrijedi

$$e^{ix} = cos(x) + isin(x)$$

Provjerimo formulu u Pythonu:

```
>>> from math import (sin, cos,
                      pi as π, e)
>>> x = 1
>>> e **(x*1j)
(0.54030230586813+0.841470984807899j)
>>> complex (cos(x), sin(x))
(0.54030230586814+0.841470984807897j)
>>> x = π
>>> e **(x*1j)
(-1+1.2246467991473532e-16j)
>>> complex (cos(x), sin(x))
(-1+1.2246467991473532e-16j)
```

Ako bismo pokušali izdvojiti što je najvažnije za primjene i daljnje učenje programiranja, onda bi to nesumnjivo bili logički i uvjetni izrazi.

IZBOR NAREDBI ZA IZVRŠAVANJE

Pretpostavimo da imamo *k* nizova naredbi sadržanih u *P_1*, *P_2*, ..., *P_k* tekstovima i da za zadani *i*, $1 \le i \le k$, treba izvršiti samo naredbe sadržane u tekstu *P_i*. Rješenje je jednostavno:

```
P = (i==1)*P_1 or (i==2)*P_2 or ... or
    (i==k)*P_k
exec (P)
```

Ako nijedan uvjet nije ispunjen, `P` će biti jednako `''`, pa neće biti izvršen nijedan *P_i*. Za ilustraciju izbora naredbi za izvršavanje i ponavljanja izvršavanja naredbi u nastavku dajemo četiri primjera izračunavanja i ispisa interesantnih izraza.

INTERESANTNI IZRAZI (2)

Na kraju smo prvog poglavlja prikazali kako se mogu ispisati četiri interesantna izraza. Sada dajemo program u kojem se slučajno bira jedan od četiri izraza koji su dani u tekstu, `I_1`, `I_2`, `I_3` i `I_4`, s pozivom pridruženim im inicijalnim vrijednostima, `P_1`, `P_2`, `P_3` i `P_4`.

Interesantni_izrazi.py

```
# Interesantni izrazi i rezultati
# njihova izračunavanja
from random import *

I_1 = """
B = B*10 +i; print (B, 'x 8 +', i,
 '=', B*8 +i); i += 1 """
P_1 = "B = 0; i = 1;  exec(9*I_1)"

I_2 = """
B = B*10 +i; print (B, 'x 9 +', i+1,
 '=', B*9 +i+1); i += 1 """
P_2 = "B = 0; i = 1;  exec(9*I_2)"

I_3 = """
B = B*10 +i; print (B, 'x 9 +', i-2,
 '=', B*9 +i-2); i -= 1 """
P_3 = "B = 0; i = 9;  exec(9*I_3)"

I_4 = """
B = B*10 +1; print (B, 'x', B,
 '=', B**2) """
P_4 = "B = 0;         exec(9*I_4)"

# izbor izraza za ispis
i = randint (1, 4); print ('i =', i)
P = P_1 *(i==1) or P_2 *(i==2) or P_3
*(i==3) or P_4 *(i==4)
exec ( P )
```

```
>>>
i = 2
1 x 9 + 2 = 11
12 x 9 + 3 = 111
123 x 9 + 4 = 1111
1234 x 9 + 5 = 11111
12345 x 9 + 6 = 111111
123456 x 9 + 7 = 1111111
1234567 x 9 + 8 = 11111111
12345678 x 9 + 9 = 111111111
123456789 x 9 + 10 = 1111111111
```

DE MORGANOVO PRAVILO

Nepoznavanje De Morganovog zakona čest je uzrok logičkih pogrešaka u programiranju. Na primjer, ako je neka funkcija `f(x)` definirana za $x \in [a,b]$, tj. `f(x)` je definirano za $a \le x$ i $x \le b$, a treba nam negacija tog uvjeta, početnici u programiranju bi „izveli“ izraz $x<a$ i $x>b$. S obzirom da vrijedi $a<b$, taj uvjet ne bi nikada bio ispunjen! Pravilno je $x<a$ ili $x>b$, a to je jedno od *De Morganovih pravila*.

Ako su ***X*** i ***Y*** logičke vrijednosti (izrazi), pravila negiranja konjunkcije i disjunkcije su sljedeća:

```
not (X and Y) → not (X) or  not (Y)
not (X or  Y) → not (X) and not (Y)
```

Ako je ***X*** ili ***Y*** relacijski izraz, `(x relacija y)`, za njihovu negaciju vrijede sljedeća pravila:

```
not (x <  y) → (x >= y)
not (x <= y) → (x >  y)
not (x == y) → (x != y)
not (x != y) → (x == y)
not (x >= y) → (x <  y)
not (x >  y) → (x <= y)
```

EVALUIRANJE LOGIČKIH IZRAZA

Evaluiranje logičkih izraza se ne izvršava do kraja izraza ako se na nekom mjestu jednoznačno utvrdi njegova vrijednost. Ako disjunkciju

```
x or y
```

prikažemo kao uvjetni izraz

```
x if x else y
```

rezultat će biti jednak ***x***, ako je `bool(x)` jednako `True` i prekida se daljnje evaluiranje bez obzira na vrijednost `bool(y)`. Ako konjunkciju

```
x and y
```

napišemo kao uvjetni izraz

```
x if not x else y
```

rezultat će biti jednak ***x***, ako je `bool(x)` jednako `False` i prekida se daljnje evaluiranje bez obzira na vrijednost `bool(y)`. Na primjer, ako želimo izračunati vrijednost funkcije `x**0.5`, a znamo da je definirana za `x >= 0`, možemo napisati *LAMBDA funkciju*:

```
>>> f1 = lambda x : x**0.5 if x >= 0 \
                    else "x < 0!"
```

```
>>> f1(2)              1.4142135623730951
>>> f1(-1)             'x < 0!'
```

Funkcija f2 definirana kao

```
>>> f2 = lambda x : "x < 0!" if x < 0 \
                    else x**0.5
```

dala bi iste rezultate:

```
>>> f2(2)              1.4142135623730951
>>> f2(-1)             'x < 0!'
```

Poruka "x < 0!" bit će ispisana ako je x < 0, a to je negacija od x >= 0, pa zaključujemo da možemo definirati i funkciju f3 koja nema uvjetne izraze:

```
>>> f3 = lambda x : (x < 0)* "x < 0!"\
                    or x**0.5
```

Ako je x < 0 rezultat će biti "x < 0!" i neće se izvršiti izraz x**0.5:

```
>>> f3(2)              1.4142135623730951
>>> f3(-1)             'x < 0!'
```

Zaključak je: Ako uvjetni izraz sadrži uvjete koji su negacija jedan drugom i jedan od njih prethodi izračunavanju funkcije koja je definirana ako je uvjet ispunjen (istinit), koristit ćemo operaciju or tako da prvo pišemo uvjet koji je negacija uvjeta domene, jer će se njegovim ispunjenjem prekinuti daljnje izračunavanje izraza, a time se „neće ni znati“ da je funkcija trebala biti pozvana s vrijednošću izvan njezine domene. Dakako, ako koristimo uvjetni izraz, takvih problema ne bismo imali.

POJEDNOSTAVLJENJE LOGIČKIH IZRAZA

Ponekad je moguće pojednostaviti logičke izraze i time povećati preglednost. Pri tome se mora paziti da vrijednost izraza ostane nepromijenjena. Ako su r_1 i r_2 relacije, evo nekoliko primjera najčešćih pojednostavljenja:

```
a r1 x and x r2 b  → a r1 x r2 b
x r1 b and b r2 y  → x r1 b r2 y
P == True          → P
P == False         → not P
X = True if x >= y else False
     → X = x >= y
X < a and X > a       → False
1 if x > y else 0     → 1*(x > y)
```

```
guess > 10 and guess < 20
           ↓
10 < guess and guess < 20
   → 10 < guess < 20
```

Također treba koristiti logičke varijable. Na primjer:

```
P = G % 4 == 0 and G % 100 != 0 \
               or  G % 400 == 0
print (P *'PRIJESTUPNA!',
       (not P) *'NIJE PRIJESTUPNA')
```

PROVJERA ULAZNIH PODATAKA

Važno: ne možemo biti sigurni da program radi korektno ako nisu učitani podaci iz očekivane domene. Shema:

```
INPUT = """
X     = Input ("Unesi podatak X ")
Ok    = <logički izraz>
exec ( INPUT *(not Ok) ) """
exec (INPUT)
```

<logički izraz> će sadržavati uvjet koji će biti istinit ako je uneseni podatak u domeni, odnosno, neistinit ako nije. Tada će not Ok biti jednako 1, pa će se ponoviti unos. Ako je podatak u domeni, not Ok će biti jednako 0 pa će INPUT *0 biti prazan string i neće se ponoviti unos. Na primjer, program „Cajger na cajgeru“ može se napisati kao što slijedi:

Cajger_na_cajgeru.py

```
Input = """
T  = eval (input ("Zadaj sat "))
Ok = type (T) == int and (0 <= T <= 23)
Ok = (not Ok) *Input;  exec (Ok) """
exec (Input)
S = T % 12      # S je od 0 do 11
m = S *60/11.0 # m min poslije T sati
M = int (m)
s = round ((m-M)*60, 2) # s sekunde
print (T, ':', M, ':', s)
>>>
Zadaj sat 24
Zadaj sat 16.5
Zadaj sat 16          16 : 21 : 49.09
```

LOGIČKE LAMBDA FUNKCIJE

S obzirom na to da *LAMBDA funkcija* u svojoj definiciji ima sintaksnu strukturu ***izraz***, u posebnom slučaju to može biti ***logički_izraz***, pa možemo, na primjer, definirati *LAMBDA funkcije* logičkih operacija not, and

i or. Još smo dodali i definiciju implikacije ($x \Rightarrow y$ ekvivalentno je $\neg x \vee y$).

```
>>> NOT = lambda x    : not (x)
>>> AND = lambda x, y : x and y
>>> OR  = lambda x, y : x or y
>>> # implikacija, x => y
>>> IMP = lambda x, y : not (x) or y
>>> F   = False; T = True
>>> NOT (F)   True  >>> NOT (T)   False
>>> AND (F,T) False >>> OR  (F,T) True
>>> IMP (F,F) True  >>> IMP (F,T) True
>>> IMP (T,F) False >>> IMP (T,T) True
>>> AND (1+2,5)   5 >>> OR (0,'0')  '0'
```

UPORABA UVJETNIH IZRAZA

Prije svega, uočimo da iz semantike uvjetnih izraza slijedi definicija logičkih operacija:

```
not x   -> False if x else True
x or y  -> x if x else y
x and y -> x if not x else y
```

Uvjetne izraze ćemo rabiti kad treba izračunati vrijednost funkcije definirane po segmentima. Možda je dobar primjer za to rješenje sljedećeg zadatka.

Zadatak 3.4

Pretpostavimo da je cijena razgovora na mobitelu jednaka C kn (s PDV-om!) po minuti, s obračunskom jedinicom 15 sekundi. Uspostava poziva je C0 kn. Treba izračunati koliko će koštati poziv ako ste razgovarali M minuta i S (od 0 do 59) sekundi. Trajanje razgovora unosi se kao realni broj u formatu M.SS.

Analizom problema dolazimo do formule za izračunavanje cijene:

```
Cijena = C0 + C*(M+K)
```

gdje je:

```
M    minute
S    sekunde
K = 0.00 za S = 0
    0.25 za S od  1 do 15
    0.50 za S od 16 do 30
    0.75 za S od 31 do 45
    1.00 za S od 46 do 59
```

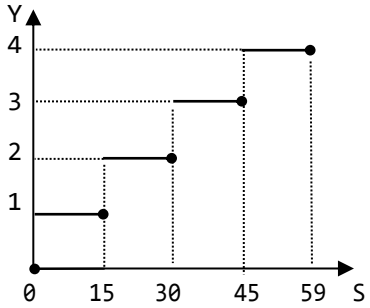
Udio sekundi, K, u cijeni razgovora nije problem izračunati:

```
K =  0.00 if S ==  0 else \
     0.25 if S  < 16 else \
     0.50 if S  < 31 else \
     0.75 if S  < 45 else 1.00
```

ili, još jednostavnije:

```
K = 0.25 *((S>0) +(S>15) +(S>30)
   +(S>45))
```

Ali, ni to nije sve! Iz drugog izraza za izračunavanje koeficijenta K vidimo da je izraz u zagradi zapravo step funcija Y što se može prikazati kao:



Lako se uvjerimo da se Y može izračunati formulom:

```
Y = (S +14) // 15
```

pa će koeficijent K biti:

```
K = ((S +14) // 15) *0.25
```

Evo sada niza naredbi koji izračunava cijenu razgovora uz pretpostavku da je C0=0.25 i C=0.49. Da bismo izbjegli probleme s numerikom, obavezno umjesto funkcije int() rabimo *LAMBDA funkciju* Int(), opisanu ranije.

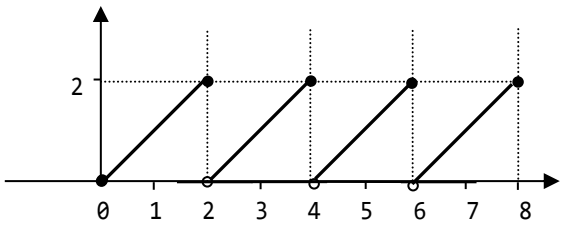
3.31 Cijena telefonskog poziva

```
>>> k   = lambda : ((S +14) // 15) *0.25
>>> C0  = 0.25;  C = 0.49
>>> Int = lambda x : int (round (x))
>>> T   = eval (input (
'Upiši trajanje razgovora, M.SS '))
Upiši trajanje razgovora, M.SS 1.16
>>> M = Int(T); S = Int (100*(T -M))
>>> K = k(); print(
"Cijena razgovora je %0.2f kn"
% (C0 +C*(M+K)))
Cijena razgovora je 0.98 kn
```

I „pilaste funkcije" su podesne za primjene uvjetnih izraza. Slijede dva primjera koja to potvrđuju u dovoljnoj mjeri:

Zadatak 3.5

Dana je „pilasta funkcija“ prikazana na slici.



Izračunati f(x), x∈[0,8], odnosno ispisati „nije definirano“ ako je x izvan tog intervala. Napomena: vrijednost funkcije u točki x=0 je 0, a u točkama 2, 4, 6 i 8 je 2.

››› 3.32 “Pilasta funkcija”

```
>>> # "Pilasta funkcija" f(x) na
>>> # intervalu [0, 8]
>>> f = lambda x : (
  'izvan domene' if not 0 <= x <= 8 else
  0              if x == 0          else
  2.0            if not x % 2       else
  x % 2
>>> print (f(-1), f(6.5), f(8),
           f(8.01))
izvan domene 0.5 2.0 izvan domene
```

Evo još jednoga rješenja:

```
>>> t = 'izvan domene'
>>> fx  = lambda x : \
          t   if x < 0   else \
          x   if x <= 2  else \
          x-2 if x <= 4  else \
          x-4 if x <= 6  else \
          x-6 if x <= 8  else \
          t
>>> print (fx(-1), fx(6.5), fx(8),
           fx(8.01))
izvan domene 0.5 2 izvan domene
```

REKURZIJE

LAMBDA funkcije omogućuju poziv same sebe, za što kažemo da dopuštaju „rekurzivne pozive” ili, jednostavno, „rekurzije”. To slijedi iz definicije sintakse *LAMBDA funkcije* koja sadrži izraz, a izraz može biti uvjetni izraz koji kao svoju alternativu može sadržati izraz s funkcijom kao operandom, u posebnom slučaju s pozivom funkcije u čijoj se definiciji nalazi.

Zadatak 3.6

Funkcija f(x) definirana je kao:

$$f(x)=\begin{cases} x+10, & \text{za } x \geq 10 \\ f(x+10), & \text{za } x < 100 \end{cases}$$

Napisati program koji će za zadani x izračunati f(x).

Evo rješenja koje sadrži rekurziju:

```
>>> f = lambda x : \
        x+10 if x >= 100 else f(x+10)
>>> f(-113)           117
```

Faktorijel nenegativnog cijelog broja definiran je rekurzivno:

$$fac(n)=\begin{cases} \text{"nije definirano"} & \text{ako je } n<0 \\ 0 & \text{ako je } n=0 \\ n*fac(n-1) & \text{ako je } n>0 \end{cases}$$

Na primjer, faktorijel broja 5 bio bi jednak:

```
fac(5) = 5 *fac(4)
           ⇓
       = 5 *4 *fac(3)
              ⇓
       = 5 *4 *3 *fac(2)
                 ⇓
       = 5 *4 *3 *2 *fac(1)
                    ⇓
       = 5 *4 *3 *2 *1 *fac(0)
                       ⇓
       = 5 *4 *3 *2 *1 *1
       = 120
```

Sada možemo definirati *LAMBDA funkciju* za izračunavanje faktorijela cijelog broja:

```
>>> fac = lambda n : \
    "nije definirano" if n<0  else    \
                   1  if n==0 else    \
          n*fac(n-1) # if n>0
>>> fac (5)                          120
>>> fac (100)
933262154439441526816992388562667004907
159682643816214685929638952175999932299
156089414639761565182862536979208272237
582511852109168640000000000000000000000
00
```

Kao što ćemo kasnije vidjeti, prednost uporabe rekurzije jest u kompaktnosti algoritma i izravnom „prepisivanju“ definicije problema. Međutim, postoje

određene zamke. Jedna od njih je da rekurzivni poziv funkcije povećava zahtjeve za radnom memorijom i postoji granica ili maksimalna „dubina“ rekurzivnih poziva. To će se u našem primjeru dogoditi ako trebamo izračunati faktorijel od 1025:

```
>>> fac (1025)
...
RuntimeError: maximum recursion depth
exceeded
```

Dakle, funkcija `fac()` definirana je za domenu `[0,1024]` i ako nam treba, na primjer, faktorijel od `5000`, morat ćemo tražiti druga rješenja. Pokazat ćemo da je to moguće kad uvedemo još neke naredbe, ali u ovom slučaju znamo da postoji funkcija `factorial()` iz modula `math` koja korektno računa faktorijel za brojeve `1025`, a i za `5000`. Provjerimo:

```
>>> from math import *
>>> factorial (1025)
55539920159603287151538449933009511064 5
...0000000000000
```

To je broj s 2643 znamenke (pozicionirajte kursor iza posljednje znamenke i pročitajte broj kolone). Možemo probati i `factorial(5000)`. Rezultat je broj sa 16326 znamenki!

I rekurzivna LAMBDA funkcija u rješenju Zadatka 3.6 ima ograničenja. Na primjer:

```
>>> f(-9820)          110
>>> f(-9830)
...
RecursionError: maximum recursion depth
exceeded in comparison
```

Nerekurzivna funkcija može se napisati kao

```
>>> f2 = lambda x : (x+10)*(x >= 100) or
(110 +x % 10)
>>> f2(-9830)          110          >>>
f2(-999999)           111
```

Član Fibonaccijevog niza (3)

Često se izračunavanje n-tog člana Fibonaccijevog niza rješava rekurzijom. Razlog za to jest da se Fibonaccijev niz definira rekurzivno:

1) Prvi član niza, Fib_1, jednak je 1.
2) Drugi član niza, Fib_2, također je jednak 1.
3) n-ti član niza, za $n>2$, jednak zbroju prethodna dva člana:

$$Fib_n = Fib_{n-2} + Fib_{n-1}$$

```
>>> fib = lambda n : \
    1                   if n <= 2 else \
    fib(n-2) + fib(n-1)
>>> fib(10)     55
```

Možemo probati izračunati `20`-ti, `30`-ti i `40`-ti. član. Primijetit ćemo da je vrijeme izračuna ovoga posljednjeg bilo malo duže. Ako probamo izračunati `45`-ti član, izračun će trajati 5-6 minuta. Ako probamo izračunati 70-ti član, izračun bi trajao oko godinu dana! Normalno, pod uvjetom da imamo beskonačno memorije, da je računalo stalno bilo uključeno... Izračun `100`-tog člana trajao bi, vjerovali ili ne, oko 3 milijuna godina. Morali bismo svojim nasljednicima dati upute da čekaju rezultat! Malo se šalimo, ali dokaz trajanja izračuna dali smo u dvanaestom poglavlju.

Dakle, zaključujemo da uporaba rekurzije nije uvijek najbolje rješenje. U ovom je primjeru bolje rabiti klasičnu iteraciju koju smo opisali u prvom i drugom poglavlju. Izračun `1000`-tog člana je trenutan! Evo rezultata:

```
100-ti član  : 354224848179261915075
1000-ti član :
434665576869374564356885276750406258025
646605173717804024817290895365554179490
518904038798400792551692959225930803226
347752096896232398733224711616429964409
065331879382989696499285160037044761377
95166849228875
```

P R O G R A M I

Na ovom mjestu imamo dovoljno znanja da možemo pisati malo „ozbiljnije” programe. Dani primjeri programa sumiraju sve što smo dosad naučili. Istodobno pokazuju pristup disciplini programiranja, posebno u provjeri ulaznih podataka. Prvo proširujemo modul `Moj_modul.py` s uvozom standardnog modula `cmath` i *LAMBDA funkcijom* `z()` za unos kompleksnih brojeva.

```
from cmath  import *
```

```
z = lambda s='', x='x = ', y='y = ' : \
    complex (Input (s+x), Input (s+y))
```

U preostalim programima je prvo pokazano kako se uporabom kompleksnih podataka može jednostavnije napisati algoritam za izračunavanje udaljenosti dviju točaka u ravnini i površina trokuta sa zadanim koordinatama svojih vrhova u ravnini. Posebno je pokazana uporaba kompleksnih brojeva i funkcija na njima u rješavanju problema nalaženja rezultirajuće sile (zbroj vektora) ili koordinate težišta trokuta.

Potom je u nekoliko programa pokazana uporaba uvjetnih izraza u rješavanju problema iz prakse, rješenje programa iz prethodnih poglavlja na drugi način, nul-točke kvadratne jednadžbe na još dva načina, „grafički“ prikaz funkcije `sin(x)` i računanje trećeg korijena.

Udaljenost dviju točaka u ravnini (2)

Ako su koordinate dviju točaka X i Y u ravnini zadane kao kompleksni brojevi, izračunavanje njihove udaljenosti jednako je

```
abs (X -Y)
```

pa je nova inačica programa:

Udaljenost_dviju_točaka_2.py

```
from Moj_modul import *
d = lambda X, Y : round (abs (X -Y), 2)
A = z ('Točka A, A') # z iz Moj_modul
B = z ('Točka B, B')
print ('d =', d(A, B))
```

```
>>>
Točka A, Ax = 12.5
Točka A, Ay = 1.5
Točka B, Bx = -2.2
Točka B, By = -3
d = 15.37
```

Površina trokuta (3)

Primjenom kompleksnih brojeva za koordinate vrhova trokuta, A, B i C, problem izračunavanja površine trokuta prilično je jednostavan. Evo programa:

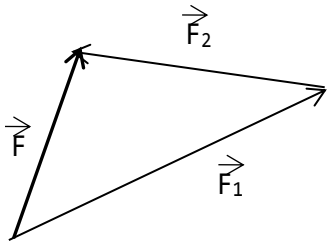
Površina_trokuta_3.py

```
# Izračunavanje površine trokuta sa
# zadanim koordinatama vrhova A, B i C
# kompleksnim brojevima
from Moj_modul import *

d = lambda X, Y : round (abs (X -Y), 2)
A, B, C = z('Vrh A, A'), z('Vrh B, B'), \
          z('Vrh C, C')
a, b, c = d (B, C), d (A, C), d (A, B)
s = (a +b +c) /2; P = sqrt (s *(s-a)
        *(s-b) *(s-c)); print ("P =", P)
```

```
>>>
Vrh A, Ax = 0
Vrh A, Ay = 0
Vrh B, Bx = 0
Vrh B, By = 4
Vrh C, Cx = 3
Vrh C, Cy = 0
P = 6.0
```

Rezultanta dviju sila

Treba izračunati rezultirajuću silu F dviju sila F_1 i F_2.



Problem se svodi na zbrajanje vektora $\vec{F}_1$ i $\vec{F}_2$. Vektor je određen svojim intenzitetom i napadnim kutom, pa ga možemo interpretirati kao kompleksni broj uz pomoć funkcije `cmath.rect(r,φ)`, gdje je r intenzitet, a φ napadni kut (u radijanima).

Tada se problem zbrajanja vektora svodi na zbrajanje kompleksnih vrijednosti F1 i F2, pa je rezultujuća sila F, interpretirana kao kompleksni broj, F = F1+F2. Poslije toga preostaje da se F konvertira u vektor. To čini funkcija `cmath.polar(F)` koja vraća `abs(F)`, a to je modul, odnosno intenzitet rezultirajuće sile, i napadni kut (u radijanima).

Rezultanta.py

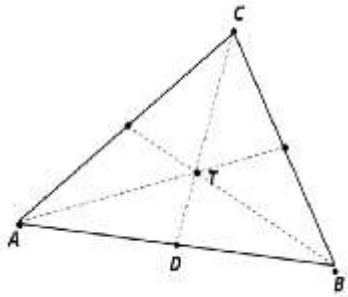
```
# Rezultanta dviju sila (zbroj vektora)
from Moj_modul import *
vektor  = lambda s : rect (Input
('Intenzitet sile F' +s),
radians (Input ('i napadni kut      ')))
F1, F2  = vektor('1 '), vektor('2 ')
F       = F1 +F2; r, Fphi = polar(F) phi
= degrees (Fphi)
print ("Intenzitet sile F  %0.2f" % r)
print ("i napadni kut      %0.2f" % phi)
```

```
>>>
Intenzitet sile F1 100
i napadni kut      45
```

```
Intenzitet sile F2 100
i napadni kut      135
Intenzitet sile F  141.42
i napadni kut      90.00
```

TEŽIŠTE TROKUTA

Zadan je trokut s koordinatama svojih vrhova *A*, *B* i *C*. Izračunati kordinate težišta. Iz matematike je poznato da je težišnica trokuta dužina koja spaja vrh s poloviš-tem nasuprotne stranice te dijeli trokut na dva dijela jednake površine. Sve tri težišnice se sijeku u jednoj točki, težištu trokuta. Težište *T* dijeli svaku od težišnica u omjeru 1:2.



Ideja rješenja je da se koordinate vrhova konvertiraju u kompleksne brojeve. Težište će biti na 1/3 težišnice koja spaja točku polovišta stranice između bilo koja dva vrha i vrha nasuprot. Na primjer, neka to bude točka D, polovište stranice AB, i vrh C. Polovište se može dobiti sa:

```
D = (A + B) / 2
```

pa se točka T može dobiti izračunavanjem izraza

```
D + (C - D) / 3 = (3*D +C -D) / 3
                = (2*D +C) / 3
                = (2 *(A+B)/2 +C) / 3
                = (A + B + C) / 3
```

Težište_trokuta.py

```
# Koordinate težišta trokuta s vrhovima
# A, B i C
from Moj_modul import *
print (NL, 'KOORDINATE TEŽIŠTA '
'TROKUTA', NL, sep = '')
A = z ('Koordinate točke A ')
B = z ('Koordinate točke B ')
C = z ('Koordinate točke C ')
T = (A +B +C) / 3
print (NL, 'Koordinate težišta T,',
       sep = '', end = ' ')
print ("T (%0.2f, %0.2f)"
       % (T.real, T.imag))
```

```
>>>
KOORDINATE TEŽIŠTA TROKUTA
Koordinate točke A x = 1
Koordinate točke A y = 1
Koordinate točke B x = 3
Koordinate točke B y = 2
Koordinate točke C x = -1
Koordinate točke C y = -2

Koordinate težišta T, T (1.00, 0.33)
```

REZULTIRAJUĆI OTPOR SERIJSKOG STRUJNOG KRUGA

Iz elektrotehnike je poznato da je električna impedancija slična električnom otporu koji je mjera suprostavljanja prolasku istosmjerne električne struje kroz strujni krug. U samom računu impedancija idealnog otpora jednaka je otporu za istosmjernu struju, R , a zavojnicama induktiviteta L i konden-zatorima kapaciteta C dodjeljuje se čisto imaginarni „reaktivni otpor“ ili reaktancija:

$$R_L = j\omega L$$
$$R_C = 1/j\omega C$$

gdje su:

j – imaginarna jedinica
ω – frekvencija strujnog kruga

Impedancija serijskog strujnog kruga izmjenične str-uje izračunava se prema formuli:

$$Z = \sqrt{R^2 + (R_L - R_c)^2}$$

gdje su:

R – omski otpor
R_L – induktivni otpor
R_C – kapacitivni otpor

S obzirom na to da su imaginarne vrijednosti, rezul-tirajući otpor (impedancija) jednaka je:

$$Z = |R + R_L - R_C|$$

Otpor.py

```
# Ω
from Moj_modul import Input

R, RC, RL = Input ('Zadaj omski, '
'kapacitivni i induktivni otpor u Ω ')
Ω = abs (R +RL*1j -RC*1j)
print ('Rezultirajuci otpor',
       round (Ω, 2), 'Ω')
```

```
>>>
```

```
Zadaj omski, kapacitivni i induktivni
otpor 0, 300, 700
Rezultirajuci otpor 400.0Ω
>>>
Zadaj omski, kapacitivni i induktivni
otpor u Ω 100, 0, 100
Rezultirajuci otpor 141.42 Ω
```

KISELOST TLA

Klasifikacija raspona kiselosti tla, pH, definiran je u sljedećoj tablici. https://en.wikipedia.org/wiki/PH

kiselo - alkalno	pH	kiselo - alkalno	pH
ultra kiselo	< 3.5	neutralno	6.6 - 7.3
izuzetno kiselo	3.5 - 4.4	lagano alkalno	7.4 - 7.8
vrlo kiselo	4.5 - 5.0	umjereno alkalno	7.9 - 8.4
jako kiselo	5.1 - 5.5	jako alkalno	8.5 - 9.0
umjereno kiselo	5.6 - 6.0	vrlo jako alkalno	> 9.0
blago kiselo	6.1 - 6.5		

Ovo je "školski primjer" za uporabu uvjetnih izraza. Prethodi im unos vrijednosti pH i p
onavljanje unosa ako nije u domeni od 0 do 14,

pH.py

```
PH = """
pH =  eval( input(
'Unesite pH vrijednost (od 0 do 14) '))
Ok =  0 <= pH <= 14
print( 'IZVAN DOMENE!' *(not Ok) )
exec( PH *(not Ok) ) """
exec (PH)
k, a = ' kiselo', ' alkalno'
print( 'ultra'    +k if pH <   3.5 else
       'izuzetno' +k if pH <=  4.4 else
       'vrlo'     +k if pH <=  5.0 else
       'jako'     +k if pH <=  5.5 else
       'umjereno' +k if pH <=  6.0 else
       'blago'    +k if pH <=  6.5 else
       'NEUTRALNO'   if pH <=  7.3 else
       'blago'    +a if pH <=  7.8 else
       'umjereno' +a if pH <=  8.4 else
       'jako'     +a if pH <=  9.0 else
       'vrlo'     +a )
>>>
Unesite pH vrijednost (od 0 do 14) 14.5
IZVAN DOMENE!
Unesite pH vrijednost (od 0 do 14) 5.5
jako kiselo
```

ISPIT

n studenata polaže ispit iz predmeta „Programiranje u Pythonu". Prolaznost je 90%, od čega su vjerojatnosti dobivenih ocjena: 25% studenata, ocjena 2, 45% ocjena 3, 20% ocjena 4 i 10% studenata, ocjena 5. Da bismo bolje sagledali strukturu programa, tekst nismo prikazali zelenom bojom. Samo smo označili početak i kraj.

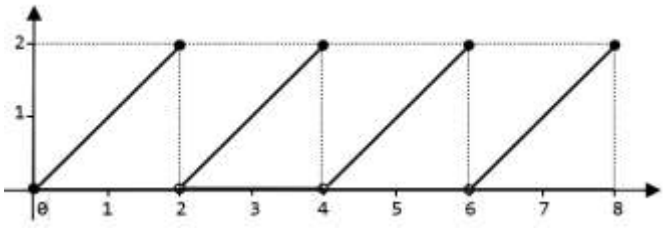
Ispit.py

```
# n studenata polaže ispit iz predmeta
# "Programiranje u Pythonu".
# Prolaznost je 90%, od čega su
# vjerojatnosti dobivenih ocjena:
# 25% 2, 45% 3, 20% 4 i 10% 5
from Moj_modul import *
_2 = _3 = _4 = _5 = 0  # Brojači ocjena
Inp =  """
n  = Input (
'Koliko je studenata izašlo na ispit?')
Ok = type(n) == int and n > 0
print ((not Ok) * 'n < 1 ili nije '
'cijeli broj. Ponovi upis!')
exec (Inp * (not Ok)) """
Ispit = """
x = randint (1, 100)
O = 1 if random() < 0.1 else \
    2 if x <= 25        else \
    3 if x <= 70        else \
    4 if x <= 90        else \
    5
_2 += O == 2
_3 += O == 3
_4 += O == 4
_5 += O == 5  """
exec (Inp); exec (Ispit *n)
s = _2 +_3 +_4 +_5
p = round (100.0/s, 2)
print ("Položilo %d (%0.2f%c)" % (s,
round (100.0 *s/n, 2), '%'), NL)
isp = "%3d     %-3d  %6.2f"
print ("Ocjena  BS      %")
print (isp % (2, _2, _2*p))
print (isp % (3, _3, _3*p))
print (isp % (4, _4, _4*p))
print (isp % (5, _5, _5*p))
>>>
Koliko je studenata izašlo na ispit? 32
```

```
  Položilo 28 (87.50%)

  Ocjena  BS      %
    2     9     32.13
    3     7     24.99
    4     8     28.56
    5     4     14.28
>>>
Koliko je studenata izašlo na ispit? 100

Položilo 89 (89.00%)

Ocjena  BS      %
  2     20    22.40
  3     39    43.68
  4     13    14.56
  5     17    19.04
```

PILASTA FUNKCIJA

Dana je „pilasta funkcija“ prikazana na slici.



Treba napisati lambda funkciju f koja će izračunati $f(x), x \in [0,8]$, odnosno ispisati „nije definirano“ ako je x izvan tog intervala.

Pilasta_funkcija.py

```
1-----------------------------------1
f = lambda x : \
 'nije def.' if x<0 or x>8 else \
 x           if x <= 2     else \
 4-x         if x <= 4     else \
 x-4         if x <= 6     else \
 8-x
2-----------------------------------2
f2 = lambda x : \
 'nije def.' if x<0 or x>8 else \
 x %4        if x %4 <= 2  else \
 4 -(x %4)

x = eval (input (
   'Zadaj x iz intervala [0, 8] '))
print ('1. f(x)  =', f(x) )
print ('2. f2(x) =', f2(x))
>>>
Zadaj x iz intervala [0, 8] -1
1. f(x)  = nije def.
2. f2(x) = nije def.
>>>
Zadaj x iz intervala [0, 8] 2.5
1. f(x)  = 1.5
2. f2(x) = 1.5
```

CIJENA PARKIRANJA U ZRAČNOJ LUCI ZAGREB

Pretpostavimo da je „Cjenik parkirališta za putnike i posjetitelje” u Zračnoj luci Zagreb:

Vrijeme zadržavanja	Cijene u HRK
do **1** sat	27 HRK
2 sata	47 HRK
3 sata	70 HRK
od **3 do 6** sata	78 HRK
od **6 do 12** sati	110 HRK
od **12 do 24** sata	150 HRK
od **24 do 48** sati	210 HRK
od **3. do 5. dana** svaki sljedeći dan	72 HRK
od **6. dana** svaki sljedeći dan*	68 HRK

Svaki započeti novi sat računa se kao cijeli. Svaki započeti novi dan također se računa kao cijeli. Cjeniku ćemo dodati da se boravak do prvih 10 minuta ne naplaćuje. Priloženi program za zadano trajanje parkiranja u danima, D, $D \geq 0$, satima, S, $0 \leq S \leq 23$ i minutama, M, $0 \leq M \leq 59$ izračunava ukupnu cijenu usluge parkiranja.

Parking_Zračna_luka.py

```
# Usluga parkiranja na parkiralištu
# Zračne luke Zagreb
Inp = """
D, S, M = eval ( input ("Trajanje
parkiranja (Dana, Sati, Minuta)? ") )
Ok = type(D)==type(S)==type(M) == int \
     and D >= 0 and 0 <= S <= 23 and  \
     0 <= M <= 59
exec ( Inp *(not Ok) ) """
exec ( Inp )
T = round (D*24 +S +M/100.0, 2)
d = D +((T % 1) > 0)
C = 0  if T <=  0.1  else  \
   27  if T <=  1.0  else  \
   47  if T <=  2.0  else  \
   70  if T <=  3.0  else  \
   78  if T <=  6.0  else  \
```

```
    110  if T <= 12.0  else  \
    150  if T <= 24.0  else  \
    210 +( 0 if d <= 2 else
           72 *(d-2) if d <=5 else
           72 *3 +68 *(d-5) )
# ili 210 + 72 *(d-2) *(2<d<=5)
# +(72 *3 + 68 *(d-5)) *(d>5)
print ( "Usluga parkiranja za %d dana,
%d sati i %d min... %d kn"
% (D, S, M, C) )
```

```
>>>
Trajanje parkiranja (Dana, Sati,
Minuta)? 0, 0, 9
Usluga parkiranja za 0 dana, 0 sati i 9
min... 0 kn
>>>
Trajanje parkiranja (Dana, Sati,
Minuta)? 0, 2, 1
Usluga parkiranja za 0 dana, 2 sati i 1
min... 70 kn
>>>
Trajanje parkiranja (Dana, Sati,
Minuta)? 2, 0, 1
Usluga parkiranja za 2 dana, 0 sati i 1
min... 282 kn
>>>
Trajanje parkiranja (Dana, Sati,
Minuta)? 30, 1, 1
Usluga parkiranja za 30 dana, 1 sati i
1 min... 2194 kn
```

ZBROJ ZNAMENKI PRIRODNOG BROJA (2)

Treba zbrojiti znamenke (brojke) velikog prirodnog broja. Rješenje je sadržano u programu

Zboj.py

Ako je n zadani broj, dijelili smo ga s 10 i zbrajali ostatke:

```
S = 0
n, b = divmod (n, 10); S += b
```

Postupak treba ponoviti nad novom vrijednošću broja n sve dok je n>0. Postupak je bio ponovljen k puta, gdje k duljina broja n, `len (str (n))`. Sada znamo kako ćemo okončati postupak kad n postane 0. Evo druge inačice. Poslije provjere valjanosti ulaznog podatka ponavljamo izvršavanje niza naredbi sadržanih u tekstu `Zbroj` sve dok je ostatak dijeljenja s 10 veći od 0.

Zboj_2.py

```
# Unos i provjera domene
Unos = """
N   = eval ( input (
'Zadaj cijeli broj veći od 0 ') )
INT = type (N) == int
exec ( Unos * ( not INT or INT
                and N <= 0) )
"""
exec (Unos)
# Izračunaj zbroj znamenki
Zbroj = """
N, B = divmod (N, 10)
S   += B
exec ( Zbroj *(N > 0) ) """
S = 0; exec (Zbroj)
print (S)
```

```
>>>
Zadaj cijeli broj veći od 0 0
Zadaj cijeli broj veći od 0 12.0
Zadaj cijeli broj veći od 0
848029428374283742389742384763245676543
785643785634786543785634875643876537485
674385634875643786534785634785634875634
875328765387265843756348756374865348765
473865473856743825438725638 47
986
```

NUL-TOČKE KVADRATNE JEDNADŽBE (3)

Evo još jedne inačice programa za nalaženje nul-točaka kvadratne jednadžbe. Bez obzira na to što još uvijek ne možemo dati potpuno rješenje, jer ne znamo još neke naredbe, dani program prikazuje kako se mogu rabiti logičke varijable, uvjetni i znakovni izrazi. Pokazali smo i kako prikazati rješenja koja nisu u realnoj domeni (konjugirano su kompleksna).

Nul_točke_3.py

```
from Moj_modul import *
a, b, c = Input (
   'Zadaj koef. kv. jed. (a, b, c) ')
A = a != 0
p = ('' if A else
     'Nije kvadratna jednadžba (a=0)!')
D = b**2 -4*a*c;
_2a = 2*a if A else 1.0
a1  = round (-b/_2a, 4)
a2  = round (abs(D)**0.5 /_2a, 4)
a2  = a2 *1j if D < 0 else a2
x1  = a1 +a2 if A     else ''
x2  = a1 -a2 if A     else ''
```

```
t1  = 'x1 =' * bool (x1)
t2  = 'x2 =' * bool (x2)
print ( p, t1, x1, t2, x2 )
```

```
>>>
Zadaj koef. kv. jed. (a, b, c) 0, 1, 2
Nije kvadratna jednadžba (a=0)!
>>>
Zadaj koef. kv. jed. (a, b, c) -2, 2, 3
 x1 = -0.8229 x2 = 1.8229
>>>
Zadaj koef. kv. jed. (a, b, c) 1, 2, 3
 x1 = (-1+1.4142j) x2 = (-1-1.4142j)
```

NUL-TOČKE KVADRATNE JEDNADŽBE (4)

Prethodni program provjerava valjanost ulaznih podataka. Ako je `a==0` program radi „na prazno“ jer nismo zahtijevali ponavljanje upisa. Evo još jedne inačice programa u kojoj smo pokazali kako se uvjetni izraz može iskoristiti za izbor niza naredbi sadržanog u tekstu ili stringu koji će biti izvršen. Na taj smo način izbjegli rad programa „na prazno“.

Nul_točke_4.py

```
from Moj_modul import *
a, b, c = Input (
     'Zadaj koef. kv. jed. (a, b, c) ')
KJ = """
D = b**2 -4*a*c;
_2a = 2*a
a1  = round (-b/_2a, 4)
a2 = round (abs(D)**0.5 /_2a, 4)
a2     = a2 *1j if D < 0 else a2
x1, x2 = a1 +a2, a1 -a2
print ( 'x1 =', x1, 'x2 =', x2 )
"""
Y = (KJ if a != 0 else "print ("
  "'Nije kvadratna jednadžba (a=0)!')")
exec (Y)
```

```
>>>
Zadaj koef. kv. jed. (a, b, c) 0, 1, 2
Nije kvadratna jednadžba (a=0)!
>>>
Zadaj koef. kv. jed. (a, b, c) -2, 2, 3
x1 = -0.8229 x2 = 1.8229
```

CRTANJE SINUSOIDE

Priloženi program „crta“ sinusoidu u tzv. „znakovnoj grafici“. Sadržaj skripte `Sin` (tekst) „otvorili“ smo da biste je mogli bolje analizirali.

Sinusoida.py

```
# Crtanje funkcije sin(x) na intervalu
[0, 2*π]
from math import sin, pi as π
x = 0;  n = 35;  m = 20
print ( 'x       sin(x)' +' '*3 +'-1' +'
'*(m-1) +'0' +' '*(m-1) +'1' )
print ( ' '*17 +'-'*(2*m+2) )
Sin = """
y = n +int (round (m *sin(x)))
s = ("%s*%s|" %(' '*(y-12), ' '*(n-y-1))
    if y <  n  else "%s*" % (' '*(y-12))
 if y == n else
 "%s|%s*" % (' '*(n-12), ' '*(y-n)))
print ( "%6.4f%8.4f" % (x, sin(x)), s )
Ok = '' if x >= 2*π else ' ';
x += π/10
exec (Ok and Sin) """
exec ( Sin )
```

```
>>>
x       sin(x)   -1                  0                  1
                 ----------------------------------------

0.0000  0.0000                       *
0.3142  0.3090                       |     *
0.6283  0.5878                       |           *
0.9425  0.8090                       |              *
1.2566  0.9511                       |                 *
1.5708  1.0000                       |                  *
1.8850  0.9511                       |                 *
2.1991  0.8090                       |              *
2.5133  0.5878                       |           *
2.8274  0.3090                       |     *
3.1416  0.0000                       *
3.4558 -0.3090                 *     |
3.7699 -0.5878              *        |
4.0841 -0.8090           *           |
4.3982 -0.9511        *              |
4.7124 -1.0000     *                 |
5.0265 -0.9511        *              |
5.3407 -0.8090           *           |
5.6549 -0.5878              *        |
5.9690 -0.3090                 *     |
6.2832 -0.0000                       *
```

IZRAČUNAVANJE TREĆEG KORIJENA

Izračunavanje trećeg korijena realnog broja *x* izvršava se iterativnim numeričkim postupkom poznatim kao *Newtonova aproksimacija*. Ako je *y=x* početna aproksimacija trećeg korijena, nova se aproksimacija dobiva formulom:

$$z = \frac{2y + \frac{x}{y^2}}{3}$$

S obzirom na to da je u svakom koraku *z* bliži stvarnoj vrijednosti trećeg korijena iz *x*, postupak se ponavlja sve dok nova aproksimacija ne zadovolji zadanu preciznost *d*:

$$| z^3 - x | < d$$

Dajemo rekurzivno rješenje, uz pomoć *LAMBDA funkcije* `f`, koja za izračunavanje nove vrijednosti `z` u svakom koraku koristi također *LAMBDA funkciju*, `f0`.

Treći_korijen.py

```
# Newtonova metoda za izračunavanje
# trećeg korijena
x  = eval ( input ('Izračunavam treći
korijen iz broja? ') )

f0 = lambda x, z : (2*z +x/(z*z))/3
f  = lambda x, z : \
f0 (x, z) if abs (z*z*z -x) < 0.00001 \
          else f (x, f0 (x, z))
print ( f(x, x) )
```

```
>>>
Izračunavam treći korijen iz broja? 8
2.0
>>> f (-8, -8)                    -2.0
>>> f (1000, 1000)                10.0
>>> f (3, 3)          1.4422495703074112
```

4.

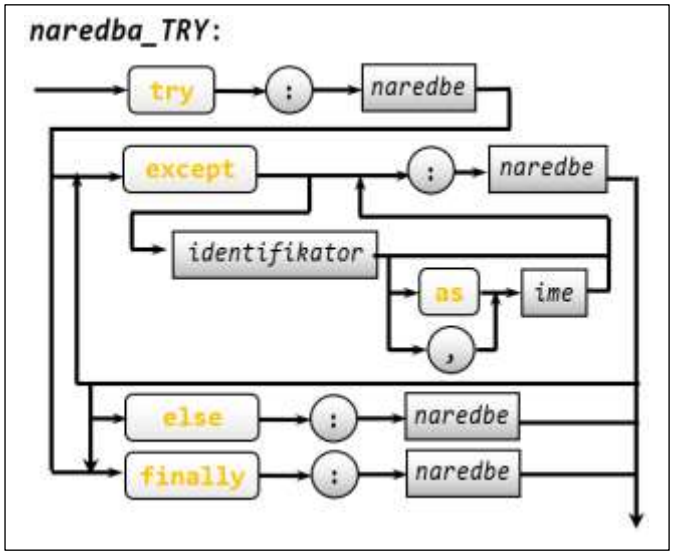
IZNIMKE, SELEKCIJA

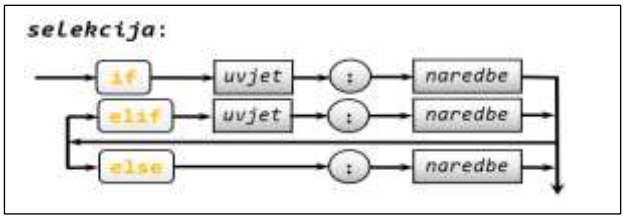
Primitivne ili jednostavne naredbe sintaksne su strukture ili dio programa koje se pišu u jednoj liniji programa. Strukturirane ili složene naredbe objedinjuju više primitivnih ili strukturiranih naredbi u jednu cjelinu.

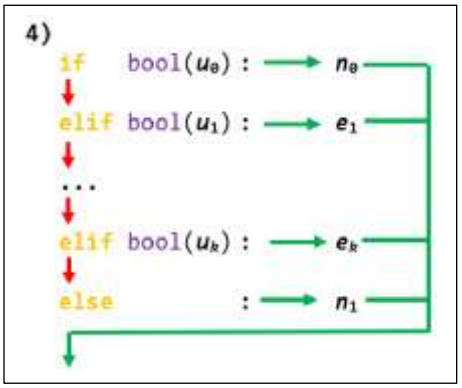
U ovom su poglavlju uvedene i opisane takve dvije naredbe – naredba TRY ili "pokušaj izvršenja naredbi" i selekcija ili „naredba za odabir", kako ćemo je ponekad zvati.

Naredba TRY nam dopušta provjeru izvršivosti pojedinih naredbi (tamo gdje očekujemo da do toga može doći), dojavimo pogrešku i eventualno tražimo unos valjanih podataka.

Primjenom naredbe za odabir moguće je provjeriti postavljeni uvjet i na temelju njegova ishoda odlučiti što dalje raditi. Zbog toga se kaže da naredba za odabir čini osnovu pisanja „inteligentnih" programa. Definicija selekcije u Pythonu objedinjuje i naredbu „grananja" poznatu u nekim jezicima (naredbu CASE u Pascalu ili „switch" naredba u nekim jezicima za programiranje).







Iznimke

Jedan je dio brojčanih funkcija definiran nad ograničenom domenom. Na primjer, logaritamske funkcije su definirane za argument `x>0`, a funkcija `math.sqrt()` za `x≥0`. Pokušajem poziva s argumentom koji je izvan domene, bila bi dojavljena pogreška:

```
>>> from math import *; sqrt (-1)
ValueError: math domain error
```

Pokušajem poziva s argumentom koji nije odgovarajućeg tipa, bila bi dojavljena pogreška `TypeError`:

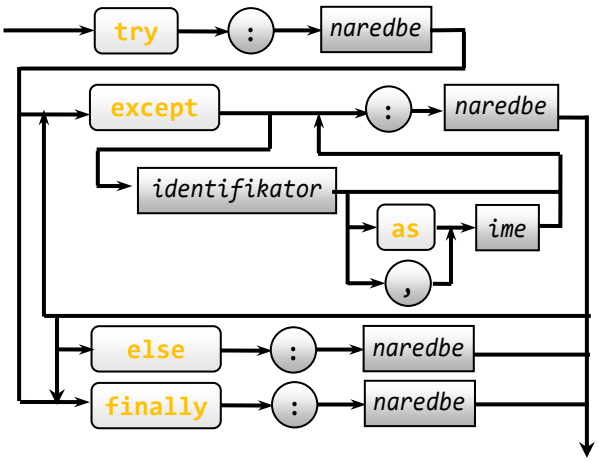
```
>>> '123'**2
TypeError: unsupported operand type(s)
for ** or pow(): 'str' and 'int'
```

U pokušaju dijeljenja s nulom bila bi dojavljena pogreška:

```
>>> 12/0
ZeroDivisionError: integer division or
modulo by zero
```

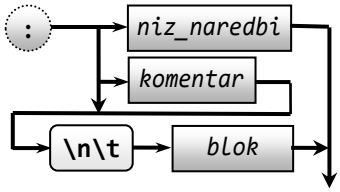
Poslije svake dojave pogreške prekida se daljnje izvršavanje programa. Ako ne želimo da se to dogodi, postoji posebna naredba, *naredba TRY*, koja nam dopušta provjeru izvršivosti pojedinih naredbi (tamo gdje očekujemo da do toga može doći), dojavimo pogrešku i eventualno tražimo unos valjanih podataka.

***naredba_TRY*:**



Naredba TRY ili iznimke prva je složena naredba koju ćemo definirati. Sastoji se iz glavnog dijela, koji započinje rezerviranom rječju `try`, i nekoliko grana, koje započinju rezerviranim riječima `except`, `else` i `finally`. Općenitoj sintaksi *naredbi TRY* dodajemo:

***naredbe*:**



***identifikator*:** *ime_pogreške*

ime_pogreške : ValueError| SyntaxError| TypeError| NameError| ZeroDivisionError

Danoj sintaksi *naredbe TRY* dodajemo i sljedeća kontekstna pravila:

1) ***naredbe*** mogu biti ***niz_naredbi*** ili ***blok***

2) ***niz_naredbi*** piše se iza ":", u jednoj liniji programa

3) Naredbe bloka pišu se u novom redu (prikazali smo s **\n** – novi red, i **\t** –tabulator) i počinju u istoj koloni pomaknutoj desno barem jedno mjesto u odnosu na kolonu u kojoj je napisan `try`, `except`, `else` ili `finally`. Ako *naredba TRY* sadrži naredbe u bloku, dopušteno je pisati komentar u početnoj liniji (iza dvotočke). Predefinirano je pomicanje naredbi bloka za 4 znaka udesno. Tu postavku možemo promijeniti biranjem **Options**, **Configure IDLE**, potom s klizačen u postavci **Indentation Width** postavimo, na primjer, 2. Sada, ako napišemo `try:` pa **Enter**, kursor će biti pozicioniran u trećoj koloni novoga reda. Možemo ga, ako želimo, pomaknuti još udesno. Poslije unosa naredbe u tom redu i prelaskom u novi red kursor će biti u koloni početka prethodnog reda. Ako pišemo novu granu, vraćamo kursor u prvu kolonu.

4) `try` i pripadajuće grane `except`, `else` i `finally` pišu se u istoj koloni (jer su dio istog bloka, osnovnog ili na nižoj razini).

5) *Naredba TRY* u glavnom dijelu i u svim granama sadrži stukturu ***blok***. S obzirom da ***blok*** može sadržavati *naredbu TRY*, moguće je na svim mjestima pisati ugniježđenu *naredbu TRY*.

Ako *naredbu TRY* pišemo u interaktivnom modu, poslije upisa:

```
>>> try :
        _
```

označili smo poziciju kursora u novom redu za pisanje naredbi bloka. Početak ostalih grana *naredbe TRY* piše se od prve kolone:

```
>>> try :
        ...
except :
    ...
else :
    ...
>>>
```

Kraj upisa je poslije praznoga reda poslije čega se izvršavaju naredbe počevši od glavne grane.

SEMANTIKA

Iz sintakse *naredbe TRY* slijede dva slučaja njezina pisanja:

1) `try ... [ except ...]+ [else ...]? [finally ...]?`
2) `try ... finally ...`

U prvom slučaju mora biti jedna ili više *EXCEPT grana*, što je označeno znakom "+" u ekponentu, potom *ELSE grana*, koja može biti izostavljena, označeno znakom "?" u eksponentu, i *FINALLY grana*, koja također može biti izostavljenja.

Izvršavanje *naredbe TRY* započinje glavnom, *TRY granom*, i naredbama u nizu naredbi ili bloku. Ako nema pogreške, preskaču se *EXCEPT grane* i nastavlja na *ELSE grani* (ako je ima) i završava s *FINALLY granom* (ako je ima).

Ako se pojavi pogreška, provjerava se redom istinitost uvjeta iza *EXCEPT grana*. Izvršava se blok unutar prve *EXCEPT grane* u kojoj je identifikator jednak imenu pogreške ili *EXCEPT grana* koja nema identifikatora.

Ako ne postoji takva *EXCEPT grana* izvršava se blok unutar *ELSE grane* i prelazi na *FINALLY granu* (ako postoji) i završava s *FINALLY granom*, ako postoji. Analizirajmo semantiku *naredbe TRY* u sljedećem primjeru:

TRY.py

```
from math import sqrt
try :
  x = eval (input ("Unesite broj: " ))
  print ( 'sqrt(x) =', sqrt (x) )
  try    : print ( '10 /x   =', 10 /x )
  except : print (
     "nije dopušteno dijeliti s nulom")
except ValueError :
  print ("nije definiran drugi "
         "korijen negativnog broja ")
except TypeError  :
  print ("pogreška u tipu argumenta ")
except : print ("sintaksna pogreška "
                "pri unosu odataka")
else :
  # Ako nije aktivirana nijedna
  #          EXCEPT grana
  print ( "U ELSE grani sam!" )
finally : # Ovdje će se UVIJEK doći
  print ( "U FINALLY grani sam!" )
print ( "NASTAVLJAM iza naredbe TRY" )
```

```
>>>
Unesite broj: 125
sqrt(x) = 11.180339887498949
10 /x   = 0.08
U ELSE grani sam!
U FINALLY grani sam!
NASTAVLJAM iza naredbe TRY
```

Nije bilo pogreške pa su izvršene naredbe *grana ELSE* i *FINALLY* te se program nastavio dalje izvršavati.

```
>>>
Unesite broj: -10
nije definiran drugi korijen negativnog
broja
U FINALLY grani sam!
NASTAVLJAM iza naredbe TRY
```

Bila je pogreške pa grana ELSE nije bila aktivirana. Ni *naredba TRY* unutar glavne *TRY grane* nije bila izvršena. Aktivirana je *FINALLY grana* i program se nastavio dalje izvršavati.

```
>>>
Unesite broj: 0
sqrt(x) = 0.0
nije dopušteno dijeliti s nulom
U ELSE grani sam!
U FINALLY grani sam!
NASTAVLJAM iza naredbe TRY
```

Izvršena je prva naredba u *TRY grani*. Bila je pogreška u *naredbi TRY* unutar *TRY grane* pa je izvršena njezina *EXCEPT grana*. Nije bilo pogrešaka u prvoj *TRY grani* pa su izvršene naredbe njezine *ELSE grane* te bezuvjetno *FINALLY grana* i program se nastavio dalje izvršavati.

```
>>>
```

```
Unesite broj: 1 2
sintaksna pogreška pri unosu podataka
U FINALLY grani sam!

NASTAVLJAM iza naredbe TRY
```

Dojavljena je pogreška u unosu ulaznog podatka. Ide se na FINALLY granu i program se nastavio dalje izvršavati.

```
>>>
Unesite broj: '1'
pogreška u tipu argumenta
U FINALLY grani sam!
NASTAVLJAM iza naredbe TRY
```

Dojavljena je pogreška u *EXCEPT grani*, aktivirana je *FINALLY grana* i program se nastavio dalje izvršavati.

Kao drugi primjer, napišimo program za izračunavanje površine trokuta zadanih stranica, koristeći *naredbu TRY*:

TRY_Površina_trokuta.py

```
from math import sqrt
a, b, c = eval (input (
          'Zadaj stranice trokuta '))
s = (a +b +c) /2
try    :
  sqrt(s); sqrt(s-a); sqrt(s-b)
  sqrt(s-c);
  D = s *(s -a) *(s -b) *(s -c)
  print ( 'P =', round (sqrt (D), 4) )
except : print ( 'Nije trokut!' )
```

```
>>>
Zadaj stranice trokuta 3, 4, 5
P = 6.0
>>>
Zadaj stranice trokuta 10, 11, 12
P = 51.5212
>>>
Zadaj stranice trokuta 10, 20, 40
Nije trokut!
>>>
Zadaj stranice trokuta 10, 10, -10
Nije trokut!
```

U ovom smo se programu poslužili „trikom”: napisali smo niz izraza, poziva funkcije `sqrt()` s različitim argumentima, koji svi moraju imati vrijednost veću od `0` da bi zadane stranice `a`, `b` i `c` mogle činiti trokut. Nailaskom na prvi argument koji to ne zadovoljava prekida se daljnje izvršavanje niza naredbi *TRY grane* i prelazi na *EXCEPT granu*.

Selekcija

Sada napišimo program semantički ekvivalentan danom programu, ali uz pomoć složene naredbe „Selekcija”, koju uvodimo u ovom poglavlju:

SELEKCIJA_Površina_trokuta.py

```
from math import sqrt
a, b, c = eval (input (
          'Zadaj stranice trokuta '))
s = (a +b +c) /2
if s > 0 and s-a > 0 and s-b > 0 \
   and s-c > 0 :
   D = s *(s -a) *(s -b) *(s -c);
   print ('P =', round (sqrt (D), 4))
else : print ('Nije trokut!')
```

```
>>>
Zadaj stranice trokuta 10, 11, 12
P = 51.5212
>>>
Zadaj stranice trokuta 10, 10, -10
Nije trokut!
>>>
Zadaj stranice trokuta 10, 20, 40
Nije trokut!
```

Niz poziva funkcije `sqrt()` ovdje je zamijenjen logičkim izrazom:

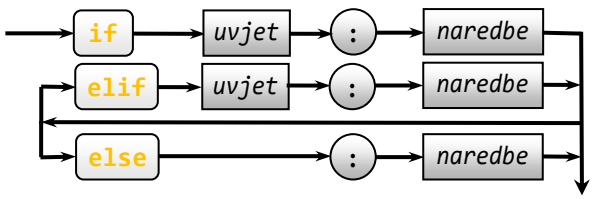
```
s > 0 and s-a > 0 and s-b > 0 and s-c > 0
```

Reći ćemo da je to uvjet jer mu prethodi rezervirana riječ `if` (ako). Slijedi opis sintakse i semantike selekcije.

SINTAKSA

Pravilo pisanja selekcije prikazano je u sljedećem sintaksnom dijagramu, gdje je sintaksna kategorija ***naredbe*** definirana u sintaksi *naredbe TRY*.

selekcija:



Kontekstna pravila pisanja selekcije jednaka su konteksnim pravilima pisanja *naredbe TRY*. Rezervirane riječi `if`, `elif` i `else` pišu se u istoj koloni (jer su dio istog bloka, osnovnog ili na nižoj razini). Dijelovi

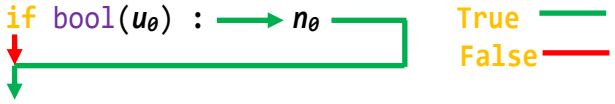
koji počinju s `elif` i `else` zvat ćemo *ELIF* i *ELSE grane* selekcije. Primjeri:

```
if Error : print ("Greška broj",
                  Error )

if x >= 0 : y = x **0.5
else      : print ( "x < 0" )

if   X >  Y : print ( "X >  Y" )
elif X == Y : print ( "X == Y" )
elif X <  Y : print ( "X <  Y" )
```

SEMANTIKA

Značenje pojedinih slučajeva pisanja selekcije (ili „naredbe za odabir"), gdje su u_0 do u_k uvjeti, a n_0, n_1 i e_1 do e_k naredbe, prikazano je sljedećim dijagramima:

1)



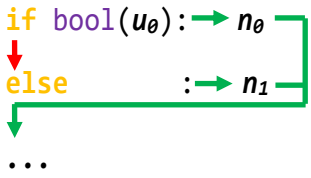

```
... (prva naredba iza selekcije)
```

Prvo se izračuna `bool`(u_0). Ako je jednak `True`, izvršit će se naredbe n_0 i nastaviti na prvoj naredbi iza selekcije. Ako je jednak `False`, naredbe n_0 neće biti izvršene, već se izvršavanje programa nastavlja na prvoj naredbi iza selekcije. Na primjer:

```
a = abs (eval (input ())); A = int(a)
if a % 1 : A += 1
...
```

Varijabli `A` bit će dodano `1` samo ako `a` ima decimalni dio veći od `0`.

2)

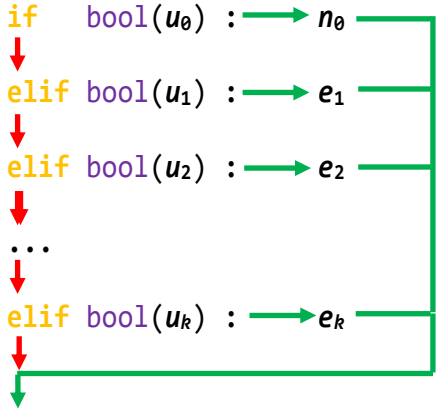


Prvo se izračuna `bool`(u_0). Ako je jednak `True`, izvršit će se naredbe n_0 i nastaviti na prvoj naredbi iza selekcije. Ako je jednak `False`, bit će n_1, naredbe *ELSE grane* i izvršavanje programa će se nastaviti na prvoj naredbi iza selekcije. Na primjer:

```
x = eval (input())
if x >= 0 : y = x**0.5
else      : print ("Broj izvan domene!")
...
```

Ako je `x >= 0`, varijabli `y` bit će pridružena vrijednost drugog korijena iz `x`, inače, bit će ispisana poruka `"Broj izvan domene!"`.

3)

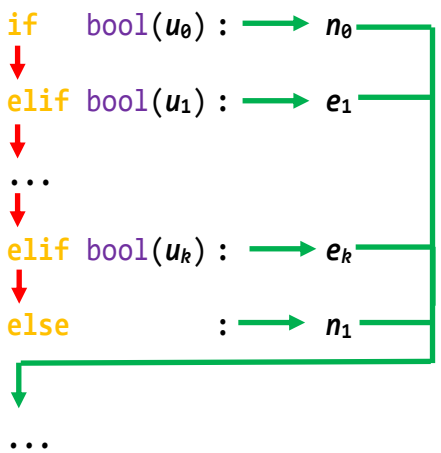


Opet se kreće od glavne grane i računa se U_0 = `bool`(u_0). Ako je U_0 jednako `True`, izvršit će se naredbe n_0 i nastaviti na prvoj naredbi iza selekcije. Ako je U_0 jednako `False`, prelazi se na prvu *ELIF granu* i računa U_1. Ako je jednako `True`, bit će izvršene naredbe e_1, i nastavit će se na prvoj naredbi iza selekcije. Ako je jednako `False`, prelazi se na sljedeću *ELIF granu* i ponavlja postupak. Ako se dosegne posljednja *ELIF grana* i U_k je `False`, nastavlja se s prvom naredbom iza selekcije. Na primjer:

```
from random import *
i = randint (1, 5)
if   i == 1 : B = 'jedan'
elif i == 2 : B = 'dva'
elif i == 3 : B = 'tri'
elif i == 4 : B = 'četiri'
elif i == 5 : B = 'pet'
...
```

Kreće se od glavne grane. Ako je `i==1`, varijabli `B` bit će pridružen string `'jedan'` i nastavlja se s prvom naredbom iza selekcije, a ako nije, prelazi se na prvu *ELIF granu*. S obzirom na to da je `i` broj od `1` do `5` bit će istinit jedan od postavljenih uvjeta.

4)



Značenje selekcije s ovakvom strukturom jednaka je selekciji iz prethodnog slučaja, osim ako su svi uvjeti u_0 do u_k neistiniti, tada će biti izvršene naredbe n_1 *ELSE grane*.

GOVORIMO PYTHONSKI

Mi ćemo u našim programima rabiti *naredbu TRY* koja će imati strukturu s *EXCEPT granom* i, eventualno, s *ELSE granom*. Bit će to na mjestima gdje ne želimo nasilni prekid izvršavanja programa zbog pogreške u unosu podataka, a da to nekom drugom naredbom, na primjer selekcijom, ne možemo nadići. Na primjer, ako pri izvršenju naredbe:

```
a, b, c = eval (input (
'Zadaj stranice trokuta '))
```

upišemo:

```
Zadaj stranice trokuta 11 12 13
...
    11 12 13
        ^
SyntaxError: invalid syntax
```

ili

```
>>> v = eval (input ('Zadaj brzinu, v '))
Zadaj brzinu, v v
NameError: name 'v' is not defined
```

Da bismo to nadišli, možemo rabiti *naredbu TRY*. Na primjer:

```
try    : a, b, c = eval (input (
         'Zadaj stranice trokuta '))
except : print (
         'Pogreška u ulaznim podacima!')
else   :
   # nastavak programa
```

Ako bismo htjeli ispisati preciznu poruku o vrsti pogreške, trebamo imena pogrešaka i rabiti ih u EXCEPT granama naredbe TRY. Na primjer:

```
except ZeroDivisionError: print (
    ('Nedopušteno dijeljenje s nulom!')
```

No, možda u ovim našim „školskim" programima to nije posebno važno, pa ćemo ponoviti izvršavanje programa i upisati ulazne podatke bez pogreške. U praksi, kod „ozbiljnijih" i kompleksnijih programa naredbu TRY ćemo rabiti na mjestima gdje ćemo morati provjeriti valjanost nekih međurezultata koji ne smiju biti izvan domene ili nisu valjanog tipa.

LOGIČKI IZRAZI

Selekcija je prva od nekoliko složenih naredbi Pythona koja sadrži logičke izraze (uvjete) u glavnom dijelu i *ELIF granama*. O vrijednosti napisanih logičkih izraza ovisi što će se dalje raditi u programu. Zbog toga je posebno važno u razvoju algoritma, rješavajući zadani problem, biti siguran da će svi logički izrazi koje smo napisali dati potpuni prijevod, presliku, onoga što smo osmislili. Da bismo u to bili sigurni, najprije moramo biti sigurni da smo u potpunosti usvojili značenje logičkih operacija, koje su u Pythonu definirane za operande bilo kojega tipa, i da nepogrešivo znamo vrijednosti izračunavanja bilo kojeg binarnog izraza. Ako niste sigurni u to, vratite se na prethodno poglavlje.

Često je i „izbjegavanje" uporabe logičkih varijabli. Čak i vrsni programeri umjesto logičkih vrijednosti `False` i `True` češće koriste cijele brojeve `0` i `1`. Ili, ako se već i koriste logičke varijable, onda im se rijetko pridružuje vrijednost izračunavanja složenijeg logičkog izraza već se za to koristi selekcija, pa se ovisno o vrijednosti izraza logičkoj varijabli pridružuje `True` ili `False`. Evo dvaju takvih slučajeva i naredbi koja imaju jednaku semantiku:

```
if uvjet: X = True
else    : X = False
```
⟹ `X = bool(uvjet)`

ili

```
if uvjet: X = False
else    : X = True
```
⟹ `X = not bool(uvjet)`

Na primjer, umjesto selekcije:

```
if (63<Temp) and (Temp<78) : T = True
else                       : T = False
```

možemo napisati samo jednu naredbu dodjeljivanja:

```
T = 63 < Temp < 78
```

Ako je uvjet logički izraz, funkciju `bool()` treba izostaviti. Na primjer, umjesto selekcije:

```
if G % 4 == 0 and \
   G % 100 <> 0 or G % 400 == 0 :
   Pr_god = True
else : Pr_god = False
```

treba pisati:

```
Pr_god = G % 4 == 0 and \
         G % 100 <> 0 or G % 400 == 0
```

Da se pojam logičke vrijednosti ne prihvaća u svom punom značenju, svjedoče nam primjeri pisanja logičkih izraza, najčešće kao uvjeta u selekciji, gdje se vrijednost logičke varijable uspoređuje s `True` ili `False`. Ako su `Ok` i `Kraj` logičke varijable, evo primjera semantički ekvivalentnih selekcija:

```
if Ok    == True    : naredbe
   → if Ok : naredbe
if Kraj == False   : naredbe
   → if not Kraj : naredbe
```

Uvođenjem selekcije dobiven je mehanizam za upravljanje tokovima izvršavanja programa. To, zajedno s ostalim elementima jezika, pruža mogućnosti pisanja programa za rješavanje složenijih problema i za izražavanje složenijih algoritama računanja. Naredba za odabir, zajedno s naredbom za prekid izvršavanja programa, omogućuje uspostavljanje nadzora nad radom programa i automatizme njegovoga prekida.

No, sada se mogu lakše provjeravati i ulazne vrijednosti. Njihovom provjerom može se zaštititi program od nepotrebnoga rada ili, što je možda važnije, od pogrešnih rezultata.

Često se, u knjigama koje opisuju Python, naredbe unutar pojedinih grana pišu kao blokovi, bez obzira što su dio niza naredbi. Na primjer:

```
if x >= 0 :
   y := x **0.5
else      :
   print ( "x < 0" )
```

Mi ćemo ih pisati kao niz naredbi, u istoj liniji. Na primjer:

```
if x >= 0 : y := x **0.5
else      : print ( "x < 0" )
```

VALJANOST ULAZNIH PODATAKA (2)

U prethodnom smo poglavlju izveli općenitu strukturu dijela programa za provjeru valjanosti ulaznih podataka koji je sadržavao varijablu `Ok`,

```
Ok = '' if uvjet else ' '
```

i naredbu

```
exec ( Ok and Input )
```

gdje je `Input` tekst koji sadrži naredbe za unos i provjeru podataka. Sada te dvije naredbe možemo zamijeniti jednom:

```
if not (uvjet) :
  # ... naredbe
  exec ( Input )
```

Na primjer, u sljedećem dijelu programa unosimo polumjer kruga koji mora biti veći od `0`.

```
Input = """
  r = eval (input (
  'Zadaj polumjer kruga ') )
  if r <= 0 : # not (r > 0)
    print (
    "Polumjer mora biti veći od 0! "
    "Ponovite unos!" )
    exec ( Input ) """
  exec ( Input )
>>>
  Zadaj polumjer kruga -5
  Polumjer mora biti veći od 0! Ponovite
  unos!
  Zadaj polumjer kruga 0
  Polumjer mora biti veći od 0! Ponovite
  unos!
  Zadaj polumjer kruga 4.5
```

Ne može se dati općeniti „recept“ za provjeru ulaznih podataka. Morat ćemo se dovijati od slučaja do slučaja. Evo primjera kako provjeriti valjanost ulaznih podataka, koeficijenata *a*, *b* i *c* kvadratne funkcije ax^2+bx+c, $a \neq 0$, bez uporabe *naredbe TRY*. Za unos koeficijenata rabimo funkciju `Inp()` iz modula `Moj_modul`.

```
>>> abc = """
from Moj_modul import *
a, b, c = Inp ('a, b, c? ')
Ok      = '' if a else ' '
print ('a = 0. Ponovi upis!' *bool(Ok))
exec (Ok and abc) """
```

```
>>> exec (abc)
a, b, c? 0, 1, 2
a = 0. Ponovi upis!
a, b, c? -1, 1, 2
```

Iz ovoga primjera možemo izvesti općenitu strukturu dijela programa za provjeru valjanosti ulaznih podataka koji će sadržavati varijablu Ok,

```
Ok = '' if uvjet else ' '
```

i naredbu

```
exec (Ok and Input)
```

gdje je Input tekst koji sadrži naredbe za unos i provjeru podataka. Primijetiti da smo na taj način „nesvjesno" uveli rekurziju, jer imamo dio programa koji poziva sam sebe. Unesena vrijednost će biti korektna kada kontrolna varijabla Ok postane '' , što je ujedno argument naredbe EXEC, pa se više ne izvršava niz naredbi Input. Kasnije ćemo vidjeti da postoji drugi način za provjeru ulaznih podataka primjenom *WHILE petlje*.

Često osim provjere domene ulaznih podataka trebamo biti sigurni i da je podatak određenog tipa. Na primjer, funkcija faktorijel je definirana za nenegativni cijeli broj (cijeli broj veći ili jednak 0), pa ako napišemo dio programa za unos broja za kojeg treba izračunati faktorijel:

```
>>> Input = """
n   = eval (input(
'Računam faktorijel za cijeli broj n. '
'n >= 0 ' ))
Ok  = '' if n >= 0 else ' '
print ('n < 0. Ponovi upis!' *bool(Ok))
exec (Ok and Input) """
>>> exec (Input)
Računam faktorijel za cijeli broj n, n
>= 0 -1
n < 0. Ponovi upis!
Računam faktorijel za cijeli broj n, n
>= 0 12.5
>>> n          12.5
```

Dakle, ako je unesena vrijednost negativni broj, ponavlja se unos sve dok ne unesemo 0 ili broj veći od 0. Ali, nismo provjerili je li broj cijeli. Možda bi u ovom slučaju rješenje bilo da dodamo provjeru decimalnog dijela unesenog broja, pa ako nije jednak nuli, ponoviti upis:

```
Ok = '' if n>=0 and not(n %1) else ' '
```

Uz još neke dopune sada bi dio programa za unos bio:

```
>>> Input = """
n  = eval (input (
'Računam faktorijel za cijeli broj n, '
'n >= 0 ' ))
Ok = '' if n>=0 and not(n % 1) else ' '
print ('n < 0 ili nije cijeli broj. '
       'Ponovi upis!' *bool(Ok))
n = int(n)
exec (Ok and Input)  """
>>> exec (Input)
Računam faktorijel za cijeli broj n, n
>= 0 12.5
n < 0 ili nije cijeli broj. Ponovi upis!
Računam faktorijel za cijeli broj n, n
>= 0 12.0
```

Ovdje smo morali dodati i pretvorbu u cjelobrojnu vrijednost ako bude učitan realni broj čiji je decimalni dio jednak 0.0.

PONAVLJANJE IZVRŠAVANJA NAREDBI (2)

U prethodnom smo poglavlju pokazali kako se niz naredbi sadržanih u teksu ***A*** može ponoviti zadani broj puta, ***n***. Sada, uporabom selekcije, možemo graditi nizove naredbi čije će se izvršavanje ponavljati na dva načina:

1) Sve dok postavljeni uvjet ne postane istinit.
2) Sve dok je postavljeni uvjet istinit.

Struktura dijela programa za provjeru valjanosti ulaznih podataka primjer je ponavljanja izvršavanja naredbi prvog načina, dok će struktura ponavljanja izvršavanja naredbi na drugi način biti:

```
... (inicijalizacija varijabli )
WHILE = """
if uvjet :            if uvjet :
  naredbe               naredbe
  exec ( WHILE ) """    exec ( WHILE )
exec ( WHILE )
```

Radi bolje preglednosti strukturu naredbi smo prikazali desno. Kao što ćemo vidjeti u narednom poglavlju, dana struktura dijela programa semantički je ekvivalentna WHILE petlji, pa smo za ime tekstualne varijable, ne slučajno, izabrali WHILE.

UVJETNI IZRAZI I SELEKCIJA

Uvjetni izrazi čija se vrijednost izračunavanja pridružuje nekoj varijabli *y* mogu se prevesti u selekciju. Ako općenito napišemo

```
y = I1 if U1 else ... In if Un else I
```

selekcija ekvivalentna ovoj naredbi može se napisati kao

```
if   U1  : y = I1
elif U2  : y = I2
...
elif Un  : y = In
else     : y = I
```

Na primjer, pridruživanje

```
Kn = 15 *T if T <= 3 else 45 +3 *(T -3)
```

može se napisati kao selekcija s ekvivalentnim značenjem:

```
if T <= 3 : Kn = 15 *T
else      : Kn = 45 +3 *(T -3)
```

Ili, dio programa **Parking.py** u kojem se izračunava usluga parkiranja do 48 sati može se napisati kao selekcija s ekvivalentnim značenjem:

```
if   T <=  0.1 : C = 0
elif T <=  1.0 : C = 27
elif T <=  2.0 : C = 47
elif T <=  3.0 : C = 70
elif T <=  6.0 : C = 78
elif T <= 12.0 : C = 110
elif T <= 24.0 : C = 150
else           : C = 210
```

Ako bismo sada rekli što koristiti u programiranju, uvjetne izraze ili selekciju, odgovor je: ovisi o strukturi rješenja problema. Ako u rješenju treba izračunati jednu vrijednost (funkciju) koja je definirana po segmentima, kao što je bilo u oba primjera, prednost bismo dali uvjetnim izrazima.

P R O G R A M I

Dalje će pisanje programa (skripti) biti nezamislivo bez selekcije. Na ovom mjestu nismo još u mogućnosti dati "prave" primjene, jer ne znamo mnoge naredbe Pythona, pa dajemo nekoliko jednostavnih programa koji zasad u dovoljnoj mjeri prikazuju primjene selekcije. Prije toga dodajmo u **Moj_modul.py** logičku funkciju `Int()` koja provjerava je li podatak tipa `int`:

```
Int   = lambda x : type(x) == int
```

POVRŠINA TROKUTA (2)

Za zadane stranice trokuta, *a*, *b* i *c*, treba izračunati njegovu površinu. U programima

TRY-Površina_trokuta.py
SELEKCIJA-Površina_trokuta.py

rabili smo Heronov obrazac (formulu),

$$P = \sqrt{s(s-a)(s-b)(s-c)}$$

gdje je:

$$s = (a+b+c)/2$$

Tada smo koristili iznimke provjeravajući je li vrijednost `s` ili bilo kojeg faktora ispod korijena manja od nule. Ako jest, (a što ako je jednaka nuli?) bila bi dojavljena pogreška. U selekciji smo provjeravali jesu li `s` i svi faktori veći od nule, pa ako jesu, izračunali smo površinu. U ovom ćemo rješenju rabiti činjenicu da stranice *a*, *b* i *c* mogu činiti trokut ako vrijedi

$$s > max\ (a,\ b,\ c)$$

Površina_trokuta_2.py

```
a, b, c = eval (input (
          'Upiši stranice trokuta '))
s       = (a +b +c) /2
if s > max (a, b, c) :
   P = round ((s *(s-a) *(s-b)
       *(s-c))**0.5, 4)
   print ('a =', a, 'b =', b,
          'c =', c, '  P =', P)
else : print ('NIJE TROKUT!')
```

```
>>>
Upiši stranice trokuta 10, 10, 12
a = 10 b = 10 c = 12   P = 48.0
>>>
Upiši stranice trokuta 1, 2, 5
NIJE TROKUT!
```

TREĆI KUT TROKUTA (2)

Ako su zadana dva kuta trokuta u obliku S.MM, gdje su S stupnjevi, a MM minute, izračunati vrijednost trećeg kuta.

Treći_kut.py

```
# TREĆI KUT TROKUTA
a, b   = eval (input ('Zadaj dva '
         'kuta trokuta u obliku S.MM '))
Sa, Sb = int(a), int(b)
Ma, Mb = round (100 *(a -Sa)), \
         round (100 *(b -Sb))
if 0 <= Ma < 60 > Mb >= 0 :
  M = (Sa +Sb)*60 +Ma +Mb
  if M < 180*60 :
    c      = 180*60 -M
    Sc, Mc = divmod (c, 60)
    print ('treći kut =', Sc +Mc/100)
  else : print ('Greška. Nije trokut!')
else : print ('Greška u podacima!')
```

NUL-TOČKE KVADRATNE FUNKCIJE (5)

Sada smo u mogućnosti dati potpuni program za izračunavanje nula kvadratne funkcije $f(x) = ax^2 + bx + c$, $a \neq 0$. Znamo da nule te funkcije nalazimo formulom:

$$x_{1,2} = \frac{-b \pm \sqrt{D}}{2a}$$

gdje je $D = b^2 - 4ac$, i naziva se diskriminanta. Ako je $D \geq 0$, nule kvadratne funkcije su realne. U suprotnom su konjugirano kompleksne.

Kvadratna_2.py

```
# Nule kvadratne funkcije
# a*x**2 +b*x +c = 0, a <> 0
a, b, c = eval (input (
    'Upiši koeficijente kvadr. jed. '))
if a != 0 :
   _2a = 2*a;  D  = b**2 -4*a*c
   a1 = round (-b/_2a, 2)
   a2 = round (abs(D)**0.5/_2a, 2)
   if D > 0 :
      x1 = a1 +a2; x2 = a1 -a2
      x1, x2 = min(x1, x2), max(x1, x2)
      # x1 <= x2
      print ('x1 =', x1, ' x2 =', x2)
   elif D == 0 :
      x1 = x2 = a1
      print ('x1 = x2 =', x1)
   else : # D < 0
      print (
   'Rješena su konjugirano kompleksna')
      x1 = a1 +a2*1j; x2 = a1 -a2*1j
      print ('x1 =', x1, ' x2 =', x2)
else : print (
      "Nije kvadratna jednadžba (a=0)")
```

Ako promatramo strukturu ovoga programa, uočavamo ugniježđenu selekciju u glavnoj grani prve selekcije. Uvlačenje pojedinih blokova traži posebnu koncentraciju, jer se pomicanjem nekih naredbi može dobiti drugo značenje. Na primjer, ako naredbu za ispis napisane u *ELSE grani* ugniježđene selekcije pamaknemo dva mjesta ulijevo, ne bi bila dojavljena pogreška, ali bi imala značenje prve naredbe iza ugniježđene selekcije, pa bismo osim ispisa ako je bilo D >= 0 imali još jedan.

RASTUĆI NIZ BROJEVA (2)

Zadane brojeve treba ispisati u rastućem nizu. Već za tri broja, A, B i C, početnici često taj problem pokušavaju riješiti tražeći minimalnu, srednju i maksimalnu vrijednost, ili ispisuju složene logičke izraze u kojima uspoređuju vrijednosti varijabli. Ako i dođu do „rješenja" koje „radi", najčešće je to niz selekcija:

```
A, B, C = eval (input (
          'Unesi tri broja '))
if A < B < C : print (A, B, C)
if A < C < B : print (A, C, B)
if B < A < C : print (B, A, C)
if B < C < A : print (B, C, A)
if C < A < B : print (C, A, B)
if C < B < A : print (C, B, A)
>>>
Unesi tri broja 6, 2, 1        1 2 6
>>>
Unesi tri broja 10, 9, 8       8 9 10
```

Osim što je napisano šest složenih logičkih izraza, uz veliku vjerojatnost da se pogriješi pišući imena varijabli, rješenje nije dobro jer neće dati rezultat ako su bilo koje dvije zadane vrijednosti jednake!

```
>>>
Unesi tri broja 10, 2, 10
>>>
```

Nema ispisa! Dobro, pogriješili smo, ispravimo to s preuređenim selekcijama:

```
if A <= B <= C : print (A, B, C)
```

```
if A <= C <= B : print (A, C, B)
if B <= A <= C : print (B, A, C)
if B <= C <= A : print (B, C, A)
if C <= A <= B : print (C, A, B)
if C <= B <= A : print (C, B, A)
```

Testirajmo ovo rješenje prvo s tri različite vrijednosti:

```
>>>
Unesi tri broja 10, 9, 8
8 9 10
```

Uređeni niz brojeva dobiven je aktiviranjem ispisa selekcije s uvjetom C <= B <= A. Zadajmo dva jednaka broja:

```
>>>
Unesi tri broja 10, 2, 10
2 10 10
2 10 10
```

Pogreška! Postoje dva istinita uvjeta, pa je dvaput ispisan niz uređenih brojeva! No, pogledajmo rezultate ako su sva tri broja jednaka:

```
>>>
Unesi tri broja 7, 7, 7
7 7 7
7 7 7
7 7 7
7 7 7
7 7 7
7 7 7
```

Svih 6 uvjeta će biti istiniti! Uređeni niz je ispisan 6 puta! Zaključujemo da je nedostatak ovog rješenja što će u slučaju dva jednaka broja biti istinita dva uvjeta, a u slučaju tri jednaka broja, svih šest. Ako uvedemo *ELIF grane* problem će biti riješen, jer nailaskom na prvi istinit uvjet bit će ispisan rezultat i preskočit će se preostale grane:

```
if   A <= B <= C : print (A, B, C)
elif A <= C <= B : print (A, C, B)
elif B <= A <= C : print (B, A, C)
elif B <= C <= A : print (B, C, A)
elif C <= A <= B : print (C, A, B)
elif C <= B <= A : print (C, B, A)
>>>
Unesi tri broja 10, 9, 8        8 9 10
>>>
Unesi tri broja 10, 2, 10       2 10 10
>>>
Unesi tri broja 7, 7, 7         7 7 7
```

Ovo rješenje je korektno. No, kako bismo riješili, na primjer, problem ispisa rastućeg niza 4, 5 ili više brojeva? Lako se može dokazati da bi za n brojeva bilo $n!$ uvjeta (grana) selekcije. Na primjer, za n=2 imali bismo 2=2! uvjeta, za n=3 imali smo 6=3!, za n=4 24 uvjeta i za n=5 120 uvjeta!

Problem uređenja niza brojeva rastući, ili opadajući, dio je teorije algoritama, posebno sortiranja. Ovdje dajemo dva rješenja koja su korektna i za više od tri vrijedosti. Dajemo primjer s četiri vrijednosti, a, b, c i d.

U prvom se postupku, poznatom kao „sortiranje razmjenom", u prvom koraku, pretpostavi da je a najmanje. Potom se u tri selekcije uspoređuje s b, c i d. Ako se pronađe vrijednost manja od a, razmijene se vrijednosti. Na kraju će prvog koraka a sadržavati minimalnu vrijednost. Postupak se ponavlja s varijablom b, koja će se usporediti s preostalim varijablama i na kraju će sadržavati minimum od b, c i d, itd. U drugom se postupku, poznatom kao "sortiranje u valovima", uspoređuju susjedne vrijednosti i razmjenjuju ako je prva veća od druge. U koraku (1) uspoređuju se a i b, b i c, te c i d. Na kraju će varijabla d sadržavati maksimalnu vrijednost od a, b, c i d. Postupak se ponavlja, korak (2), nad varijablama a, b i c. Na kraju će c sadržavati maksimalnu vrijednost itd.

Rastući_niz.py

```
T = '\t'
try :
  A, B, C, D = eval (input (
               'Unesi 4 broja '))
  # 1. SORTIRANJE RAZMJENOM
  a, b, c, d = A, B, C, D
  (1)
  if b < a : a, b = b, a
  if c < a : a, c = c, a
  if d < a : a, d = d, a
  #  a = min (a, b, c, d)
  (2)
  if c < b : b, c = c, b
  if d < b : b, d = d, b
  #  b = min (b, c, d)
  (3)
  if d < c : c, d = d, c
  #  c = min (c, d)
  print ('1.', T, a, T, b, T, c, T, d)
  # 2. SORTIRANJE U VALOVIMA
  a, b, c, d = A, B, C, D
```

```
    (1)
    if a > b : a, b = b, a
    if b > c : b, c = c, b
    if c > d : c, d = d, c
    #  d = max (a, b, c, d)
    (2)
    if a > b : a, b = b, a
    if b > c : b, c = c, b
    #  c = max (a, b, c)
    (3)
    if a > b : a, b = b, a
    #  b = max (a, b)
    print ('2.', T, a, T, b, T, c, T, d)

  except : print ("Pogrešan unos!")
```

```
>>>
Unesi 4 broja -10, 20, -1.5, 3.2
1.  -10   -1.5   3.2   20
2.  -10   -1.5   3.2   20
>>>
Unesi 4 broja 4, 3, 2, 1, 1
Pogrešan unos!
```

Najveća zajednička mjera

Poznat je problem računanja najveće zajedničke mjere dvaju pozitivnih cijelih brojeva. Često se za to koristi Euklidov algoritam, premda nije posebno efikasan ako je velika razlika ulaznih brojeva. Ako su `M` i `N` dva cijela broja veća od nule, ponavlja se niz naredbi sadržan u tekstu `Euclid`:

```
Euclid = """
if m <> n :
  if m  > n : m -= n
  if n  > m : n -= m
  exec (Euclid) """
exec (Euclid)
```

Na kraju je `m` ili `n` (vrijedi `m==n`) najveća zajednička mjera učitanih vrijednosti `M` i `N`. Na primjer, za `M=24; N=18` najveća zajednička mjera je 6, a za `M=31; N=97` najveća zajednička mjera je 1, jer su `M` i `N` prosti (prim) brojevi.

Ovdje ćemo problem riješiti na efikasniji način, proširenjem značenja Euklidovog algoritma. Ako su `m` i `n` dva cijela broja veća od nule, ponavljat će se niz naredbi:

```
i = min (m, n)
if m > n and m % i : m %= i
if n > m and n % i : n %= i
```

sve dok uvjet

```
m % i == n % i == 0
```

ne postane istinit. Najveća zajednička mjera bit će jednaka `i`.

NZM.py

```
# Najveća zajednička mjera
from Moj_modul import *
Inp = """
m, n = Input (
'Zadaj dva cijela broja veća od 0 ')
Ok = Int(m) and Int(n) and \
     m > 0 and n > 0
if not Ok : exec (Inp) """
exec (Inp)
print ('Nzm (', m, ',', n, ') =',
        end = ' ')
NZM = """
i = min (m, n)
if m > n and m % i : m %= i
if n > m and n % i : n %= i
if m % i or n % i  : exec (NZM) """
exec (NZM);  print (i)
```

```
>>>
Zadaj dva cijela broja veća od 0 24, 18
Nzm ( 24 , 18 ) = 6
>>>
Zadaj dva cijela broja veća od 0 13, 97
Nzm ( 13 , 97 ) = 1
>>>
Zadaj dva cijela broja veća od 0
2*3*5*7*9, 13*11*7
Nzm ( 1890 , 1001 ) = 7
>>>
Zadaj dva cijela broja veća od 0
2*3*5*7*11, 13*17*5*7
Nzm ( 2310 , 7735 ) = 35
```

No, modul `math` sadrži funkciju `gcd()` – najveću zajedničku mjeru dvaju cijelih brojeva `x` i `y`:

```
>>> help (gcd)
gcd(...)
  gcd(x, y) -> int
  greatest common divisor of x and y
```

Provjerimo rezultate iz naših primjera:

```
>>> gcd (18, 24)                    6
>>> gcd (13, 97)                    1
>>> gcd (2*3*5*7*9, 13*11*7)        7
>>> gcd (2*3*5*7*11, 13*17*5*7)    35
```

Skraćivanje razlomka

Skratiti razlomak *m*/*n*, gdje su *m* i *n* prirodni brojevi. Rezultat prikazati kao mješoviti razlomak.

Skraćivanje_razlomka.py

```
from Moj_modul import *

Unos = """
M, N = Input (
   'Unesi dva cijela broja veća od 0 ')
Ok = Int(M) and M > 0 and Int(N) \
            and N > 0
if not Ok : exec (Unos) """

exec (Unos)
q = gcd (M, N);  m, n = M //q, N //q
print (M, '/', N, ' = ', sep = '',
       end = ' ')
if   m % n == 0 : print ( m //n)
elif m > n      :
  print ("%d %d/%d" % (m //n, m %n, n))
else            :
  print ( "%d/%d" % (m, n) )
```

```
>>>
Unesi dva cijela broja veća od 0 11*5,
22*5
55/110 =  ½

>>>
Unesi dva cijela broja veća od 0 1234,
322
1234/322 =  3 134/161

>>>
Unesi dva cijela broja veća od 0 77, 33
77/33 =  2 1/3
```

Nul-točke kvadratne funkcije (6)

Evo još jednog, „kratkog" rješenja, u kojem smo nule prikazali kao realne, ako je `D>=0`, inače, kao konjugirano kompleksne. Koristili smo uvjetni izraz za izračunavanje komponente `a2`.

Kvadratna_6.py

```
a, b, c = eval (input (
           'Zadaj koef. kvadr. jed. '))

if a :
  D  = b**2 -4*a*c;  _2a = 2.0*a;
  a1 = -b/_2a
  a2 = D**0.5/_2a if D >= 0 else \
       (-D)**0.5/_2a *1j
  x1 = a1 +a2;  x2 = a1 -a2
  print ('x1 =', x1, ', x2 =', x2)
else : print (
  "Nije kvadratna jednadžba (a=0)")
```

Funkcija zadana po segmentima

Funkcija *f(x)* zadana je po segmentima:

$$f(x)=\begin{cases} 0 & x \le 0 \\ x^2 & 0 < x \le 1 \\ 1 & 1 < x \le 2 \\ x^2 - 6x + 9 & 2 < x \le 3 \\ 0 & x > 3 \end{cases}$$

Treba napisati dio programa koji će za danu vrijednost varijable *x* izračunati *f(x)*. Ovo je „školski primjer" uporabe uvjetnih izraza u rješenju.

Funkcija.py

```
# Funkcija zadana po segmentima
f = lambda x : \
    0           if x <= 0 else \
    x**2        if x <= 1 else \
    1           if x <= 2 else \
    x**2 -6*x +9 if x <= 3 else \
    0
x = eval (input ('Zadaj x '))
print (f (x))
```

```
>>>
Zadaj x 2.55          0.20250000000000057
>>> f(2.55)           0.20250000000000057
>>> f(1.5)                              1
>>> f(0.77)                        0.5929
```

Tablica

U sljedećem smo programu, ispisu tablice brojeva od 1 do n, njihovih kvadrata i korijena, prikazali dva načina ponavljanja izvršavanja naredbi. Unos gornje granice prikaza brojeva u tablici primjer je ponavljanja naredbi, sve dok je ispunjen postavljeni uvjet `i <= n`.

Tablica_2.py

```
# TABLICA i, i**2, i**0.5,
# i = 1,2,..., n, n > 0

from Moj_modul import *
```

```
UNOS = """
n  = Input (
     'Ispisujem tablicu od 1 do n = ')
Ok = type (n) == int and n > 0

if not Ok :
  print (
    'Pogreška u ulaznom podatku. '
    ' Ponovi unos' )
  exec ( UNOS )  """

exec (UNOS)
print (' i       i**2       i**0.5',
       '\n', '-'*26 )
i = 1

TABLICA = """
if i <= n :
  print ("%2d %10d  %11.4f"
         % (i, i**2, i**0.5) )
  i += 1
  exec ( TABLICA ) """
exec ( TABLICA )
```

```
>>>
Ispisujem tablicu od 1 do n = 10

 i       i**2       i**0.5
--------------------------
 1          1       1.0000
 2          4       1.4142
 3          9       1.7321
 4         16       2.0000
 5         25       2.2361
 6         36       2.4495
 7         49       2.6458
 8         64       2.8284
 9         81       3.0000
10        100       3.1623
```

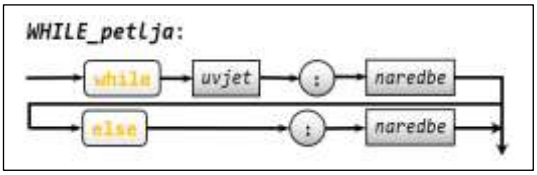
U ovom primjeru programa, s obzirom da će uvijek biti ispisan prvi red kad je `i=1`, struktura je rješenja bliža prvom načinu ponavljanja izvršanja naredbi:

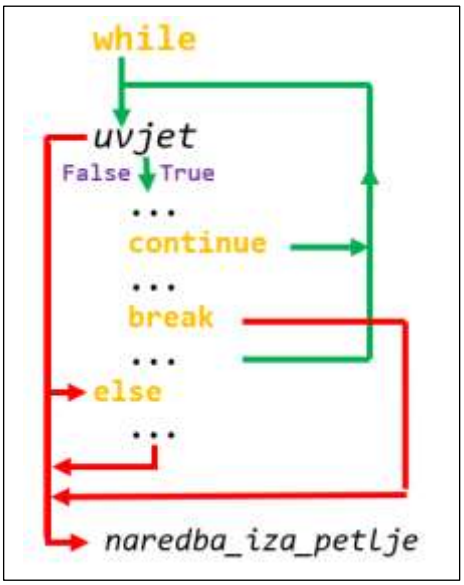
```
TABLICA2 = """
i += 1
print ("%2d %10d  %11.4f"
       % (i, i**2, i**0.5) )
# ponovi ispis ako nije dosegnut n:
if i != n : exec (TABLICA2) """
i = 0
exec ( TABLICA2 )
```

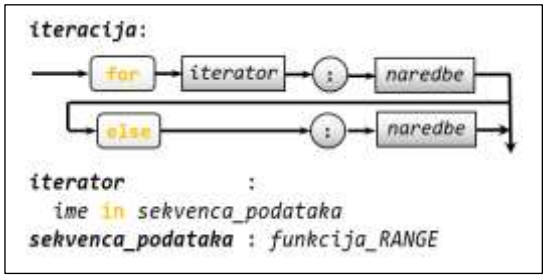
5.

PETLJE

Primjenom selekcije i naredbe EXEC možemo pisati programe koji sadrže grupu naredbi čije se izvršavanje ponavlja. Međutim, postoje dvije složene naredbe, WHILE petlja i FOR petlja, koje imaju ugrađene mehanizme za to. Te su dvije naredbe poznate u većini jezika za programiranje. Ali, njihova sintaksa i semantika u Pythonu prilično se razlikuje od onih u drugim jezicima. U odjeljcima GOVORIMO PYTHONSKI i PROGRAMI to je zorno prikazano u velikom broju primjera.







Duljina_krivulje.py

```
# Duljina krivulje funkcije f(x)
# na intervalu [a, b]
from math import *;  π = pi
y  = lambda x : eval (fx)
while 1 :
  fx = input ('Upiši f(x) =  ')
  try :
    a, b = eval( input(
           'interval a, b ( π ) ') )
    y(a), y(b)
    break
  except : print ( 'Pogreška!' )

f  = lambda x : complex (x, y(x))
P = 1e-6; 'točnost (preciznost)'
n = 2; L0 = 0;
while True :
  d = abs (a-b) /n;  A = f(a)
  L = 0; x = a+d

  while x < b+d/2 :
    if x > b -d : x = b
    B  = f(x); L += abs(A-B); A = B
    x += d
  if L != L0 and abs(L-L0) < P : break
  L0 = L; n *= 2
print( 'd =', L )
```

```
>>>
Upiši f(x) =  sin(x) +cos(x)

interval a, b ( π ) 0, 2*π

d = 8.73775247166
```

Uvod

U prethodnom smo poglavlju pokazali kako se mogu provjeravati ulazni podaci i kako ponoviti unos ako nisu korektni, Tablica_2.py. Također smo pokazali kako se može ponavljati izvršavanje niza naredbi. U oba smo slučaja koristili tekst kao program koji je sadržavao selekciju i *naredbu EXEC*, kao, na primjer, u sljedećem programu:

EXEC_Tablica.py

```
# TABLICA i, i**2, i**0.5, i = 1, ..., n
UNOS = """
n = eval( input(
    'Ispisujem tablicu od 1 do n = ' ))
Ok = type (n) == int and n > 0
if not Ok : exec ( UNOS ) """
exec (UNOS); print (
    '\n', ' i       i**2       i**0.5',
    '\n', '-'*26, sep = '' )
i = 1
TABLICA = """
if i <= n :
  print( "%2d %10d  %11.4f"
         % (i, i**2, i**0.5) )
  i += 1
  exec( TABLICA )  """
exec( TABLICA )
```

```
>>>
Ispisujem tablicu od 1 do n = 10

 i       i**2       i**0.5
--------------------------
 1          1       1.0000
 2          4       1.4142
 3          9       1.7321
 4         16       2.0000
 5         25       2.2361
 6         36       2.4495
 7         49       2.6458
 8         64       2.8284
 9         81       3.0000
10        100       3.1623
```

Međutim, postoji jedna strukturirana naredba u Pythonu čije značenje odgovara dijelovima UNOS i TABLICA. To je *WHILE petlja* i najčešće je rabljena naredba za ponavljanje izvršavanja niza naredbi u većini jezika za programiranje. Sljedeći je program napisan uz pomoć *WHILE petlje* i semantički je ekvivalentan programu **EXEC-Tablica.py**.

WHILE_Tablica.py

```
# TABLICA i, i**2, i**0.5, i = 1,
# 2,..., n
Ok = False
while not Ok : # UNOS
   n  = eval( input(
     'Ispisujem tablicu od 1 do n = '))
   Ok = type (n) == int and n > 0
print (
    '\n', ' i       i**2       i**0.5',
    '\n', '-'*26, sep = '' )
i = 1
while i <= n : # TABLICA
  print( "%2d %10d  %11.4f"
         % (i, i**2, i**0.5) )
  i += 1
```

```
>>>
Ispisujem tablicu od 1 do n = 10
 i       i**2       i**0.5
--------------------------
 1          1       1.0000
 2          4       1.4142
 3          9       1.7321
 4         16       2.0000
 5         25       2.2361
 6         36       2.4495
 7         49       2.6458
 8         64       2.8284
 9         81       3.0000
10        100       3.1623
```

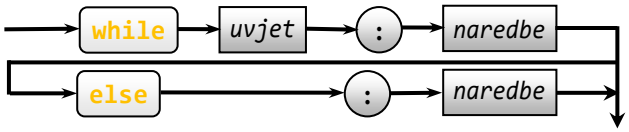
WHILE petlja

WHILE petlja je strukturirana (složena) naredba. Njezinom primjenom moguće je ponavljati izvršavanje naredbi bloka. Sada preostaje da definiramo njezinu sintaksu i semantiku.

SINTAKSA

Pravilo pisanja *WHILE petlje* dano je sljedećim sintaksnim dijagramom:

***WHILE_petlja*:**



SEMANTIKA

WHILE petlju općenito možemo napisati bez *ELSE grane*:

```
while uvjet : naredbe
naredba_iza_petlje
```

ili s ELSE granom:

```
while uvjet : naredbe
else        : naredbe
naredba_iza_petlje
```

značenje je ekvivalentno nizu naredbi:

```
While = """          While_else = """

if uvjet :           if uvjet :
   naredbe              naredbe
   exec(While)          exec (While_else)
"""                  """
exec (While)         exec (While_else)
naredba_iza_petlje   naredbe
                     naredba_iza_petlje
```

Ili, riječima, značenje *WHILE petlje* jest: izvršavanje naredbi bit će ponavljano sve dok je vrijednost uvjeta istinita. Kad vrijednost uvjeta postane `False` (što može biti već pri njezinom prvom izvršavanju), daljnje se izvršavanje programa nastavlja na naredbama iza `else` (ako postoji), pa na prvoj naredbi iza petlje, odnosno na prvoj naredbi iza petlje ako ne postoji *ELSE grana*. Iz svega toga slijedi da mogu postojati dva posebna slučaja upotrebe *WHILE petlje*:

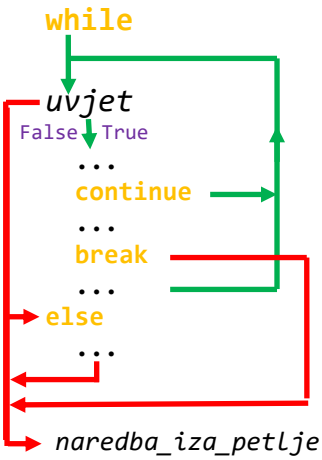
1) Naredba unutar petlje neće biti izvršena nijedanput ako je vrijednost logičkog izraza pri dolasku na početak petlje jednaka `False`.
2) Naredba unutar petlje bit će izvršavana beskonačno ("beskonačna petlja") ako je vrijednost logičkog izraza uvijek jednaka `True`.

Posljedica slučaja (2) jest da logički izraz *WHILE petlje* mora sadržati bar jednu varijablu i da naredbe unutar petlje moraju mijenjati vrijednost varijabli logičkog izraza i time osigurati da vrijednost logičkog izraza u jednom trenutku postane jednaka `False`. Na primjer, pogledajmo što će biti ispisano poslije izvršenja dva dijela programa:

```
i = 0
while i < 10 : print( i, end = ' ' ); \
               i += 1
print( '\n*** (izvan WHILE petlje)' )
>>>
0 1 2 3 4 5 6 7 8 9
  *** (izvan WHILE petlje)
i = 0
while i < 10: print( i, end = ' ' ); \
              i += 1
else        : print( 'kraj, i =', i )
print( '*** (izvan WHILE petlje)' )
>>>
0 1 2 3 4 5 6 7 8 9 kraj, i = 10
*** (izvan WHILE petlje)
```

Naredba BREAK i naredba CONTINUE

Postoje dvije naredbe, *naredba BREAK* i *naredba CONTINUE*, koje se smiju pisati samo unutar naredbi *WHILE petlje*. U svim drugim slučajevima bila bi dojavljena sintaksna pogreška. Značenje ovih dviju naredbi prikazano je na sljedećem crtežu:



Naredba BREAK ima značenje prekida izvršavanja naredbi petlje i nastavak na prvoj naredbi iza petlje. *Naredba CONTINUE* ima značenje vraćanja na početak *WHILE petlje*.

WHILE_continue_break.py

```
i = 0
while i < 100 :
  if 1 <= i <= 3 or 11 <= i <= 13 :
     i += 1; continue
  print( i, end = ' ' )
  if i >= 20 : break
  i += 1
>>>
0 4 5 6 7 8 9 10 14 15 16 17 18 19 20
```

„REPEAT petlja"

Neki jezici za programiranje imaju još jednu vrstu petlji – *REPEAT petlju* (Pascal) ili *DO WHILE petlju* (jezik C). To su petlje u kojima se naredbe unutar petlje

izvrše, a potom se izračunava postavljeni uvjet, pa ako je istinit, prekida se daljnje ponavljanje izvršavanja naredbi, a ako nije, naredbe unutar petlje ponovo se izvršavaju.

Takva se petlja u Pythonu može izvesti beskonačnom *WHILE petljom* koja će sadržavati uvjetovani prekid, a to je selekcija koja sadrži *naredbu BREAK*:

```
while True :
  ...
  if uvjet : [ niz_naredbi ; ] break
  ...
naredba_iza_petlje
```

Ubuduće ćemo *WHILE petlju* s ovakvom strukturom, a to je ponavljanje izvršavanja niza naredbi sve dok postavljeni uvjet ne postane istinit, nazivati *REPEAT petlja*.

FOR petlja

Druga petlja, *FOR petlja*, ima ugrađeni „mehanizam" za ponavljanje naredbi bloka zadani broj puta. Prije nego što opišemo sintaksu i semantiku *FOR petlje*, evo programa FOR-Tablica.py koji je semantički ekvivalentan programu WHILE-Tablica.py.

FOR_Tablica.py

```
# TABLICA i, i**2, i**0.5,i = 1,..., n
while 'Unos' :
   n  = eval( input(
   'Ispisujem tablicu od 1 do n = ' ))
   if type (n) == int and n > 0 : break
print(
   '\n', ' i       i**2       i**0.5',
   '\n', '-'*26, sep = '' )
for i in range( 1, n+1 ) :
    print( "%2d %10d  %11.4f"
           % (i, i**2, i**0.5) )
```

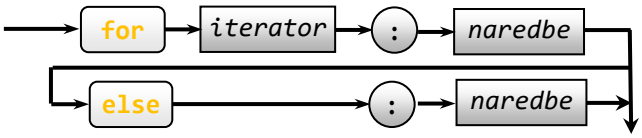
```
>>>
Ispisujem tablicu od 1 do n = 5

 i       i**2       i**0.5
--------------------------
 1          1       1.0000
 2          4       1.4142
 3          9       1.7321
 4         16       2.0000
 5         25       2.2361
```

Napominjemo da će ovdje biti riječi o nepotpunoj sintaksi i semantici *FOR petlje*, koja ima značenje slično nekim drugim jezicima za programiranje, a u Pythonu je to zapravo „iteracija" koja ima šire značenje i odnosi se na rad sa strukturama podataka danih u sljedećim poglavljima.

SINTAKSA

Pravilo pisanja FOR petlje dano je sljedećim sintaksnim dijagramom:

iteracija:




```
iterator          :
  ime in sekvenca_podataka
sekvenca_podataka : funkcija_RANGE
```

FUNKCIJA range()

Funkcija range() („opseg", „domena", „rang") generira sekvencu cijelih brojeva koji predstavljaju aritmetičke nizove, s diferencijom (korakom) *d* različitom od nule, ali zasad samo kao „mehanizam" iteracije *FOR petlje*. Piše se prema pravilu:

```
range ( do | od, do [, korak ] )
od, do, korak : cjelobrojni_izraz
```

5.1 Funkcija range()

```
>>> range (10)        range(0, 10)
>>> range (1,10)      range(1, 10)
>>> range (1,100,2)   range(1, 100, 2)
>>> range (10,0,-1)   range(10, 0, -1)
>>> range (20,30,-2)  range(20, 30, -2)
```

Iz pravila pisanja funkcije range() moguća su tri slučaja generiranja niza cjelobrojnih vrijednosti (aritmetičkog niza), sa sljedećim značenjem:

```
1) range (do      )  → od = 0; korak = 1
   0, 1, ..., do -1
2) range (od, do )   → korak = 1
   od, od+1, ..., do -1
3) range (od, do, korak)
   od, od+korak, od+2*korak, ..., do -1
```

ISPIS GENERIRANOG NIZA

Članovi generiranog niza su uređeni rastući, ako je

```
od < do and korak > 0
```

ili opadajući ako je

```
od > do and korak < 0
```

Možemo ih ispisati naredbom za ispis napisanoj prema pravilu:

```
print( * funkcija_RANGE )
```

5.2 Ispis generiranog niza

```
>>> range(10); print( *range(10) )
range(0, 10)
0 1 2 3 4 5 6 7 8 9
>>> range(1,10); print( *range(1, 10))
range(1, 10)
1 2 3 4 5 6 7 8 9
>>> range(1, 100,12); print(
    *range (1, 100, 12))
range(1, 100, 12)
1 13 25 37 49 61 73 85 97
>>> range(10, 0, -1); print(
    *range (10, 0, -1))
range(10, 0, -1)
10 9 8 7 6 5 4 3 2 1
>>> range(20, 30, -2); print(
    *range (20, 30, -2))
range(20, 30, -2)
```

U posljednjem primjeru nije bilo ispisa jer od 20 do 30 nije definiran opadajući niz (korak je -2).

RELACIJA in

Niz generiran funkcijom range() jest objekt klase range. Na primjer:

```
type (range (10))           <class
'range'>
```

Možemo mu pridružiti neko ime:

```
ime = funkcija_RANGE
```

Na primjer:

```
>>> R = range (10); type(R); R
<class 'range'>
range(0, 10)
```

Generirane članove možemo ispisati naredbom za ispis napisanoj prema pravilu:

```
print( * ime )
```

Na primjer:

```
>>> print( *R )      0 1 2 3 4 5 6 7 8 9
```

Nad generiranim nizom definirana je relacija in, napisana prema pravilu:

```
x in range ()
```

gdje je *x* cjelobrojni izraz, koja vraća True, ako je vrijednost izraza x sadržana u nizu podataka koje funkcija range() generira, inače vraća False.

5.3 Relacija in

```
>>> 0   in range( 10 )          True
>>> 10  in range( 10 )          False
>>> 3   in range( 1, 10, 2 )    True
>>> 8   in range( 1, 10, 2 )    False
>>> 10  in range( 10, 20, -1 )  False
>>> 5   in range( 10, 2, -2 )   False
>>> 25  in range( 1, 100, 5 )   False
>>> 1.5 in range( 15)           False
```

FUNKCIJE len(), min() I max()

Funkcija len() za argument *funkcija_RANGE* vraća duljinu (broj generiranih članova) niza. Prazan niz, a to je niz koji se ne može generirati iz zadane donje, gornje granice i koraka, ima duljinu 0. Standardne funkcije min() i max() vraćaju minimalnu, odnosno maksimalnu vrijednost članova generiranog niza.

5.4 Broj članova niza

```
>>> len (range(10)), min (range(10)), \
    max (range(10))          (10, 0, 9)
>>> len (range(1, 10)), \
    min (range(1, 10))          (9, 1)
>>> len( range(10, 20, -1) )          0
>>> A = range(10, 0, -1); A; len(A)
range(10, 0, -1)
10
>>> B = range(10, 5, -1); B; len(B); \
    print( *B, sep = ', ' ); min(B), \
    max(B)
range(10, 5, -1)
5
10, 9, 8, 7, 6
(6, 10)
```

ATRIBUTI FUNKCIJE range()

Nad članovima niza generiranog funkcijom range() postoji nekoliko atributa danih u nastavku.

.index()

Svakom članu generiranog aritmetičkog niza funkcije range() pridružen je indeks, broj 0, za prvi član, 1 za drugi itd. do *n*-1 za posljednji član, gdje je *n* duljina niza. Funkcija index() napisana prema pravilu:

```
( ime_funkcije_RANGE| funkcija_RANGE )
  .index ( cjelobrojni_izraz )
```

vraća indeks člana niza generiranog funkcijom range() jednakog vrijednosti cjelobrojnog izraza, ako postoji, inače se dojavljuje pogreška:

```
ValueError: 0 is not in range
```

››› 5.5 .index()

```
>>> range(10). index(5)                5
>>> range(10). index(10)
ValueError: 10 is not in range
>>> A = range(10,0,-1)
>>> print (*A); A. index(2)
10 9 8 7 6 5 4 3 2 1
8
```

.start, .stop i .step

Atribut **start** vraća prvi element niza. Ako je funkcija range() napisana kao **range**(***do***), onda je to 0, inače je to vrijednost cjelobrojnog izraza ***od***, za **range** (***od***, ***do***[, ***korak***]).

Atribut **stop** vraća gornju granicu ***do***, a atribut **step** ***korak***, ako je napisan, 0 inače.

››› 5.6 .start, .stop i .step

```
>>> range(10).start, range(10).stop, \
    range(10).step              (0, 10, 1)
>>> A = range(10,0,-1); \
    print(*A,sep = ',  ')
10, 9, 8, 7, 6, 5, 4, 3, 2, 1
>>> A.start, A.stop, A.step  (10, 0, -1)
```

SEMANTIKA

FOR petlja, za razliku od *WHILE petlje*, ima ugrađeni „mehanizam" za ponavljanje naredbi bloka sadržan u zaglavlju:

```
for ime in funkcija_RANGE
```

Izabrano ime predstavlja „kontrolnu varijablu" *FOR petlje*. Njemu će u svakom koraku iteracije biti pridružene redom vrijednosti niza generiranog funkcijom range(). Na primjer:

```
>>> for i in range(20):
    print( i, end = ' ' )
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
17 18 19
```

U sljedećem smo programu, range.py, prikazali semantiku *FOR petlje WHILE petljom*. Ako je I ime učitane range() funkcije, atributi su pridruženi varijablama:

```
i, do, k = I.start, I.stop, I.step
```

range.py

```
from Moj_modul import Input, NL
while 1 :
  try : I = Input(
    'Zadaj domenu iteriranja, '
  + 'range([od,] do [,korak]) '); break
  except : print(
      'Pogreška! Ponovi unos.' )
print( I, ' ', len(I), ' iteriranja',
       NL, 'WHILE petlja:', sep = '' )
i, do, k = I.start, I.stop, I.step
while i < do if k > 0 else i > do :
  print( i, end = ' ' );  i += k
print ('\n');
print ('FOR petlja:' )
for i in I : print( i, end = ' ' )
```

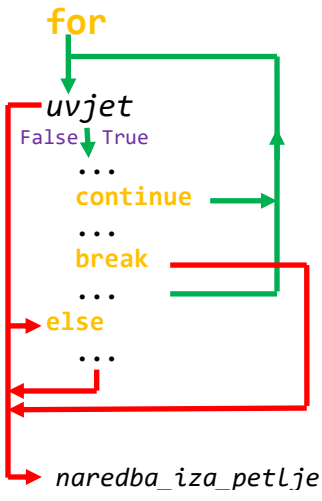
```
>>>
Zadaj domenu iteriranja, range([od,] do
[,korak]) range (1, 11, 2)
range(1, 11, 2) 5 iteriranja
WHILE petlja:
1 3 5 7 9
FOR petlja:
1 3 5 7 9
```

```
>>>
Zadaj domenu iteriranja, range([od,] do
[,korak]) range (20, 0, -3)
range(20, 0, -3) 7 iteriranja
WHILE petlja:
20 17 14 11 8 5 2
FOR petlja:
20 17 14 11 8 5 2
```

Ako je

uvjet : ime in ***funkcija_RANGE***

semantiku FOR petlje možemo grafički prikazati sa:



Naredba BREAK ima značenje prekida izvršavanja naredbi petlje (kao i u *WHILE petlji*) i nastavak na

prvoj naredbi iza petlje. *Naredba CONTINUE* ima značenje povratka na zaglavlje petlje i pridruživanje sljedećeg elementa iteriranja kontrolnoj varijabli petlje.

FOR_continue_break.py

```
for i in range( 100 ) :
   if 1 <= i <= 3 or 11 <= i <= 13 :
      i += 1; continue
   print( i, end = ' ' )
   if i >= 20 : break
```

```
>>>
0 4 5 6 7 8 9 10 14 15 16 17 18 19 20
```

GOVORIMO PYTHONSKI

Mehanizam iteriranja zadan u zaglavlju *FOR petlje* pamti se prvim dolaskom na njega. Na primjer, ako je

```
>>> I = range (2, 21, 2)
```

Izvršenjem dijela programa (u interaktivnom modu):

```
>>> for i in I :
    print( i, end= ' ' )
    i += 5
    if i == 15 : I = range( 35, 56 )

2 4 6 8 10 12 14 16 18 20
```

vidimo da `i += 5` nije promijenilo vrijednost kontrolne varijable `i` u iteraciji. Jedino je za `i=10` bio ispunjen uvjet `i == 15`, pa je izvršena naredba `I = range (35, 56)`, ali ni to nije utjecalo na domenu petlje. Na kraju su varijable `i` i `I` imale vrijednosti:

```
>>> i          25
>>> I          range(35, 56)
```

PREPORUKE ZA UPORABU PETLJI

Poslije uvođenja *WHILE petlje* više nećemo za ponavljanje izvršavanja naredbi koristiti tekstove i *naredbu EXEC*. Za vježbu se možemo vratiti programima u kojima smo to radili i „prevesti" ih u strukturu s *WHILE petljom*. Na primjer, ispis izraza:

```
1 x 8 + 1 = 9
12 x 8 + 2 = 98
... 123456789 x 8 + 9 = 987654321
```

za koji smo koristili tekst i ponavljali njegovo izvršavanje *naredbom EXEC*, sada možemo napisati koristeći *WHILE petlju*:

```
>>> b = 0; i = 1
>>> while i < 10 :
    b = b*10 +i; print( b, 'x', 8, '+',
        i, '=', b*8+i ); i += 1
```

```
1 x 8 + 1 = 9
12 x 8 + 2 = 98
123 x 8 + 3 = 987
1234 x 8 + 4 = 9876
12345 x 8 + 5 = 98765
123456 x 8 + 6 = 987654
1234567 x 8 + 7 = 9876543
12345678 x 8 + 8 = 98765432
123456789 x 8 + 9 = 987654321
```

Postavlja se pitanje: kad treba koristiti *WHILE petlju*, a kada *FOR petlju*? Odgovor bi mogao biti: onda kad se struktura rješenja problema može preslikati u strukturu *WHILE petlje*, odnosno, *REPEAT petlje* ili *FOR petlje*.

U problemima gdje je potrebno izvršiti slijed operacija dok vrijedi postavljeni uvjet, najbolje je uporabiti *WHILE petlju*, a u problemima kad treba ponavljati izvršavanje slijeda operacija sve dok se ne ispuni postavljeni uvjet - *REPEAT petlju*. Česta i „prirodna" uporaba *REPEAT petlje* bit će pri testiranju ulaznih podataka.

REPEAT petlja se može prevesti u "običnu" *WHILE petlju*

```
while True :                    Ok = False
  naredbe                       while not Ok :
  if uvjet :
    niz_naredbi; break            naredbe
  ...                             Ok = uvjet
naredba_iza_petlje              naredba_iza_petlje
```

ali je struktura *REPEAT petlje* bolje rješenje.

Ako je poznat broj ponavljanja naredbi unutar bloka petlje i ako se kontrolna varijabla petlje rabi unutar bloka, *FOR petlja* je najbolje rješenje. To bi, na primjer, vrijedilo za prethodni primjer:

```
>>> b = 0
>>> for i in range( 1,10 ):
  b = b*10 +i
  print( b, 'x', 8, '+', i, '=', b*8+i )
```

```
1 x 8 + 1 = 9
12 x 8 + 2 = 98
123 x 8 + 3 = 987
1234 x 8 + 4 = 9876
12345 x 8 + 5 = 98765
123456 x 8 + 6 = 987654
1234567 x 8 + 7 = 9876543
12345678 x 8 + 8 = 98765432
123456789 x 8 + 9 = 987654321
```

Ako je vrijednost kontrolne varijable niz cijelih brojeva iz neke domene, bolje je rabiti *FOR petlju*. Na primjer, u ispisu brojeva od 1 do 100, po deset u svakom redu:

```
>>> for i in range( 1, 101 ) :
    N = i % 10 == 0;
    print( "%3d" %i,
        sep = ' ' if not N else '',
        end = ' ' if not N else '\n' )
```

```
  1   2   3   4   5   6   7   8   9  10
 11  12  13  14  15  16  17  18  19  20
 21  22  23  24  25  26  27  28  29  30
 31  32  33  34  35  36  37  38  39  40
 41  42  43  44  45  46  47  48  49  50
 51  52  53  54  55  56  57  58  59  60
 61  62  63  64  65  66  67  68  69  70
 71  72  73  74  75  76  77  78  79  80
 81  82  83  84  85  86  87  88  89  90
 91  92  93  94  95  96  97  98  99 100
```

FOR petlju ćemo rabiti i u slučaju kad je poznat broj iteracija, kao u primjeru za izračunavanje n-tog člana Fibonaccijevog niza (dio teksta smo „otvorili" da bismo ga bolje vidjeli):

```
>>> Fib_n = """
n = eval( input(
          'n-ti član Fib. niza, n = ') )
a, b = 1, 1
for i in range( 1, n ) : a, b = b, a + b
print( a ) """
>>> exec( Fib_n *2 )
n-ti član Fib. niza, n = 100
354224848179261915075
n-ti član Fib. niza, n = 200
280571172992510140037611932413038677189525
```

BESKONAČNE PETLJE

Ponekad ćemo, često u razvoju programa, napisati *WHILE petlju* a da smo zaboravili osigurati promjenu vrijednosti postavljenog uvjeta koji će osigurati prekid izvšavanja petlje. Ako se to dogodi, izvršavanje programa možemo prekinuti s **Ctrl_C** (ili ukinuti Shell prozor).

TESTIRANJE PROGRAMA

Često ćemo pri razvoju nekog programa imati potrebu ponoviti ga s više različitih ulaznih podataka da bismo provjerili je li korektan. Cijeli će program biti unutar jedne beskonačne petlje koja će okončati ako ne unesemo podatak (podatke).

Testiranje.py

```
# Testiranje.py
while 'Test' :
    Unos = input(
        'Enter za prekid izvšavanja'
      + ' programa ili unesi polumjer ' )
    if not Unos : break
    # podaci = eval (Unos)
    # ...
```

```
>>>
Enter za prekid izvšavanja programa ili
unesi polumjer 16.66
Enter za prekid izvšavanja programa ili
unesi polumjer <Enter>
```

PRIM-BROJ

U sljedećem primjeru za zadani prirodni broj, n, veći od 1 treba odgovoriti je li prost (prim) broj. Program koji je ovdje dan nije jedino rješenje toga problema, ali zorno prikazuje primjenu *WHILE petlje*. Brojevi 2 i 3 su prosti brojevi. Ako je n>3 i nije prost broj, tada postoje dva broja p i q tako da je:

```
n = p *q
```

i vrijedi p<=q. Zato ćemo, krenuvši od i=2, provjeravati djeljivost n s i. n=2, prost je broj. Ako je n>2, dijelimo ga s i, i=2, 3, ... n**0.5. Ako postoji i s kojim je n djeljiv, prekida se daljnje izvršavanje programa i dojavljuje da broj nije prim-broj. U suprotnom, broj je prim-broj.

Dajemo dvije verzije provjere je li zadani broj n prim-broj. U drugoj smo verziji pokazali kako se može koristiti *ELSE granu WHILE petlje* da bi se izbjegla uporaba pomoćne varijable.

Prim_broj.py

```
while 'n < 2' :
  n = eval( input(
 'Zadaj cijeli broj veći ' + 'od 1 ') )
  if type(n) == int and n > 1 : break
```

```
# 1. ---------------------------------
i = 2;  Prim = True
while i <= n**0.5 :
  if not n % i : Prim = False;  break
  i += 1
print ( n, end = ' ' )
if Prim : print( 'je prim-broj!' )
else    : print( 'nije prim-broj! '
               + 'Djeljiv je s', i )
# 2. ---------------------------------
i = 2
while i <= n**0.5 :
  if not n % i : # nije prim-broj
    print( n, 'nije prim-broj! '
            + 'Djeljiv je s', i ); break
  i += 1
else : print( n, 'je prim-broj!' )
```

```
>>>
Zadaj cijeli broj veći od 1 100
100 nije prim-broj! Djeljiv je s 2
>>>
Zadaj cijeli broj veći od 1 997
997 je prim-broj!
997 je prim-broj!
```

REKURZIJE ZDESNA I ITERACIJE

Nastavljamo priču o rekurzijama koje smo opisali u trećem poglavlju. Tada smo definirali *LAMBDA funkciju* za izračunavanje faktorijela cijelog broja većeg ili jednakog 0:

```
>>> fac = lambda n : (
        "nije definirano" if n < 0  else
        1                 if n == 0 else
        n * fac( n-1 ) ) #if n>0
```

Ukazali smo i na nedostatak uporabe dane funkcije ako želimo izračunati faktorijel broja većeg od 1024, i uveli funkciju `math.factorial()`.

Ovo je bio primjer „rekurzije zdesna". Takav se rekurzivni algoritam može zamijeniti „repeticijom". Na primjer, program za izračunavanje faktorijela cijelog broja većeg ili jednakog nuli može se napisati kao:

Faktorijel_2.py

```
from Moj_modul import Int, NL
while 'Unos' :
  n = input( 'Zadaj cijeli broj veći ili'
      +' jednak 0 ili <Enter> za kraj ')
  if not n : break
  n = eval( n )
  if Int( n ) and n >= 0 :
     Fac = 1
     for i in range( 2, n+1 ) : Fac *= i
     print( NL, "\n%d! = %d"
            % (n, Fac), NL )
```

```
>>>
Zadaj cijeli broj veći ili jednak 0 ili
<Enter za kraj> 100

 100! =
933262154439441526816992388562667004907
159682643816214685929638952175999932299
156089414639761565182862536979208272237
582511852109168640000000000000000000000
00

Zadaj cijeli broj veći ili jednak 0 ili
<Enter za kraj> <Enter>
>>>
```

FIBONACCIJEVI BROJEVI (3)

Rješenje koje prilažemo u nastavku vrlo je jednostavno. Svaki novi član jednak je zbroju prethodna dva. Vrijeme izvršavanja je zanemarivo, gotovo trenutno i za 1000-ti član!

Fibonacci_3.py

```
# n-ti član Fibonaccijevog niza
from Moj_modul import Input, Int
while 'n < 2' :
  n = Input(
  'Zadaj cijeli broj veći '+'od 1 ')
  if Int(n) and n > 1 : break
a = b = 1; i = 2
while i < n : a, b = b, a+b;  i += 1
print( "\na%d =\n" % n, b )
```

```
>>>
Zadaj cijeli broj veći od 1 1500

a1500 =
135511256685631019516369368671484083777
860107124184972421335431532214873108735
287506122593540357172653003737788143473
202576992570823565500453499141029242495
959974839822286992875272419318113250950
996424476212422002092544399201969604653
214384983053458933789325853933815390935
494792961948008381459961871225833548980
00
```

ISKLJUČUJUĆA DISJUNKCIJA I IMPLIKACIJA

Evo kako možemo definirati dvije logičke operacije, isključujuću disjunkciju i implikaciju, kao lambda funkcije `xor()` i `imp()`.

xor_imp.py

```
xor = lambda x, y : \
      x and not y  or  not x and y
imp = lambda x, y : not x or y
```

Tablicu istinitosti tih operacija dobit ćemo poslije izvršenja dvije *WHILE petlje*:

```
x = False
while 1 :
  y = False
  while 2 :
    print( x, 'xor', y, '->', '\t',
           xor (x, y) )
    print ( x, '=> ', y, '->', '\t',
            imp (x, y), '\n' )
    if y : break
    y = True
  if x : break
  x = True
```

```
>>>
False xor False ->      False
False =>  False ->      True

False xor True ->       True
False =>  True ->       True

True xor False ->       True
True =>  False ->       False

True xor True ->        False
True =>  True ->        True
```

P R O G R A M I

U nastavku dajemo nekoliko programa koji objedinjuju sve dosad naučene naredbe Pythona u rješavanju relativno jednostavnih problema.

TABLICA MNOŽENJA

Treba ispisati tablicu množenja brojeva od 1 do n, n>0. Dajemo rješenje s potpunom provjerom ulaznog podatka.

Tablica_množenja.py

```
from Moj_modul import *
while 'Unos' :
   try : n = Input(
   'Tablica množenja od 1'+' do n > 1 ')
   except :
     print( 'Pogreška u ulaznom',
            'podatku! Ponovi unos.');
     continue
   if Int(n) and n > 1 : break
   print( 'Ponovi unos!' )
N = range( 1, n+1 )
for i in N :
    print( (NL +'    ') *(i==1),
      "%4d" % i, sep = '',
      end = ('' if i != n else
             NL +'    ' +'----' *n +NL) )
for i in N :
    print( "%2d " %i, end = '' )
    for j in N : print( "%4d" % (i*j),
            sep ='',
            end = '' if j != n else NL )
```

```
>>>
Tablica množenja od 1 do n > 1 10

    1   2   3   4   5   6   7   8   9  10
 ----------------------------------------
 1   1   2   3   4   5   6   7   8   9  10
 2   2   4   6   8  10  12  14  16  18  20
 3   3   6   9  12  15  18  21  24  27  30
 4   4   8  12  16  20  24  28  32  36  40
 5   5  10  15  20  25  30  35  40  45  50
 6   6  12  18  24  30  36  42  48  54  60
 7   7  14  21  28  35  42  49  56  63  70
 8   8  16  24  32  40  48  56  64  72  80
 9   9  18  27  36  45  54  63  72  81  90
10  10  20  30  40  50  60  70  80  90 100
```

IZRAČUNAVANJE TREĆEG KORIJENA (2)

Pretpostavimo da ne postoji operacija potenciranja (**) i da trebamo izračunati treći korijen realnog broja *x*. Za to ćemo rabiti poznati numerički postupak (Newtonova aproksimacija) izračunavanjem niza konvergentnih aproksimativnih vrijednosti *z* prema formuli:

$$z = \frac{2y + \frac{x}{y^2}}{3}$$

gdje je y prethodna aproksimacija. Početna je aproksimacija *y=x*.

treći_korijen.py

```
# IZRAČUNAVANJE TREĆEG KORIJENA
# (Newtonova metoda)

from Moj_modul import *
x = Input(
'Izračunavam treći korijen iz broja? ')
z = x
if x :
  while 1 :
    z = (2*z +x/(z*z))/3
    if abs (z*z*z -x) < 0.00001 : break
print( z )
```

```
>>>
Izračunavam treći korijen iz broja? -27
-3.0000000017936714
```

NAJVEĆA ZAJEDNIČKA MJERA (2)

U prethodnom smo poglavlju dali prvu verziju algoritma za traženje najveće zajedničke mjere dvaju pozitivnih cijelih brojeva koji se sada može napisati kao:

```
i = min (m, n)
while m % i or n % i :
  if m > n and m % i : m %= i
  if n > m and n % i : n %= i
  i = min (m, n)
```

Također smo opisali Euklidov algoritam za traženje najveće zajedničke mjere dvaju pozitivnih cijelih brojeva.

Euclid_2.py

```
# Euklidov algoritam za NZM
from Moj_modul import *
while True :
  M, N = Input(
  'Zadaj dva cijela broja veća od 0 ' )
  Ok = Int(M) and Int(N) and min(M, N)>0
  if Ok : break
m, n = M, N
while m != n :
  if m > n : m -= n
  if n > m : n -= m
print ( 'Nzm (', M, ',', N, ') =', m )
```

```
>>>
Zadaj dva cijela broja veća od 0 2222,
22
Nzm ( 2222 , 22 ) = 22
>>>
Zadaj dva cijela broja veća od 0
12345678, 2
Nzm ( 12345678 , 2 ) = 2
```

U drugom je primjeru izračunavanje najveće zajedničke mjere trajalo par sekundi. Razlog tomu je izvršavanje naredbe

```
if m > n : m -= n
```

6,172,839 puta!, jer je toliko puta m (inicijalno M) bio veći od n (jednako N), što smo dobili iz:

```
>>> M /2      6172839
```

PRIM-BROJEVI

Proširimo program Prim_broj.py tako da ispisuje sve prim-brojeve iz intervala [2, N], N≥2.

Prim_brojevi.py

```
from Moj_modul import *
Poruka = (
      'Ispis prim brojeva od 2 do N, '
      'N>1. Zadaj N ')
while 1 :
  N  = int (input (Poruka))
  if N > 1 : break
  Poruka = ('Broj mora biti veći od 1! '
            'Ponovi upis ')
n = 2
while n <= N :
  i = 2; Prim = True
  while Prim and i <= n **0.5 :
     Prim = bool( n % i );  i += 1
  if Prim : print( n, end = ' ' )
  n += 1
```

```
>>>
Ispis prim brojeva od 2 do N, N>1. Zadaj
N 50
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
```

KOSI HITAC (2)

Program **Kosi_hitac.py**, dan u drugom poglavlju, izračunavao je parametre kosog hica: maksimalnu visinu *H*, krajnji domet *D* u odnosu na polaznu točku u horizontalnoj ravnini i vrijeme leta *T*, prema formulama:

$$T = \frac{2V_0}{g} \quad D = TV_x \quad H = \frac{V_y^2}{2g}$$

$V_x = V_0 cos \propto_0 \quad V_y = V_0 sin \propto_0$

Pored toga, možemo izračunati koordinate položaja točke x i y, u vertikalnoj ravnini za vrijeme leta. Formule su sljedeće:

$$x = V_x t \quad y = V_y t - \frac{gt^2}{2}$$

gdje je g = 9.81 - ubrzanje zemljine teže, t vrijeme, $0 \leq t \leq T$, V_x i V_x su projekcije početne brzine na x-os i y-os ravnine.

Kosi_hitac_2.py

```
# Proračun putanje materijalne točke
# - kosi hitac
g = 9.81;   'ubrzanje zemljine teže'
""" varijable:
V, Vx, Vy   - početna brzina i
              projekcije brzine na x, y
Alfa        - početni kut [stup.]
R           - početni kut u radijanima
Tm, D, H    - vrijeme leta, domet i
visina leta, [m]
x, y        - koordinate materijalne
točke
t           - vrijeme
Dt          - korak iteriranja
i           - pomoćna varijabla """
from Moj_modul import *
from math      import sin, cos
V, α     = Input(
           'Upišite početnu brzinu, m/s'
           'i kut u st. ' )
r        = round;  R = radians (α)
Vx, Vy   = V *cos(R), V *sin(R)
Tm       = 2 *Vy /g
D        = r (Tm *Vx, 2)
H        = r (Vy**2 /(2*g), 2)
Prikaz = """
Tmax  = %10.2f sec
Domet = %10.2f m
Hmax  = %10.2f m """
print (Prikaz % (round (Tm,2),
       round (D,2), round (H,2)))
while 'korak' :
  n = int( input(
      'Zadaj broj koraka (>4) '
      'za ispis putanje ' ) )
  if n > 4 : break
print (NL,
       '  t            x           y',
       NL, '-'*31 )
t  = 0
Dt = Tm /n
while t <= Tm :
  x, y = Vx *t, Vy*t -g*t**2/2
  print ( "%6.2f"    % r(t, 2),
          "%12.2f"*2 % (r(x, 2),
                        r(y, 2)) )
  t += Dt

if Tm -t +Dt > 0.1 :
   print ("%6.2f" % r (Tm, 2),
          "%12.2f"*2 % (r (D, 2), 0.0))
```

```
>>>
Upišite početnu brzinu, m/s i kut u st.
100, 45
Tmax  =      14.42 sec
Domet =    1019.37 m
Hmax  =     254.84 m

Zadaj broj koraka (>4) za ispis putanje
10
   t            x           y
 --------------------------------
  0.00         0.00        0.00
  1.44       101.94       91.74
  2.88       203.87      163.10
  4.32       305.81      214.07
  5.77       407.75      244.65
  7.21       509.68      254.84
  8.65       611.62      244.65
 10.09       713.56      214.07
 11.53       815.49      163.10
 12.97       917.43       91.74
 14.42      1019.37        0.00
```

ZBROJ ČLANOVA REDA

Vrijednost funkcija $sin(x)$ i $cos(x)$ može se izračunati kao zbroj članova reda:

$$sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots = \sum_{i=1}^{n} (-1)^{i+1} \frac{x^{2i-1}}{(2i-1)!} \quad -\infty < x < \infty$$

$$cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots = \sum_{i=0}^{n} (-1)^{i} \frac{x^{2i}}{(2i)!} \quad -\infty < x < \infty$$

Napišimo program koji će za izabranu funkciju, 'sin' ili 'cos', i zadani kut α u stupnjevima, izračunati njezinu vrijednost. Poslije svake iteracije provjeravat ćemo je li razlika nove i prethodne vrijednosti manja od zadane točnosti, **1e-12** u našem programu. Ako jeste, postupak se prekida.

Evo kompletnog programa koji kompaktno objedinjuje gotove sve naredbe i izraze koje smo dosad naučili:

sin_cos.py

```
# Zbroj prvih P članova reda
# 1 -X**2/2! +X**4/4! -... (cos(X))
# X -X**3/3! +X**5/5! -... (sin(X))

from Moj_modul import Input, TAB, NL
from math      import (
     radians, factorial, sin, cos )
while 1 :
  fun = input ('Računam cos ili sin ')
  if fun == 'cos' or fun == 'sin' :
     break
  print( 'Pogrešno ime funkcije. '
         'Ponovi unos!' )
COS = fun == 'cos'
α = Input( 'Zadajte argument, '
           'kut u stupnjevima ' )
x = radians (α)
Σ = C = 1 if COS else x
i = 2;  print( 1, TAB, Σ )
while True :
  Fact = factorial (2*(i -1)) if COS \
         else factorial (2*i -1)
  C *= -x**2;  Σ0 = Σ;  Σ += C /Fact
  print( i, TAB, Σ )
  if abs (Σ0 -Σ) < 1e-12 : break
  i += 1
print ('     ' +fun +'(', α, ') = ',
  Σ, NL, 'math.' +fun +'(', α, ') = ',
  eval (fun +'(x)'), sep = '')
```

```
>>>
Računam cos ili sin 30
Pogrešno ime funkcije. Ponovi unos!
Računam cos ili sin cos
Zadajte argument, kut u stupnjevima 30
1    1
2    0.8629221610959812
3    0.8660538834157472
4    0.8660252641005711
5    0.8660254042103523
6    0.8660254037835535
7    0.8660254037844399
     cos(30) = 0.8660254037844399
math.cos(30) = 0.8660254037844387
>>>
Računam cos ili sin sin
Zadajte argument, kut u stupnjevima 60
1    1.0471975511965976
2    0.8558007815651173
3    0.8662952837868347
4    0.8660212716563725
5    0.8660254450997811
6    0.8660254034934827
7    0.8660254037859597
8    0.8660254037844324
9    0.8660254037844385
     sin(60) = 0.8660254037844385
math.sin(60) = 0.8660254037844386
```

Binomni koeficijenti (Pascalov trokut)

Iz matematike je poznato da se koeficijenti uz pojedine članove u formuli za potenciranje binoma *(a+b)* na potenciju *n* mogu dobiti kao:

$$(a+b)^n = \sum_{r=0}^{n} \binom{n}{r} a^{n-r} b^r$$

Koeficijenti $\binom{n}{r}$ nazivaju se „binomni koeficijenti". Zapis $\binom{n}{r}$ čita se „*n* povrh *r*", a značenje mu je:

$$\binom{n}{r} = \frac{n \times (n-1) \times \ldots \times (n-r+1)}{1 \times 2 \times 3 \times \ldots \times r}$$

Pri izračunavanju koeficijenata za dani stupanj *n* vrijednost za *r* mijenjat će od 0 do *n*. Još je po definiciji:

$$\binom{n}{0} = 1$$

Na primjer, koeficijenti uz pojedine članove pri izračunavanju binoma $(a+b)^3$ su:

$$(a+b)^3 = \binom{3}{0}a^3b^0 + \binom{3}{1}a^2b^1 + \binom{3}{2}a^1b^2 + \binom{3}{3}a^0b^3$$
$$= 1a^3 + 3a^2b + 3ab^2 + 1b^3$$

Da bismo napisali program koji bi izračunavao binomne koeficijente za dani stupanj *n*, napišimo danu formulu za izračunavanje binomnih koeficijenata

$$\binom{n}{r} = \frac{n}{1} \times \frac{n-1}{2} \times \ldots \times \frac{n-r+1}{r}$$

Prevođenjem ovih formula u Python, dobiva se program za izračunavanje binomnih koeficijenata (Pascalovog trokuta) do danoga stupnja S:

Pascalov_trokut.py

```
# Binomni koeficijenti (Pascalov trokut)
from Moj_modul import Input, Int, NL
while 1 :
  S = Input(
  'Ispis binomnih koeficijenata '
  '(max. 15)? ')
  if Int (S) and 0 <= S <= 15 : break
  print( 'Pogrešan unos. Ponovite! ' )
print( " n      (a + b) **n" )
for n in range (S+1) :
  print ("%2d  " % n, end = ' ')
  for r in range (n+1) :
    k = 1
    for i in range (n, n-r, -1) :
        k = k*i //(n-i+1)
    print( "%4d" %k,
           end=' ' if r<n else NL )
```

Ovaj jednostavni program ilustrira pisanje petlje u petlji, kao i primjenu petlje s negativnim korakom. Na primjer, za zadani stupanj jednak 7 bilo bi ispisano:

```
>>>
Ispis binomnih koeficijenata (max. 15)?
7
 n      (a + b) **n
 0     1
 1     1    1
 2     1    2    1
 3     1    3    3    1
 4     1    4    6    4    1
 5     1    5   10   10    5    1
 6     1    6   15   20   15    6    1
 7     1    7   21   35   35   21    7    1
```

STOLNI TENIS

Sljedeći program simulira stolni tenis u 4 dobivena seta. Set se igra do 11 poena pri čemu razlika osvojenih poena mora biti veća od 1. Ako nije, nastavlja se igra sve dok jedan od natjecatelja ne bude vodio s dva poena razlike.

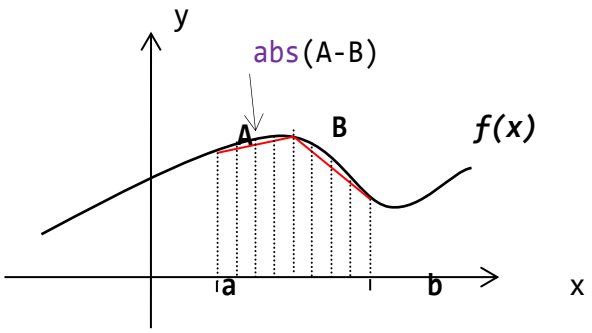
Stolni_tenis.py

```
# Stolni tenis u četiri dobivena seta
from random import *
A = B = 0
while 1 :
  a = b = 0
  while 2 :
    if random() < 0.5 : a += 1
    else              : b += 1
    if (a >= 11 or b >= 11) \
       and abs(a-b)>1 : break
  A += a > b;  B += b > a
  print( "%2d : %2d (%d : %d)"
         % (a,    b,   A,   B) )
  if A == 4 or B == 4 : break
```

```
>>>
 6 : 11 (0 : 1)
 8 : 11 (0 : 2)
 8 : 11 (0 : 3)
11 :  9 (1 : 3)
11 :  7 (2 : 3)
12 : 14 (2 : 4)
```

DULJINA KRIVULJE FUNKCIJE

Treba izračunati duljinu krivulje zadane funkcije $f(x)$, na intervalu $[a,b]$.



Počinjemo s unosom funkcije i intervala i provjerom korektnosti podataka. Dodali smo modul `math` da bismo osim standardnih funkcija mogli koristiti i funkcije iz tog modula (trigonometrijske, logaritamske i druge funkcije).

Duljina_krivulje.py

```
# Duljina krivulje funkcije f(x)
# na intervalu [a, b]
from math import *;  π = pi
y  = lambda x : eval (fx)
while 1 :
  fx = input ('Upiši f(x) =  ')
  try :
    a, b = eval( input(
           'interval a, b ( π ) ') )
    y(a), y(b)
    break
  except : print ( 'Pogreška!' )
```

Rješenje koje dajemo temelji se na izračunavanju zbroja dužina između parova točaka $(f(x), f(x+d))$, gdje je d korak (prirast) za n točaka intervala

[*a*,*b*], *d*=abs(*a*-*b*)/*n*. To je približna vrijednost, ali možemo zadati koliko će se razlikovati od stvarne ako uspoređujemo razliku dvaju duljina, za *n* i 2**n* intervala, jer ćemo u svakoj novoj iteraciji udvostručiti broj intervala.

U programu su inicijalne vrijednosti: broj intervala, n = 2, duljina krivulje, L0 = 0, i preciznost, P = 1e-6. Parovi točaka su kompleksni brojevi A i B između kojih se računa udaljenost s abs(A-B). Postupak okončava kad je postignuta preciznost P.

```
f  = lambda x : complex (x, y(x))
P = 1e-6; 'točnost (preciznost)'
n = 2; L0 = 0;
while True :
  d = abs (a-b) /n;  A = f(a)
  L = 0; x = a+d

  while x < b+d/2 :
    if x > b -d : x = b
    B  = f(x); L += abs(A-B); A = B
    x += d
  if L != L0 and abs(L-L0) < P : break
  L0 = L; n *= 2
print( 'd =', L )
```

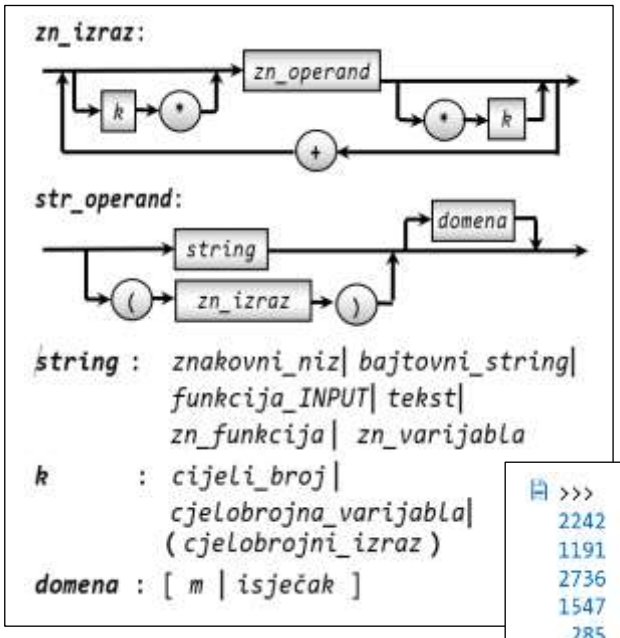
```
>>>
Upiši f(x) =  sin(x)
interval a, b ( π ) 0
Pogreška!
Upiši f(x) =  (25 -x**2) **0.5
interval a, b ( π ) -3, 6
Pogreška!
Upiši f(x) =  (25 -x**2) **0.5
interval a, b ( π ) -2, 5
d = 9.91156565889
>>> n # broj intervala
32768
>>>
Upiši f(x) =  sin(x)
interval a, b ( π ) 0, π
d = 3.82019767156
>>>
Upiši f(x) =  sin(x) +cos(x)
interval a, b ( π ) 0, 2*π
d = 8.73775247166
>>>
Upiši f(x) =  (9 -x**2)**0.5
interval a, b ( π ) -3, 3
d = 9.4247775402

>>> abs(3*π -L) # odstupanje u odnosu na
>>> # točnu vrijednost (poluopseg, r*π)
4.20564834513e-07
>>> n # broj segmenata
3276
```

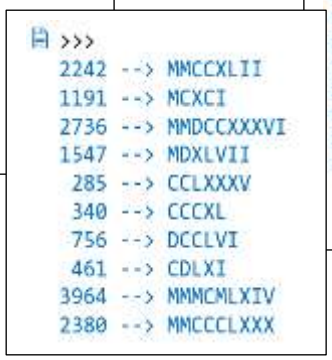
6.

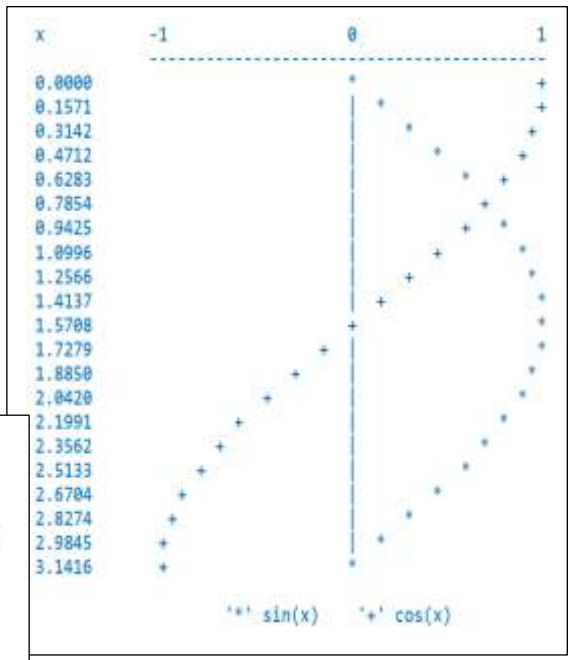
ZNAKOVI I ZNAKOVNI NIZOVI

Vrijednosti bez komponenata, koje predstavljaju same sebe i dalje se ne dijele, nazvali smo primitivnim tipovima. Nositelji takvih vrijednosti bile su primitivne varijable. Polazeći od primitivnih tipova moguće je definirati strukturirane tipove, skupove vrijednosti čija struktura ima određeni smisao. Nositelji strukturiranih podataka bit će strukturirane varijable koje se mogu koristiti tako da se odnose na strukturiranu vrijednost u cjelini ili na pojedine komponente. Python ima nekoliko standardnih strukturiranih tipova podataka. Prva koju ćemo obraditi u ovom poglavlju jest znakovni niz ili string.

String smo opisali još u prvom poglavlju. Uradili smo to da bismo ga mogli rabiti u opisu zahtjeva za unos podataka i izlaznih rezultata. Jedan oblik pisanja stringa u više redova, tekst, rabili smo u pisanju programa kao teksta. U ovom ćemo ga poglavlju prikazati u potpunosti, opisati funkcije koje su definirane nad njim i koje omogućuju jednostavno rješavanje mnogih problema s tekstom.







Uvod

Znakovni niz ili string struktura je podataka definirana nad znakovima. Vrijednosti „stringovnog" tipa - nizovi znakova - pišu se prema pravilu opisanom u osnovnoj leksičkoj strukturi Pythona. Na primjer, stringovi su:

```
'Python'   ''   "I''m"   "C''est la vie"
'***'
```

Nizovi koji su počinjali i završavali s jednim navodnikom ili polunavodnikom pišu se u jednoj liniji. Osim njih koristili smo i tekstove, nizove znakova koji počinju i završavaju se s tri navodnika ili polunavodnika i mogu biti napisani u više redova. Nizove znakova, kao vrijednosti, rabili smo u naredbi za ispis. Tekstove smo koristili kao dokumentaciju u programima ili smo ih pridruživali nekoj varijabli kao dijelove programa koje smo mogli izvršiti *naredbom EXEC*.

Osim eksplicitne uporabe stringova postoje i varijable koje pamte takve vrijednosti. Također su definirani znakovni izrazi i nekoliko korisnih znakovnih funkcija ili funkcija drugoga tipa koje za argumente imaju nizove znakova. U ovom je poglavlju u potpunosti opisan znakovni tip, te sintaksa i semantika naredaba i procedura koje koriste znakovne vrijednosti.

Znakovi

Znak jest jedinstvena, nedjeljiva cjelina, kao što su, na primjer, slova, znamenke, +, -, *, /, (,), [,] itd. Drugim riječima, to je ono što je označeno na tipkovnici (uključujući i razmak ili "blank"), što se interpretira kao znak na ekranu ili drugom izlaznom mediju, uz dodatak kontrolnih znakova. Znakovni tip čini skup znakovnih vrijednosti.

Godine 1968. standardizirani su kodovi znakova koji su se rabili na kompjuterima. Uvedena je ASCII tablica (American Standard Code for Information Interchange). Sadržavala je 128 znakova (brojke, slova, znakove interpunkcije itd), s kodovima od 0 do 127. Na primjer, znaku (slovu) 'A' bio je dodijeljen kôd 65, znaku 'a' 97 itd.

U tom su skupu znakova bila samo velika i mala slova engleskog alfabeta, što znači da nije bilo slova s naglascima, kao što su 'ä', 'é', 'è', 'ö' ili 'ü', pa jezici koji su u svojem alfabetu imali te znakove nisu mogli rabiti ASCII tablicu.

Tada su proizvođači kompjutera (IBM, Univac, CDC, DEC itd.) za naručitelje njihovih stojeva iz Francuske, Njemačke ili iz jugoistočne Europe zamjenjivali „nepotrebne" znakove '@', '[', '\', ']', '^', '`', '{', '|', '}' i '~' s posebnim slovima, na primjer, u Hrvatskoj, sa 'Ž', 'Š', 'Đ', 'Ć', 'Č', 'š', 'đ', 'ć' i 'č'.

Pojavom osobnih računala početkom 80-tih godina prošloga stoljeća, koja su bila 8-bitna, ASCII tablica je proširena s dodatnim znakovima koji su imali kodove od 128 do 255. U taj su se dio dodavali slova s naglascima, grafički znakovi i drugi znakovi, kao što su neka grčka slova. Tada je uveden i pojam „kodnih stranica", na primjer, 437 (DOS), 850 (Latin-1) i 852 (Latin-2), koje su sadržavale različite skupove znakova. Pojavom Windowsa uvedene su još neke kodne stranice, na primjer 1252 (MS Windows – Latin 1) i 1250 (MS Windows – Latin 2).

Znamo da funkcija chr(i), gdje je i cijeli broj od 0 do 1,114,111 (0x10FFFF u bazi 16), vraća znak čiji je *Unicode* jednak i. Kontrolni znakovi imaju kôd od 0 do 31, a ostali znakovi od 32 (praznina ili blank) sve do dane gornje granice.

Na primjer, chr(65) vraća znak 'A', dok chr(8364) vraća znak '€'.

Program **ASCII.py**, dan u nastavku, ispisuje tablicu *Unicode* znakova, od koda 33 do 255, koju ćemo iz tradicionalknih razloga zvati ASCII tablica znakova.

ASCII.py

```
print( '    0 1 2 3 4 5 6 7 8 9' );
print( '-'*23 );
for i in range (32, 256) :
  if i % 10 == 0 : print (); \
     print( "%2d " % (i//10),
            end = ' ' )
  print( chr(i), end = ' ' )
>>>
```

```
    0 1 2 3 4 5 6 7 8 9      0 1 2 3 4 5 6 7 8 9
  ---------------------    ---------------------
  3       ! " # $ % & '    15 - – ˜ ™ š › œ ? ? Ÿ
  4 ( ) * + , - . / 0 1    16   ¡ ¢ £ ¤ ¥ ¦ § ¨ ©
  5 2 3 4 5 6 7 8 9 : ;    17 ª « ¬ - ® ¯ ° ± ² ³
  6 < = > ? @ A B C D E    18 ´ µ ¶ · ¸ ¹ º » ¼ ½
  7 F G H I J K L M N O    19 ¾ ¿ À Á Â Ã Ä Å Æ Ç
  8 P Q R S T U V W X Y    20 È É Ê Ë Ì Í Î Ï Ð Ñ
  9 Z [ \ ] ^ _ ` a b c    21 Ò Ó Ô Õ Ö × Ø Ù Ú Û
 10 d e f g h i j k l m    22 Ü Ý Þ ß à á â ã ä å
 11 n o p q r s t u v w    23 æ ç è é ê ë ì í î ï
 12 x y z { | } ~ ?       24 ð ñ ò ó ô õ ö ÷ ø ù
 13                       25 ú û ü ý þ ÿ
 14
```

FUNKCIJA ord()

Funkcija ord(*Ch*), gdje je *Ch* znak, inverzna je funkcija funkciji chr() i vraća Unicode kôd (redni broj) argumenta. Na primjer, ispišimo redne brojeve malih i velikih slova koji ne pripadaju ASCII tablici:

```
>>> for c in 'čćđšžČĆĐŠŽ' :
    print (c, ord(c), end = ' ')

č 269 ć 263 đ 273 š 353 ž 382 Č 268 Ć
262 Đ 272 Š 352 Ž 381
```

UNICODE STANDARDIZACIJA

Uvođenjem kodnih stranica različiti su strojevi imali različite kodove, što je dovelo do problema razmjene podataka. Osim toga, bilo je premalo mjesta da se paralelno rabe dva pisma, na primjer latinica u zapadnoj Europi i ćirilica u nekim drugim zemljama Europe.

Rješenje se problema naziralo u drugom dijelu 80-tih godina, pojavom 16-bitnih procesora koji su imali na raspolaganju 2^{16}=65536, umjesto 2^8=256, različitih vrijednosti. Uvedena je "Unicode" standardizacija. Početni je cilj bio da *Unicode* sadrži pisma svakog pojedinog ljudskog jezika. Ispalo je da čak 16 bita nije dovoljno da zadovolji taj cilj, pa sada moderna *Unicode* specifikacija koristi širi spektar kodova, 0 od do 1,114,111 (0x10ffff heksadecimalno).

Unicode standard opisuje kako su znakovi predstavljeni kodnim točkama. Kodna točka je cjelobrojna vrijednost, obično prikazana kao heksadecimalni broj. Unicode standard sadrži mnogo tablica znakova i njihovih kodnih točaka. Na primjer, Windows-1250 je kodna stranica u Microsoft Windowsima koja se koristi za pisanje tekstova u jezicima istočne Europe (poljski, češki slovački, ..., hrvatski). Program **ASCII.py** ispisuje tablicu znakova kodne stranice 1250. Ako bismo u interaktivnom modu napisali

```
>>> 'Č','č','Ć','ć','Đ','đ','Š','š', \
    'Ž', 'ž'
('\xc8', '\xe8', '\xc6', '\xe6', '\xd0',
'\xf0', '\x8a', '\x9a', '\x8e', '\x9e')
```

vidimo da su ispisani heksadecimalni kodovi navedenih znakova (slova). Međutim, naredba za ispis „radi" kako se očekuje:

```
>>> print( 'Č', 'č', 'Ć', 'ć', 'Đ',
           'đ', 'Š', 'š', 'Ž', 'ž')
Č č Ć ć Đ đ Š š Ž ž
```

Program dan u nastavku ispisuje ruski alfabet.

Ruski_alfabet.py

```
i = 1040;  k = i +31
print( 'Ruski alfabet: \n ' )
while i <= 1103 :
  print( chr (i), end = ' ' )
  if i == k : print ( )
  i += 1
L = u'\u041b'; e = u'\u0435'
v = u'\u0432'
N = u'\u041d'; i = u'\u0438'
k = u'\u043a'; o = u'\u043e'
l = u'\u043b'; a = u'\u0430'
c_= u'\u0447'; T = u'\u0422'
s = u'\u0441'; t = u'\u0442'
j = u'\u0439'; A = u'\u0410'
n = u'\u043d'; K = u'\u041a'
r = u'\u0440'
print( '\n'*2, 'Primjer:' )
print( L+e+v, N+i+k+o+l+a+e+v+i+c_,
       T+o+l+s+t+o+j+':',
       A+n+n+a, K+a+r+e+n+i+n+a )
```

```
>>>
А Б В Г Д Е Ж З И Й К Л М Н О П Р С Т У
Ф Х Ц Ч Ш Щ Ъ Ы Ь Э Ю Я
а б в г д е ж з и й к л м н о п р с т у
ф х ц ч ш щ ъ ы ь э ю я
Лев Николаевич Толстой: Анна Каренина
```

UREĐENJE ZNAKOVA

Uređenje znakovnih vrijednosti određeno je Unicode kodovima. Na primjer, vrijedi:

```
'A' < 'B' <... < 'Z' <... 'a' <... < 'z'
```

Također vrijedi: '0' < '1' < ... < '8' < '9'. Drugim riječima, ako su *c1* i *c2* dva znaka, vrijedi

```
c1 < c2
```

ako i samo ako je ord(*c1*) < ord(*c2*).

››› 6.1 usporedba znakova

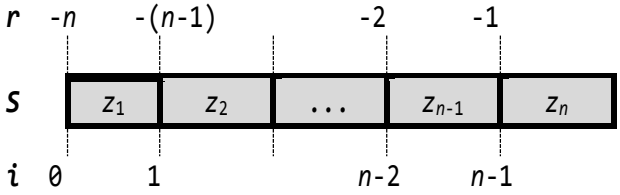
```
>>> 'A'<'B' True    >>> '0'<'1'    True
>>> 'a'<'A' False   >>> '9'<'0'    False
>>> '9'<'+' False   >>> '+'<'-'    True
>>> 'š'<'T' False   >>> '9'<'A'    True
>>> 'a'<'b'<'c'                   True
```

Znakovni nizovi

Objekt, *x*, neke klase *X*, može biti eksplicitno napisan podatak ili ime varijable iza kojih će slijediti točka pa ime atributa ili metode. Zapamtimo da točka nije dio imena pa se može pisati odvojeno od objekta i atributa (metode). Na primjer, ako je `'Python'` podatak klase `str`, može se napisati

```
>>> 'Python'.upper()          'PYTHON'
>>> 'Python' .   upper()      'PYTHON'
```

Znakovni niz ("string" ili "povorka") predstavlja strukturu podataka. Čine je znakovi napisani jedan za drugim, omeđeni s polunavodnicima ili navodnicima (v. sintaksu u prvom poglavlju). Ako je ***S*** znakovni niz sačinjen od *n* znakova z_i, "$z_0 z_1 \ldots z_{n-1}$", interpretira se kao uređena sekvenca u kojoj je svakom znaku pridružen indeks ***i***, od `0` do ***n***`-1`, slijeva nadesno, odnosno, relativni indeks ***r***, od `-1` do ***–n***, zdesna nalijevo:



gdje je ***n*** duljina niza ***S***, a dobije se pozivanjem funkcije `len()`, ***n***=`len(`***S***`)`.

Pojedinom se elementu znakovnoga niza, znaku, može pristupiti pišući string kojeg slijedi njegov indeks ili relativni indeks između uglatih zagrada:

S*`[`*i*`]` ***0 ≤ i ≤ n-1 ili ***S*`[`*r*`]` ***-n ≤ r < 0***

Relativni indeks ***r*** indeksa ***i*** jednak je ***i-n***.

››› 6.2 znak znakovnog niza

```
>>> len ('abcd')     4
>>> 'abcd'[3]        'c'
>>> 'abcd'[0]        'a'
>>> 'abcd'[4]
... string index out of range
```

U sljedećoj je vježbi pokazano kako možemo ispisati sve znakove niza znakova, redom, od znaka s indeksom `0` do posljednjeg, s indeksom *n*`-1`, gdje je *n* duljina niza znakova.

››› 6.3 ispis znakovnog niza s razmakom

```
>>> i = 0
>>> while i < len( 'Python' ):
    print( 'Python'[i], end = ' ')
    i += 1

P y t h o n
```

ZNAKOVNE VARIJABLE

Nizovna varijabla, ili varijabla sa strukturom znakovnog niza, bit će ime kojem je pridružen string. To se može postići naredbom za pridruživanje kad je na mjestu izraza znakovni izraz:

naredba_za_pridruživanje :
ime { = *ime* } = *str_izraz*

SEMANTIKA

Značenje naredbe za pridruživanje jest: pridruživanje navedenom imenu (imenima, ako ih ima više od jednog) stringa dobivenog izračunavanjem stringovnog izraza. Izvršenjem te naredbe *ime* će poprimiti svojstvo „stringovna varijabla". Pristup pojedinoj kompo-nenti stringovne varijable bit će sa:

***nv*`[`*i*`]`

gdje je ***nv*** varijabla znakovnog niza („stringovna varijabla"). Ako je ***n*** duljina niza znakova sadržanih u ***nv***, ***i*** je cjelobrojni izraz čija je vrijednost (indeks i relativni indeks) iz intervala:

-n ≤ i ≤ n-1

Potpuni opis stringovnog izraza dan je malo kasnije. Zasad ćemo koristiti njegov najjednostavniji oblik, pa je jednostavno pridruživanje definirano kao

ime = *znakovni_niz* | *funkcija_INPUT*

U sljedećoj smo vježbi znakovni niz pridružili varijabli Py i potom ga ispisali, znak po znak.

››› 6.4 ispis znakova znakovnoga niza

```
>>> Py = 'Python'
>>> for i in range (len (Py)) :
    print( Py[i], end = ' ' )
P y t h o n
```

OPERATORSKO PRIDRUŽIVANJE

Nad znakovnim varijablama definirana su dva operatorska pridruživanja: += i *=. Ako je ***str_var*** stringovna varijabla, pravilo pisanja je:

str_var (+= ***str_izraz*** | *= ***cjelobrojni_izraz***)

Značenje operatora += je dopisivanje stringa dobivenog evaluacijom stringovnog izraza prethodnom sadržaju stringovne varijable:

str_var = ***str_var*** + ***str_izraz***

Značenje operatora *= multipliciranje prethodnog sadržaja stringovne varijable s vrijednošću (cijelim brojem) dobivenim izračunavanjem cjelobrojnog izraza:

str_var = ***str_var*** * (***cjelobrojni_izraz***)

RELACIJE SA ZNAKOVNIM NIZOVIMA

Nad znakovnim nizovima definirane su standardne relacije kao i nad primitivnim tipovima, uz dodatak relacija in i not in:

relacijski_izraz:
zn_izraz (*relacija* | in | not in) *zn_izraz*

STANDARDNE RELACIJE

Uspoređivanje dvaju znakovnih nizova, općenito dobivenih kao rezultat izračunavanja znakovnih izraza, svodi se na uspoređivanje njihovih znakova na poziciji jednakog indeksa. Dva su stringa, ***X*** i ***Y***, jednaka, ***X***==***Y***, ako je

`len(X) == len(Y)` i ako je `X[i] == Y[i]` za sve `i = 0, 1, ..., len(X)-1`.

Niz ***X*** je manji od niza ***Y***, ***X***<***Y***, ako uspoređujući `X[i]` s `Y[i]` postoji indeks $i, 0 \le i \le$ `min( len(X),len(Y) )` za koji je `X[i]<Y[i]`. Ili, ako je

`X[i] == Y[i]` za sve `i = 0, 1, ..., len(X)-1` i `len(X) < len(Y)`

››› 6.5 standardne relacije

```
>>> '100000' <  '2'          True
>>> 'abc'    >  'abcABC'     False
>>> 'Čičak'  <  'Mak'        False
>>> 'Python' >= 'Pascal'     True
```

RELACIJE in I not in

Nad stringom je definirana i relacija pripadnosti (sadržanosti) jednog znaka ili stringa u drugom stringu. To je relacija in, onosno not in, pa je pravilo pisanja relacijskog izraza prošireno sa:

relacijski_izraz:
zn_izraz (in | not in) *zn_izraz*

Niz *S* duljine *n* ima *n**(*n*+1)/2 +1 podnizova, a to su: prazan niz, podnizovi duljine 1 (znakovi), podnizovi duljine 2, ... podniz duljine *n*.

››› 6.6 podnizovi niza znakova

```
>>> S = 'abc';  n = len(S)
>>> print( n*(n+1)/2 +1, 'podnizova' )
7 podnizova
>>> '' in S   # prazan string      True
>>> 'a' in S, 'd' in S # zn stringa
(True, False)
>>> 'ab' in S, 'bc' in S  # podnizovi
(True, True)
>>> S in S # cijeli string         True
>>> 'ac' not in S                  True
```

ITERIRANJE

Podaci znakovnog tipa ili stringovi pripadaju skupini podataka u Pythonu za koje ćemo reći da imaju *strukturu sekvence*. Nad strukturom sekvence definirana je složena naredba *FOR petlja* – naredba za iteriranje ili iteracija. Sintaksu smo dali u prethodnom poglavlju. Ovdje proširujemo značenje sekvence podataka:

sekvenca_podataka : *zn_izraz*

SEMANTIKA

Ako iteraciju prikažemo sa:

```
for i in _s : n1
else        : n2
```

gdje su:

i	- ime iteratora
_s	- string dobiven izračunavanjem znakovnog izraza *zi*
n1, *n2*	- naredbe

značenje je ekvivalentno *WHILE petlji* sa sljedećom strukturom:

```
i_ = 0;  _s = ni
while i_ < len (_s) :
  i = _s[i_]
  n1
  i_ += 1
else : n2
```

Prvo se rezultat izračunavanja znakovnog izraza, *ni*, pridruži varijabli _s. Ako je _s neprazan string, ponavljat će se izvršavanje naredbi *n1* pridružujući znakovnoj varijabli iteratora, *i*, znakove stringa _s, redom od _s[0] do _s[len(_s)-1]. Ako iteracija sadrži *ELSE granu*, naredbe *n2* bit će izvršene odmah, ako je _s prazan string, ili poslije završetka ponavljanja izvršavanja naredbi *n1*.

U sljedećoj smo vježbi usporedili ispis znakova stringa 'Python' uz pomoć *WHILE petlje* i naredbom za iteriranje. U ovom je slučaju primjerenija uporaba iteriranja.

››› 6.7 *WHILE petlja* i iteriranje

```
>>> Py = 'Python';  i = 0
>>> while i < len (Py) :
    print ( Py[i], end = ' ' ); i += 1
P y t h o n
>>> # for
>>> for C in Py : print( C, end = ' ' )
P y t h o n
```

U sljedećoj smo vježbi varijabli A dodavali vrijednost iteratora x u svakom koraku.

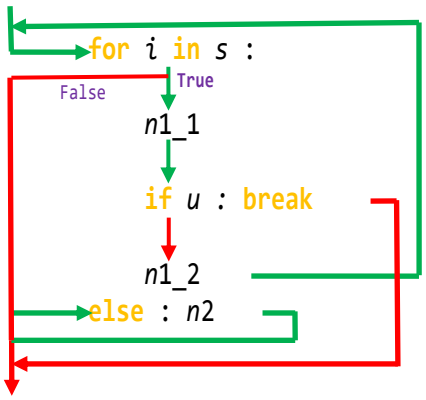
››› 6.8 iteriranje

```
>>> A = 'Python'
>>> for x in A :
    print( x, end = ' ' );  A += x

P y t h o n
>>> x                                  'n'
>>> A                      'PythonPython'
```

Vidimo da je iterator x imao vrijednosti stringa A na početku iteracije, bez obzira što je varijabla A mijenjala svoj sadržaj unutar iteracije.

PREKID ITERACIJE

Posljedica definicije semantike iteracije: iteriranje je uvijek konačno i imat će len(_s) ponavljanja izvršavanja naredbi *n1*. To će biti točno ako iteracija ne sadrži *naredbu BREAK*. Ako iteracija sadrži *naredbu BREAK* unutar svojih naredbi, značenje je kao i u *WHILE petlji*: prekid daljnjeg izvršavanja ponavljanja naredbi i nastavak izvršavanja programa prvom naredbom iza iteracije. Tada se neće izvršavati naredbe *ELSE grane*, ako postoji. Sve smo to shematski prikazali na sljedećem crtežu:



Na primjer, u sljedećem se programu učitava rečenica (niz znakova) i iz nje ekstrahiraju riječi, a to su stringovi koji ne sadrže znakove interpunkcije, IntPun.

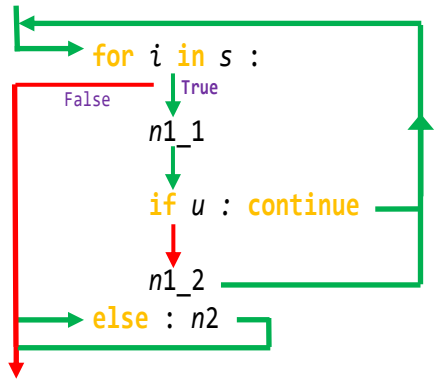
for_break.py

```
S      = input ('Učitaj rečenicu:\n')
IntPun = " .:,;!?-()+-*/'\""
i = 0
while True :
  w = ''
  for s in S[i:] :
    i += 1
    if s in IntPun :
      if w       : print (w, end = ' ')
      if s != ' ': print (s, end = ' ')
      break
    w += s
  else :
    if w : print ( w )
    break
```

```
>>>
Učitaj rečenicu: (10+20)*55.45-30
( 10 + 20 ) * 55 . 45 - 30
```

PRELAZAK NA SLJEDEĆI ELEMENT SEKVENCE

S obzirom na to da iteracija ima "ugrađen mehanizam" pridruživanja elemenata (znakova) sekvence podataka varijabli iteriranja, postoji *naredba CONTINUE* čijim se izvršenjem prelazi na sljedeći element ignorirajući dio naredbi do kraja naredbi iteracije. *Naredba CONTINUE* se uvijek izvršava pod određenim uvjetima, tj. piše se u selekciji, kao što je prikazano na sljedećem crtežu:



Vidimo da će se naredbe *ELSE grane*, ako postoji, uvijek izvršiti. Sljedeći program izbacuje razmake iz ulaznog niza.

for_continue.py

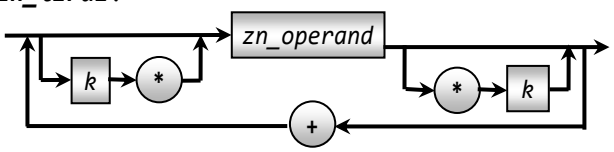
```
S = input( 'Učitaj rečenicu:\n' )
w = ''
for s in S :
  if s == ' ' : continue
  w += s
print( w )
```

```
>>>
Učitaj rečenicu:
Ova    rečenica      ima                puno
razmaka.
Ovarečenicaimapunorazmaka.
```

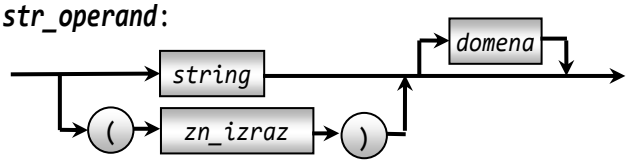
ZNAKOVNI IZRAZI

Poslije pisanja jednostavnih znakovnih izraza i njihove prvobitne sintakse dane u prvom poglavlju, sada dajemo potpuno pravilo njihovog pisanja:

zn_izraz:



str_operand:



string : *znakovni_niz* | *bajtovni_string* | *funkcija_INPUT* | *tekst* | *zn_funkcija* | *zn_varijabla*

k : *cijeli_broj* | *cjelobrojna_varijabla* | (*cjelobrojni_izraz*)

domena : [*m* | *isječak*]

isječak: [*m*] : [*n*] [: [*l*]]

gdje su ***m*** i ***n*** cjelobrojni izrazi čija je vrijednost u intervalu od -**len**(*string*) do **len**(*string*), a ***l*** cjelobrojni izraz čiji rezultat izračunavanja mora biti različit od **0**.

SEMANTIKA

Ako nizovni izraz prikažemo pojednostavljeno kao

[*k* *] *no* [* *k*] { + [*k* *] *no* [* *k*] }

gdje je *no* nizovni operand, značenje operacija je:

- \+ nastavljanje (dopisivanje ili konkatenacija) dvaju stringova
- * multipliciranje stringa (nastavljanje *k* stringova, ako je *k*>0, prazan string za *k*=0)

Operacija nastavljanja nije komutativna, tj. ako su *no*1 i *no*2 dva nizovna operanda, *no*1≠*no*2, vrijedi

*no*1 + *no*2 ≠ *no*2 + *no*1

Evaluiranje nizovnog izraza odvija se prema sljedećem prioritetu:

1) izraz u zagradi
2) nizovna funkcija
3) multipliciranje stringa
4) nastavljanje stringova

6.9 evaluiranje znakovnog izraza

```
>>> '000' +'123'*5 '000123123123123123'
>>> 2*('abc' +'**')        'abc**abc**'
>>> '+' +'---+'*5
'+---+---+---+---+---+'
>>> '|' +'   |'*5
'|   |   |   |   |   |'
```

Nizovni operand koji sadrži domenu možemo prikazati sa

s [*i* | [*m*] : [*n*] [: [*o*]]]

gdje je $s=c_0c_1c_2...c_{k-1}$ niz znakova c_i duljine *k*, *m* i *n* su cjelobrojni izrazi koji predstavljaju početni i krajnji indeks stringa i *o* je cjelobrojni izraz, *o*≠0, koji predstavlja korak povećanja ili umanjenja indeksa. Postojat će sljedeći slučajevi:

Operand	*Značenje*
s **[** ***i*** **]**	Vraća znak c_i stringa ***s***, ako je ***-k≤i<k***, inače, dojavljuje se: IndexError: string index out of range

```
>>> '0123456789'[5]          '5'
>>> '0123456789'[-6]         '4'
>>> '0123456789'[]
SyntaxError: invalid syntax
```

`s [ : ]` String **s**.
```
>>>'0123456789'[:]  '0123456789'
```
`s [m : ]` Vraća:
1) string **s**, ako je `m == 0 or m <= -k`.
2) podstring stringa **s** od znaka s indeksom *m* do kraja niza, $c_m...c_{k-1}$, ako je `m != 0 and -k < m < k`.
3) prazan string, `''` , ako je `m >= k`.

```
>>> s = '0123456789'; i = 1
>>> while i <= len (s) :
    print( s[i :] ); i += 1
123456789
23456789
3456789
456789
56789
6789
789
89
9
```

U sljedećim smo slučajevima *m* i *n* sveli na apsolutne indekse:

```
m+i*o < n -> i < (n-m)/o
  if m < 0 : m += len(s)
  if n < 0 : n += len(s)
```

Operand	*Značenje*
`s [ m:n ]`	1) jednako je `s[m: ]` ako je `n>=k`. 2) podstring stringa **s** od znaka s indeksom *m* do znaka s indeksom *n*-1, ako je `m < n`. 3) prazan string, `''` , ako je `m >= n`.
`s [ :n ]`	`= s [ 0 : n ]`
`s [ : : ]`	`= s [ : ]`
`s [ m: : ]`	`= s [ m :  ]`
`s [ m:n: ]`	`= s [ m : n ]`

`s [ m:n:o ]`	`=` $c_m c_{m+o}...c_{m+i*o}$, i=1,...,$(n-m-1)/o$
	ako je `o>0 and m<n`
	`=` $c_m c_{m+o}...c_{m+i*o}$, i=1,...,$(m-n+1)/o$
	ako je `o<0 and m>n`.
	Inače, vraća prazan niz.
	`>>> '0123456789'[5:1:2]  ''`
	`>>> '0123456789'[-10:1:2]  '0'`
	`>>> '0123456789'[10:0:-1]`
	`'987654321'`

`s [ m:  :o ]`	`= s [ m : k : o ]`
	`>>> '0123456789'[10::-1]`
	`'9876543210'`

`s [ : n: o ]`	`= s [ 0  : n : o ]` ako je `o > 0`

	`= s [ k-1 : n : o ]`
	ako je `o < 0`
	`>>> '0123456789'[ : 2 :-2]`
	`'9753'`

`s [ : : o ]`	`= s [ : k : o ]`
	ako je `o > 0`
	`= s [ k-1 : : o ]`
	ako je `o < 0`
	`>>> '0123456789'[::2]  '02468'`
	`>>> '0123456789'[::2][::-1]`
	`'86420'`

STANDARDNE FUNKCIJE NAD ZNAKOVNIM NIZOVIMA

Nad stringovima su definirane znakovne, nizovne, cjelobrojne i logičke funkcije. Također su i neke poznate brojčane funkcije definirane nad znakovnim argumentom.

Znakovne funkcije

Standardne znakovne funkcije vraćaju znak kao rezultat svoga izračunavanja. Osim funkcije `chr()` i `unichr()` to su i poznate funkcije `min()` i `max()` koje nad stringom kao argumentom imaju značenje dano u sljedećoj tablici:

Tablica 6.1 – Standardne znakovne funkcije

Funkcija()	*Tip*	*Opis*	*Primjeri*
`max (s)`	z	Znak stringa *s* koji ima najveći ASCII kod.	`max('abcAc')` `'c'`
`min (s)`	z	Znak stringa *s* koji ima najmanji ASCII kod.	`min('a1b1c')` `'1'`

s-string, *z*-znak

Stringovne funkcije

Stringovne fukcije vraćaju string kao rezultat izračunavanja. Jedan dio standardnih stringovnih funkcija uveli smo još u trećem poglavlju. Bile su to funkcije za konverziju cijelih brojeva u binarni, oktalni ili heksadecimalni zapis. Sada im možemo dodati još jednu funkciju, `str()`, koja konvertira broj u string.

6.10 stringovne funkcije

```
>>> bin (2**16)   '0b10000000000000000'
>>> bin (11111)        '0b10101101100111'
>>> bin (97531)   '0b10111110011111011'
>>> bin (0b111**4)       '0b100101100001'
>>> oct (2**16)               '0o200000'
>>> oct (11111)                '0o25547'
>>> oct (0o777**4)       '0o774005774001'
>>> hex (2**16)                '0x10000'
>>> hex (0xABCDEF)             '0xabcdef'
```

```
>>> str (12345)                    '12345'
>>> str()                             ''
>>> str(12.55)                     '12.55'
>>> str(1+2*3)                        '7'
>>> str(2**7-1)                     '127'
>>> str('a')                          'a'
>>> a = 10**2; str(a)               '100'
>>> str(a**2)                     '10000'
```

MODUL str

Standardni modul **str** je ugrađeni modul (klasa). Sadrži veliki broj funkcija (metoda) korisnih za rad sa znakovnim vrijednostima.

```
>>> dir (str)
[..., 'capitalize', 'casefold', 'center',
'count', 'encode', 'endswith',
'expandtabs', 'find', 'format',
'format_map', 'index', 'isalnum',
'isalpha', 'isdecimal', 'isdigit',
'isidentifier', 'islower', 'isnumeric',
'isprintable', 'isspace', 'istitle',
'isupper', 'join', 'ljust', 'lower',
'lstrip', 'maketrans', 'partition',
'replace', 'rfind', 'rindex', 'rjust',
'rpartition', 'rsplit', 'rstrip', 'split',
'splitlines', 'startswith', 'strip',
'swapcase', 'title', 'translate', 'upper',
'zfill']
```

Ako je ***s*** string, ove se funkcije pozivaju prema pravilu:

s.*ime_funkcije* ([*argumenti*])

U nastavku dajemo pregled znakovnih funkcija.

s. capitalize()

Vraća string u kojem su sve pojave slova prevedene u mala slova, a prvi znak, ako je slovo, u veliko slovo.

```
>>>'pYtHON-1'.capitalize ()     'Python-1'
>>> for x in 'ććžšđ' :
    print( x.capitalize(), end = ' ' )
Ć Ć Ž Š Đ
```

s. center(*c* [, *z*])

Vraća string centriran unutar polja širine *c* dopunjen sa znakom *z* ili razmacima ako je *z* izostavljeno.

```
>>> '123'.center(2)              '123'
>>> '123'.center(8,'-')          '--123---'
>>> '123'.center(8)              '  123   '
```

s. ljust(*c* [, *z*])

Vraća string dopunjen u polju duljine *c* znakovima *z* ili razmacima ako je *z* izostavljeno.

```
>>> s = '12345'
>>> s.ljust (12)              '12345       '
>>> s.ljust (12, '-')         '12345-------'
```

s. lower()

Vraća string u kojem su sva velika slova engleskog alfabeta i slova 'Č', 'Ć', 'Đ', 'Š', 'Ž' konvertirana u ista takva mala slova.

```
>>> 'AbC.123'.lower()            'abc.123'
>>> 'ČĆŽŠĐ'.lower()              'čćžšđ'
>>> 'abc.123'.lower()            'abc.123'
>>> grčki_alfabet = 'Α Β Γ Δ Ε Ζ Η Θ Ι Κ Λ
Μ Ν Ξ Ο Π Ρ Σ Τ Υ Φ Χ Ψ Ω'
>>> print( grčki_alfabet ); print (
            grčki_alfabet.lower() )
Α Β Γ Δ Ε Ζ Η Θ Ι Κ Λ Μ Ν Ξ Ο Π Ρ Σ Τ Υ Φ Χ
Ψ Ω
α β γ δ ε ζ η θ ι κ λ μ ν ξ ο π ρ σ τ υ φ χ
ψ ω
```

s. lstrip([*z*])

Vraća string iz kojeg su izbačeni vodeći znakovi *z*, odnosno vodeći razmaci ako *z* nije navedeno.

```
>>> lstrip('   +++123')            '+++123'
>>> '+++123'.lstrip('+')           '123'
```

s. replace(*s1*, *s2* [, *c*])

Vraća string u kojem su sva pojavljivanja stringa *s1* zamijenjena stringom *s2*, odnosno najviše *c* pojavljivanja.

```
>>> s = '1,2,3,4.5.6'
>>> s. replace (',', '')      '1234.5.6'
>>> s. replace ('.', '')     '1,2,3,456'
>>> s. replace (',', '').replace ('.', '')
'123456'
>>> '121212'.replace ('12', 'ab', 1)
'ab1212'
```

s. rjust(*c* [,*z*])

Vraća string u polju duljine *c* koji ima prefiks sa znakovima *z* ili razmacima ako je *z* izostavljeno.

```
>>> s = '12345';  s. rjust (12)
'       12345'
>>> s. rjust (12, '-')
'-------12345'
```

s. rstrip([*s0*])

Vraća string iz kojeg su izbačeni znakovi *z* stringa *s0* ako se pojavljuju kao sufiks, ili razmaci ako *s0* nije navedeno.

```
>>> '   spacious   '.rstrip()
'   spacious'
```

```
>>> 'mississippi'.rstrip('ipz')
'mississ'
```

s. strip([s0])

Vraća string iz kojeg je izbačen prefiks i sufiks znakova *z* stringa *s0* ili razmaci ako *s0* nije navedeno.

```
>>> '     a b c     '.strip()      'a b c'
>>> '   spacious    '.strip()   'spacious'
>>> 'www.example.com'.strip('cmowz.')
'example'
```

s. swapcase()

Vraća string u kojem su mala slova konvertirana u ista takva velika, a velika u mala.

```
>>> print( 'ćčšđžĆČŠĐŽ'. swapcase() )
ĆČŠĐŽćčšđž
>>> print( 'ČćĆčŽžŠšĐđ'. swapcase() )
čĆčĆžŽšŠđĐ
```

s. title()

Vraća string u kojem su početna mala slova svake riječi konvertirana u ista takva velika slova, a ostala u mala.

```
>>> 'i1 b. c' . title()          'I1 B. C'
>>> 'i1 b. 123 c'.title()    'I1 B. 123 C'
>>> \
'čedomil ćićarija đurđa ŽELJKO' title()
'Čedomil Ćićarija Đurđa Željko'
```

s. upper()

Vraća string u kojem su sva mala slova engleskog alfabeta i slova 'č', 'ć', 'đ','š', 'ž' konvertirana u ista takva velika slova.

```
>>> 'abc-12xyzčćđ'. upper()
'ABC-12XYZČĆĐ'
>>> 'šš'. upper()                    'ŠŠ'
```

s. zfill(*d*)

Dopisuje *d*-`len(s)` vodećih nula brojčanom stringu. Ako je *d*-`len(s)`<0 vraća s bez promjene.

```
>>> '123'.  zfill (5)              '00123'
>>> '123'.  zfill (2)                '123'
>>> '123.5'. zfill (12)     '0000000123.5'
```

LOGIČKE FUNKCIJE NAD STRINGOVIMA

Ako je ***s*** string, postoji veći broj logičkih funkcija (svojstava ili atributa) nad stringovima, tablica 6.2, koje se pozivaju prema pravilu:

***s*.*ime_funkcije*()**

Tablica 6.2 – Standardne logičke funkcije

***Funkcija*()**	***Opis***
s.isalnum()	True, ako su svi znakovi stringa *s* alfanumerički (slova i/ili brojke). '123ABCĆŽ缁' .isalnum() True 'Python 2.7.8' .isalnum() False '0123456789' .isalnum() True
s.isalpha()	True, ako su svi znakovi stringa *s* slova. 'SamoSlova' .isalpha() True 'Samo slova' .isalpha() False 'abcčČćĆžŽšŠđĐД'.isalpha() True
s.isdigit()	True, ako su svi znakovi stringa *s* brojke. '0123456789' .isdigit() True '123ABCĆŽ' .isdigit() False '3.1415926' .isdigit() False
s.islower()	True, ako su svi znakovi stringa *s* mala slova. 'MALASLOVA' .islower() False 'velikaslova' .islower() True
s.isspace()	True, ako su svi znakovi stringa *s* blankovi (razmaci). ' ' .isspace() True 10*' ' .isspace() 10 (10*' ') .isspace() True '... --- ...' .isspace() False
s.istitle()	True, ako riječi stringa *s* koje počinju slovom počinju s velikim slovom. 'I B C' .istitle() True 'I1 B. C' .istitle() True 'I1 B. 123 C' .istitle() True 'I... abc' .istitle() False
s.isupper()	True, ako su sva slova stringa *s* velika. 'A * B - 15' .isupper() True '123.5 * a**2' .isupper() False

s. endswith(*s0* [,*i1* [,*i2*]])

Vraća **True** ako ***s*** završava sufiksom ***s0***, inače **False**. To vrijedi ako su indeksi ***i1*** i ***i2*** izostavljeni. Značenje funkcije **endswith()** s indeksima je sljedeće:

s.endswith(*s0*,*i1*) → s[i1:].endswith(*s0*)
s.endswith(*s0*,*i1*,*i2*)
→ s[i1:i2].endswith(*s0*)

```
>>> '0123456789'. endswith ('9')      True
>>> '0123456789'. endswith ('9', -1) True
>>> '0123456789'. endswith ('67')    False
>>> '0123456789'. endswith ('67',0,7)
False
>>> '0123456789'. endswith ('67',0,8)
True
```

s. startswith(*s0* [,*i1* [,*i2*]])

Vraća **True** ako ***s*** počinje s prefiksom ***s0***, inače **False**. Ako je zadan indeks ***i1***, provjerava se prefiks stringa **s[i1:]**, a ako je zadan i indeks ***i2***, prefiks stringa **s[i1:i2]**. Primjeri:

```
>>> '012345'.startswith('9')      False
>>> '012789'.startswith ('9',-1)   True
>>> '0136789'.startswith ('0')     True
>>> '0123456'.startswith ('5',5)   True
>>> '0123456'.startswith ('234',2) True
```

IMENA

Znamo da se naši dijakritički znakovi mogu rabiti u imenima varijabli i funkcija, premda smo to izbjegavali. Međutim, u nekim smo programima koristili neke posebne znakove (slova) iz cijelog skupa znakova, posebno grčkog alfabeta. Grčki se alfabet nalazi od rednog broja 913 do 969:

```
    0 1 2 3 4 5 6 7 8 9
----------------------
 91       Α Β Γ Δ Ε Ζ Η
 92 Θ Ι Κ Λ Μ Ν Ξ Ο Π Ρ
 93 ? Σ Τ Υ Φ Χ Ψ Ω Ϊ Ϋ
 94 ά έ ή ί ϋ α β γ δ ε
 95 ζ η θ ι κ λ μ ν ξ ο
 96 π ρ ς σ τ υ φ χ ψ ω
>>> 'φ'. upper(), ord ('φ')    ('Φ', 981)
```

Sada možemo dati konačno pravilo za tvorbu imena u Pythonu:

```
ime     : prefiks ( prefiks | brojka )*
prefiks : _ | alpha_znak
brojka  : [0-9]
```

gdje je ***alpha_znak*** bilo koji znak ***c*** Unicode tablice za kojeg je ***c.isalpha()*** jednako True.

```
>>> chr (960)                          'π'
>>> 'π'.isalpha ()                    True
>>> EUR = chr(892)
>>> print( EUR, EUR.isalpha() )  ϵ True
```

CJELOBROJNE FUNKCIJE NAD STRINGOVIMA

Osim funkcije len(), nad stringovima ***s*** definirano je još nekoliko cjelobrojnih funkcija:

s. find(*s0* [,*i1* [,*i2*]])

Vraća najmanji indeks pozicije pojavljivanja stringa ***s0*** u stringu ***s***, ako je ***s0*** in ***s[i1:i2]***, inače -1.

```
>>> '012323456'.find ('2', 1)          2
>>> '0134563456'.find ('345')          3
>>> '0123453456'.find ('345', 2, 10)   3
>>> '0123453456'.find ('345', 2, 5)   -1
```

s. rfind(*s0* [,*i1* [,*i2*]])

Vraća najveći indeks (prvi zdesna) pozicije pojavljivanja stringa ***s0*** u stringu ***s***, ako je ***s0*** in ***s[i1:i2]***, inače -1.

```
>>> '01234560123'.rfind ('2', 1)       9
>>> '01234560123456'.rfind ('345')    10
>>> '0134563456'.rfind ('345',2,10)    6
>>> '01234503456'.rfind ('345',2,5)   -1
```

s. index(*s0* [,*i1* [,*i2*]])

Vraća najmanji indeks pozicije pojavljivanja stringa ***s0*** u stringu ***s***, ako je ***s0*** in ***s[i1:i2]***, kao i find(), ali dojavljuje ValueError ako ***s0*** not in ***s[i1:i2]***.

```
>>> '0123456'.index ('2', 1)           2
>>> '123'.index ('0')
ValueError: substring not found
```

PRETVORBA STRINGA U BROJ

Poznate standardne cjelobrojne i realne funcije, int() i float(), „rade“ i za argument znakovnog tipa, ali pod uvjetom da može biti interpretiran kao cijeli ili realni broj.

››› 6.11 int() i float()

```
>>> int ('1'+'23')                    123
>>> int ('1'+'2'*3)                  1222
>>> int ('1' +'0'*10)          10000000000
>>> int ('12.5')
ValueError: invalid literal for int()
>>> int ('1'+'2*3')
ValueError: invalid literal for int()
>>> float ('123')                    123.0
>>> float ('924324234234')
924324234234.0
>>> float ('123'*10)
1.2312312312312312e+29
>>> x='123.456789'; float (x) 23.456789
>>> round (float (x), 4)          123.4568
```

MODUL string

Modul string sadrži nekoliko konstanti, jednu funkciju i dvije klase za rad sa znakovnim nizovima.

››› 6.12 modul string

```
>>> import string
>>> dir (string)
```

```
['Formatter', 'Template', '_ChainMap',
..., 'ascii_letters', ascii_lowercase',
'ascii_uppercase', 'capwords', digits',
'hexdigits', 'octdigits', 'printable',
'punctuation', 'whitespace']
```

>>> 6.13 konstante

```
>>> from string import *
>>> ascii_letters
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKL
MNOPQRSTUVWXYZ'
>>> ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'
>>> ascii_uppercase
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> digits                 '0123456789'
>>> hexdigits  '0123456789abcdefABCDEF'
>>> octdigits                '01234567'
>>> printable
'0123456789abcdefghijklmnopqrstuvwxyzAB
CDEFGHIJKLMNOPQRSTU...
>>> punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
>>> whitespace      ' \t\n\r\x0b\x0c'
```

GOVORIMO PYTHONSKI

Proširujemo Moj_modul.py sa

```
from string import *
α_    = lambda x : x.isalpha()
slova = lambda x : (
'slov'+'o'*(len(x)==1)
  +'a'*(len(x)>1) if α_ (x) else
  'nije slovo'    if len(x) == 1
  else 'nisu slova' )
>>> from Moj_modul import *
>>> slova ('A1')           'nisu slova'
>>> slova ('1')            'nije slovo'
>>> slova ('Šansona')      'slova'
>>> α_    ('Šansona')      True
```

MALO PRETRAŽIVANJA UNICODE TABLICE ZNAKOVA

Različitim kodnim točkama i imenima možete pristupiti pretraživanjem weba, opisom potrebnih znakova ili pomoću određenog web mjesta kao što je

http://unicode-table.com.

Ako kliknete na „Unicode blocks“ dobit ćete pregled ranga kodnih stranica skupina (alfabeta) znakova. Na primjer:

```
0000–001F Control character
0020–007F Basic Latin
0080–00FF Latin-1 Supplement
0100–017F Latin Extended-A
0180–024F Latin Extended-B
0250–02AF IPA Extensions
02B0–02FF Spacing Modifier Letters
0300–036F Combining Diacritical Marks
0370–03FF Greek and Coptic
0400–04FF Cyrillic ...
```

```
>>> for i in range (0x0370, 0x03FF +1) :
    print( chr (i), end = ' ')
Ͱ ͱ Ͳ ͳ ʹ ͵ Ͷ ͷ □ □ ͺ ͻ ͼ ͽ ; Ϳ □ □ □ □ ΄ ΅ Ά
· Έ Ή Ί □ Ό □ Ύ Ώ ΐ Α Β Γ Δ Ε Ζ Η Θ Ι Κ Λ Μ Ν
Ξ Ο Π Ρ □ Σ Τ Υ Φ Χ Ψ Ω Ϊ Ϋ ά έ ή ί ΰ α β γ δ
ε ζ η θ ι κ λ μ ν ξ ο π ρ ς σ τ υ φ χ ψ ω ϊ ϋ
ό ύ ώ Ϗ ϐ ϑ ϒ ϓ ϔ ϕ ϖ ϗ Ϙ ϙ Ϛ ϛ Ϝ ϝ Ϟ ϟ Ϡ ϡ Ϣ
ϣ Ϥ ϥ Ϧ ϧ Ϩ ϩ Ϫ ϫ Ϭ ϭ Ϯ ϯ ϰ ϱ ϲ ϳ ϴ ϵ ϶ Ϸ ϸ Ϲ
Ϻ ϻ ϼ Ͻ Ͼ Ͽ
>>> ͼ = ͼ_kn = .54124; 125 *ͼ      942.655
```

Prepustimo se malo svojoj mašti pa pretražimo Unicode tablicu znakova:

```
>>> for i in range(2**15, 2**15-100,-1):
    print (chr (i), end = ' ')
耀 翿 翾 翽 翼 翻 翺 翹 翸 翷 翶 翵 翴 翳 翲 翱 翰
... 羫 羪 義 羨 羧 羦 羥 群 羣 羢 羡 羠 羟 羞 羝
>>> chr (k -1)                      '翿'
>>> chr (k -255)                    '缁'
>>> '缁' .isalpha()                 True
>>> '胿' .isalpha()                 True
```

HRVATSKA ABECEDA

Poseban je problem leksičko uređenje riječi hrvatskoga jezika. Znakovi (slova) hrvatskoga jezika koji se ne nalaze u engleskoj abecedi imaju redne brojeve koji su “raštrkani” u Unicode tablici i nisu u rastućem nizu. Na primjer, kodovi (redni brojevi) hrvatskih slova su:

```
>>> hr = """
HR = 'ČčĆćĐđŠšŽž'
for c in HR : print( c, end = '   ' )
```

```
print ()
for c in HR :
  print( ord(c), end = ' ' ) """
>>> exec( hr )
Č   č   Ć   ć   Đ   đ   Š   š   Ž   ž
268 269 262 263 272 273 352 353 381 382
```

Da bismo mogli usporediti dva niza koji mogu sadržavati i slova hrvatske abecede moramo oformiti niz, A, koji će predstavljati hrvatsku abecedu, prvo velika, potom mala slova, i nije ovisan o kodnoj stranici:

HR_abeceda.py

```
A_ = 'ABCČĆDĐ'
for k in range( ord( 'E' ),
      ord( 'S' ) +1 ) : A_ += chr( k )
A_ += 'Š'
for k in range( ord( 'T' ),
      ord( 'Z' ) +1 ) : A_ += chr( k )
A_ += 'Ž'; a_ = A_.lower()
HR_ = A_ + a_
>>> print( A_ ); print( a_ )
ABCČĆDĐEFGHIJKLMNOPQRSŠTUVWXYZŽ
abcčćdđefghijklmnopqrsštuvwxyzž
```

Ako su s1 i s2 dva niza koje treba usporediti, evo dijela programa koji to čini:

usporedba_HR.py

```
from HR_abeceda import HR_
Greška = ("Moraju biti samo slova
hrvatske" + "abecede")
while 'unos' :
  s1 = input ('Unesi prvi niz znakova  ')
  if not s1 : break
  if not s1.isalpha() :
     print( Greška );  continue
  s2 = input(
       'Unesi drugi niz znakova ' )
  if not s2.isalpha() :
     print( Greška );  continue
  k = min( len(s1), len(s2) )
  for i in range( k ) :
    c1, c2 = s1[i], s2[i]
    if c1 not in HR_ or c2 not in HR_ :
      print (Greška);  break
    if c1 != c2 :
      i1, i2 = ( HR_.find( c1 ),
                 HR_.find( c2 ) )
      if i1 < i2 : print(s1, s2)
      else       : print(s2, s1)
      break
  else :
    print( (s1 +' ' +s2)
          if len(s1) < len(s2)
          else s2 +' ' +s1 )
>>>
Unesi prvi niz znakova  Ljubić
Unesi drugi niz znakova Lovrić
Ljubić Lovrić
>>>
Unesi prvi niz znakova  Čačić
Unesi drugi niz znakova Ćaćić
Čačić Ćaćić
>>>
Unesi prvi niz znakova  Marković
Unesi drugi niz znakova Markov
Markov Marković
>>>
Unesi prvi niz znakova  <Enter>
```

I to je to, pomislit ćemo. Ali, nije! Zaboravili smo da hrvatski jezik ima tri glasa (fonema) koji se pišu s po dva slova. To su 'DŽ', 'LJ' i 'NJ'. 'DŽ' je u hrvatskoj abecedi poslije 'D', 'LJ' poslije 'L' i 'NJ' poslije 'N'. Da bismo to postigli jedno od mogućih rješenja jest da za svaki fonem imamo dva znaka. Onim znakovima koji iza sebe nemaju hrvatske foneme bit će dodan razmak. Na primjer, 'A' će biti 'A '. Znaku iza kojeg slijedi slovo hrvatskog alfabeta bit će dodan znak 'X'. To su znakovi 'C', 'D', 'S' i 'Z'. Uređenje 'C'<'Č'<'Ć' postignuto je tako da se 'Č' prevodi u 'CY', a 'Ć' u 'CZ'. Dakle, ako trebamo usporediti dva ulazna niza, uz dodatni uvjet da velika i mala slova imaju isti redni broj, preveli bismo ih u dva pomoćna niza prema danim pravilima, odnosno tablici za iznimke:

```
C  Č  č  Ć  ć  D  Đ  đ  S  Š  š  Z  Ž  ž
CX CY CY CZ CZ DX DZ DZ SX SY SY ZX ZY ZY
```

Potom bismo pretražili te nizove i zamijenili svako pojavljivanje podniza 'N J ' u 'NJ' i podniza 'L J ' u 'LJ', te podniz 'DXZY' koji je nastao od 'DŽ' u 'DY', jer je 'DŽ' ispred 'Đ' ('DZ'). Evo kompletnog programa za prevođenje "neuređenog" niza u uređeni.

Sort_HR.py

```
# Uređenje niza koji sadrži hrvatske
# foneme Č, Ć, ...
A = ('C  Č  č  Ć  ć  D  Đ  đ  S  '
   + 'Š  š  Z  Ž  ž  ')
B = ('CX CY CY CZ CZ DX DZ DZ SX '
   + 'SY SY ZX ZY ZY ')
```

```
Unos  = """
Naziv = input( 'Upiši niz: ' ); Q = ''
for c in Naziv :
    k = A. find( c +'  ' )
    Q += B[k :k+2] if k >= 0 else \
         c.upper() +' '

Q = Q. replace( 'DXZY', 'DY' )
Q = Q. replace( 'N J ', 'NJ' )
Q = Q. replace( 'L J ', 'LJ' )  """
exec( Unos ); N1, Q1 = Naziv, Q
exec( Unos ); N2, Q2 = Naziv, Q
S = (N1 +' ' +N2) if Q1 <= Q2 else \
    (N2 +' ' +N1)
print( S )
```

```
>>>
Upiši niz: Ljubić
Upiši niz: Lovrić
Lovrić Ljubić
```

DVOZNAČNOSTI ZNAKOVA

Ako pregledamo prvih desetak kodnih stranica Unicode tablice uočit ćemo ponavljanje mnogih znakova. Mislimo na njihov grafički prikaz. Na primjer:

```
>>> chr(65), chr(913)          ('A', 'Α')
>>> ord ('A'), ord ('Α')        (65, 913)
```

Vidimo da nema razlike u grafičkom prikazu slova `'A'`, koji ima redni broj (kôd) jednak 65, i velikog grčkog slova alfa, koji ima kod 913! Ako napišemo:

```
>>> A = 123;  print( A )
... A = 123;  print ( A )
NameError: name 'A' is not defined
```

Očigledno su ime A u naredbi za pridruživanje i ime A u *naredbi PRINT* dva različita imena! Da bi *naredba PRINT* ispisala sadržaj varijable A, moramo je kopirati iz naredbe za pridruživanje:

```
>>> print( A )                    123
```

Dakle, izbjegavajmo takve dvoznačnosti. Ako koristimo grčka slova onda za imena izaberimo mala slova, kopirajući ih iz tablice dane u modulu **Moj_modul.py**.

DULJINA CIJELOG BROJA

Pretvorbom cijeloga broja u string i uporabom funkcije `len()` možemo dobiti broj znamenki (duljinu) cijeloga broja. Na primjer:

```
>>> len (str (2**1024))           309
```

Često ćemo trebati dani niz „skratiti" za posljednji znak. Na primjer, ako učitavamo linije teksta koje završavaju s `'\n'`. Tada ćemo koristiti

```
s = s[:-1].
```

PALINDROMI

Palindromi su stringovi koji se jednako čitaju slijeva i zdesna. String *s* je palindrom ako vrijedi:

```
s.lower() == s[::-1].lower()
```

Treba ispisati sve brojeve iz intervala 1 do 1000000 koji se jednako čitaju slijeva i zdesna (palindromi), a da su im drugi korijeni cijeli brojevi. To su 1, 4, 9, 121, 484 itd. Konverzijom kvadrata brojeva, *n*, u stringove *b*, rješenje je jednostavno. Palindromi su brojevi *n* za koje je *b* jednako inverznom *b*. Gornja granica iteriranja je 1000 (drugi korijen iz 1000000).

Palindrom_n2.py

```
print( "   n  n**2\n-------------" )
for n in range( 1, 1001 ) : #
    b = str( n**2 )
    if b == b[::-1] :
       print( "%4d  %s" % (n, b) )
```

```
>>>
   n  n**2
------------
   1  1
   2  4
   3  9
  11  121
  22  484
  26  676
 101  10201
 111  12321
 121  14641
 202  40804
 212  44944
 264  69696
 307  94249
 836  698896
```

Ako analiziramo rečenice, onda je to palindrom ako se jednako čita slijeva i zdesna, ignorirajući razmake i znakove interpunkcije, kao na primjer „U Rimu idu ljudi u miru!".

Neka je W rečenica koju provjeravamo je li palindrom. Treba je reducirati kako smo opisali, svesti na mala slova i dodati dio preslikavanja naših glasova dž, lj i nj u #, $, %, jer bi njihov obrnuti niz bio žd, jl i jn.

Je_li_palindrom.py

```
from string import *
W = input ('Unesi rečenicu:\n')
w = ''
for c in W :
  w += (c if c not in punctuation +' '
        else '')
w = w.lower()
# -- ako je rečenica HR --
w = w.replace ('dž', '#')
w = w.replace ('lj', '$')
w = w.replace ('nj', '%')
# ------------------------
P = w == w[::-1]
print ('PALINDROM'*P
       +'NIJE PALINDROM' *(not P) )
```

```
>>>
Unesi rečenicu:
U Rimu idu ljudi u miru!
PALINDROM
>>>
Unesi rečenicu:
Anja sebe sanja.
PALINDROM
>>> w # reducirana rečenica
'a%asebesa%a'
```

ISPIS PORUKE PRI UNOSU PODATAKA

Funkcija `input()` sadrži string kao poruku za unos podataka. Ako unosimo niz podataka možemo u poruku „ugraditi" redni broj podatka. Na primjer:

```
>>> i = 0
>>> while 'ima podataka' :
    i += 1; x  = eval(
       input('Unesi %d. podatak ' % i) )
    if x == -1 : break

Unesi 1. podatak 181
Unesi 2. podatak 178
Unesi 3. podatak 182
Unesi 6. podatak -1
```

P R O G R A M I

Osam primjera programa danih u ovom dijelu potpuno ilustriraju rad s znakovnim tipom podataka. Neke smo programe priložili bez dodatnog komentara jer smatramo da su trivijalni.

ALFABET STRINGA

Učitati string i ispisati alfabet (znakove) od kojih je sačinjen. Razmak (blank) ne prikazivati kao dio alfabeta.

Alfabet.py

```
s = input( 'Upiši niz znakova\n' )
S = s.replace (' ', '')
A = ''
for c in S : A += c *(c not in A)
print( * A )
```

Ovdje smo umjesto

```
if c not in A : A += c
```

pisali

```
A += c *(c not in A)
```

```
>>>
Upiši niz znakova
Ovaj niz znakova napisan je nad
alfabetom:
O v a j n i z k o p s e d l f b t m :
```

PREBROJAVANJE ZNAKOVA STRINGA

Sljedeći program prebrojava koliko ima slova, brojki i ostalih znakova u učitanom znakovnom nizu. Pokazali smo kako se može riješiti problem „čitanja" izlaznih podataka vodeći računa da riječi slovo, brojka ostali znak budu napisani u odgovarajućem padežu jednine i/ili množine. Da bismo mogli testirati kako će te riječi biti „izgovorene" za veći broj znakova, dodali smo varijablu `Test` koja nam dopušta da unosimo izraze sa znakovnim nizovima. Njezinim isključenjem, učitava se string z bez promjene.

Prebroj.py

```
# Prebrojavanje slova, brojki i ostalih
# znakova ulaznog znakovnog niza
_2_4 = lambda x : ( 2 <= x%10 <= 4 and
       ( x%100 < 12 or x%100 > 14) )
Test = True
while 1 :
  z = input( 'Unesi niz znakova ' )
  if not z : break
  if Test  : z = eval(z)
  S = B = O = 0
  for c in z :
    if c.isupper() or c.islower() :
                      S += 1
```

```
    elif c.isdigit() : B += 1
    elif c != ' '    : O += 1
  s, b, o = (
    'slovo' if S == 1 or S%10 == 1 and
              S%100 != 11 else 'slova',
    'brojka' if B == 1   else
    'brojke' if _2_4 (B) else 'brojki',
    'ostali znak'  if O == 1   else
    'ostala znaka' if _2_4 (O) else
    'ostalih znakova' )
  print(
    ("%d " +s +", %d " +b +" i %d " +o)
     % (S, B, O) )
```

```
>>>
Unesi niz znakova "Python 3.9.1, Dec  7
2020, 17:08:21"
9 slova, 14 brojki i 6 ostalih znakova
Unesi niz znakova 'a0.' *114
114 slova, 114 brojki i 114 ostalih
znakova
Unesi niz znakova 'x9*'
1 slovo, 1 brojka i 1 ostali znak
Unesi niz znakova <Enter>
```

PROVJERA ZAPORKE

Treba provjeriti je li zaporka napisana prema danim pravilima, a potom provjeriti je li ponovljena zaporka jednaka originalu. Pravila pisanja zaporke su sljedeća:

1) duljina je 6 do 12 znakova
2) dopuštena je uporaba svih znakova osim razmaka
3) mora sadržavati najmanje jedno veliko i jedno malo slovo engleske ili hrvatske abecede
4) mora sadržavati najmanje jednu brojku

Zaporka.py

```
V = M = B = R = False;
z = input( 'Unesi zaporku ' )
if 5 < len(z) < 13 :
  for c in z :
    V = V or c.isupper()
    M = M or c.islower()
    B = B or c.isdigit()
    R = R or c.isspace()
  if V and M and B and not R :
    if z == input( 'Ponovi zaporku ' ) :
       print( 'Zaporka je prihvaćena' )
    else : print(
           'Zaporka NIJE prihvaćena!' )
  else : print('Ne može biti zaporka!' )
  else   : print( 'Zaporka mora imati '
           '6 do 12 znakova!' )
```

```
>>>
Unesi zaporku Žacques1
Ponovi zaporku Žacques1
Zaporka je prihvaćena
```

NAJVEĆI PALINDROM

U sljedećem programu izračunava se najveći palindrom dobiven umnoškom dvaju dvoznamenkastih, troznamenkastih ili četveroznamenkastih brojeva.

Palindrom.py

```
palindrom = lambda n : (str(n) ==
                        str(n)[::-1])
while 1 :
  b = input( 'Zadaj broj znamenki ' )
  if b in "234" and len(b) == 1 : break
b = int(b);  p = 10**b -1;  q = p//10
Max = 0
for i in range( p, q, -1 ) :
  for j in range( i, q, -1 ) :
    n = i*j
    if palindrom (n) and n > Max :
      I, J, Max = i, j, n; print( Max )
print( Max, '=', I, '*', J )
```

```
>>>
Zadaj broj znamenki 2
9009
9009 = 99 * 91
>>>
Zadaj broj znamenki 3
580085
906609
906609 = 993 * 913
>>>
Zadaj broj znamenki 4
99000099
99000099 = 9999 * 9901
```

PRETVORBA CIJELOG BROJA U BAZU 2 DO 16

Znamo, uz pomoć funkcija `bin()`, `oct()` i `hex()`, pretvoriti dekadski cijeli broj u binarni, oktalni ili heksadecimalni cijeli broj: Ovdje ćemo pokazati kako ga možemo pretvoriti i u preostale baze.

Cijeli broj N_{10}, napisan u bazi 10, pretvaramo u N_b, u bazi b, tako što ga dijelimo s bazom b i zapisujemo

količnik i ostatak. Ostatak zapisujemo, a to je posljednja znamenka broja N_b. Nastavljamo s dijeljenjem količnika s b i svaki put dopisujemo ostatak dijeljenja ispred pretvorenoga broja. Postupak je okončan kad količnik bude jednak 0. Znamenke broja N_b su od 0 do b-1. Ako je b>10, znamenke veće od 9 označene su slovima a, b, ..., f. Sada nije problem napisati program koji će zadani dekadski broj pretvoriti u brojeve u bazi 2 do 16.

Baza_2_do_16.py

```
# Prevodjenje dekadskih cijelih brojeva
# u drugu bazu, od 2 do 16

from Moj_modul import *

Z = '0123456789abcdef'
while 1 :
  N = Input ('Upišite cijeli broj ')
  if Int (N) and N >= 0 : break
print( NL, N, '(10)' )

for b in range( 2, 17 ) :
  k = N;  Nb = ''
  while k > 0 : k,i = divmod( k, b ); \
                Nb = Z[i] +Nb
  print( '   = (%2d)' % b, Nb )
```

```
>>>
Upišite cijeli broj 2**31

2147483648 (10)
   = ( 2) 10000000000000000000000000000000
   = ( 3) 12112122212110202102
   = ( 4) 2000000000000000
   = ( 5) 13344223434043
   = ( 6) 553032005532
   = ( 7) 104134211162
   = ( 8) 20000000000
   = ( 9) 5478773672
   = (10) 2147483648
   = (11) a02220282
   = (12) 4bb2308a8
   = (13) 282ba4aab
   = (14) 1652ca932
   = (15) c87e66b8
   = (16) 80000000
```

Ovdje treba napomenuti da funkcija int() pretvara broj napisan u bilo kojoj bazi od 2 do 36 u dekadski broj. Piše se prema sintaksi:

```
int ( string, baza )
```

Primjeri:

```
>>> int ('282ba4aab', 13)   2147483648
>>> int ('282ba4aab', 20)    61604836211
>>> int ('123456789', 36) 2984619134745
>>> int ('ZZZZ', 36)            1679615
>>> int ('xyz', 36)               44027
>>> 35 +34*36 +33*36**2 # provjera
44027
>>> int ('xyz', 35)
ValueError: invalid literal for int()
with base 35: 'xyz'
```

Igra križić-kružić

Popularna igra križić-kružić (tic-tac-toe) može nam poslužiti kao pregled mogućnosti primjene strukture stringa. Ako polja matrice 3x3 označimo brojevima:

```
789
456
123
```

možemo definirati string S koji će sadržavati oznake redova, stupaca i dijagonala te matrice:

```
S = '789 456 123 147 258 369 159 357'
```

Igraju igrači označeni s 'X' i 'O'. Igru započinje slučajno odabrani igrač I i postavlja svoj znak na jednu od slobodnih lokacija (na početku su slobodna sva polja, string B). Izabrana lokacija bit će zamijenjena znakom igrača na svim mjestima pojavljivanja u stringu S. Provjerava se je li ispunjen uvjet

```
3*I in S
```

Ako jest, igrač I je pobjednik jer ima označen redak, stupac ili jednu od dijagonala. Inače, brojač poteza b povećava se za 1 i označeno se polje izbacuje iz B. Postupak se nastavlja sve dok jedan od igrača ne ispuni dani uvjet ili su sva polja popunjena, bez pobjednika.

Križić_kružić.py

```
# Igra križić - kružić
from random import random
"""
   789
   456
   123 """
S = '789 456 123 147 258 369 159 357'
T = ' '*4; NLT = '\n' +T
Prikazi = lambda : print( T +S[ :4] +NLT
                  + S[4:7] +NLT +S[8:11])
X = 'X';  O = 'O';  B = '123456789'
I = X if random()<0.5 else O; Prikazi()
```

```
while B :
  while 1 :
    a = input( 'Igra ' +I +' > ' )
    if len( a ) == 1 and a in B : break
  B = B. replace( a, '' )
  S = S. replace( a, I ); Prikazi()
  if 3*I in S :
     print(  'BRAVO,', I ); break
  I = X if I == 0 else 0
else : print( 'NERIJEŠENO!' )
```

```
>>>
   789          789          089       089
   456          4X6          4X6       4X6
   123          123          123       X23

Igra X > 5  Igra O > 7  Igra X > 1  Igra O > 9

   08O          0XO          0XO       0XO
   4X6          4X6          4X6       XX6
   X23          X23          XO3       XO3

Igra X > 8  Igra O > 2  Igra X > 4  Igra O > 6

   0XO          0XO
   XXO          XXO
   XO3          XOX

Igra X > 3  NERIJEŠENO!
```

Crtanje funkcija sin() i cos()

sin_cos.py

```
# Crtanje funkcija sin(x) i cos(x) na
# intervalu [0.0, π]

from math import cos, sin, pi as π
A = 20;  print ( )
print( 'x' +' '*11 +'-1' +' '*(A-1)+'0'
       +' '*(A-1) +'1' )
print( ' '*12 +'-'*(2*A+2) )
s0 = ' '*(A+5) +'|' +' '*A;  x = 0

while x <= π :
  y = A+5 +int( round( A *sin(x) ) )
  s = s0[:y] +'*' +s0[y+1:]
  y = A+5 +int( round( A *cos(x) ) )
  s = s [:y] +'+' +s [y+1:]
  print( "%6.4f  %s" % (round( x, 4 ),
         s) )
  x += π /20

print()
print(
    " "*A +"'*' sin(x)    '+' cos(x)" )
```

```
x          -1                  0                  1
           ----------------------------------------
0.0000                         *                  +
0.1571                         |  *               +
0.3142                         |     *            +
0.4712                         |        *         +
0.6283                         |           *   +
0.7854                         |             +
0.9425                         |           +   *
1.0996                         |        +        *
1.2566                         |     +           *
1.4137                         |  +              *
1.5708                         +                 *
1.7279                    +    |                 *
1.8850                 +       |                 *
2.0420              +          |                 *
2.1991           +             |                 *
2.3562        +                |              *
2.5133     +                   |           *
2.6704  +                      |        *
2.8274 +                       |     *
2.9845 +                       |  *
3.1416 +                       *
                  '*' sin(x)    '+' cos(x)
```

Prevođenje arapskih brojeva u rimske

Na kraju dajemo program za prevođenje arapskih brojeva u rimske. Arapski broj `a` prevodi se u string `A` duljine 4, s vodećim nulama ako je broj manji od `1000`. Inicijalno rimski broj `Rim` sadrži tisuće (ako ih ima). Potom se nastavlja s generiranjem prijevoda stotica desetica i jedinica definirano uvjetnim izrazom.

String `R` za indeks `k` jednak `0`, `3` i `6` sadrži prijevode stotica, za `9`, `12` i `15` desetica i za `18`, `21` i `24` jedinica ovisno o brojki `i`. Provjerili smo valjanost algoritma na uzorku od `10` slučajnih arapskih brojeva.

Arap_rim.py

```
# Prevođenje arapskih brojeva u rimske
from Moj_modul import *
R  = "CM D  C  XC L  X  IX V  I  "
#     0  3  6  9  12 15 18 21 24
r  = lambda i : R[i: i+2]
for n in range( 10 ) :
  a = randint(1, 3999 ); A = "%04d" % a
  Rim = 'M' *int(A[0]);  k = 0
  for I in range (1, 4) :
    i = int (A[I])
    if not i : k += 9; continue
    Rim += (
    r(k+6) *i       if i <= 3 else
    r(k+6) +r(k+3) if i == 4 else
    r(k+3) +r(k+6) *(i-5) if i < 9 else
    r(k) )
    k += 9
```

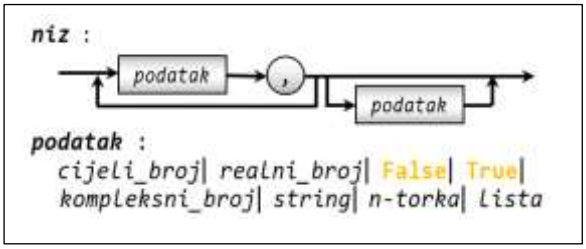
```
    Rim = Rim. replace( ' ', '' )
    print( "%4d --> %s" % (a, Rim) )
>>>
2242 --> MMCCXLII
1191 --> MCXCI
2736 --> MMDCCXXXVI
1547 --> MDXLVII
 285 --> CCLXXXV
 340 --> CCCXL
 756 --> DCCLVI
 461 --> CDLXI
3964 --> MMMCMLXIV
2380 --> MMCCCLXXX
```

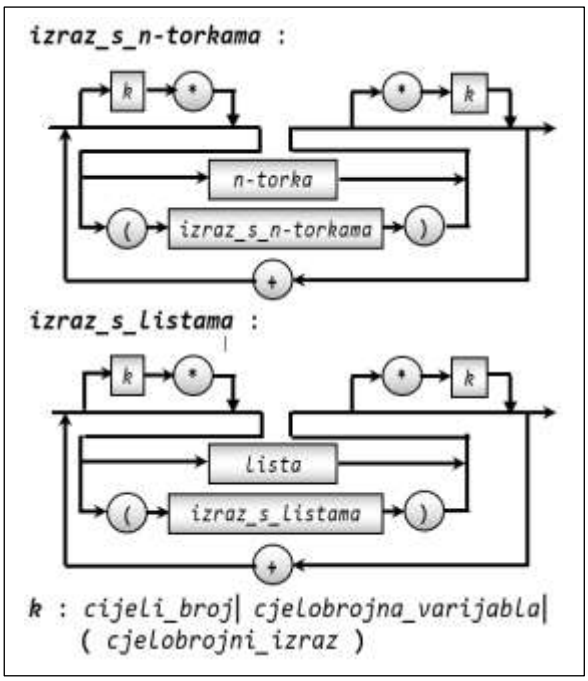
7.

NIZOVI: *n*-TORKE I LISTE

Python ima dvije standardne strukture podataka, n-torke i liste, koje predstavljaju kolekcije uređenih nizova podataka. S obzirom na to da te dvije strukture podataka imaju dosta zajedničkih svojstava, opisujemo ih zajedno u ovom poglavlju. Razlika između njih je što su n-torke nepromjenljiv („immutable"), a liste promjenljiv tip (klasa) podataka.

Znajući da su nizovi važni u programiranju, posebno pružaju „elegantna" rješavanja mnogih problema, dali smo veliki broj programa.





Eratos.py

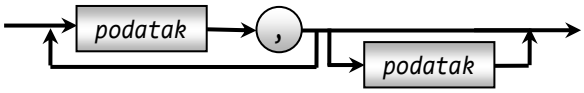
```
# Eratostenovo sito - prim brojevi
# od 2 do N
while 'N < 2' :
  N = int( eval( input(
      'Zadaj gornju granicu >=2 za '
      'ispis prim brojeva ')))
  if N >= 2 : break
P = ['X']*2 +list( range(2, N+1) )
for i in range( 2, int (N**0.5) +1 ) :
    if P[i] == 'X' : continue
    p = 2 *i
    P [p: : i] = ['X'] *len(P [p: : i])
while 'X' in P : P. remove( 'X' )
print( * P )
```

```
>>>
Zadaj gornju granicu >=2 za ispis prim
brojeva 100
2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
37, 41, 43, 47, 53, 59, 61, 67, 71, 73,
79, 83, 89, 97
```

Niz

Niz je struktura podataka koja sadrži sekvencu primitivnih i/ili složenih podataka, bilo kojeg tipa, odvojenih zarezom:

niz :



podatak :
cijeli_broj| *realni_broj*| `False`| `True`|
kompleksni_broj| *string*| *n-torka*| *lista*

Dva su tipa (klase) podataka sa strukturom niza: *n*-torke i liste:

n-torka : ([*niz*])
lista : [[*niz* | *podatak*]]

n-torka (klasa `tuple`) je *niz* omeđen okruglim zagradama. `()` je prazna *n*-torka. ***lista*** (klasa `list`) je *niz* ili *podatak* omeđen uglatim zagradama. `[]` je prazna lista.

SEMANTIKA

Ako je ***N*** *niz* takva se struktura podataka interpretira kao uređena sekvenca u kojoj je svakom njezinom elementu, podatku dobivenom kao rezultat izračunavanja određenog izraza, pridružen indeks, od `0` do *n*-`1`, slijeva nadesno, odnosno `-1` do `–`*n*, zdesna nalijevo:

r	-*n*	-(*n*-1)		-2	-1
N	el_1	el_2	...	el_{n-1}	el_n
i	0	1		*n*-2	*n*-1

gdje je ***n*** broj elemenata ili duljina *niza* ***N***, a izračunavamo ga poznatom funkcijom `len()`, *n* = `len(`***N***`)`. Prazna *n-torka* i prazna lista imaju duljinu jednaku `0`.

```
>>> ()                                  ()
>>> []                                  []
>>> type ( () )            <class 'tuple'>
>>> type ( [] )             <class 'list'>
>>> len ( () ), len ( [] )          (0, 0)
```

n-torka s jednim elementom piše se prema pravilu:

```
( element , )
```

Na primjer:

```
>>> (10, )      (10,)
```

No, ako napišemo:

```
>>> (10)      10
```

to nije *n-torka* već cijeli broj!

››› 7.1 nizovi

```
>>> # prazna n-torka
>>> ()                              ()
>>> # n-torka s 3 prazne
>>> ( (),(),() )                    ((), (), ())
>>> # n-torka cijelih brojeva
>>> (1, 4, 9, 25)                   (1, 4, 9, 25)
>>> # n-torka realnih brojeva
>>> (0.5, -3.66, 1.0, -6.32, 1.5, 2.5)
(0.5, -3.66, 1.0, -6.32, 1.5, 2.5)
>>> # n-torka logičkih vrijednosti
>>> (False, True, True, True
(False, True, True, True)
>>> # n-torka znakova
>>> ('a', 'b', 'c')          ('a', 'b', 'c')
>>> # lista s jednim elementom
>>> [ 12.5 ]                        [12.5]
>>> # lista stringova
>>> ['Java','C++','Pascal','Python']
['Java', 'C++', 'Pascal', 'Python']
>>> # n-torka kompleksnih brojeva
>>> (1+2j, -2+3j, -1-1j)
((1+2j), (-2+3j), (-1-1j))
>>> # "mješovita" n-torka:
>>> (1,'jedan', 2,'dva', 3, 'tri',
     4,'četiri')
(1, 'jedan', 2, 'dva', 3, 'tri', 4,
'četiri')
```

Ispis *n-torke* u interaktivnom modu postiže se pisanjem *n-torke* ili *naredbom PRINT*, a u programskom modu samo *naredbom PRINT*.

PISANJE NIZOVA U VIŠE REDOVA

Već smo u prethodnom primjeru pokazali da se *n*-torke i liste, kao i izrazi u zagradi, mogu pisati u više redova, u potpuno slobodnom formatu, bez znaka „\“ za prelazak u novi red. Pogledajmo još primjer pisanja u sljedećoj vježbi:

››› 7.2 Pisanje nizova

```
>>> (1,  2,  3)                    (1, 2, 3)
>>> (      1, 2,
                3 )                (1, 2, 3)
>>> [ (1, 2, 3),
       (4, 5, 6), (7, 8, 9) ]
[(1, 2, 3), (4, 5, 6), (7, 8, 9)]
```

```
>>> ['Java', 'C#', 'C++', 'BASIC',
     'FORTRAN', 'Pascal', 'Python']
['Java', 'C#', 'C++', 'BASIC',
'FORTRAN', 'Pascal', 'Python']
```

Dakako, ne treba pretjerivati i zlorabiti tu slobodu pisanja. U ovoj vježbi jedino posljednji primjer, lista *n*-torki, ima smisla pisati u tri reda iz kojih je razvidno da se dana lista može interpretirati kao matrica.

EVALUIRANJE NIZA

Funkcija `eval()` može evaluirati *n*-torku ili listu. Na primjer:

```
>>> eval ( '[1,2,3]' )
[1, 2, 3]
>>> eval ( '( (1,2,3), [4,5,6] )' )
((1, 2, 3), [4, 5, 6])
```

VARIJABLE S NIZOVIMA

Varijable sa strukturom niza (*n-torke* i *liste*) definiraju se naredbom za pridruživanje koja je napisana prema pravilu:

```
ime_n-torke = niz | n-torka |
              eval ( funkcija_INPUT )
ime_liste   = lista |
              eval ( funkcija_INPUT )
```

Poslije izvršenja ove naredbe *ime* će poprimiti svojstvo „varijabla sa strukturom *n-torke* (ili *liste*)" ili „ime *n-torke* (ili *liste*)" i bit će joj pridružen navedeni *niz* ili *n-torka* (koja može biti i prazna), odnosno *lista*. Ako se rabi *funkcija_INPUT*, unosi se *n-torka* ili *lista*.

››› 7.3 Varijable s nizovima

```
>>> A = (1,2,3); type (A)
<class 'tuple'>
>>> L = []; L # prazna lista          []
>>> type( L )              <class 'list'>
>>> X = eval (input ('Upiši listu '))
Upiši listu [10, 9, 9, 10,    'kraj']
>>> X
[10, 9, 9, 10, 'kraj']
>>> Y = eval (input (
        'Upiši n-torku  ')); Y
Upiši n-torku  '**', '//', '%'
('**', '//', '%')
```

INICIJALIZACIJA NIZA

Inicijalizacija varijable sa strukturom niza postiže se pridruživanjem imenu prazne *n*-torke ili funkcije `tuple()` bez argumenata, odnosno prazne liste ili funkcije `list()` bez argumenata:

```
ime = () | tuple() | [] | list()
```

PRIDRUŽIVANJE NIZA FUNKCIJOM `input()`

S obzirom na to da funkcija `input()` učitava string, funkcijom `eval()` ćemo ga evaluirati i pridružiti nekom imenu. Na primjer:

```
>>> A = eval (input (
        'lista ili n-torka ')); A
lista ili n-torka [1, 2, 3]
[1, 2, 3]
>>> A = eval (input (
        'lista ili n-torka ')); A
lista ili n-torka ((1, 2, 3), [4, 5, 6])
((1, 2, 3), [4, 5, 6])
>>> A = eval (input (
        'lista ili n-torka ')); A
lista ili n-torka (1, 2, 3) +(4, 5, 6)
(1, 2, 3, 4, 5, 6)
>>> A = eval (input (
       'lista ili n-torka ')); A
lista ili n-torka [0] *10
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

ISPIS NIZA

U interaktivnom modu napisani niz ili niz sadržan u varijabli sa strukturom niza bit će ispisan u sljedećem redu. Možemo ga ispisati *naredbom PRINT* pišući ga kao argument. Ispis niza iz programskog moda moguć je samo *naredbom PRINT*.

››› 7.4 Ispis niza

```
>>> 10, 5, 12, ('a', 'b'), 'kraj'
(10, 5, 12, ('a', 'b'), 'kraj')
>>> Y = ( 1, False, ('a', 'b'),
          3.15, 'python', 10  )
>>> Y; print( Y )
(1, False, ('a', 'b'), 3.15, 'python',
10)
(1, False, ('a', 'b'), 3.15, 'python',
10)
```

ELEMENTI NIZA

Pojedinom se elementu *n-torke* ili *liste*, općenito padatku bilo kojeg primitivnog ili složenog tipa, može pristupiti pišući *n-torku* ili *listu* koju slijedi njegov indeks `i` između uglatih zagrada:

```
n     = len (N)
N[i] -n ≤ i ≤ n-1      N[i] == N[i-n]
```

Prvi je element `N[0]` (ili `N[-n]`), a posljednji `N[n-1]` (ili `N[-1]`). Ako je `i` izvan domene indeksa niza, dojavljuje se pogreška. Provjera je li podatak *x* element niza *X* rezultat je izračunavanja relacije:

```
x in X
```

Odgovor će biti `True` ako je *x* sadržan u *X*, inače `False`.

Ispis elemenata niza možemo ostvariti na dva načina: promjenom indeksa niza *N* od `0` do `len(N)-1` ili iteracijom. Ova potonja je prikladnija za to jer iteracija ima „ugrađeni mehanizam“ za izuzimanje elemenata niza, redom, od prvoga do posljednjega. Opisat ćemo je u nastavku knjige.

››› 7.5 Ispis elemenata niza

```
>>> X = ( 1, False, ('a', 'b'), 3.15,
          'python', 10 )
>>> i = 0
>>> while i <= len( X ) -1 :
    print( X[i], end = '  ' );  i += 1
1  False  ('a', 'b')  3.15  python  10
```

Ako želimo ispisati sve elemente niza možemo koristiti *naredbu PRINT* prema sintaksi:

```
print ( * niz )
```

Elementi će bi ti ispisani s jednim razmakom:

```
>>> print ( * X )
1 False ('a', 'b') 3.15 python 10
```

››› 7.6 Element niza

```
>>> (1, 4, 9, 25)[0]   # prvi          1
>>> (1, 4, 9, 25)[-1]  # poslj.       25
>>> ('a', 'b', 'c')[0] # prvi        'a'
>>> ('a', 'b', 'c')[1] # drugi        'b'
>>> ['a', 'b', 'c'][2] # treći        'c'
>>> ['a', 'b', 'c'][3] # četvrti
IndexError: list index out of range
>>> [(1,'jedan'), (2,'dva'),
     (3,'tri')][-1]           (3,'tri')
>>> 'jedan' in [(1,'jedan'),
      (2,'dva'), (3,'tri')]       False
>>> (1,'jedan') in [(1,'jedan'),
      (2,'dva'), (3,'tri')]          True
>>> A = (1, False, ('a', 'b'), 3.15,
         'python', 10);  a = 10
>>> 'a' in A                       False
>>> ('a', 'b') in A                 True
>>> a in A   True  >>> 2*5 in A     True
>>> False in A                      True
>>> 'Python' not in A               True
```

ITERIRANJE

S obzirom na to da *n*-torke i liste imaju strukturu sekvence, definirana je naredba za iteriranje (iteracija) ili „*FOR petlja*“, s njihovim elementima prema poznatoj sintaksi u kojoj je:

```
sekvenca_podataka :
   gen_n-torke | gen_liste
```

SEMANTIKA

Semantika iteracije jednaka je onoj koju smo opisali u prethodnom poglavlju, samo je ovdje niz, *n*-torka ili lista, sekvenca podataka iteratora: ponavljaja se izvršavanje naredbi za sve elemente niza, od njegova početka do kraja.

Kontrolna varijabla iteriranja poprimit će vrijednost, a time i tip, jednak tekućem podatku elementa niza. Dakle, može se mijenjati tijekom iteriranja jer niz može sadržavati različite primitivne i složene podatke. Iteriranje može biti prekinuto *naredbom BREAK*.

Ako iteracija sadrži *naredbu CONTINUE*, njezinim izvršenjem prijeći će se na sljedeći element iteriranja. Poslije normalnog završetka interiranja prelazi se na *ELSE granu*, ako postoji. U sljedećoj se vježbi primjenom iteracije ispisuje tablica istinitosti logičkih operacija `and` i `or`.

››› 7.7 Iteriranje

```
>>> Iter = """
FT = (False, True);
isp = "%-5s  %-5s   %-8s  %-8s"
print(isp % ('x','y','x and y','x or y'))
for x in FT :
  for y in FT : print( isp %
        (x, y, x and y, x or y) ) """
>>> exec( Iter )
x      y      x and y   x or y
False  False  False     False
False  True   False     True
True   False  False     True
True   True   True      True
```

ISJEČAK NIZA

Nad n-torkom ili listom, slično kao i nad stringom, može se ekstrahirati isječak. Ako je ***N*** niz napisan s domenom:

niz_s_domenom : ***N*** [*i* | [*m*] : [*n*] [: [*o*]]]

gdje je ***N***=(e_0, e_1, e_2, ..., e_{k-1}) niz elemenata e_i duljine *k*, *m* i *n* su cjelobrojni izrazi koji predstavljaju početni i krajnji indeks niza i *o* je cjelobrojni izraz, *o*≠0, koji predstavlja korak povećanja ili umanjenja indeksa, postojat će sljedeći slučajevi:

Operand	*Značenje*

N [*i*]	Vraća element e_i niza ***N***, ako je $-k \le i < k$, inače, dojavljuje se:
	`IndexError: list index out of range`
	`>>> (1,2,3,4,5) [4]  5`
	`>>> (1,2,3,4,5) [-5]  1`
	`>>> (1,2,3,4,5) []`
	`SyntaxError: invalid syntax`

N [:]	Cijeli niz ***N***.
	`>>> (1,2,3,4,5) [:]`
	`(1, 2, 3, 4, 5)`

N [*m* :]	Vraća: 1) niz ***N***, ako je *m* == 0 or *m* <= -*k*. 2) isječak niza ***N*** od znaka s indeksom *m* do kraja niza, $e_m \ldots e_{k-1}$, ako je *m* <> 0 and –k < *m* < *k*. 3) prazan niz, [ ] ili (), ako je *m* >= *k*.

U sljedećim smo slučajevima *m* i *n* sveli na apsolutne indekse:

```
m+i*o < n -> i < (n-m)/o
```

```
if m < 0 : m += len(N)
if n < 0 : n += len(N)
```

N [*m*: *n*] 1) jednako je ***N***[*m*:] ako je *n* >= *k*.
2) podniz niza ***N*** od elementa s indeksom *m* do elementa s indeksom *n*-1, ako je *m* < n.
3) prazan niz, [ ] ili ( ), ako je *m* >= n.

N [: *n*] = ***N*** [0: *n*]

N [: :] = ***N*** [:]

N [*m*: :] = ***N*** [*m*:]

N [*m*: *n*:] = ***N*** [*m*: *n*]

N [*m*: *n*: *o*] = $e_m e_{m+o} \ldots e_{m+i*o}$, *i*=1,2,...,(*n*-*m*-1)/*o*
ako je *o*>0 and *m*<*n*
= $e_m e_{m+o} \ldots e_{m+i*o}$, *i*=1,2,...,(*m*-*n*+1)/*o*
ako je *o*<0 and *m*>*n*. Inače, vraća praznu listu.

```
>>> (0,1,2,3,4,5)[5: 1: 2]          ()
>>> (0,1,2,3,4,5)[-10: 1: 2]      (0, )
>>> [0,1,2,3,4,5,6,7,8,9][ 10: 0:-1]
[9, 8, 7, 6, 5, 4, 3, 2, 1]
```

N [*m*: :*o*] = ***N*** [*m*: *k*: *o*]

```
>>> (0,1,2,3,4,5,6,7,8,9][10: :-1]
(9, 8, 7, 6, 5, 4, 3, 2, 1, 0)
```

N [: *n*: *o*] = ***N*** [0: *n*: *o*] ako je o > 0
= ***N*** [k-1: *n*: *o*] ako je o < 0

```
>>> (0,1,2,3,4,5,6,7,8,9)[ :2 :-2]
(9, 7, 5, 3)
```

N [: :*o*] = ***N*** [:*k* :*o*] ako je o > 0
= ***N*** [k-1: :*o*] ako je o < 0

```
>>> (0,1,2,3,4,5,6,7,8,9)[ : : 2]
(0, 2, 4, 6, 8)
>>> (0,1,2,3,4,5,6,7,8,9)[ : :2][ : :-1]
(8, 6, 4, 2, 0)
>>> (0,1,2,3,4,5,6,7,8,9)[ : :-2]
(9, 7, 5, 3, 1)
>>> (0,1,2,3,4,5,6,7,8,9)[ 10: 2:-2]
(9, 7, 5, 3)
>>> (0,1,2,3,4,5,6,7,8,9)[ 5: 0:-3]
(5, 2)
```

BRISANJE NIZA

Može se izbrisati (ukinuti) samo cijelu *n*-torku ili listu, što se postiže pisanjem *naredbe DEL*:

del (*ime_n-torke* | *ime_liste*)

››› 7.8 Brisanje niza

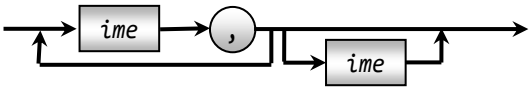
```
>>> A = (1,2,3); B = ('a','b','c')
>>> A; B
(1, 2, 3)
('a', 'b', 'c')
>>> del B
>>> A; B
(1, 2, 3)
NameError: name 'B' is not defined
>>> L = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> L   [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> del L[1 : len(L)+1 : 2]
>>> L   [1, 3, 5, 7, 9]
```

Pridruživanje nizova

Uvođenjem *n*-torke i liste vraćamo se konkurentnom pridruživanju i definiramo potpuno pravilo njegova pisanja:

konkurentno_pr :
(*niz_imena* | (*niz_imena*))
= *gen_n-torke*

niz_imena:



Ako naredba za pridruživanje *n*-torke sadrži više od jednog imena, onda je to konkurentno pridruživanje sa značenjem jednakim značenju „običnog" konkurentnog pridruživanja. Drugim riječima, konkurentno je pridruživanje, premda to ranije nismo rekli jer nismo uveli strukturu *n*-torke, ekstrahiranje elemenata *n*-torke i pridruživanje imenima nevedenim na lijevoj strani naredbe za pridruživanje. Broj komponenti (elemenata) *n*-torke mora biti jednak broju imena.

```
>>> A = (1, 2);  A,      ((1, 2),)
>>> *A,                  (1, 2)
>>> *A, 5                (1, 2, 5)
>>> *A, 'kraj'           (1, 2, 'kraj')
>>> a, b = 'Python'
ValueError: too many values to unpack...
>>> a, b = '12'  # znakovni niz!
>>> a, b                 ('1', '2')
```

PRIDRUŽIVANJE NORMALNOG NIZA

Bilo koji niz ili drugi ponovljivi niz vrijednosti mogu se dodijeliti bilo kojem nizu imena, sve dok su duljine jednake. Ovaj temeljni obrazac za dodjeljivanje redoslijeda djeluje u većini konteksta dodjele:

```
>>> a,b,c,d = [1,2,3,4]; a, d        (1, 4)
>>> for (a,b,c) in [[1,2,3], [4,5,6]] :
    print( a, b, c )
1 2 3
4 5 6
```

PRIDRUŽIVANJE PROŠIRENOG NIZA

Pridruživanje niza se proširuje da bi se omogućila kolekcija proizvoljno mnogo podataka, dodavanjem jedne prefiks varijable u cilj dodjele sa zvijezdom; kada se koristi, duljine sekvenci ne moraju se poduda-rati, a ime sa zvjezdicom prikuplja sve inače nepodu-rarne stavke na novoj listi. Na primjer:

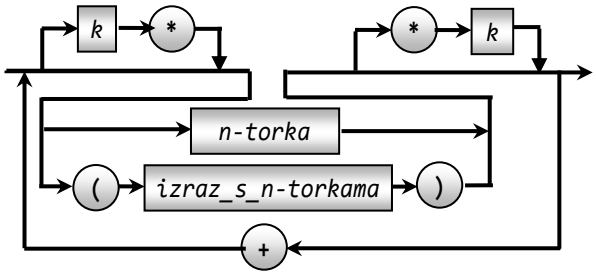
```
>>> a, *b = [1, 2, 3, 4]
>>> a, b                    (1, [2, 3, 4])
>>> a, *b, c = (1, 2, 3, 4)
>>> a, b, c                 (1, [2, 3], 4)
>>> *a, b = 'abcd'
>>> a, b          (['a', 'b', 'c'], 'd')
>>> for (a, *b) in [
                  [1, 2, 3], [4, 5, 6]] :
    print( a, b )
1 [2, 3]
4 [5, 6]
```

7.9 Konkurentno pridruživanje

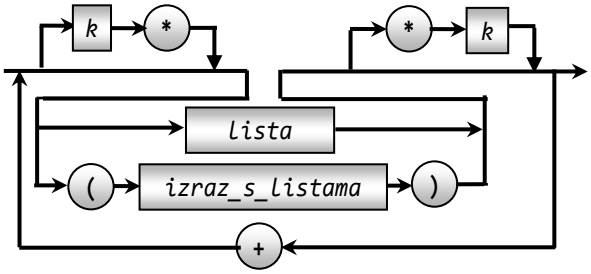
```
>>> A = 1, 2, 3
>>> a, b, c = A
>>> print ( a, b, c )                1 2 3
>>> # Još jedan način gen. nizova
>>> P = (1, 2, 3); Q = (3, 4, 5)
>>> P +Q                (1, 2, 3, 3, 4, 5)
>>> (*P, *Q)            (1, 2, 3, 3, 4, 5)
>>> P +Q == (*P, *Q)                 True
>>> *2*P, *Q  (1, 2, 3, 1, 2, 3, 3, 4, 5)
>>> *P, (5,5)            (1, 2, 3, (5, 5))
>>> *3*Q
SyntaxError: can't use starred exp here
>>> *3*Q, *P
(3, 4, 5, 3, 4, 5, 3, 4, 5, 1, 2, 3)
>>> *3*P,     (3, 4, 5, 3, 4, 5, 3, 4, 5)
>>> *[1], *[2]                      (1, 2)
>>> [*[1], *[2]]                    [1, 2]
>>> [*[1], *(1,2,3)]          [1, 1, 2, 3]
>>> *(1,2)
SyntaxError: can't use starred exp here
>>> *(1,), *(2,)                    (1, 2)
>>> X = list (range (8))
>>> X             [0, 1, 2, 3, 4, 5, 6, 7]
>>> x, *y, z = X; x                      0
>>> y                   [1, 2, 3, 4, 5, 6]
>>> z                                     7
>>> Y = (1, 100)
>>> Prvi, *Srednji, Zadnji = Y
>>> Prvi                                  1
>>> Srednji                              []
>>> Zadnji                             100
>>> Y = (1, 50, 100)
>>> Prvi, *Srednji, Zadnji = Y
>>> Prvi                                  1
>>> Srednji                           [50]
>>> Zadnji                             100
```

Definirani su i izrazi s *n*-torkama i listama, prema sintaksi:

izraz_s_n-torkama :



izraz_s_listama :



k : *cijeli_broj*| *cjelobrojna_varijabla*|
(*cjelobrojni_izraz* **)**

Samo su dvije standardne operacije definirane s nizovima:

\+ dodavanje niza i
* multipliciranje niza

Operacijom dodavanja niza B nizu istoga tipa A, A+B, dobije se rezultirajući niz sačinjena od elemenata niza A kojima su dopisani elementi niza B. Ako se rezultirajući niz pridružuje nizu A, možemo pisati

```
A = A + B
```

ili, može se koristiti operatorsko pridruživanje +=

```
A += B
```

Kao i kod nastavljanja nizova znakova, za operaciju dodavanja niza ne vrijedi zakon komutacije.

Niz ***N*** pomnožen s cijelim brojem *k*, *k*>1, bit će multipliciran *k* puta, odnosno, bit će dodan *k*-1 puta sam sebi. Ako je *k*==1, rezultirajući niz jednak je ***N***, a ako je *k*=0, rezultat je prazna n-torka ili lista. Ako se rezultirajući niz pridružuje nizu koji se multipricira, može se koristiti operand *=.

››› 7.10 Dodavanje i multipliciranje n-torke

```
>>> A = (1,2); B = ('a','b')
>>> A + B                (1, 2, 'a', 'b')
>>> 2*A +2*B
(1, 2, 1, 2, 'a', 'b', 'a', 'b')
>>> (A+B)*2
(1, 2, 'a', 'b', 1, 2, 'a', 'b')
>>> A += B; A           (1, 2, 'a', 'b')
>>> B *= 2; B       ('a', 'b', 'a', 'b')
```

››› 7.11 Proširenje n-torke

```
>>> A  = 1, 2, 3;  A     (1, 2, 3)
>>> A += (4, );  A       (1, 2, 3, 4)
>>> A  = (0, ) +A;  A    (0, 1, 2, 3, 4)
>>> A *= 2;  A
(0, 1, 2, 3, 4, 0, 1, 2, 3, 4)
```

Značenje operacije * u izrazu *N*k* jest multipliciranje niza *N* *k* puta. Ako je *k*≤0, rezultat je prazna *n*-torka ili lista.

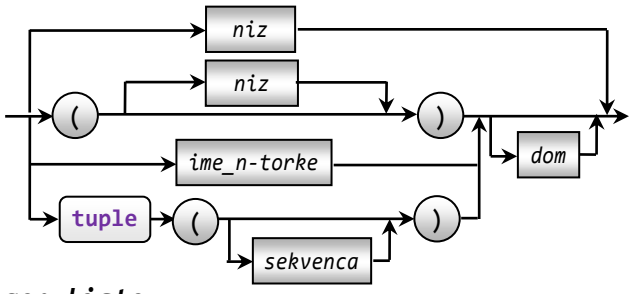
Ako su *X* i *Y* nizovi istoga tipa, značenje operacije + jest nastavljanje niza X nizom Y. Ako treba proširiti n-torku X, moramo rabiti naredbu za pridruživanje u kojoj će izraz s *n*-torkama sadržavati *n*-torku X kao operand. Ako je X prvi operand možemo rabiti operatorsko pridruživanje += i *=.

```
>>> (1, 2, 3) *4
(1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3)
>>> ((1,2,3) + ('a', 'b')) *2
(1, 2, 3, 'a', 'b', 1, 2, 3, 'a', 'b')
>>> X = (1, 2, 3);  Y = ('a', 'b', 'c')
>>> X +Y        (1, 2, 3, 'a', 'b', 'c')
>>> X += 2
TypeError: can only concatenate tuple
(not "int") to tuple
>>>
>>> X                          (1, 2, 3)
>>> X *= 2; X         (1, 2, 3, 1, 2, 3)
>>> A = ['ha'];  B = ['.']
>>> A +B                     ['ha', '.']
>>> 2*A*3
['ha', 'ha', 'ha', 'ha', 'ha', 'ha']
>>> (A +B)*3
['ha', '.', 'ha', '.', 'ha', '.']
>>> C = (1, 2, 3) +(4, )    (1, 2, 3, 4)
>>> C = C[:2] +(8, 9, 10) +C[2:];  C
(1, 2, 8, 9, 10, 3, 4)
```

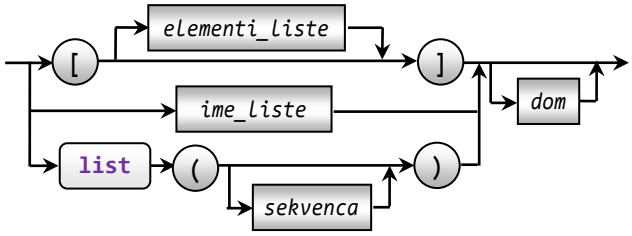
GENERIRANJE NIZA

Potpuna pravila generiranja niza, *n-torke* i liste, dana su sljedećim sintaksnim dijagramima:

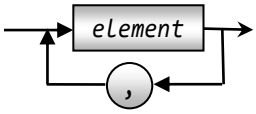
gen_n-torke:



gen_liste:



elementi_liste:



element: *izraz* | *n-torka* | *lista* | *izraz_s_n-torkama* | *izraz_s_listama*

dom : [*m* | *isječak*]

isječak: [*m*] : [*n*] [: [*l*]]

gdje su ***m*** i ***n*** cjelobrojni izrazi čiji je rezultat izračunavanja u intervalu od `-len(`*n-torka*`)` do `len(`*n-torka*`)`, a ***l*** cjelobrojni izraz čiji rezultat izračunavanja mora biti različit od `0`.

››› 7.12 Generiranje niza

```
>>> 1, 2, 3                          (1, 2, 3)
>>> (1, 2, 3)                        (1, 2, 3)
>>> x = 2;
>>> x, x**2, x**3, x**4    (2, 4, 8, 16)
>>> eval ( " tuple (range (10)) " )
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> eval ( " list (range (10)) "  )
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

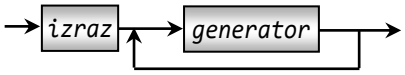
UVJETNO GENERIRANJE NIZA

Moguće je generirati niz pod nekim uvjetima, sa sintaksom danom u nastavku. „Tijelo“ generatora je između okruglih zagrada, za *n*-torku, ili između uglatih zagrada za listu.

n-torka: (***uvjetno_generirani_niz***)

lista: [***uvjetno_generirani_niz***]

uvjetno_generirani_niz:



generator:

››› 7.13 Uvjetno generiranje niza

```
>>> X = [x for x in range(21) if x %2 ]
>>> X
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
>>> print ( * X )
1 3 5 7 9 11 13 15 17 19
>>> Y = [x**2 for x in range (1, 11)]
>>> Y
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

PROMJENA SADRŽAJA LISTE

Osim potpune promjene vrijednosti sadržaja varijable sa strukturom liste, navodeći njezino ime na lijevoj strani naredbe za pridruživanje, dopuštena je promjena bilo kojeg njezinog elementa ili isječka. Ako je ***L***, ***L***=[e_0, e_1, e_2, ...,e_{k-1}], dopuštena je promjena vrijednosti za bilo koji indeks i, $-n \le i \le n-1$, gdje je n duljina liste. Pravilo pisanja je:

L [*i*] = *izraz* | *lista*

Promjena sadržaja isječka liste postiže se naredbom:

L [[*m*] : [*n*] [: [*o*]]] = *lista*

gdje su *m*, *n* i *o* parametri domene. Pritom mora biti zadovoljen kontekstni uvjet: duljina isječka liste ***L*** i duljina liste koja se pridružuje mora biti jednaka.

››› 7.14 Promjena sadržaja liste

```
>>> L = range (1, 9); L
[1, 2, 3, 4, 5, 6, 7, 8]
>>> L[-1] = ['a', 'b']; L
[1, 2, 3, 4, 5, 6, 7, ['a', 'b']]
>>> L[::2] = (len(L) -len(L[::2]))*[0]
>>> L
[0, 2, 0, 4, 0, 6, 0, ['a', 'b']]
```

Lista je objekt ili instanca klase `list()`. Zbog toga pridruživanje varijable sa strukturom liste nekoj drugom imenu ima značenje uvođenja novog imena iste instance. Na primjer, ako je:

```
>>> A = [1, 2, 3]; B = A; print( A, B )
[1, 2, 3] [1, 2, 3]
```

Sadržaji su jednaki, a iz:

```
>>> B == A    True    >>> B is A     True
>>> A is B    True
>>> id(A), id(B)    (19149888, 19149888)
```

potvrđuje se da se radi o istoj instanci klase `list()`. Ako sada promijenimo sadržaj liste B,

```
>>> B += ['+']; A      [1, 2, 3, '+']
```

vidimo da se "promijenio" i sadržaj liste A. Međutim, ako sada napišemo:

```
>>> B = [1, 2, 3, '+']
>>> B == A    True    >>> B is A     False
```

vidimo da su sada A i B dvije različite instance.

Ako elementu `i` neke liste A pridružimo listu B, tada će svaka promjena sadržaja liste B mijenjati sadržaj elementa `A[i]`:

```
>>> A = [1,2,3]; A[0] = B = ['a', 'b']
>>> A; B
[['a', 'b'], 2, 3]
['a', 'b']
>>> id(A[0]),id(B) (32938224, 32938224)
>>> B.append('c'); B; A
['a', 'b', 'c']
[['a', 'b', 'c'], 2, 3]
```

Ili, promjena sadržaja `A[i]` mijenja sadržaj liste B:

```
>>> A[0].remove ('b')
>>> A; B
[['a', 'c'], 2, 3]
['a', 'c']
```

Razmotrimo još nekoliko primjera:

1) Inicijalizirajmo varijablu (listu) X:

```
>>> X = [[1]]*3;  X   [[1], [1], [1]]
>>> for x in X :
        print (id(x), end = ' ')
34388208 34388208 34388208
```

Lista X sarži tri elementa, tri liste locirana na istoj adresi.

2) Ako metodom `append()` proširimo sadržaj jednog elementa, „promijenili" su se i sadržaji preostala dva elementa:

```
>>> X[0].append(2);  X
[[1, 2], [1, 2], [1, 2]]
```

3) Isto će se dogoditi ako sadržaj jednog elementa promijenimo operandom +=:

```
>>> X[1] += [3];  X
[[1, 2, 3], [1, 2, 3], [1, 2, 3]]
```

4) Ali, ako sadržaj bilo koje elementa promijenimo naredbom za pridruživanje, taj će element dobiti novu adresu i novu vrijednost. Adrese preostalih elemenata i njihove vrijednosti ostat će nepromijenjene:

```
>>> X[2] = [1,2,3,4];  X
[[1, 2, 3], [1, 2, 3], [1, 2, 3, 4]]
>>> for x in X: print(id(x),end = ' ')
34388208 34388208 34380816
```

Da smo inicijalnu listu X oformili sa:

```
>>> X = [ [1], [1], [1] ]
>>> for x in X : print( id(x), end = ' ')
34388848 34388128 34394760
```

ili sa:

```
>>> X = [ [1] for x in range (3) ]
>>> X                    [[1], [1], [1]]
>>> for x in X : print (id(x), end = ' ')
34387928 34387088 34394240
```

njezini bi elementi imali različite (neovisne) adrese lokacija.

BRISANJE LISTE

Može se izbrisati (ukinuti) cijela lista ili njezin isječak. Brisanje cijele liste ili isječka liste postiže se *naredbom DEL*:

```
del ime_liste [ domena ]
```

››› 7.15 Brisanje liste

```
>>> A = [1,2,3]; B = A; print( A, B )
[1, 2, 3] [1, 2, 3]
>>> del A; print( B )          [1, 2, 3]
>>> P = list (range (2, 11));  P
[2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> del P [2 : :2];  P  [2, 3, 5, 7, 9]
```

Na kraju ovog poglavlja analizirat ćemo odnose između stringova, lista i *n*-torki i uvesti još neke funkcije.

GENERIRANJE *n*-TORKE IZ STRINGA

Generiranje n-torke znakova (ili pretvorba) iz stringa postiže se funkcijom `tuple()` sa stringom kao argumentom:

```
tuple ( string ) [ domena ]
```

››› 7.16 Pretvorba stringa u n-torku znakova

```
>>> tuple ('abcdef')
('a', 'b', 'c', 'd', 'e', 'f')
>>> tuple ('0123456789') [::2]
('0', '2', '4', '6', '8')
>>> tuple ('0123456789') [::-2]
('9', '7', '5', '3', '1')
```

GENERIRANJE STRINGA IZ NIZA

Funkcijom `join()` može se generirati string u kojem će biti spojeni stringovi niza bez delimitera (prazan string) ili sa zadanim delimiterom, nepraznim stringom. Pravilo pisanja je:

```
s. join ( seq )
```

gdje je `s` delimiter (string), a `seq` niz stringova.

››› 7.17 join()

```
>>> A = tuple ('abc');  A
('a', 'b', 'c')
>>> ''.join (A)                    'abc'
>>> '*'.join (A)                   'a*b*c'
>>> a = 'a', '1', '2', '**', '=='
>>> ' '.join (a)              'a 1 2 ** =='
>>> ', '.join (a)    'a, 1, 2, **, ==='
>>> x = (1, '2', 5)
>>> ' '.join (x)
TypeError: sequence item 0: expected
string, int found
>>> L = list ('abc'); L
['a', 'b', 'c']
>>> ''.join (L)                    'abc'
>>> '.'.join (L)                   'a.b.c'
>>> a = ['a', '1', '2', '**', '==']
>>> ' '.join (a)              'a 1 2 ** =='
>>> ', '.join (a)     'a, 1, 2, **, =='
>>> x = [1, '2', 5]
>>> ' '.join (x)
TypeError: sequence item 0: expected
string, int found
```

PARTICIJA STRINGA

Ako je ***s*** string i *sep* separator, također string, funkcija `partition()` napisana prema pravilu:

```
s . partition (sep)
```

vraća *n*-torku (trojku) sačinjenu od tri dijela stringa ***s*** u odnosu na prvo pojavljivanje separatora: string prije separatora, separator, string iza separatora. Ako ***s*** ne sadrži separator kao podstring, vraća trojku (***s***, '', ''). Postoji i funkcija `rpartition()`:

```
s . rpartition (sep)
```

vraća *n*-torku (trojku) sačinjenu od tri dijela stringa ***s*** u odnosu na posljednje pojavljivanje separatora: string prije separatora, separator, string iza separatora. Ako ***s*** ne sadrži separator kao podstring, vraća trojku (***s***, '', '').

››› 7.18 Particija stringa

```
>>> '123.56'.partition ('.')
('123', '.', '56')
>>> '123.56 -3.56'.partition ('.')
('123', '.', '56 -3.56')
>>> '123.56'.partition (',')
('123.56', '', '')
>>> '123.56 -3.56'.rpartition ('.')
('123.56 -3', '.', '56')
>>> '-3.56'.rpartition ('.')
('-3', '.', '56')
```

FUNKCIJA `list()`

Funkcija `list()` piše se prema pravilu:

```
list ( [ string | n-torka | lista ] )
```

Značenje je ove funkcije generiranje:

- prazne liste, [ ], ako je argument izostavljen,
- liste znakova ako je string argument,
- liste s elementima jednakim elementima *n*-torke, ako je *n-torka* argument
- liste jednake listi kao argumentu

››› 7.19 Funkcija `list()`

```
>>> list (1, 2, 3)
TypeError: list() takes at most 1
argument (3 given)
>>> list ()                  []
>>> list ('abc')             ['a', 'b', 'c']
>>> list( (12,3)),           [1, 2, 3]
>>> list( ['abc', (12,3)] )
['abc', (12, 3)]
>>> X = ( 1, 2, 3, 2, 1 ); print ( X )
```

```
(1, 2, 3, 2, 1)
>>> print( tuple( list( X ) ) )
(1, 2, 3, 2, 1)
>>> Y = [ 1, 2, 1, 2 ]; print( X )
[1, 2, 1, 2]
>>> print( list( tuple( Y ) ) )
[1, 2, 1, 2]
```

FUNKCIJA filter()

Funkcija filter() definirana je nad nizovima kao sekvencom podataka:

```
filter (funkcija, n_torka | lista)
```

Značenje je: generiranje sekvence dobivene selekcijom elemenata ulaznog niza (*n*-torke ili liste) koja zadovoljava uvjet (predikat) definiran funkcijom.

››› 7.20 Selekcija elemenata niza

```
>>> tuple (filter( None,
    [1,2,3,0,0,5,'','']))    [1, 2, 3, 5]
>>> list( filter( None,
          [1, 2, 3, 10, 20, -1]) )
[1, 2, 3, 10, 20, -1]
>>> L = list( range( 1, 11 ));  L
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> odd = lambda x : x % 2 != 0
>>> list( filter( odd, L ) )
[1, 3, 5, 7, 9]
>>> [x for x in L if odd(x)]
[1, 3, 5, 7, 9]
>>> tuple( filter( None,
          (1, 2, 3, 0, 0, 5, '', '')) )
(1, 2, 3, 5)
>>> tuple( filter(None,
           (1, 2, 3, 10, 20)) )
(1, 2, 3, 10, 20)
>>> odd = lambda x : x % 2 != 0
>>> tuple( filter( odd,
           (1,2,3,4,5,6,7,8,9,10)) )
(1, 3, 5, 7, 9)
```

Primijetiti da je značenje funkcije filter() nad listom ekvivalentno generatoru:

```
[ x for x in lista if funcija(x) ]
```

FUNKCIJE NAD NIZOVIMA

Nad n-torkama i listama definirane su funkcije count() i index(). Također je definirana i standardna funkcija sum().

. *funkcija*()	*Opis*
count (*el*)	Broj pojavljivanja elementa *el* u *n*-torki. Vraća 0 ako ne postoji. (1,1,2,1,'1') .count(1) 3 ['a','b','ab'].count('a') 1
index (*el*)	Indeks prvog pojavljivanja elementa *el* u nizu. Vraća ValueError ako ne postoji. (1,1,2,2) .index(1) 0 [1,1,2,2] .index(2) 2 (1,1,2,2) .index(3) ValueError
sum ((*n_torka* | *lista*) [, *start*])	Vraća zbroj svih elemenata *n*-torke ili liste, ako je *start* izostavljen, ili zbroj svih elemenata od indeksa *start* do kraja. Elementi moraju biti brojčanog tipa. sum((1,2,3,4,5,6,7,8,9)) 45 A = list(range(1, 10)) sum([i**2 for i in A]) 285 sum((10, 20, 'a')) TypeError sum (1, 2, 3) TypeError

U pretposljednjem primjeru funkcija sum() nije definirana jer niz sadrži nebrojčani element, a u posljednjem primjeru funkcija sum() nije definirana jer 1, 2, 3 nije *n*-torka.

METODE NAD LISTAMA

Nad listama kao promjenljivom nizu podataka definirano je nekoliko metoda koje mijenjaju sadržaj ili uređenje liste. Dane su u sljedećoj tablici:

L. metoda()	*Opis*
append (*el*)	Dodaje *el* na kraj liste. X = []; X.append(10); print(X) [10] X.append('***'); print(X) [10,'***']
clear ()	Izbacuje sve elemente iz liste. Poslije toga je lista prazna. X. clear(); print(X) []
extend (*x*)	L += list (*x*) gdje je *x* string, n-torka ili lista. X = [1,2,3]; X.extend('ab'); print(X) [1, 2, 3, 'a', 'b'] Y = []; Y.extend ((10,20)); print(Y) [10, 20]
insert (*i*, *el*)	Umeće *el* ispred elementa s indeksom *i*. Ako je *i* ≥len(*L*), *el* će biti dodan na kraj liste, a ako je i≤-len(L), ispred prvog elementa. L = [1]; L.insert(2,2); print(L) [1, 2] L.insert(-1, 3); print (L) [1, 3, 2] L.insert(-10, 4); print (L)

	`[4, 1, 3, 2]`
	`L.insert(-2, 5); print (L)`
	`[4, 1, 5, 3, 2]`

pop([*i*])	Brisanje i vraćanje elementa liste s indeksom *i*. Ako je indeks izostavljen, briše se posljednji (s indeksom -1).
	`A = [1,2,3];`
	`print(A.pop(), A)` `3 [1, 2]`
	`B = A.pop(0); print( B, A )`
	`1 [2]`

remove(*el*)	Ukidanje prvog elementa *el* liste ***L***. Dojavljuje pogrešku ako *el* nije u ***L***.
	`[1,2,'1'].remove('1')` `[1,2]`
	`a = [1,2,1,1]`
	`while 1 in a: a.remove(1);`
	`print( a )`
	`[2, 1, 1]`
	`[2, 1]`
	`[2]`

reverse()	Reverzna lista liste ***L***.
	`A = [10,20,15,33]`
	`A.reverse(); print( A )`
	`[33,15,20,10]`

sort()
sort(
reverse = True**)**
	Sortirana lista ***L***, u rastućem nizu.
	Sortirana lista ***L***, u opadajućem nizu.
	`A = [33,15,20,10]`
	`A.sort(); print( A )`
	`[10,15,20,33]`
	`A.sort(reverse=True);`
	`print(A)`
	`[33,20,15,10]`
	`B = A.sort(); print( B )`
	`None`

Metode mijenjaju sadržaj liste (objekta) i vraćaju None kao rezultat. Jedino metoda pop() vraća element koji je ukinut i može biti ispisan ili pridružen nekom imenu (v. primjere iz tablice).

Uvjetni izrazi

Ako proširimo sintaksu izraza sa:

izraz *: gen_n-torke*

onda se u uvjetnom izrazu može pisati i *gen_n-torke* na mjestu izraza.

LAMBDA FUNKCIJE

Poslije uvođenja može se definirati *LAMBDA funkciju* koja će na mjestu *izraz* imati *n*-torku ili listu. Funkcija srt(), dana u nastavku, generira *n*-torku sačinjenu od tri broja uređena rastući:

```
>>> srt = lambda a, b, c : \
    ( min(a,b,c),
      a+b+c-min(a,b,c) - max(a,b,c),
      max(a,b,c) )
>>> srt (-10, -20, 5)      (-20, -10, 5)
>>> srt (10, 8, 10)          (8, 10, 10)
```

Ili, generiranje „znamenki“ rimskih brojeva može se postići *LAMBDA funkcijom* definiranom sa:

```
>>> RB = lambda x, y='', z='' :       \
   ('', x, 2*x, 3*x) if x == 'M' else \
   ('', x, 2*x, 3*x, x+y, y,
    y+x, y+2*x, y+3*x, x+z )
>>> RB ('M') # tisuće:
('', 'M', 'MM', 'MMM')
>>> RB ('I', 'V', 'X')  # jedinice:
('', 'I', 'II', 'III', 'IV', 'V', 'VI',
'VII', 'VIII', 'IX')
```

LAMBDA funkcija kao element niza

Prema dosad danoj sintaksi element niza (*n*-torke ili liste) može biti vrijednost bilo kojeg primitivnog ili složenog tipa podataka. Osim toga, lista može sadržavati ime funkcije, standardne ili vlastite, a *LAMBDA funkcije* su takve:

element_niza *: Lambda | ime_Lambda_fun*

Na primjer, lista L:

```
>>> L = ( lambda x : x**2,
          lambda x : x**3 )
```

sadrži dva elementa tipa function

```
>>> type (L[0])   <type 'function'>
```

koji referiraju na adresu *LAMBDA funkcija*, na primjer:

```
>>> L[0]
<function <lambda> at 0x02015BF0>
```

Izvršavanje tih funkcija bit će pozivom njihovih „imena“ L[0] ili L[1] i argumenta u zagradi:

```
>>> L[0](5)     25     >>> L[1](5)    125
```

U sljedeće smo dvije vježbe definirali *n*-torke s *LAMBDA funkcijama* kao elementima.

››› 7.21 *LAMBDA funkcija* kao element n-torke (1)

```
>>> from math import sin, cos, radians
>>> Y = ( lambda x : sin(x),
          lambda x : cos(x) )
```

```
>>> X = ()
>>> for i in range(10) :
    X += (i*10,);  i+= 1
>>> for x in X :
    r = radians(x)
    print( "%2d" % x, end = ' ' )
    for y in Y :
      print( "%7.4f" % y(r), end = ' ')
      print()
 0  0.0000  1.0000
10  0.1736  0.9848
20  0.3420  0.9397
30  0.5000  0.8660
40  0.6428  0.7660
50  0.7660  0.6428
60  0.8660  0.5000
70  0.9397  0.3420
80  0.9848  0.1736
90  1.0000  0.0000

>>> print( Y )
(<function <lambda> at 0x038944F8>,
<function <lambda> at 0x02E3C1E0>)
```

Vidimo da n-torka (par) `Y` sadrži definiciju dvaju lambda funkcija. Mogli smo ih zadati i eksplicitno, s pridruženim imenima `a` i `b`, kao u sljedećoj vježbi.

››› 7.22 *LAMBDA funkcija* kao element n-torke (2)

```
>>> a = lambda x : x**2; \
    b = lambda x : x**3
>>> Y = (a, b)
>>> for x in (1.5, 2.5, 3.5) :
    print( "%6.2f" % x, end = " " )
    for y in Y :
        print( "%8.2f" % y(x),
               end = " " )
        print()
  1.50     2.25     3.38
  2.50     6.25    15.62
  3.50    12.25    42.88
```

PRIDRUŽIVANJE ELEMENATA *n*-TORKE

Ako trebamo pridružiti jedan element s indeksom ***i*** *n*-torke ***T***, činili smo to s

ime = ***T***[***i***]

Ako smo željeli pridružiti više elemenata nizu imena, koristili bismo konkurentno pridruživanje. Međutim, u posebnom slučaju, ako je broj imena kojima treba pridružiti vrijednosti jednak broju elemenata *n*-torke, može se pisati:

$i_1, i_2, \ldots, i_n = T,\ n = \text{len}(T)$
ili $[i_1, i_2, \ldots, i_n] = T,\ n = \text{len}(T)$

››› 7.23 Pridruživanje elementa n-torke

```
>>> T = (10, 20, 30)
>>> x = T            # x je n-torka
>>> x, y = T
ValueError: too many values to unpack
>>> x, y, z   = T          # x, y i z su
cjelobrojne varijable
>>> # ili
>>> (x, y, z) = T
>>> x               10
>>> y               20
>>> z               30
>>> FUN = ( lambda x : x**2,
            lambda x : x**0.5 )
>>> f1, f2 = FUN; print( f1(2), f2(2) )
4, 1.4142135623730951
```

NABRAJANJE *n*-TORKE

Elementima *n*-torke funkcijom `enumerate()` može se generirati *n*-torka koja će sadržavati parove u kojima je svakom elementu pridružen „redni broj“. Pravilo pisanja je:

enumerate (*gen_n-torke* [,*start*])

Ako je *start* izostavljen, redni broj počinje od `0`.

enumerate.py

```
god_doba = ( 'proljeće', 'ljeto',
             'jesen', 'zima' )
GD = tuple( enumerate( god_doba, 1 ) )
print( god_doba ); print( GD )
for x in GD : print( x[0], x[1] )
```

```
>>>
('proljeće', 'ljeto', 'jesen', 'zima')
((1, 'proljeće'), (2, 'ljeto'), (3,
'jesen'), (4, 'zima'))
1 proljeće
2 ljeto
3 jesen
4 zima
```

GOVORIMO PYTHONSKI

Nizovi, *n*-torke i liste, važne su standardne strukture podataka u Pythonu. Ako smo svjesni što se sve može učiniti njihovom uporabom, naši će programi biti jednostavniji i čitljiviji. Na primjer, umjesto dijela programa `1.`:

```
while True :
  d = eval( input(
      'Upiši redni broj dana u tjednu,'
      ' 1 pon, 2 uto, ... ') )
  if type(d) == int and 1 <= d <= 7 :
     break
1.
Dan = (
'pon' if d == 1 else 'uto' if d == 2 else
'sri' if d == 3 else 'čet' if d == 4 else
'pet' if d == 5 else 'sub' if d == 6 else
'ned' )
```

bolje je rješenje `2.`

```
2.
Dani = ('', 'pon', 'uto', 'sri',
            'čet', 'pet', 'sub', 'ned')
Dan  = Dani [d]
```

n-torke ćemo koristiti kad se struktura podataka programa najbolje može predstaviti *n*-torkom. To će u pravilu biti vektori i matrice s nepromjenljivim podacima. Na primjer, u sljedećoj *n*-torci dani su prijevodi brojeva od 1 do 5 na francuski i engleski jezik:

```
( (1, 'un',     'one'   ),
  (2, 'deux',   'two'   ),
  (3, 'trois',  'three' ),
  (4, 'quatre', 'four'  ),
  (5, 'cinq',   'five'  ) )
```

Ili, umjesto niza uvjeta:

```
B = (
'jedan'if b==1 else 'dva'    if b==2 else
'tri'  if b==3 else 'četiri' if b==4 else
'pet'  if b==5 else 'pogreška')
```

bolje je koristiti *n*-torke:

```
T = ('pogreška', 'jedan', 'dva', 'tri',
     'četiri', 'pet')
if b < 1 or b > 5 : b = 0
B = T[b]
```

KONTROLIRANI UNOS PODATAKA

Ako želimo, na primjer, unijeti tri realna broja, stranice trokuta, možemo rabiti generirani niz koji će sadržavati funkciju `input()`:

```
>>> for i in 'abc' :
    exec (i +" = float( input ("
          +"'" +i +"'" +"' = '" +"))")
a = 10
b = 11
c = 12
>>> print( a, b, c, sep = '  ' )
10.0  11.0  12.0
```

GENERIRANJE LISTE SLUČAJNIH UZORAKA

Često je u nekim simulacijama potrebno generirati listu s k slučajnih uzoraka iz niza ili stringa duljine n, $k \leq n$. Uzorci se biraju sa slučajno izabranim indeksima, bez ponavljanja. Za to se može koristiti funkcija `sample()` iz modula `random`:

```
>>> from random import sample
```

Pravilo pisanja funkcije `simple()` je:

```
sample ( uzorci, broj_uzoraka )
uzorci       : lista | n-torka | string
broj_uzoraka : cjelobrojni_izraz
```

```
>>> sample( '12345',1)              ['5']
>>> sample( '12345',1)              ['3']
>>> sample( list( range( 1, 21 )), 5)
[18, 19, 6, 14, 15]
>>> sample( '12345',3 ) ['2', '4', '3']
>>> sample( '12345',3 ) ['4', '2', '1']
>>> sample( '1223333445', 5)
['2', '3', '3', '1', '4']
```

MODIFIKACIJA n-TORKE

Evo primjera dviju *LAMBDA funkcija* za modifikaciju *n*-torke, izbacivanje elementa sa zadanim indeksom i umetanje novog elementa ispred zadanog indeksa.

```
>>> DEL = lambda i, n_ : (
    n_[:i] +n_[i+1:] if 0 <= i < len(n_)
    else n_ )
```

```
>>> A = (1, 2, 8, 9, 10, 3, 4)
>>> A = DEL (4, A);  A
(1, 2, 8, 9, 3, 4)
>>> INS = lambda i, X, Y : (
    Y[:i] +X +Y[i:] if 0 <= i < len(Y)
    else Y )
>>> A = INS (2, ('Python', ), A)
>>> A      (1, 2, 'Python', 8, 9, 3, 4)
```

n-TORKE I FORMATIRANI STRING

Prema pravilima „starog“ generiranja formatiranog stringa vrijednosti koje će biti umetnute na definirana polja unutar stringa koji će biti generiran pišu se iza stringa i znaka %. Sada, poslije uvođenja *n*-torke, može se reći da je to *n*-torka vrijednosti koja može biti pridružena varijabli sa strukturom *n*-torke. Na primjer:

```
>>> f = (10, 20, 30); "%d " *len(f) % f
'10 20 30 '
```

n-TORKE I KOMPLEKSNI BROJEVI

Poseban slučaj *n*-torke jesu one s dva elementa ili uređeni parovi, ako su elementi brojčanog tipa. Parovi (*x*,*y*) mogu biti koordinate *x* i *y* točaka u ravnini, posebno vrhova trokuta. Na primjer:

```
>>> A = (-2,-1); B = (1,3); C = (3,-1)
```

Možemo definirati *LAMBDA funkciju* za izračunavanje udaljenosti između dvaju točaka X i Y:

```
>>> d = lambda X, Y : \
      ((X[0] -Y[0])**2 +(X[1] -
        Y[1])**2)**0.5
```

i izračunati stranice trokuta:

```
>>> a, b, c = d (B,C), d(A,C), d(A,B)
```

Konverzija para (točke *X*) u kompleksni broj može se realizirati *LAMBDA funkcijom*. Na primjer:

```
>>> z = lambda X : complex (X[0], X[1])
>>> A = z( (1, 2) );  A          (1+2j)
```

I obratno, ako je *Z* kompleksni broj, njegova konverzija u par može se realizirati sa:

```
>>> par = lambda Z : (Z.real, Z.imag)
```

Na primjer:

```
>>> par(A)                    (1.0, 2.0)
```

LISTA REZERVIRANIH RIJEČI PYTHONA

Modul keyword sadrži listu svih rezerviranih riječi Pythona.

Keywords.py

```
import keyword
print( keyword.iskeyword( 'if' ) )
Keywords = keyword.kwlist
print( Keywords )
```

```
>>>
True
['False', 'None', 'True', 'and', 'as',
'assert', 'break', 'class', 'continue',
'def', 'del', 'elif', 'else', 'except',
'finally', 'for', 'from', 'global', 'if',
'import', 'in', 'is', 'lambda',
'nonlocal', 'not', 'or', 'pass', 'raise',
'return', 'try', 'while', 'with',
'yield']
```

LISTA METODA MODULA

Pregled sadržaja modula (lista imena podataka, atributa i funkcija - metoda) može se dobiti izvršavanjem *naredbe DIR*:

```
dir ( ime_modula | ime_klase )
```

Sada se može reći da je *naredba DIR* u biti funkcija dir() koja vraća listu imena modula ili klase. Na primjer, dir(list) vraća listu imena metoda klase list. Sljedeći program ekstrahira samo one metode koje nemaju prefiks '_'.

Metode.py

```
# Generira listu metoda zadane klase
while 1 :
  Klasa = input (
    "\nGeneriram listu metoda klase? " )
  if not Klasa : break
  try    :
    exec ("Metode = dir ( "
          +Klasa +" )")
    metode = [ x for x in Metode
                        if x[0] != '_' ]
    print( "\nMetode klase "
           +Klasa +":" )
    print( * metode )
  except : continue
```

```
>>>
Generiram listu metoda klase? list

Metode klase list:
```

```
append clear copy count extend index
insert pop remove reverse sort

Generiram listu metoda klase? tuple

Metode klase tuple:
count index

Generiram listu metoda klase? str

Metode klase str:
capitalize casefold center count encode
endswith expandtabs find format format_map
index isalnum isalpha isascii isdecimal
isdigit isidentifier islower isnumeric
...
translate upper zfill

Generiram listu metoda klase? int

Metode klase int:
as_integer_ratio bit_length conjugate
denominator from_bytes imag numerator real
to_bytes

Generiram listu metoda klase? bool

Metode klase bool:
as_integer_ratio bit_length conjugate
denominator from_bytes imag numerator real
to_bytes

Generiram listu metoda klase? <Enter>
```

UPORABA STRUKTURA PODATAKA

Na ovom mjestu znamo osnovne tipove i tri strukture podataka Pythona (znakovne nizove, n-torke i liste), te gotovo sve primitivne i složene naredbe. To nam daje posebne mogućnosti u rješavanju mnogih problema, jer ćemo kombinirajući te tri strukture podataka moći dobiti pregledne i efikasne algoritme. Pritom ćemo koristiti veliki broj funkcija i metoda.

Kada koristiti te strukture podataka? Ako rješavamo probleme sa stringovima, prirodna je i njihova uporaba. Ponekad ćemo konvertirati brojeve u stringove da bismo lakše riješili neki problem, kao u primjeru ispisivanja duljine cijeloga broja i zbroja njegovih znamenki. Evo dva rješenja:

```
>>> N = \
234324234238542383289574328957489047280437
728095734089574389057348574353499999999999
217777777777777777777777771233166666
>>> # 1.
>>> S = k = 0; n = N
>>> while n > 0 :
    k += 1; n, b = divmod (n, 10); S += b
>>> print ( 'duljina =', k, 'zbroj =', S)
duljina = 119 zbroj = 647
>>> # 2.
>>> print ( "len(N) =", len (str (N)),
    "\nsuma brojki =", sum (
    [ int(x) for x in list (str (N)) ])
len(N) = 119
suma brojki = 647
>>> # Generiranje liste
>>> # [m *1, m *2, ..., m *5]
>>> m = list (range (1, 6))
>>> for x in [
    [ i*j for i in m ] for j in m ] :
    print ( x )

[1, 2, 3, 4, 5]
[2, 4, 6, 8, 10]
[3, 6, 9, 12, 15]
[4, 8, 12, 16, 20]
[5, 10, 15, 20, 25]
```

SKALARNI PRODUKT VEKTORA

Definicija skalarnog produkta (umnoška) vektora $\mathbf{a}=[a_1, a_2, \ldots, a_n]$ i vektora $\mathbf{b}=[b_1, b_2, \ldots, b_n]$ dana je sa:

$$\mathbf{a} \times \mathbf{b} = \sum_{i=1}^{n} a_i \times b_i = a_1 \times b_1 + a_2 \times b_2 + \ldots + a_n \times b_n$$

Vektor se može prikazati n-torkom, s brojčanim elementima, jednake duljine, pa je skalarni produkt, rabeći funkciju `sum()`, jednostavan za implementaciju. Na primjer:

```
>>> avec = (10, 20, 30, 40, 50)
>>> bvec = ( 7,  5,  3, 10, 20)
>>> sum (a*b for a,b
        in zip (avec, bvec))      1660
```

FUNKCIJA `reduce()`

Funkcija `reduce()` bila je standardna do inačice Pythona 3.0. Sada se nalazi u modulu **`functools`**. Piše se prema pravilu:

`reduce` (*`funkcija, niz`*)

gdje je funkcija zasad samo *LAMBDA funkcija* koja se

kontinuirano primjenjuje na *niz* i vraća jednu vrijednost. Funkciju *LAMBDA* nije potrebno nužno imenovati, može se napisati izravno kao prvi argument. Na primjer:

```
>>> from functools import reduce
>>> L = list (range (1, 101))
>>> reduce (lambda x, y : x+y, L)
5050
```

Dakle, značenje funkcije `reduce()` u ovom slučaju jednako je iteraciji:

```
>>> S = 0
>>> for x in L : S += x

>>> S
5050
```

FORMATIRANJE STRINGOVA

Uvođenjem niza proširuje se značenje formatiranja stringova. Pokazali smo to u sljedećim primjerima:

```
>>> a = (1,2,3)
>>> print (*a)
1 2 3
>>> *a
SyntaxError: can't use starred expression
here
>>> "{}, {}, {}".format (*a)  '1, 2, 3'
>>> print ("{}, {}, {}". format (*a))
1, 2, 3
>>> ("{}, "*(len(a)-1)+"{}").format(*a)
'1, 2, 3'
>>> a = ('Python', 'Pascal', 'C++')
>>> ("{}, "*(len(a)-1)+"{}").format(*a)
'Python, Pascal, C++'
>>> print( ' '.join ('%2d' % n for n in
                     range (10)) )
 0  1  2  3  4  5  6  7  8  9
>>> ['%2d' % n for n in range (6)]
[' 0', ' 1', ' 2', ' 3', ' 4', ' 5']
```

PRETRAGA LISTE

Ponekad je potrebno znati na kojim se mjestima (indeksima) pojavljuje element *a* u listi *X*. Dajemo primjer dva rješenja. U prvom rješenju inicijalno pretražujemo cijelu listu, od indeksa *k*=0. Ako lista sadrži *a* na poziciji *i*, *k* se uvećava za *i*. Potom pomičemo interval indeksa udesno, počevši od *k*+1.

```
>>> X = [1, 2] *4; a = 1; k = 0; I = []
>>> while a in X[k:] :
    k += X[k:].index(a)
    I.append (k); k += 1
>>> if I : print ("Lista", X, "sadrži",
                  a, "na indeksima:", I)
    else : print ("Lista", X,
                  "ne sadrži", a)
Lista [1, 2, 1, 2, 1, 2, 1, 2] sadrži 1
na indeksima: [0, 2, 4, 6]
```

U drugom rješenju koristimo pomoćnu listu *Y*. Bilježi se svaki indeks *k* pojavljivanja *a* u *Y* i *a* se zamjenjuje s praznim stringom. Time se dobivaju apsolutni indeksi pojavljivanja elementa *a* u listi *X*.

```
>>> X = [1, 2] *4
>>> Y = [] +X; a = 1; I = []
>>> while a in Y :
    k    = Y.index(a); I.append (k)
    Y[k] = ''
>>> if I : print( "Lista", X, "sadrži",
           a, "na indeksima:", I)
    else : print ("Lista", X,
           "ne sadrži", a )
Lista [1, 2, 1, 2, 1, 2, 1, 2] sadrži 1
na indeksima: [0, 2, 4, 6]
>>> Y      ['', 2, '', 2, '', 2, '', 2]
```

SORTIRANJE STRINGA I n-TORKE

Funkcija `sorted()` vraća sortiranu listu argumenta. Ako argument nije lista, potrebno je potom dobivenu listu pretvoriti u strukturu jednaku tipu argumenta (string ili n-torka). Za to ćemo definirati i dodati u **Moj_modul.py** naše funkcije `srt_s()`, za sortiranje stringa i `srt_t()` za sortiranje *n-torke*.

✚Moj_modul.py

```
srt_s = lambda x, y = False : \
    ''.join( sorted( x, reverse = y ) )
srt_t = lambda x, y = False : \
    tuple( sorted ( x, reverse = y ) )
>>> from Moj_modul import *
>>> srt_s( 'cgazolkjutr' )
'acgjklortuz'
>>> for x in srt_s( 'czsdžšđćčŽĐŠĆČ' ) :
    print( x, end = ' ' )
c d s z Ć ć Č č Đ đ Š š Ž ž
```

ITERIRANJE

Naredba za iteriranje kad je sekvenca iteriranja lista dopušta njezinu promjenu tijekom iteriranja. Na primjer, što će se dobiti izvršenjem dijela programa:

```
>>> L = [1, 1, 2, 3, 3, 2, 2]
>>> for x in L :
      if x % 2 : L.remove (x)
>>> L                        [1, 2, 3, 2, 2]
>>> L = [1, 1, 2, 3, 3, 2, 2];  i = 0
>>> for x in L :
    print( i, ' ', L, '\t', x )
    if x % 2 : L.remove (x)
    print( 3*' ', L )
    i += 1

0   [1, 1, 2, 3, 3, 2, 2]     1
    [1, 2, 3, 3, 2, 2]
1   [1, 2, 3, 3, 2, 2]        2
    [1, 2, 3, 3, 2, 2]
2   [1, 2, 3, 3, 2, 2]        3
    [1, 2, 3, 2, 2]
3   [1, 2, 3, 2, 2]     2
    [1, 2, 3, 2, 2]
4   [1, 2, 3, 2, 2]     2
    [1, 2, 3, 2, 2]
>>> L = [1, 1, 2, 3, 3, 2, 2];  i = 0
>>> for x in ( [] +L ) :
    print( i, ' ', L, '\t', x )
    if x % 2 : L.remove (x)
    print( 3*' ', L );  i += 1
0   [1, 1, 2, 3, 3, 2, 2]     1
    [1, 2, 3, 3, 2, 2]
1   [1, 2, 3, 3, 2, 2]        1
    [2, 3, 3, 2, 2]
2   [2, 3, 3, 2, 2]     2
    [2, 3, 3, 2, 2]
3   [2, 3, 3, 2, 2]     3
    [2, 3, 2, 2]
4   [2, 3, 2, 2]        3
    [2, 2, 2]
5   [2, 2, 2]     2
    [2, 2, 2]
6   [2, 2, 2]     2
    [2, 2, 2]
```

Dakle, zaključujemo da nije poželjno mijenjati sadržaj varijable koja se koristi kao sekvenca za iteriranje u *FOR petlji* jer će se dobiti neočekivani rezultati.

UPORABA FUNKCIJE `range()`

Funkcija `range()` je generator niza cijelih brojeva koji predstavljaju aritmetički niz. To bi trebala biti i njezina primarna uporaba u programima. Na primjer, kad još nismo bili uveli funkciju `range()`, umjesto niza naredbi:

```
B = [];  i = 1
while i <= N : B.append(i); i += 1
```

sada je bolje pisati:

```
B = list( range (1, N+1) )
```

Već smo ukazali da se semantika naredbe za iteriranje razlikuje od sličnih naredbi u drugim jezicima. Zbog toga inzistiramo da se ne koristi naziv „FOR petlja", već „iteriranje". To je posebno važno ako se kao sekvenca podataka iteratora koristi lista `list (range())` koja sadrži veliki broj podataka. Tada umjesto:

```
for i in list( range( 2, N+1 ) ) : ...
```

bolje je:

```
i = 2
while i <= N :
  ...
  i += 1
```

FAKTORIJEL (2)

Evo još jednog rješenja problema izračunavanja faktorijela, lambda funkcije `Ft()`, koja za zadani `n>1` izračunava umnožak elemenata liste `range(1, n+1)`.

```
>>> from functools import reduce; \
   Ft = lambda n : (
   'nije definirano' if type(n) == int
           and n < 0 or type(n) != int
    else 1 if n in [0, 1]
    else reduce (lambda x, y :
    x*y, [a for a in range (1, n+1)]) )

>>> Ft(10.5)    'nije definirano'
>>> Ft (-10)    'nije definirano'
>>> Ft (50)
30414093201713378043612608166064768844377641568960512000000000000
```

KARTEZIJEV PRODUKT

Primjer dan u nastavku prikazuje kako je moguće generirati Kartezijev produkt dviju listi.

```
# Kartezijev_produkt.py
boje     = ['bijelo', 'plavo']
veličine = ['S', 'M', 'L']

artikli  = [(b, v) for b in boje for v
            in veličine]
for art in artikli: print (art)
```

P R O G R A M I

Došli smo do točke kada imamo „alat" da možemo rješavati mnoge probleme, brojčane i tekstualne. Sada se možemo vratiti na neke probleme iz prethodnih poglavlja i pokazati kao se mogu napisati jednostavna rješenja primjenom *n*-torki i lista.

Petnaestak programa koje dajemo u ovom poglavlju u dovoljnoj mjeri prikazuje primjene svih dosad naučenih struktura podataka.

Plaćanje računa s najmanjim brojem apoena

U drugom smo poglavlju, programu

Plaćanje_računa.py

dali niz naredbi za izračunavanje najmanjeg broja apoena pri plaćanju računa u kunama. Ideja je bila da se zadani iznos dijeli, počevši od najvećeg apoena (1000 kn), pamti količnik, a ostatak se dalje dijeli sa sljedećim apoenom itd. Uvođenjem nizova i FOR petlje može se napisati preglednije rješenje. Prilažemo dva programa koji to potvrđuju.

Placanje_2.py

```
# Plaćanje iznosa u kunama s najmanjim
# brojem apoena
while 1 :
  C = kn = int (input ('Zadaj vrijednost
u kunama bez lipa '))
  if kn >= 1 : break
N = (1000, 500, 200, 100, 50,
     20, 10, 5, 2, 1); PL = ()
for n in N :
    x, kn = divmod( kn, n ); PL += (x, )
X  = "%4d"*10 % PL
kn = "%4d"*10 % N
print( "Iznos %d kn platiti sa:" % C )
print( X );  print( "   x"*10 )
print( kn )
```

```
>>>
Zadaj vrijednost u kunama bez lipa 1888
Iznos 1888 kn platiti sa:
   1   1   1   1   1   1   1   1   1   1
   x   x   x   x   x   x   x   x   x   x
1000 500 200 100  50  20  10   5   2   1
```

Placanje_3.py

```
# Plaćanje iznosa u kunama s najmanjim
# brojem novčanica
while 1 :
  kn = int (input ('Zadaj vrijednost '
            'u kunama bez lipa '))
  if kn >= 1 : break
print( 'Iznos od %d kn bit će plaćen'
       ' sa:\n' % kn )
N = (1000, 500, 200, 100, 50, 20, 10,
     5, 2, 1)
PL = ''
for n in N : x, kn = divmod (kn, n); \
    PL += "%4d x %d\n"                \
    % (x, n) if x else ''
print ( PL )
```

```
>>>
Zadaj vrijednost u kunama bez lipa 1888
Iznos od 1888 kn bit će plaćen sa:
   1 x 1000
   1 x 500
   1 x 200
   1 x 100
   1 x 50
   1 x 20
   1 x 10
   1 x 5
   1 x 2
   1 x 1
>>>
Zadaj vrijednost u kunama bez lipa 1001
Iznos od 1001 kn bit će plaćen sa:

1000 x 1
   1 x 1
```

Rastavljanje broja na faktore

Dani broj *n* treba napisati kao umnožak prostih (prim) brojeva:

$$n = f_1 \times f_2 \times \ldots \times f_k$$

Problem se svodi na nalaženje najmanjega višekratnika od *n*, označimo ga sa f_1, pa najmanjega višekratnika od (n / f_1), označimo ga s f_2, itd. Postupak se ponavlja sve dok se ne dobije broj jednak $n / (f_1 \times f_2 \times \ldots \times f_{k-1})$ koji je istodobno i najmanji višekratnik, označimo ga s f_k. Evo programa koji tako radi.

Faktori.py

```
from Moj_modul import *
while 1 :
  N = n = Input (
          'Zadaj cijeli broj > 0 ')
  if Int (N) and N > 0 : break
i = 2; Faktori = []
while i <= int (N**0.5) and n > 1 :
  if n % i == 0 :
    Faktori. append (i);  n //= i
  else          : i += 1
if n > 1 : Faktori. append (n)
print ( * Faktori )
```

```
>>>
Zadaj cijeli broj > 0
230556796394551842475310214733175607 0
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
53 59 61 67 71 73 79 83 89 97
```

RADNI SATI PO MJESECIMA

Treba izračunati koliko je bilo radnih sati u 2020. godini. Računa se po 8 sati za sve dane osim subota i nedjelja. Zna se da je 1.1.2020. bila srijeda.

Radni_sati_2020.py

```
# Radni sati po mjesecima u 2020. godini
M  = [0, 31, 29, 31, 30, 31, 30,
         31, 31, 30, 31, 30, 31]
# Dn = ['uto', 'sri', 'čet', 'pet',
#       'sub', 'ned', 'pon']
RS = [0]
for i in range (1, 13) :
  RS.append (sum ([8 for k in
     range (sum (M[:i]) +1,
     sum (M[:i]) +M[i]+1)
     if k % 7 not in [4, 5]]))

form = "%4d" *12
print( 'mjesec    : ', form %
       tuple (range (1, 13)) )
print( 'rad. sati : ', form %
       tuple (RS[1:]) )
print( 'Ukupno', sum (RS),
       'sati u 2020. godini' )
```

```
>>>
mjesec    :    1   2   3   4   5   6
   7   8   9  10  11  12
rad. sati :   184 160 176 176 168 176
 184 168 176 176 168 184
Ukupno 2096 sati u 2020. godini
```

HRVATSKE, ENGLESKE I FRANCUSKE BROJKE

U sljedećem programu zadana brojka hrvatskog, engleskog ili francuskog jezika prevodi se u dva preostala.

HR_EN_FR.py

```
H = ( 'nula', 'jedan', 'dva', 'tri',
      'četiri', 'pet', 'šest', 'sedam',
      'osam', 'devet' )
E = ( 'zero', 'one',   'two',  'three',
      'four', 'five', 'six', 'seven',
      'eight', 'nine' )
F = ( 'zéro', 'un', 'deux', 'trois',
      'quatre','cinq','six', 'sept',
      'huit', 'neuf' )

w = input ('Prevodim brojku? ').lower()
prevedi = lambda X, Y, Z : \
  Y[X.index (w)] +' ' +Z[X.index (w)]
A = (H, E, F) if w in H else \
    (E, H, F) if w in E else \
    (F, H, E) if w in F else \
    ('Ne postoji brojka ' +w
    +' niti u jednom jeziku!')
print( prevedi (A[0], A[1], A[2])
       if type (A) == tuple else A)
```

```
>>>
Prevodim brojku? DEUX          dva two
Prevodim brojku? eight         osam huit
Prevodim brojku? Nula          zero zéro
Prevodim brojku? DESET
Ne postoji brojka deset niti u jednom jeziku!
```

PREVOĐENJE ARAPSKIH BROJEVA U RIMSKE

Problem prevođenja arapskih brojeva u rimske svodi se na ekstrahiranje znamenki arapskog broja (cijelog broja od 1 do 3999) i izravno prevođenje u znamenke rimskog broja koje su dijelovi *n*-torki tisućica, stotica, desetica i jedinica, generiranih *LAMBDA funkcijom*.

Arap_rim_1.py

```
# Prevođenje arapskih brojeva u rimske
M = ('', 'M', 'MM', 'MMM')
C = ('', 'C', 'CC', 'CCC', 'CD',
     'D', 'DC', 'DCC', 'DCCC', 'CM')
X = ('', 'X', 'XX', 'XXX', 'XL',
     'L', 'LX', 'LXX', 'LXXX', 'XC')
I = ('', 'I', 'II', 'III', 'IV', 'V',
'VI', 'VII', 'VIII', 'IX')
```

```
while 1 :
  a = input ('Broj (1 - 3999) ')
  if not a : break
  try      :
    a = int (a)
    if not (1 <= a <= 3999) : continue
  except   :
    print ('Pogrešan podatak');
    continue
  m, c = divmod (a, 1000)
  c, x = divmod (c,  100)
  x, i = divmod (x,   10)
  Rimski = M[m] +C[c] +X[x] +I[i]
  print ('', a, '-->', Rimski)
```

```
>>>
Broj (1 - 3999) 3888
3888 --> MMMDCCCLXXXVIII
Broj (1 - 3999) 1001
1001 --> MI
```

Arap_rim_2.py

```
# Prevođenje arapskih brojeva u rimske
RB = lambda x, y = '', z = '' : \
     ('', x,2*x,3*x) if x == 'M' else \
     ('', x,2*x,3*x,x+y,y,y+x,y+2*x,
      y+3*x, x+z)
R  = (
    RB ('M'), RB ('C','D','M'),
    RB ('X','L','C'), RB ('I','V','X'))
while 2 :
  a = input('Zadaj broj od 1 do 3999 ')
  if not a : break
  try    :
    a = int (a)
    if not (1 <= a <= 3999) : continue
  except :
    print('Pogrešan podatak'); continue
  I = "%04d" % a;  rim = '';  k = 0
  for i in I: rim += R[k][int(i)]; k+=1
  print ( a, '-->', rim )
```

Prevođenje arapskih brojeva u rimske i obrnuto

Prvo treba generirati listu rimskih brojeva R:

```
R = [ '','I',…,'M',…,'MMCMXCIX','MMM']
```

Arapski broj `a`, `1 <= a <= 3999`, izravno se prevodi u rimski i jednak je `R[a]`. Ista se lista rabi i u prevođenju rimskog broja `r`, `r in R`, u arapski jednak je `R.index(r)`.

ARA_1.py

```
# Prevođenje rimskih brojeva u arapske
# i obrnuto
RB = lambda x, y = '', z = '' : \
   ('', x, 2*x, 3*x) if x == 'M' else \
   ('', x, 2*x, 3*x, x+y, y,
        y+x, y+2*x, y+3*x, x+z)
M  = RB ('M');  C  = RB ('C', 'D', 'M')
X  = RB ('X', 'L', 'C')
I  = RB ('I', 'V', 'X'); i = int
r  = lambda s : (
     M [i(s[0])] +C [i(s[1])]
    +X [i(s[2])] +I [i(s[3])] )
R  = [''] +[ r ("%04d" % a)
             for a in range (1, 4000) ]
while 'input' :
  a = input('Upiši rimski ili arapski '
            'broj ').upper()
  if not a  : break
  if a in R :
    print( ' ', a, '-->', R.index(a) )
  else :
    if   a.isalpha() : print(
         ' ', a, 'nije rimski broj' )
    elif a.isdigit() :
      a = eval(a)
      if a in range (1, 4000) :
        print( ' ', a, '-->', R[a] )
      else                      :
        print('  arapski broj van '
            'domene' )
  else                      :
    print(
    '  nije rimski niti arapski broj' )
```

```
>>>
Upiši rimski ili arapski broj 1
  1 --> I
Upiši rimski ili arapski broj 1001
1001 --> MI
Upiši rimski ili arapski broj Mcmlii
MCMLII --> 1952
Upiši rimski ili arapski broj 3888
  3888 --> MMMDCCCLXXXVIII
Upiši rimski ili arapski broj 0
  arapski broj van domene
Upiši rimski ili arapski broj -1.5
  nije rimski niti arapski broj
Upiši rimski ili arapski broj MMMCMXCIX
  MMMCMXCIX --> 3999
Upiši rimski ili arapski broj <Enter>
```

ISKLJUČUJUĆI "ILI" I IMPLIKACIJA

Nad logičkim tipom podataka definirali smo jednu unarnu (`not`) i dvije binarne (`and` i `or`) operacije. Osim njih u matematičkoj logici postoji još nekoliko binarnih operacija. Evo kako su definirane dvije, isključujući „ili" i implikacija:

xor_imp_2.py

```
# definicija isključujućeg "ili"
# xor(x,y) i implikacije imp(x,y)
xor = lambda x, y : \
      x and not y or not x and y
imp = lambda x, y : not x or y
print(
   'x      y      xor(x,y)  imp(x,y)' )
print(
   '-------------------------------' )
form = "%-7s" *2 +"%-8s  %-8s"
FT = (False, True)
for x in FT :
  for y in FT : print( form
              % (x, y, xor (x, y),
              imp(x,y)) )
```

```
>>>
x      y      xor(x,y)  imp(x,y)
-------------------------------
False  False  False     True
False  True   True      True
True   False  True      False
True   True   False     True
```

DAN U TJEDNU NA ODREĐENI DATUM 2021. GODINE

Znajući da je `1.1.2021.` bio petak, za zadani datum `2021.` godine treba ispisati koji je to dan u tjednu i njegov redni broj u godini.

dan_2021.py

```
# Redni broj dana i dan u tjednu datuma
# u 2021. godini
G = 2021; M = [0, 31, 28, 31, 30, 31, 30,
                  31, 31, 30, 31, 30, 31]
D = ('četvrtak', 'petak', 'subota',
     'nedjelja','ponedjeljak',
     'utorak', 'srijeda')
while 1 :
  d, m = eval (input (
               'Upiši dan i mjesec '))
  if type(d) == type(m) == int and \
     1 <= m <= 12 and 1 <= d <= M[m] :
     break
Datum = d, m
i = 1
for i in range (1, m) : d += M[i]
print (
  "%d.%d. %s, %d. dan u 2021. godini"
  % (Datum +(D[d %7], d)) )
```

```
>>>
Upiši dan i mjesec 29, 4
29.4. četvrtak, 119. dan u 2021. godini
```

ISPIS BROJA OD 1 DO 99 SLOVIMA

Cijeli broj `n` iz intervala od `1` do `99` treba ispisati slovima. Lista `J` sadrži brojeve od `1` do `10` koji se prevode izravno pišući `J[n]`, ako je $n \leq 10$, odnosno kao `J[j]`, gdje je `j` sufiks (jedinice) dvoznamenkastog broja, $0 \leq j \leq 9$. Iz te su liste izvedene lista `T`, za brojeve `n` od `11` do `19` i lista `D` (desetice) kao prefiks brojeva od `20` do `99`.

slovima.py

```
J = ['', 'jedan', 'dva', 'tri',
     'četiri', 'pet', 'šest', 'sedam',
     'osam', 'devet', 'deset']
T = ['', 'jeda'] +J[2:4] \
   +['četr', 'pet', 'šes'] +J[7:]
D = J[:4] +['četr', 'pe', 'šez'] \
   +J[7 :9] +['deve']
while 'n not in [1, 99]' :
  n = input(
    'Zadaj broj iz intervala [1, 99] ')
  if not n : break
  n = eval (n)
  if n not in range (1, 100) : continue
  d = n //10;  j = n %10
  S = ( J[n]          if n <= 10 else
        T[j] +'naest' if n <  20 else
        D[d] +'deset' +J[j]           )
  print ( '  ', n, '-->', S )
```

```
>>>
Zadaj broj iz intervala [1, 99] 100
Zadaj broj iz intervala [1, 99] 66
   66 --> šezdesetšest
Zadaj broj iz intervala [1, 99] 77
   77 --> sedamdesetsedam
Zadaj broj iz intervala [1, 99] 14
   14 --> četrnaest
Zadaj broj iz intervala [1, 99] 16
   16 --> šesnaest
Zadaj broj iz intervala [1, 99] 40
   40 --> četrdeset
```

```
Zadaj broj iz intervala [1, 99] 99
   99 --> devedesetdevet
Zadaj broj iz intervala [1, 99] <Enter>
```

FIBONACCIJEVI BROJEVI (3)

Evo i trećeg rješenja problema generiranja Fibonaccijevih brojeva, uz pomoć lista za koje možemo reći da je najprihvatljivije. Ako je

```
fib = [0, 1]
```

inicijalna lista koja sadrži prva dva člana Fibonaccijevog niza, preostalih `n-1` članova možemo generirati proširujući list `fib` zbrojem posljednja dva člana u svakom koraku iteracije sve dok je duljina liste manja od `n+1`:

```
while len(fib) < n+1 :
  fib.append(sum (fib[-2:]))
```

Generirana lista `fib` sadržavat će Fibonaccijeve brojeve (članove) `fib[i]`, `i=0, 1, ..., n`. Evo kompletnog programa:

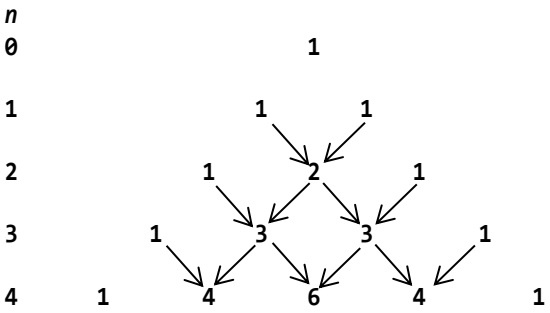
Fibonacci_3.py

```
# Fibonaccijev niz brojeva
while 'n < 2' :
  n = eval ( input (
  'Zadaj cijeli (int) broj veći od 1 '))
  if type (n) == int and n > 1 : break
fib = [0, 1]
while len(fib) < n+1 :
  fib.append(sum (fib[-2:]))
print( * fib[1:] )
```

```
>>>
Zadaj cijeli (int) broj veći od 1 20
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
987 1597 2584 4181 6765
>>> fib[10]          55
>>> fib[15]          610
```

BINOMNI KOEFICIJENTI (2)

Program za izračunavanje binomnih koeficijenata, Pascalov_trokut.py, koristeći formulu dali smo u petom poglavlju. Vidjeli smo da su koeficijenti simetrični i čine tzv. *Pascalov trokut* u kojem su na krajevima jedinice, a svaki unutrašnji član dobije se zbrajanjem susjedna dva člana iznad njega:



Pascalov_trokut_2.py

```
while 1 :
  S = int (input ('Ispis binomnih '
        +'koeficijenata (max. 15)? '))
  if 0 <= S <= 15 : break

(2)
L = []; n = 0; print ('(2)','\n', ' n')
while n <= S :
  print ("%2d" % n, end = ' ')
  Bk = [1];  i = 1
  while i < len(L) :
      Bk.append(L[i-1]+L[i]); i += 1
  if  n       : Bk.append (1)
  for k in Bk :
      print ("%4d" % k, end = ' ')
  print ();  L = Bk;  n += 1
print ()

(3)
BK = [[1] ]; print ('(3)', '\n', ' n')
for i in range (1, S+1) :
  A = BK[i-1];  B = [1]
  for j in range (len(A)) :
      B.append (sum (A[j:j+2]))
  BK.append (B)
for bk in BK :
  n = len(bk); print ("%2d" %(n-1), bk)
  # ili ("%5d"*n) %tuple(bk)
```

```
>>>
Ispis binomnih koeficijenata (max. 15)? 6

(2)
 n
 0    1
 1    1    1
 2    1    2    1
 3    1    3    3    1
 4    1    4    6    4    1
 5    1    5   10   10    5    1
 6    1    6   15   20   15    6    1
 7    1    7   21   35   35   21    7   1
```

```
(3)
 n
 0 [1]
 1 [1, 1]
 2 [1, 2, 1]
 3 [1, 3, 3, 1]
 4 [1, 4, 6, 4, 1]
 5 [1, 5, 10, 10, 5, 1]
 6 [1, 6, 15, 20, 15, 6, 1]
 7 [1, 7, 21, 35, 35, 21, 7, 1]
 8 [1, 8, 28, 56, 70, 56, 28, 8, 1]
```

NALAŽENJE PROSTIH BROJEVA (ERATOSTENOVO SITO)

Eratostenovo sito je postupak pronalaženja svih prostih brojeva u danom intervalu. Evo naprije kratkoga opisa toga postupka.

Donja je granica intervala koji se pretražuje broj 2. Ako je gornja granica N, N>2, ispišu se svi brojevi od 2 do N:

```
2 3 4 5 6 7 8 9 10 I1 12 13 14 15 16 17
... N
```

Zatim se uoči prvi prost broj, a to je 2, i prekriže redom svi brojevi koji su djeljivi njime (4, 6, 8, ...):

```
2 3 X 5 X 7 X 9 X 11 X 13 X 15 X 17 ...
N
```

Sada se uoči sljedeći neprekriženi broj. To je broj 3 pa se prekriže redom svi brojevi koji su njegovi umnošci (6, 9, 12, ...).

Postupak se nastavlja slijedećim neprekriženim brojem, sve dok se ne dođe do kraja. Brojevi koji su ostali neprekriženi jesu prosti brojevi. Postupak je okončan pri dosezanju prvoga prostoga broja većeg ili jednakoga drugom korijenu od N. Evo kompletnoga programa za ispisivanje prostih brojeva u danom intervalu.

Eratos.py

```
# Eratostenovo sito - prim brojevi
# od 2 do N
while 'N < 2' :
  N = int( eval( input(
      'Zadaj gornju granicu >=2 za '
      'ispis prim brojeva ')))
  if N >= 2 : break
P = ['X']*2 +list( range(2, N+1) )
for i in range( 2, int (N**0.5) +1 ) :
    if P[i] == 'X' : continue
    p = 2 *i
    P [p: : i] = ['X'] *len(P [p: : i])
while 'X' in P : P. remove( 'X' )
print( * P )
```

```
>>>
Zadaj gornju granicu >=2 za ispis prim
brojeva 100
2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
37, 41, 43, 47, 53, 59, 61, 67, 71, 73,
79, 83, 89, 97
```

LOTO

Ako treba generirati slučajne brojeve igre na sreću LOTO, 7 brojeva iz intervala 1..35, ili 6 iz intervala 1..45. Kao i pri rješavanju drugih problema, postoji nekoliko rješenja. Dajemo dva. U prvom se „izvlače" slučajni brojevi sve dok se ne generira lista s n različitih brojeva. Drugo je efikasnije jer rabi funkciju `sample` iz modula `random` koja vraća listu s n brojeva bez ponavljanja.

LOTO.py

```
# LOTO n od N
from random import *
n = 7;  N = 35; Ispis = (
  "print( s +'%3d' *n % tuple (Loto) )")
1.
Loto = []
while len (Loto) < n :
  b = randint (1, N)
  if b not in Loto : Loto.append(b)
s = '1.'; exec( Ispis ); Loto.sort()
s = '  '; exec( Ispis ); print( )

2.
Bubanj = range  (1, N+1)
Loto   = sample (Bubanj, n);
s = '2.'; exec( Ispis ); Loto.sort()
s = '  '; exec( Ispis )
```

Izvršenjem programa (to je ovdje izostavljeno jer morate imati svoje brojeve!) prvo će biti ispisani slučajni brojevi redom, kako su „izvučeni", potom su ispisani u rastućem nizu. Sretno!

GENERIRANJE "MINSKOG POLJA"

Minolovac (Minesweeper) poznata je igra koju je izmislio Curt Johnson devedesetih godina prošloga

stoljeća, a Robert Donner ju je 1992. godine prebacio u Windowse. Od tada se nalazi u svim inačicama Windowsa.

Ideja igre je da se pronađu sve skrivene mine postavljene u polju (matrici) dimenzija *m*×*n*. Količina mina ili „gustoća“ je određeni postotak, na primjer 15%, od ukupnog broja lokacija. Pomoć u pronalaženju mina jest oznaka broja mina koje se nalaze uz svaku lokaciju. Prazne lokacije su one koje nemaju nijednu minu u svojoj okolini. Evo programa koji generira „minsko polje“ u polju s M redova i N stupaca, s gustoćom 15%. Igru smo dali u poglavlju 14.

Mine.py

```
# Generiranje "minskog polja" za igru
# Minesweeper
from random import *
M, N = eval( input(
     'Zadaj broj redova i stupaca > '))
# broj mina u polju MxN
Mn = int (0.15 *M *N);  MP = []
for i in range (M): MP.append(['0']*N)
Bm = 0
while Bm < Mn :
  i0 = randint (0, M-1);
  j0 = randint (0, N-1)
  if MP[i0][j0] == '0' :
     MP[i0][j0] = '*'; Bm += 1
for i in range (M) :
  for j in range (N) :
    if MP[i][j] == '*' :
      '''Povećanje sadržaja lokacija
      oko mine'''
      for i0 in range (max (i-1, 0),
                       min (i+2, M)) :
        for j0 in range (max (j-1, 0),
                         min (j+2, N)) :
          x = MP[i0][j0]
          if x != '*' : x = int(x) +1; \
               MP[i0][j0] = str(x)
print ( '+' + '---+'*N )
for i in range (M) :
  print ('|', end = ' ' )
  for j in range (N) :
    x = MP[i][j]
    if x == '0' : MP[i][j] = ' '
    x = MP[i][j]
    print ( x +' |', end = ' ' )
  print( )
  print( '+' + '---+'*N )
```

```
>>>
Zadaj broj redova i stupaca > 6, 11
+---+---+---+---+---+---+---+---+---+---+---+
| * | 2 |   |   | 1 | 1 | 1 |   |   |   |
+---+---+---+---+---+---+---+---+---+---+---+
| * | 3 | 1 |   | 1 | * | 1 |   | 1 | 1 |
+---+---+---+---+---+---+---+---+---+---+---+
| 2 | * | 1 | 1 | 1 | 2 | 1 | 1 |   | 1 | * |
+---+---+---+---+---+---+---+---+---+---+---+
| 1 | 1 | 1 | 1 | * | 2 | 1 |   |   | 1 | 1 |
+---+---+---+---+---+---+---+---+---+---+---+
|   | 1 | 1 | 2 | 3 | * | 2 |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+---+
|   | 1 | * | 1 | 2 | * | 2 |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+---+
```

Igra križić-kružić (2)

U šestom smo poglavlju dali program igre križić-kružić u kojem igraju dva igrača. Ovdje ćemo pokazati kako se program može proširiti na igru „s kompjuterom”. Možda kompjuter ne igra savršeno pa vam prepuštamo eventualnu doradu programa!

Križić_kružić_2.py

```
# KRIŽIĆ - KRUŽIĆ
Y  = ['789', '456', '123', '741',
      '852', '963', '753', '951']
F  = list ('123456789');
C  = list ('51379')
NL = '\n'; T = ' '*4; NLT = NL +T
XO = lambda : \
     T +Y[0] +NLT +Y[1] +NLT +Y[2]
S  = lambda : ' '.join(Y)
Set = lambda : \
      (S().replace(a,I)).split(' ')
pos = lambda x : (
         x[0] *(x[0] in F)
         or x[1] *(x[1] in F)
         or x[2] *(x[2] in F) )
X = 'X';  O = 'O';  b = 0
while True :
  i = eval( input(
    'Prvi na potezu X ili O? ')
    . upper())
  if len(i) == 1 and i in 'XO' : break
I = _1 = i
while b < 9 :
  if b == 0 and _1 == O:
     print( XO() ); print( )
```

```
if I == X :
   if   b == 0 : a = '5'
   elif b == 1 : a = C[0]
   elif b == 2 :
      a = '9' if a in '24'   else \
          '1' if a in '6837' else '7'
   else :
     _2x, _2o, _x = [], [], []
     for y in Y :
       x    = y.count('X')
       o    = y.count('O')
       _2x += ([y] if x == 2
               and o == 0 else [])
       _2o += ([y] if o == 2
               and x == 0 else [])
       _x  += ([y] if x == 1
               and o == 0 else [])
     x = (_2x if _2x else
          _2o if _2o else _x)
     a = (pos (x[0]) if x else
          F[0] if _1 == X else C[0])
else :
   while 1 :
     a = input( 'Igra ' +I +' > ')
     if (len(a) == 1 and a in S() and
         a not in ' OX') : break
Y = Set()
F.remove (a)
if a in C : C.remove(a)
print( XO() ); print ( )
  if 3*I in Y :
    print( 'BRAVO,', I ); break
    I = O if I == X else X
    b += 1
  else : print( 'NERIJEŠENO!' )
```

```
Prvi na potezu X ili O? x
    789
    4X6
    123

Igra O > 7
    O89
    4X6
    123

    O89
    4X6
    X23

Igra O > 4
    O89
    OX6
    X23

    O8X
    OX6
    X23

BRAVO, X
>>>
```

8.

DATOTEKE

S dosad uvedenim naredbama ne bi se mogli riješiti mnogi problemi iz prakse, posebno oni koji uključuju rad s velikim brojem podataka, ili kad izlazne vrijednosti ne treba izravno pregledavati već služe kao ulazni podaci drugom, ili čak istom programu. U takvim je slučajevima primjena strukture datoteke jedino rješenje.

O općem konceptu datoteka nešto je rečeno u uvodnom dijelu ove knjige. Međutim, ako zauzmemo prilično apstraktno stanovište o podacima koji se čitaju i ispisuju programom, tada se takvi skupovi podataka nazivaju datoteke, bez obzira jesu li pohranjeni na disku, CD-u ili stiku, prikazani na ekranu monitora ili ispisani na papiru. Ovisno o mediju, neke će datoteke biti samo ulazne, tj. moći će se samo čitati. Druge će, pak, biti samo izlazne, tj. podaci će se moći samo ispisivati, a najčešće će biti one datoteke koje dopuštaju i čitanje i pisanje.

```
# Polica.py

import shelve
E = shelve.open ( 'Elementi.db' )
E ['H'] = 'vodik';  E ['O']   = 'kisik'
E ['S'] = 'sumpor'; E ['H2O'] = 'voda'
E. close()

X = shelve.open ( 'Elementi.db' )
t = '\t'
for x in X : print (x, t, X[x])
Y = list (X.keys())
Y . sort()
print ()
for y in Y : print (y, t, X[y])
>>>
H       vodik
O       kisik
S       sumpor
H2O     voda

H       vodik
H2O     voda
O       kisik
S       sumpor
```

```
>>>
Tečajna lista na dan:  01.05.2021.

RB Val Šif Par  Kupovni  Srednji  Prodajni
-----------------------------------------
 0 HRK 000 001  1.000000 1.000000 1.000000
 1 AUD 036 001  4.836606 4.851159 4.865712
 2 CAD 124 001  5.070798 5.086056 5.101314
 3 CZK 203 001  0.291101 0.291977 0.292853
 4 DKK 208 001  1.012605 1.015652 1.018699
 5 HUF 348 100  2.093366 2.099665 2.105964
 6 JPY 392 100  5.715522 5.732720 5.749918
 7 NOK 578 001  0.757432 0.759711 0.761990
 8 SEK 752 001  0.740224 0.742451 0.744678
 9 CHF 756 001  6.851968 6.872586 6.893204
10 GBP 826 001  8.658726 8.684780 8.710834
11 USD 840 001  6.225920 6.244654 6.263388
12 BAM 977 001  3.849838 3.861422 3.873006
13 EUR 978 001  7.529628 7.552285 7.574942
14 PLN 985 001  1.649462 1.654425 1.659388

Redni broj ulazne valute, iznos i redni
broj izlazne valute 13, 100, 0
100 EUR =  752.96 HRK
```

Uvod

Teško je naći nekoga u 21. stoljeću, a da ne zna što je datoteka. Ako kažemo „datoteka“, naravno, mislimo na datoteku na računalu.

Datoteka (eng. *file*) u računarstvu skup je logički povezanih binarnih podataka ili informacija, koji su spremljeni na medij dostupan programu za čitanje ili za promjenu. Operacijski sustav datoteke gleda kao niz binarnih podataka, dok na razini programa ovaj binarni podatak može se pretvoriti u: znak, broj, sliku, zvuk, izvorni program, izvršni program itd.

Obično se datoteka čuva na trajnom mediju za pohranu, npr. tvrdom disku. Jedinstveno ime i staza (*path*) rabe se u programima ili skriptama za pristup datoteci za čitanje, kopiranje i druge svrhe.

Unutar operacijskog sustava korisničke i ostale datoteke smještene su u posebni datotečni sustav, koji ima svoju posebnu strukturu. Ova struktura je obično usko povezana s programom, operacijskim sustavom, ili je u skladu s nekim dogovorenim standardom.

Struktura datoteke zove se format. Svaka datoteka također ima određenu veličinu koja je izražena u bajtovima, i obično je izražena u cijelim brojevima. Veličina je ponekada ovisna o operacijskom sustavu, ili o fizičkim svojstvima medija na kojem se datoteka nalazi.

Pojam upravljanja datotekama u kontekstu računala odnosi se na manipulaciju podacima u datoteci ili datotekama i dokumentima na računalu.

Programski jezik bez sposobnosti pohrane i dohvaćanja prethodno pohranjenih podataka bio bi jedva koristan. To se, prije svega, odnosi i na sami tekst programa koji smo pamtili u posebnoj datoteci.

Najjednostavniji zadaci koji se odnose na rad s datotekama su čitanje podataka iz datoteka i pisanje ili dodavanje podataka u datoteke. No, prije nego se upustimo u detaljan opis svega toga, opišimo strukturu ili organizaciju datoteka u Windowsima.

DATOTEKE U PYTHONU

Dosad smo rabili primitivne i složene varijable čije su vrijednosti bile pamćene unutar radne memorije i trajale su koliko i izvršavanje programa. Ako bismo željeli zapamtiti izlazne vrijednosti nekog izračuna, kao na primjer iz programa Tablica.py, označili bismo izlazne rezultate i s *Ctrl_C* pa *Ctrl_V* prenijeli u novootvorenu datoteku, općenito dokument u Notepadu, Pythonovom Shellu ili dodali nekom dokumentu u Wordu. Ono što smo prenosili u dokumente bio je tekst.

FUNKCIJA open()

Funkcija `open()` otvara postojeću ili definira strukturu nove datoteke. Pišemo je prema pravilu:

```
funkcija_OPEN :
  open ( radna_datoteka [, môd ] )
radna_datoteka : string
môd            :
  môd_tekstualne_datoteke |
  môd_binarne_datoteke
```

Funkcija `open()` otvara datoteku i vraća objekt tipa (klase) `file` koji može biti pridružen imenu - datotečnoj varijabli sa:

```
ime_dat = funkcija_OPEN
```

U stringu radne datoteke sadržano je stvarno ime pod kojim se datoteka pamti na sekundarnoj memoriji. Stvarno ime piše se prema pravilima za definiranje imena u operacijskom sustavu Windows. Može sadržavati putanju, ako datoteka nije u lokalnom folderu. *môd* je string koji indicira tip datoteke i na koji će način datoteka biti otvorena.

U programskom smo modu pisali programe (skripte) i pamtili ih, pridružujući im ime, kao tekstualne datoteke. Potom smo ih mogli učitati, preurediti i ponovno zapamtiti.

Općenito su datoteke strukture podataka koje sadrže zapise. Zapisi tekstualne datoteke su znakovi. Osim tekstualnih datoteka postoje i binarne datoteke, bez definiranoga tipa zapisa, duljine 128 bajtova.

Tekstualne datoteke

Sadržaj tekstualnih datoteka je tekst. S tekstom radimo od drugog poglavlja, a s tekstualnim datotekama od početka pisanja i pamćenja programa. S obzirom na to da su komponente tekstualne datoteke zapisi sačinjeni od znakova, ovi su grupirani u retke, koji završavaju kontrolnim znakom `'\n'`. Ovdje pojam

"redak" ne treba shvatiti doslovno, jer u internom zapisu podataka na datoteci znakovi predstavljaju neprekinuti niz koji završava kontrolnim znakom s kodom 26 (^Z) - oznakom kraja datoteke. Redak (zapis) može biti različite duljine (od nula do maksimalno jednake ukupnoj duljini datoteke). Za tekstualne datoteke postoje sljedeći modovi (načini) otvaranja:

- `'w'` Samo upisivanje podataka u datoteku. Ako je datoteka s navedenim imenom postojala, sadržaj će joj biti izbrisan, bez upozorenja!
- `'r'` Samo učitavanje podataka s datoteke. Datoteka mora postojati. U suprotnom se dojavljuje pogreška. Ako je môd izostavljen, jednak je `'r'`.
- `'a'` Dodavanje novih zapisa na kraj datoteke.
- `'r+'` Otvaranje datoteke za učitavanje i upisivanje. Datoteka mora postojati. U suprotnom se dojavljuje pogreška.
- `'w+'` Otvaranje datoteke za upisivanje i učitavanje. Ako je datoteka s navedenim imenom postojala, sadržaj će joj biti izbrisan, bez upozorenja!
- `'a+'` Dodavanje novih zapisa na kraj datoteke i/ili učitavanje zapisa neprazne datoteke.

```
>>> Dat = open( 'Podaci.TXT', 'w' )
>>> type( Dat )
<class '_io.TextIOWrapper'>
>>> print( Dat )
<_io.TextIOWrapper name='Podaci.TXT'
mode='w' encoding='cp1250'>
```

UPISIVANJE SADRŽAJA

Ako je *f* datotečna varijabla, u sljedećoj su tablici dane metode (funkcije) za upis sadržaja u tekstualne datoteke.

f. write(*s*)

Upisuje ***s*** u datoteku pridruženu datotečnoj varijabli *f*. Upis počinje od tekuće pozicije.

f. writelines(*niz*)

Upisuje string dobiven nastavljanjem stringova sadržanih u *n*-torki ili listi.

f. close()

Zatvara datoteku upisujući prije toga `Ctrl_Z` (`chr(26)`) na njezin kraj. Poslije toga nije definirana nijedna operacija nad *f*. Nastavak upisa moguć je ako se datoteka otvori s modom `'a'`.

Ako podatke upisujemo u nepostojeću datoteku ili postojeću datoteku koju želimo izbrisati, otvaramo je s modom `'w'`. Na primjer, otvorimo datoteku s radnim imenom `'Test.txt'`:

```
>>> Dat    = open ('Test.txt', 'w'); \
    Poruka = True
```

Potom upišimo nekoliko redova funkcijom `write()` dodavajući `'\n'` na kraj svakog upisa:

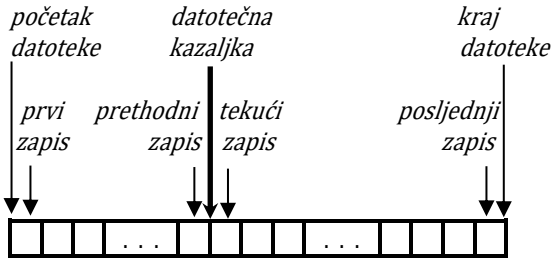
```
>>> while 1 :
    w = input( 'Upiši rečenice '
    '(Enter za prekid):\n'
        ) if Poruka else input ()
    Poruka = False
    if not w : Dat.close(); break
    Dat. write (w +'\n')
Upiši rečenice (Enter za prekid):
1. red                  7
2. red                  7
3. red                  7
<Enter>
```

Poslije upisa svakoga zapisa bila je ispisana njegova duljina zajedno s oznakom kraja reda, `'\n'`. Na kraju svih upisa obavezno zatvorimo datoteku:

```
>>> Dat . close()
```

STRUKTURA DATOTEKE

Datotečna varijabla predstavlja simboličko ime strukture radne datoteke prikazane sljedećim crtežom:



Redovi tekstualne datoteke (stringovi koji završavaju s `'\n'`) predstavljaju zapise datoteke. Broj zapisa datoteke određuje duljinu datoteke. Prazna datoteka ima duljinu jednaku nuli.

UČITAVANJE SADRŽAJA

U nastavku su dane metode (funkcije) za učitavanje sadržaja tekstualne datoteke:

f. tell()

Funkcija `tell()` vraća poziciju datotečne kazaljke datoteke f.

f.read([*n*])

Čita string od tekuće pozicije datotečne kazaljke do kraja datoteke, ako je ***n*** izostavljeno, odnosno, najviše ***n*** znakova od tekuće pozicije datotečne kazaljke.

f.readline([*n*])

Čita jednu ulaznu liniju, ako je ***n*** izostavljeno, odnosno string najviše duljine ***n***, od tekuće pozicije datotečne kazaljke do kraja linije. Poslije učitavanja datotečna kazaljka je pomaknuta iza učitanog stringa (na prvi znak sljedeće linije).

f.seek(*i* [, *k*])

Postavlja datotečnu kazaljku na:

- (apsolutnu) poziciju ***i***, ako je ***k*** izostavljeno ili jednako `0`
- ***i*** mjesta u odnosu na tekuću poziciju datotečne kazaljke, ako je ***k***=1 ili ***i***, ***i***≤0, mjesta u odnosu na kraj datoteke, ako je ***k***=2

PRIMJER

Isprobajmo sve dosad uvedene metode nad tekstualnom datotekom `'Test.txt'`.

Učitavanje zapisa

Prvo pogledajmo što je upisano u datoteku:

```
>>> Dat = open ('Test.txt', 'r')
>>> Dat . tell()      0
```

Poslije otvaranja datoteke datotečna je kazaljka na prvom znaku. Učitavamo sve zapise, odvojene s `'\n'`:

```
>>> Dat . read()
'1. red\n2. red\n3. red\n'
```

Sada je datotečna kazaljka iza posljednjeg zapisa:

```
>>> Dat . tell()      24
```

Postavimo datotečnu kazaljku na `0` i ispišimo sadržaj datoteke s `print()`:

```
>>> Dat.seek(0); Dat.tell(); \
    print (Dat.read()); Dat.tell()
0
1. red
2. red
3. red
24
```

Može se učitati pojedini zapis sa:

```
>>> Dat.seek(0); Dat.readline()
'1. red\n'
>>> Dat.readline(); Dat.readline()
'2. red\n3. red\n'
```

ili u FOR petlji, na sljedeći način:

```
>>> Dat.seek(0)
>>> for zapis in Dat: print(zapis[:-1])
1. red
2. red
3. red
```

Svaki zapis završava s `'\n'`, pa smo ga ispisali bez njega, jer `print()` ima "ugrađen" prijelaz u novi red.

Dodavanje zapisa

Da bismo dodali zapise na kraj datoteke, moramo je otvoriti u modu `'a'`:

```
>>> Dat. close(); \
    Dat = open ('Test.TXT', 'a')
>>> Dat. tell()       24
```

Dodatne ćemo redove teksta prvo generirati u stringu

```
>>> Tx = ''
>>> while 2 :
    w = input ('Upiši rečenicu: ')
    if not w : break
    Tx += w +'\n'

Upiši rečenicu: 4. red
Upiši rečenicu: <Enter>

>>> Dat. write(Tx); Dat. close()       7
```

Dodan je zapis duljine 7.

Modificiranje zapisa

Postojeću datoteku možemo modificirati ako je otvorimo s modom `'r+'`:

```
>>> f = open ('Text.txt', 'r+')
>>> f.read()
'1. red\n2. red\n3. red\n4. red\n'
>>> f.seek(0)                          0
>>> f.write ('012345678' '\n')         10
>>> f.seek(0)                          0
>>> f.read()
'012345678\nred\n3. red\n4. red\n'
```

U ovom je primjeru prepisan sadržaj datoteke od početka sa stringom `'012345678\n'`.

ATRIBUTI NAD DATOTEKAMA

Ako je *f* datoteka, postoji sedam atributa nad njom danih u nastavku.

f.closed

Logička vrijednost koja označuje trenutno stanje datoteke. Ovo je atribut samo za čitanje; metoda close() mijenja vrijednost.

f.encoding

Kodiranje koje koristi datoteka. Kada se Unicode nizovi zapišu u datoteku, pretvorit će se u bajtovne nizove pomoću ovog kodiranja. Uz to, kada je datoteka spojena na ekran, atribut daje kodiranje koje će ekran vjerojatno koristiti (te informacije mogu biti netočne ako je korisnik pogrešno konfigurirao ekran). Atribut je samo za čitanje i možda neće biti prisutan na svim datotekama.

f.errors

Obrađivač pogrešaka Unicode koji se koristi zajedno s kodiranjem.

f.mode

I / O način rada za datoteku. Ako je datoteka kreirana pomoću ugrađene funkcije open(), to će biti njegova vrijednost. Ovo je atribut samo za čitanje.

f.name

Naziv datoteke, ako je ime pridruženo stvarnoj datoteci stvoren pomoću open ().

f.softspace

Logička vrijednost koja označava treba li razmak ispisati prije druge vrijednosti kada se koristi naredba za ispis. Klase koje pokušavaju simulirati objekt datoteke također bi trebale imati atribut softspace koji se može zapisati, a koji bi trebao biti inicijaliziran na nulu. To će biti automatsko za većinu klasa implementiranih u Pythonu.

FUNKCIJE encode() I decode()

encode() funkcija koristi se za kodiranje stringa pomoću navedenog kodiranja. Ova funkcija vraća objekt klase bytes. Ako kodiranje nije navedeno, kao zadano se koristi kodiranje "utf-8".

decode() je inverzna funkcija funkciji encode(), tj koristi se za vraćanje bajtova u string.

```
>>> str_original  = 'Salut'
>>> bytes_encoded = \
str_original.encode(encoding = 'utf-8')
>>> print (type(bytes_encoded))
<class 'bytes'>
>>> str_decoded = bytes_encoded.decode()
>>> print (type(str_decoded))
<class 'str'>
>>> print ('Encoded bytes =',
           bytes_encoded)
Encoded bytes = b'Salut'
>>> print ('Decoded String =',
            str_decoded)
Decoded String = Salut
>>> print (str_original == str_decoded)
True
```

Binarne datoteke

Do sada smo koristili samo prvi parametar open, tj. naziv datoteke. Drugi parametar nije obavezan i prema zadanim je postavkama postavljen na "r" i ima značenje da se datoteka čita u tekstualnom načinu.

Drugi parametar određuje način pristupa datoteci ili drugim riječima način u kojem se datoteka otvara. Datoteke otvorene u binarnom načinu (dodajući 'b' argumentu modula) vraćaju sadržaj kao bajtovne objekte bez ikakvog dekodiranja. To ćemo pokazati u sljedećem primjeru. Za demonstraciju različitih efekata potreban nam je niz koji koristi znakove koji nisu uključeni u standardni ASCII. Zbog toga koristimo hrvatski tekst, jer koristi posebne znakove:

```
# Binarna_dat.py
X = "Često kiši i sniježi\n"
T = open ('Stanje.TXT', 'w')
T.write (X); T.close()
T = open ('Stanje.TXT', 'r') # tekst
print ('Tekst:'); print (T.read())
B = open ('Stanje.TXT', 'rb') # binarno
print ('Tekst binarno:'); print(B.read())
>>>
Tekst:
Često kiši i sniježi

Tekst binarno:
b'\xc8esto ki\x9ai i snije\x9ei\r\n'
```

Vidimo da su kodirana slova Č, š i ž, a ostala slova engleske abecede ostala su nepromijenjena. Kôd nije jednak kodu tih slova. Na primjer,

```
>>> ord('Č'), hex(268)    (268,'0x10c')
```

Niti je kodirani string jednak kodiranom stringu funkcijom encode():

```
>>> X.encode ()
b'\xc4\x8cesto ki\xc5\xa1i i
snije\xc5\xbei\n'
```

Binarne datoteke imaju sljedeće modove:

```
môd_binarne_datoteke :
    'wb'| 'rb'| 'ab'| 'rb+'| 'wb+'
```

sa značenjem:

- `'wb'` Samo upisivanje podataka u binarnu datoteku. Ako je datoteka s navedenim imenom postojala, sadržaj će joj biti izbrisan, bez upozorenja!
- `'rb'` Samo učitavanje podataka s binarne datoteke. Datoteka mora postojati. U suprotnom se dojavljuje pogreška. Ako je môd izostavljen, jednak je `'r'`.
- `'rb+'` Otvaranje binarne datoteke za učitavanje i upisivanje. Datoteka mora postojati. U suprotnom se dojavljuje pogreška.
- `'ab'` Dodavanje novih zapisa na kraj binarne datoteke.
- `'wb+'` Otvaranje binarne datoteke za upisivanje i učitavanje. Ako je datoteka s navedenim imenom postojala, sadržaj će joj biti izbrisan, bez upozorenja!

[4]

Sljedeći kôd pohranjuje popis brojeva u binarnu datoteku. Lista se prvo pretvara u niz bajtova prije pisanja. Ugrađena funkcija bytearray() vraća bajtovni prikaz objekta.

```
f   = open ("binfile.bin","wb")
num = [5, 10, 15, 20, 25]
arr = bytearray (num)
f   . write(arr)
f   . close()
```

Da bi se pročitala gornja binarna datoteka, izlaz metode read() prelijeva se na listu pomoću funkcije list().

```
f = open ("binfile.bin","rb")
L = list (f.read()); print (L)
f . close()
```

„KISELJENJE“ PODATAKA

Pretvaranje podataka u stringove (ili bajtove) za pohranu ili prijenos putem mreže uobičajena je operacija u računarstvu. Kao takav, postupak ima generički naziv: seriajalizacija (ponekad poznata i kao marširanje). „Kiseljenje“ (turšija) je specifičan za Pythonov oblik serijalizacije. Modul pickle dizajniran je za pretvaranje Pythonovih objekata u binarne sekvence. Pretvoreni tipovi objekata uključuju osnovne tipove podataka, poput cijelih brojeva i logičkih vrijednosti, i zbirke poput lista, *n*-torki i funkcija (osim LAMBDA funkcija).

dump() i load()

Modul pickle nudi nekoliko funkcija i klasa, ali obično koristimo samo dump() i load() funkcije. Funkcija dump() upisuje objekt (ili objekte) u datoteku, a load() čita objekt iz datoteke (obično objekt prethodno upisan s dumpom). Metode dump() i load() pišu se prema pravilima:

```
dump ( podatak, binarna_datoteka )
ime =  load ( binarna_datoteka )
```

Pogledajmo jedan primjer:

```
# Pickle.py
import  pickle
student =  [1012,'Pero','Perić', 'm',
           'INF', '2020-10-05', True]
pf  =  open ('Studenti.bin','wb')
pickle.dump (student, pf);  pf. close()
S = open ('Studenti.bin','rb')
Podaci =  pickle.load (S)
print (type (Podaci)); print (Podaci)
S. close()
>>>
<class 'list'>
[1012, 'Pero', 'Perić', 'm', 'INF',
'2020-10-05', True]
```

Generirana binarna datoteka ne može se „vidjeti” otvarajući je s nekim editorom teksta. Jedino se može učitati s load() i s type() možemo provjeriti tip učitanih podataka, kako je pokazano u primjeru, potom ga koristiti.

Police

Police, **shelve**, modul je Pythona koji se koristi za spremanje objekata u posebnu vrstu datoteka. Modul shelve može se koristiti kao jednostavna trajna opcija pohrane za Python objekte kada je relacijska baza podataka prekomplicirana za to. Podaci se upisuju u „bazu podataka“ koju kreira i njome upravlja poseban sustav, a pristupa im se uz pomoć jedinstvenih „ključeva“.

Da stavimo objekt na policu, prvo moramo uvesti modul `shelve`, a zatim dodijeliti vrijednost objekta na sljedeći način:

```
import shelve
polica = shelve.open ( ime_datoteke )
polica ['ključ'] = podatak
```

Da rezimiramo, ključ je string, podatak je proizvoljan objekt. Potpuno pravilo pisanja otvaranja datoteke je:

```
open (ime_datoteke, flag='c',
    protocol = None, writeback = False)
```

Parametri `flag`, `protocol` i `writeback` mogu biti izostavljeni. Parametar `flag` može biti:

`'r'` (zadano) samo za učitavanje,
`'w'` za čitanje-pisanje na postojećoj datoteci,
`'c'` za čitanje-pisanje na novoj ili postojećoj datoteci i
`'n'` za čitanje-pisanje na novoj datoteci.

Ako je imenu *d* pridružena polica s imenom *ime_dat*,

```
d = shelve.open (ime_dat, 'c')
```

u sljedećoj tablici dajemo sve operacije koje se mogu izvršiti nad policom:

`d[k] = podatak`	pohranjuje ***podatak*** u policu s ključem ***k*** (prepisuje stare podatke ako se koristi postojeći ključ)
`data = d[k]`	dohvatiti kopiju podataka pod ključem *k* (dojavljuje se KeyError ako nema takvog ključa)
`del d[k]`	izbriši podatke pohranjene s ključem ***k*** (dojavljuje se KeyError ako nema takvog ključa)
`k in d`	True ako ***k*** postoji u polici ***d***
`list( d.keys() )`	lista svih ključeva (polako!) kako je ***d*** otvoreno sa `writeback = False`

Na primjer:

```
# Polica.py
import shelve
E = shelve.open ( 'Elementi.db' )
E ['H'] = 'vodik';  E ['O']   = 'kisik'
E ['S'] = 'sumpor'; E ['H2O'] = 'voda'
E. close()

X = shelve.open ( 'Elementi.db' )
t = '\t'
for x in X : print (x, t, X[x])
Y = list (X.keys())
Y . sort()
print ()
for y in Y : print (y, t, X[y])
```

```
>>>
H       vodik
O       kisik
S       sumpor
H2O     voda

H       vodik
H2O     voda
O       kisik
S       sumpor
```

GOVORIMO PYTHONSKI

UPORABA DATOTEKA

Tekstualne ćemo datoteke češće učitavati i rabiti u svojim programima nego što ćemo ih generirati. Ako trebamo generirati neki tekst lakše je koristiti postojeće editore.

Datoteka.py

```
# Datoteka.py
# Definicija datoteke
Datoteka = """
Računalna datoteka računalni je resurs
za diskretno snimanje podataka u uređaj
računala za pohranu. Kao što se riječi
mogu pisati na papir, tako se
informacije mogu upisivati na datoteku
računala. Datoteke se mogu uređivati i
prenijeti uz pomoć nekog medija ili
putem interneta s jednog na drugo
računalo.

I ovaj program je tekstualna datoteka.
Učitava sam sebe:"""
```

```
open ("Def_datoteke.txt", "w",
  encoding = 'utf8'). write( Datoteka )
Tekst = open( "Def_datoteke.txt", "r",
  encoding = 'utf8' ). read()
print( Tekst )
print ( )
print( open( "Datoteka.py", "r",
       encoding = 'utf8' ). read() )
```

Postoji nekoliko ograničenja za predmete koji se mogu kiseliti. Opisani su u dokumentaciji modula. Kiseljenje nije rješenje za upravljanje podacima. Ono samo pretvara objekte u binarne sekvence. Te se sekvence mogu pohraniti u binarne datoteke i ponovno pročitati kako bi se mogle koristiti kao oblik postojanosti podataka.

„Kiseli krastavac“ ne pruža nikakva sredstva za pretraživanje pohranjenih predmeta ili za dohvaćanje jednog predmeta iz mnoštva pohranjenih predmeta. Morate čitati cijeli pohranjeni inventar natrag u memoriju i na taj način pristupiti objektima. Kiseli krastavac idealan je kada samo želite spasiti stanje programa kako biste ga mogli pokrenuti i nastaviti s istog položaja kao i prije (na primjer, ako ste igrali igru).

Pickle_art.py

```
# Kiseljenje artikala
import pickle
ART = (
  ('id', 'kat. br.', 'naziv', 'nc',
   'vpc', 'kol.', 'JM'),
  [1, "112/370", "AMORTIZER", 247.65,
   310.86, 0.00, "kom"],
  [2, "387 890 13 19",
   "AMORTIZER, zadnji", 638.60, 947.34,
   0.00, "kom"],
  [3, "108 353 00 42", "SEMERING, 601",
   82.98, 114.16, 4.00, "kom"],
  [5, "000 891 18 05",
   "AMORTIZER, kabine", 204.60, 388.72,
   1.00, "kom"] )
pf  =  open ('Artikli.bin','wb')
pickle.dump (ART, pf);  pf. close()
A = open ('Artikli.bin','rb')
Podaci =  pickle.load (A)
form = ("%2s %-15s %-18s " +2*"%8s "
        +"%5s %s" )
for x in Podaci :
  print (form % tuple(x))
```

```
>>>
id kat. br.         naziv
nc       vpc   kol. JM
 1 112/370          AMORTIZER
247.65    310.86    0.0 kom
 2 387 890 13 19    AMORTIZER, zadnji
638.6   1047.34    0.0 kom
 3 108 353 00 42    SEMERING, 601
82.98    114.16    4.0 kom
 5 000 891 18 05    AMORTIZER, kabine
204.6    388.72    1.0 kom
```

PROVJERA STATUSA DATOTEKE

Često ćemo pokušati učitati podatke s neke datoteke, oformiti ih ako datoteka ne postoji ili prekinuti daljnje izvršavanje.

```
try     :
  ime = open ('Datoteka', 'r')
  # učitavanje ...
except :
  # upisivanje sadržaja ili poruka
  # i prekid
```

KOSI HITAC (3)

U rijetkim ćemo prilikama imati potrebu prikazati izlazne rezultate kao tekst. Možda će to biti ako imamo rezultate nekog proračuna kao što je, na primjer, koordinate kosog hica.

Preuredimo program **Kosi_hitac_2.py** tako da izračunate vrijednosti budu zapamćene u datoteci, npr. `'Kosi_Hitac.TXT'`. Unos parametara kosoga hica isti je kao u prethodnom programu pa ćemo ga izostaviti:

Kosi_hitac_3.py

```
# Proračun putanje materijalne točke
# - kosi hitac

"... UNOS PARAMETARA KOSOG HICA KAO U
Kosi_hitac_2.py ..."
Dat = open ('Kosi_Hitac.TXT', 'w')
```

Radnu datoteku `'Kosi_Hitac.TXT'` pridružili smo imenu `Dat`, s modom `'w'`. Sada je sve spremno za upis podataka.

Vratimo se našem programu i oformimo string `Out` koji će sadržavati maksimalno vrijeme leta, `Tmax`, domet, `Domet`, i maksimalnu visinu materijalne točke kosoga hica, `Hmax`:

```
Out = 'Vo    = %10.2f (m/s)'  % V +NL \
     +'Alfa  = %10.2f (stup)' % α +NL \
     +'Tmax  = %10.2f (sec)'  % Tm+NL \
     +'Domet = %10.2f (m)'    % D +NL \
     +'Hmax  = %10.2f (m)'    % H +NL
```

Na kraju svake vrijednosti dodali smo prelazak u novi red, NL. Upis u datoteku je:

```
while 'n <= 4' :
  n = int (input (
           'Zadaj broj koraka (>4) za '
           'ispis putanje '))
  if n > 4 : break
Out += (NL +
        '  t            x           y'
       +NL +'-'*31 +NL )
t  = 0;  Dt = Tm /float(n)
while t <= Tm :
  x, y = Vx *t, Vy*t -g*t**2/2
  Out += ("%6.2f " % t +"%12.2f"*2
          % (x, y) +NL)
  t   += Dt
if Tm -t +Dt > 0.1 :
  Out += ("%6.2f " % Tm +"%12.2f"*2
          % (D, 0.0) +NL)
Dat.write (Out); Dat.close()
```

```
>>>
Upišite početnu brzinu, m/s i kut u st.
100, 45
Zadaj broj koraka (>4) za ispis putanje
10
>>> Dat = open ('Kosi_Hitac.TXT', 'r')
>>> s = Dat.read(); Dat.tell()        555
>>> print ( s )
Vo    =     100.00 (m/s)
Alfa  =      45.00 (stup)
Tmax  =      14.42 (sec)
Domet =    1019.37 (m)
Hmax  =     254.84 (m)
  t            x           y
------------------------------
  0.00         0.00        0.00
  1.44       101.94       91.74
  2.88       203.87      163.10
  4.32       305.81      214.07
  5.77       407.75      244.65
  7.21       509.68      254.84
  8.65       611.62      244.65
 10.09       713.56      214.07
 11.53       815.49      163.10
 12.97       917.43       91.74
 14.42      1019.37       -0.00
>>> Dat = open ('Kosi_Hitac.TXT', 'r')
>>> while True :
    line = Dat.readline()
    if not line : break
    print ( line )

Vo    =     100.00 (m/s)
Alfa  =      45.00 (stup)
Tmax  =      14.42 (sec)
Domet =    1019.37 (m)
Hmax  =     254.84 (m)
  t            x           y
------------------------------
  0.00         0.00        0.00
  1.44       101.94       91.74
...
>>> Dat = open ('Kosi_Hitac.TXT', 'r')
>>> while True :
      line = Dat.readline()
      if not line : break
      print( line[:-1] )
Vo    =     100.00 (m/s)
Alfa  =      45.00 (stup)
Tmax  =      14.42 (sec)
Domet =    1019.37 (m)
Hmax  =     254.84 (m)
  t            x           y
------------------------------
  0.00         0.00        0.00
  1.44       101.94       91.74
  2.88       203.87      163.10
  4.32       305.81      214.07
  5.77       407.75      244.65
  7.21       509.68      254.84
  8.65       611.62      244.65
 10.09       713.56      214.07
 11.53       815.49      163.10
 12.97       917.43       91.74
 14.42      1019.37       -0.00

>>> Dat = open ('Kosi_Hitac.TXT', 'r')
>>> for line in Dat : print line[:-1]
>>> Dat = open ('Kosi_Hitac.TXT', 'r')
>>> Dat.tell()              0
>>> Dat.read (20)
'Vo    =     100.00 ('
>>> Dat.tell()             20
>>> Dat.readline (20)      'm/s)\n'
>>> Dat.tell()             26
>>> Dat.readline (54)
```

```
'Alfa  =      45.00 (stup)\n'
>>> Dat.tell()            53
>>> Dat = open ('Kosi_Hitac.TXT', 'r')
>>> Dat.seek(500)
>>> Dat.readline()
'  917.43       91.74\n'
>>> Dat.readline()
' 14.42      1019.37       -0.00\n'
```

Pjesma

Evo pjesme (šansone) kao još jedan primjer tekstualne datoteke i centriranje izlaznih redova:

Šansona.py

```
T = """
šansona tek obična pjesma
za sva godišnja doba
pomalo dosadna muzika
šansona koju znam napamet
koju najčešće pjevam
kada me obuzmu misli sjetne

(Georges MOUSTAKI) """
open( "Šansona.txt", "w",
       encoding = 'utf8'). write( T )
Dat = open("Šansona.txt", "r",
            encoding = 'utf8')
for line in Dat:
    print( line[:-1].center( 27 ) )
```

```
>>>
 šansona tek obična pjesma
    za sva godišnja doba
   pomalo dosadna muzika
 šansona koju znam napamet
    koju najčešće pjevam
kada me obuzmu misli sjetne

      (Georges MOUSTAKI)
```

Datoteke kao označene police koristit ćemo ako nemamo puno podataka koje treba inicijalno upamtiti i povremeno ažurirati. Takva je, na primjer, tablica koja sadrži kemijske elemente (periodni sustav elemenata) ili tečajna lista.

P R O G R A M I

Hrvatsko-Englesko-Francuski brojevi 1 do 10

U sljedećem programu pokazano je kako primjenom polica možemo zapamtiti brojeve od 1 do 10 triju jezika i za zadani broj u jednom od njih dobiti prijevode u preostala dva.

Hr_En_Fr.py

```
# Prevođenje brojeva od 1 do 10
import shelve
H = ( 'jedan',   'dva',   'tri',
      'četiri',  'pet',   'šest',
  'sedam', 'osam',  'devet',  'deset')
E = ( 'one',     'two',   'three',
      'four',    'five',  'six',
  'seven', 'eight', 'nine',   'ten'   )
F = (  'un',     'deux',  'trois',
       'quatre', 'cinq',  'six',
  'sept',  'huit',  'neuf',   'dix'   )

Polica =  shelve.open ('H-E-F.sh','c')
Polica ['H'] = H; Polica ['E'] = E
Polica ['F'] = F; Polica.close()
HEF = shelve.open ('H-E-F.sh','c')
while "brojka" :
  B = input (
      'Brojka (H, E ili F) '). lower()
  if not B : break
  for b in 'HEF' :
    if B in HEF[b] :
      C = HEF[b].index(B)
      for c in 'HEF' :
        if b != c :
          print (B, '-->', HEF[c][C])
      break
  else : print ('nije brojka')
```

Prevođenje arapskih brojeva u rimske i obrnuto (2)

Dodajmo u program ARA_1.py binarnu datoteku koja će pamtiti generiranu listu rimskih brojeva.

ARA_2.py

```
# Prevođenje rimskih brojeva u arapske
# i obrnuto
RB = lambda x, y = '', z = '' : \
   ('', x, 2*x, 3*x) if x == 'M' else \
   ('', x, 2*x, 3*x, x+y, y,
        y+x, y+2*x, y+3*x, x+z)
```

```
M  = RB ('M');  C  = RB ('C', 'D', 'M')
X  = RB ('X', 'L', 'C')
I  = RB ('I', 'V', 'X')
i  = int
r  = lambda s : (
     M [i(s[0])] +C [i(s[1])]
    +X [i(s[2])] +I [i(s[3])] )
# dodano ------------------------------
from pickle import dump, load

try    :
  rim = open ('ARA.bin', 'rb')
  R   = load (rim)
except :
  R  = [''] +[ r ("%04d" % a)
               for a in range (1, 4000) ]
  rim = open ('ARA.bin', 'wb')
  dump (R, rim); rim.close()
# -----------------------------------
while 'input' : pass
# kopirati ostatak iz ARA_1.py
```

Periodni sustav elemenata

U sljedećem smo programu primjenom polica upamtili djelomični periodni sustav elemenata prikazan u tekstu E iz kojeg je generirana polica PS.

Periodni_sustav.py

```
# Djelomični periodni sustav elemenata
import shelve
E = """H  1.008 vodik
He 4.003 helij
Li 6.941 litij
Be 9.012 berilij
B  10.81 bor
C  12.01 ugljik
N  14.01 dušik
O  16.0 kisik
F  19.0 fluor
Ne 20.18 neon
Na 22.99 natrij
Mg 24.31 magnezij
Al 26.98 aluminij
Si 28.09 silicij
P  30.97 fosfor
S  32.07 sumpor
Cl 35.45 klor
Ar 39.95 argon
K  39.1 kalij
Ca 40.08 kalcij
Sc 44.96 skandij """
Dat = 'PER-SUS.db'

try    :
  PS = shelve. open (Dat, flag = 'r')
except :
  PS = shelve. open (Dat, flag = 'c')
  Persus = E.split('\n')
  for el in Persus :
    if el :
      [Sym, m, naziv]  = el.split()
      PS [Sym] = (float (m), naziv)
for el in PS : print (el, '\t', PS[el])
PS. close()
```

```
>>>
H    (1.008, 'vodik')
He   (4.003, 'helij')
...
Sc   (44.96, 'skandij')
```

Mjenjačnica

U sljedećem ćemo programu pokazati kako se može tečajna lista „skinuti" s interneta, prenijeti u tekstualnu datoteku i potom je koristiti u našoj „virtualnoj" mjenjačnici da bismo zadani iznos u jednoj valuti konvertirali u drugu. Pokazat ćemo to na primjeru tečajne liste Hrvatske narodne banke. Otiđite na internet i u svojoj tražilici napišite „tečajna lista hnb". Potom kliknite na Hrvatska narodna banka: Tečajna lista ili na Tecaj.

https://www.hnb.hr/temeljne-funkcije/monetarna-politika/tecajna-lista/tecajna-lista

Dobit ćete posljednju tečajnu listu. Ako se klikne na Formatirani zapis bit će prikazana tablica. Klikom na Opis formatiranog zapisa dano je značenje zapisa tablice. Na primjer, dana 30.04.2021. tablica je imala sljedeći izgled:

```
084300420210105202114
036AUD001       4,836606        4,851159        4,865712
124CAD001       5,070798        5,086056        5,101314
203CZK001       0,291101        0,291977        0,292853
208DKK001       1,012605        1,015652        1,018699
348HUF100       2,093366        2,099665        2,105964
392JPY100       5,715522        5,732720        5,749918
578NOK001       0,757432        0,759711        0,761990
752SEK001       0,740224        0,742451        0,744678
756CHF001       6,851968        6,872586        6,893204
826GBP001       8,658726        8,684780        8,710834
840USD001       6,225920        6,244654        6,263388
977BAM001       3,849838        3,861422        3,873006
978EUR001       7,529628        7,552285        7,574942
985PLN001       1,649462        1,654425        1,659388
```

Vidimo da se formatirani zapis tečajne liste nalazi na adresi:

https://www.hnb.hr/tecajn/htecajn.htm

Tečajna lista se primjenjuje od 01.05.2021.

Formatirani zapis tečajne liste za datum **ddmmgg** (**dd** – dan, **mm** – mjesec i **gg** – godina) može se dobiti na adresi:

https://www.hnb.hr/tecajn/f**ddmmgg**.dat

Da bismo ovu tečajnu listu zapamtili, treba učiniti sljedeće:

1) Označiti dio tablice s podacima za kopiranje (Ctrl_C)
2) Otići u program za uređenje teksta, na primjer Notepad, a može biti i Pythonov editor, prenijeti tekst s Ctrl_V i sačuvati ga u izabranom folderu pod imenom TL.TXT. Ako radimo s Pythonovim editorom pri spremanju obavezno izaberimo „Text files".

Sada nije problem napisati program koji će učitati podatke iz zapisa tekstualne datoteke, pohraniti ih u listu slogova i pretvoriti („mijenjati") zadani iznos iz jedne valute u drugu. Da bismo kune (HRK) mogli pretvoriti u drugu valutu dodali smo i tečaj kune koji je uvijek jednak 1.

Mjenjačnica.py

```
from Moj_modul import *

# UČITAJ TEČAJNU LISTU U TL
Dat = 'TL.txt'
try    : dat = open (Dat, 'r')
except : print ('NE POSTOJI DATOTEKA',
                Dat);  quit ()
datum = dat.readline()[11:19]
tl = dat.read(). replace (',', '.')
# pretvorba decimalnog zareza u točku!

# OFORMI TEČAJNU LISTU
TL0 = ['000HRK001       1.000000'
       '       1.000000       1.000000']

TL0 += tl.split ('\n')

# 'RAZBIJ' TEČAJNU LISTU
TL = []
for x in TL0 :
  x = x[3:6] +' ' +x[:3] +' ' +x[6:]
  TL.append (x.split())

# ISPIŠI TEČAJNU LISTU
print (NL +'Tečajna lista na dan: ',
      datum[:2] +'.' +datum[2:4] +'.'
     +datum[4:] +'.')
print ()
print ('RB Val Šif Par  Kupovni  '
       'Srednji  Prodajni')
print ('-----------------------------'
       '-------------')
RB = 0;  f = "%2s %s %s %s " +3*"%9s"
for x in TL :
  if x :
    t = tuple ([str(RB)] +x);
    print (f % t); RB += 1
print ()

# MJENJAČNICA
while True :
  try :
    s = input ('Redni broj ulazne '
        'valute, iznos i redni broj '
        'izlazne valute ')
    if not s : break
    i, X, j = eval (s)
    if (0 <= i < len(TL) and
        0 <= j < len(TL)) :
      Ul = TL[i] # Ul - ulazna lista
      Iz = TL[j] # Iz - izlazna lista
      print ( X, Ul[0], '= ',
        round (X *1.0/int (Ul[2])
        *eval (Ul[3] +'/' +Iz[5]), 2),
        Iz[0] )
    else : print (
          'greška, redni broj valute?')
  except :
    print (
      'POGREŠKA PRI UNOSU PODATAKA!')
    continue
```

```
>>>
Tečajna lista na dan:  01.05.2021.

RB Val Šif Par  Kupovni  Srednji  Prodajni
-----------------------------------------
 0 HRK 000 001  1.000000 1.000000 1.000000
 1 AUD 036 001  4.836606 4.851159 4.865712
 2 CAD 124 001  5.070798 5.086056 5.101314
 3 CZK 203 001  0.291101 0.291977 0.292853
 4 DKK 208 001  1.012605 1.015652 1.018699
 5 HUF 348 100  2.093366 2.099665 2.105964
 6 JPY 392 100  5.715522 5.732720 5.749918
 7 NOK 578 001  0.757432 0.759711 0.761990
 8 SEK 752 001  0.740224 0.742451 0.744678
 9 CHF 756 001  6.851968 6.872586 6.893204
10 GBP 826 001  8.658726 8.684780 8.710834
11 USD 840 001  6.225920 6.244654 6.263388
12 BAM 977 001  3.849838 3.861422 3.873006
13 EUR 978 001  7.529628 7.552285 7.574942
14 PLN 985 001  1.649462 1.654425 1.659388
```

```
Redni broj ulazne valute, iznos i redni
broj izlazne valute 13, 100, 0
100 EUR =  752.96 HRK

Redni broj ulazne valute, iznos i redni
broj izlazne valute 0, 1000, 12
1000 HRK =  258.2 BAM

Redni broj ulazne valute, iznos i redni
broj izlazne valute 13, 1000, 13
1000 EUR =  994.02 EUR

Redni broj ulazne valute, iznos i redni
broj izlazne valute <Enter>
```

U trećem smo primjeru promijenili `1000` EUR-a u EUR-e i dobili `994.02` EUR-a!? Evo odgovora. Prvo je `1000` EUR-a promijenjeno u kune:

```
Redni broj ulazne valute, iznos i redni
broj izlazne valute 13, 1000, 0
1000 EUR =  7529.63 HRK
```

potom dobivene kune u EUR-e:

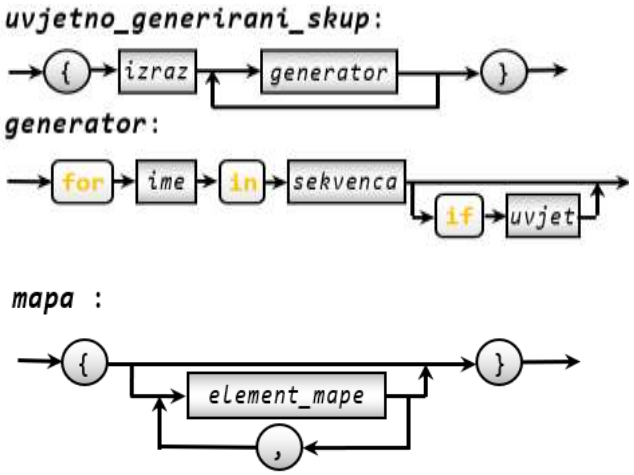
```
Redni broj ulazne valute, iznos i redni
broj izlazne valute 0, 7529.63, 13
7529.63 HRK =  994.02 EUR
```

9.

SKUPOVI I RJEČNICI (MAPE)

Važna karakteristika Pythona jest da ima skupove i rječnike (mape) kao standardne strukture (klase) podataka. Skup je struktura podataka nad kojom su definirane standardne operacije i relacije čime se pružaju posebne mogućnosti za rješavanje problema iz matematike i pisanje „elegantnih" programa. Rječnik (mapa) je sigurno jedna od najvažnijih struktura podataka.

U dijelu GOVORIMO PYTHONSKI i PROGRAMI pokazano je kako se skupovi i mape mogu koristiti u implementaciji nekih algoritama i rješavanju problema iz matematike i kemije.



element_mape: *ključ : podatak*
ključ: *element_skupa*
podatak: *br_izraz| log_izraz|string| n-torka| lista| skupovni_izraz| mapa*

ARA.py

```
# Prevođenje arapskih brojeva u rimske
# i obrnuto
from A_R_A import *
while 'input' :
  print(); a = input(
   'Upiši rimski ili arapski broj '
   ). upper()
  if not a : break
  if a in ARA : print (' ', a, '-->',
                       ARA[a])
  else        : print (' ', a,
     'nije rimski broj' if a.isalpha()
     else
      'arapski broj izvan domene'
       if a.isdigit()
     else 'nije rimski niti arapski '
          ' broj' )
```

```
>>>
Upiši izraz I**2 +II**2 +III**2 +IV**2
+V**2
  --> 55 --> LV
Upiši izraz M % VII
  --> 6 --> VI
Upiši izraz (i + i)*100
  --> 200 --> CC
Upiši izraz MMM +CM +XC +IX
  --> 3999 --> MMMCMXCIX
Upiši izraz C**3
1000000 nije rimski broj
Upiši izraz M -(CM +V)
  --> 95 --> XCV
Upiši izraz <Enter>
```

```
>>>
Upiši rimski ili arapski broj 3999
  3999 --> MMMCMXCIX
Upiši rimski ili arapski broj MLI
  MLI --> 1051
Upiši rimski ili arapski broj 4000
  4000 arapski broj van domene
```

Uvod

Pretpostavimo da trebamo riješiti sljedeći zadatak:

Zadatak 9.1

Generirati listu od 50 slučajnih cijelih brojeva iz intervala od 1 do 100 i potom izbaciti brojeve koji se ponavljaju.

Jedno od mogućih rješenja dano je u sljedećem programu:

Lista.py

```
# Lista brojeva bez ponavljanja
from Moj_modul import *
n = 50
A = [randint(1, 100) for i in range(n)]
A.sort()
print( 'Ulazni podaci: ' +NL*2, A, NL )
B = []
for a in A :
  if a not in B : B.append (a)

B.sort()
print( 'Izbačeno je', n-len(B),
       'duplikata:' +NL )
print( B )
```

```
>>>
Ulazni podaci:

[2, 4, 5, 7, 7, 8, 9, 11, 16, 16, 20,
22, 24, 24, 26, 26, 26, 28, 28, 31, 31,
36, 37, 38, 41, 42, 43, 44, 44, 47, 48,
48, 48, 49, 56, 61, 62, 62, 66, 68, 69,
73, 75, 80, 80, 93, 96, 96, 96, 98]

Izbačeno je 14 duplikata:

[2, 4, 5, 7, 8, 9, 11, 16, 20, 22, 24,
26, 28, 31, 36, 37, 38, 41, 42, 43, 44,
47, 48, 49, 56, 61, 62, 66, 68, 69, 73,
75, 80, 93, 96, 98]
```

Pretpostavimo da trebamo riješiti i sljedeći zadatak:

Zadatak 9.2

Prebrojati pojavljivanje pojedinih znakova sadržanih u nekom tekstu. Blankove ne brojati.

Jedino od mogućih rješenja s dosad uvedenim strukturama podataka jest:

Prebroj_znakove.py

```
S = input( 'Upiši znakovni niz:\n' )
C = '';  B = []
for c in S :
  if c ==' ' : continue
  if c in C  : i = C.find(c); B[i] += 1
  else       : C += c;  B.append (1)
A = list( zip ( C, B ) );  A.sort()
for (a, b) in A :
  print( "%s -->%3d " %(a, b),end = ' ' )
```

```
>>>
Upiši znakovni niz:
Prebroj sve znakove ove rečenice!
! -->  1  P -->  1  a -->  1  b -->  1
c -->  1  e -->  7  i -->  1  j -->  1
k -->  1  n -->  2  o -->  3  r -->  3
s -->  1  v -->  3  z -->  1  č -->  1
```

U ovom ćemo poglavlju uvesti dvije standardne strukture podataka (klase), skup i rječnik (mapa) koje će nam pomoći da napišemo „elegantnija“ rješenja postavljenih zadataka.

Skupovi

Iako su skupovi danas sastavni dio moderne matematike, to nije uvijek bio slučaj. Mnogi su odbacili teoriju skupova, čak i neki veliki mislioci. Jedan od njih bio je filozof Wittgenstein. Nije mu se svidjela teorija skupova i požalio se da je matematika „prožeta i pogubnim idiomima teorije skupova ...". Odbacio je teoriju skupova kao „krajnju glupost", kao „smiješnu" i „pogrešnu".

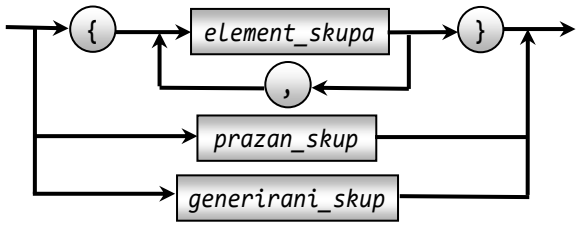
Njegova kritika pojavila se godinama nakon smrti njemačkog matematičara Georga Cantora, utemeljitelja teorije skupova. David Hilbert obranio ga je od kritičara glasno izjavivši: „Nitko nas neće protjerati iz raja koji je stvorio Cantor. Cantor je skup definirao na početku svog „Beiträge zur Begründung der transfiniten Mengenlehre" kao: „Skup je okupljanje u cjelinu određenih, različitih predmeta naše percepcije i naše misli - koji se nazivaju elementima skupa."

U današnje vrijeme na „običnom" jeziku možemo reći: Skup je dobro definirana zbirka objekata, najčešće označena velikim slovima. Elementi ili članovi skupa mogu biti bilo što: brojevi, znakovi, riječi, imena, slova abecede, čak i drugi skupovi itd. Ovo nije točna matematička definicija, ali je dovoljno dobra za sljedeće. Tip podataka „`set`", koji je vrsta zbirke, dio je Pythona od verzije 2.4.

DEFINICIJA SKUPA

Skup (*set*) je u Pythonu definiran kao struktura podataka, odnosno klasa s imenom set, koja sadrži kolekciju (sekvencu) elemenata bez ponavljanja. Element skupa može biti bilo koji primitivni tip, a od složenih tipova podataka samo nepromjenljivi: string i *n*-torka. Pravilo pisanja skupa je sljedeće:

skup :



element_skupa : *br_izraz*| *log_izraz*| *string*| *n-torka*

prazan_skup : set()

Prazan skup jest skup koji ne sadrži nijedan element. To je, prema danom pravilu, set().

››› 9.1 Skupovi

```
>>> set ( ) # prazan skup          set()
>>> { 1, 2 }                      {1, 2}
>>> type ( {1, 2} )        <class 'set'>
>>> a = 5
>>> { a, 2*a, 3*a }           {10, 5, 15)
>>> a = 2;  b = 4
>>> { a, 2*a, 3*a, b, 2*b, 3*b, 4*b }
{2, 4, 6, 8, 12, 16}
>>> { 2+1j, 'ab', (1,2) }
{(1, 2), 'ab', (2+1j)}
>>> {1, 2, 'a', [1,2,3]}
>>> # lista ne može biti element skupa!
TypeError: unhashable type: 'list'
>>> # skup sadrži elemente bez
>>> # ponavljanja:
>>> { 1, 2, 1, 2, 1, 1 }          {1, 2}
>>> { 2, 1, 2, 2, 1, 1 }          {1, 2}
>>> # skup je neuređena kolekcija
>>> # podataka:
>>> { (2, 'Python'), (1, 'Java'),
      (3, 'PHP') }

{(3, 'PHP'), (2, 'Python'), (1,
'Java')}
>>> { (1, 'Java'), (2, 'Python'),
      (3, 'PHP') }

{(3, 'PHP'), (1, 'Java'), (2,
'Python')}
```

PRIDRUŽIVANJE SKUPA

Skup može biti pridružen izabranom imenu naredbom jednostavnog pridruživanja:

ime = *skup*

Izvršenjem te naredbe *ime* će postati varijabla (objekt) tipa (klase) set. Ponekad ćemo je zvati skupovna varijabla.

››› 9.2 Pridruživanje skupa

```
>>> X = {1, 2, 3, 4};  Y = {3, 4, 5, 6}
>>> A = set();  A                 set()
>>> Ø = set()
>>> print( Ø )                    set()
>>> Jezici = {'Python', 'Pascal',
    'C++', 'PHP', 'BASIC', 'Delphi'}
>>> Jezici
{'Delphi', 'Python', 'BASIC', 'Pascal',
'PHP', 'C++'}
```

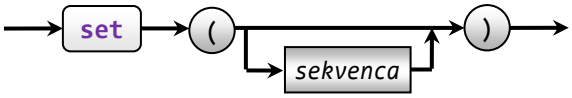
GENERIRANI SKUP

Skup može biti generiran na dva načina:

generirani_skup:

generirani_skup_iz_sekvence | *uvjetno_generirani_skup*

generirani_skup_iz_sekvence:



sekvenca: *string* | *n-torka* | *lista* | *funkcija_range* | *skup*

››› 9.3 Generiranje skupa iz sekvence

```
>>> set( (1,2,3,1,2,3) ) # n-torka
{1, 2, 3}
>>> set( [3,1,2] )       # lista
{1, 2, 3}
>>> set( 'abcd' )        # string
{'c', 'd', 'b', 'a'}
>>> set ( range(1, 10) )
{1, 2, 3, 4, 5, 6, 7, 8, 9}
>>> set( range (2, 18, 2) )
{2, 4, 6, 8, 10, 12, 14, 16}
>>> set( { 1, 2, 3} )          {1, 2, 3}
```

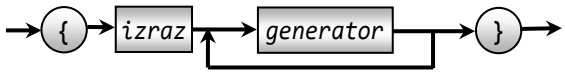
Ako je sekvenca izostavljena ili ako je jednaka praznom stringu, praznoj *n*-torci, praznoj listi ili funkciji range() koja ne generira niz, bit će generiran prazan skup.

››› 9.4 Generiranje praznog skupa

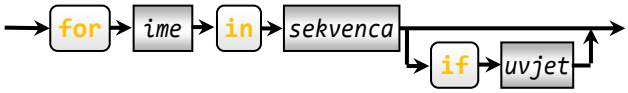
```
>>> set( '' )    set()  >>> set( )  set()
>>> set( () )    set()  >>> set([])  set()
>>> set( range( 2, 1 ) )          set()
```

Uvjetno generirani skup piše se prema sljedećem pravilu

uvjetno_generirani_skup:



generator:



Vidimo da je to pravilo jednako onome za generiranje liste, samo umjesto uglatih zagrada ovdje stoje vitičaste.

››› 9.5 Uvjetno generiranje skupa

```
>>> { x for x in range(10, 1) } set()
>>> { chr(x) for x in range( ord('a'),
      ord('z')+1 ) if chr(x) not in
      'aeiou'}
{'s', 'r', 'v', 'h', 'x', 'm', 'c', 'j',
'l', 'q', 'k', 'd', 'g', 'y', 'p', 'b',
't', 'w', 'f', 'z', 'n'}
>>> { x for x in range(16) if x%2 }
{1, 3, 5, 7, 9, 11, 13, 15}
>>> mala_grčka = {
    chr(c) for c in range(945, 970)}
>>> mala_grčka
{'δ', 'μ', 'φ', 'ε', 'χ', 'π', 'τ', 'λ',
'ι', 'ν', 'ξ', 'ο', 'ς', 'σ', 'ω', 'κ',
'γ', 'β', 'θ', 'ψ', 'ζ', 'η', 'υ', 'α',
'ρ'}
>>> len (mala_grčka)  25
>>> mala_grčka = list (mala_grčka)
>>> mala_grčka.sort(); print(
                        * mala_grčka )
α β γ δ ε ζ η θ ι κ λ μ ν ξ ο π ρ ς σ τ υ
φ χ ψ ω
```

Funkcija range() i generiranje skupa

S obzirom na to da je funkcija range() generator sekvence može se koristiti u generiranju skupa cjelobrojih vrijednosti iz zadanih granica i koraka funkcije range().

```
>>> set( range( 1, 11 ) )
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
>>> set( range( 10, 0, -1 ) )
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
>>> set( range( 2, 21, 2 ) )
{2, 4, 6, 8, 10, 12, 14, 16, 18, 20}
```

Iz prva dva primjera vidimo da je generiran isti skup, jer skup nije uređena struktura podataka!

SKUPOVNI IZRAZ

Pridruživanje vrijednosti skupovnim varijablama moguće je naredbom za inicijalizaciju i naredbom za dodjeljivanje. U naredbi za inicijalizaciju skupovna se vrijednost zadaje pišući eksplicitno skup s konstantama. Pridruživanje vrijednosti naredbom za dodjeljivanje postiže se pišući skupovni izraz na mjestu izraza (desna strana naredbe za dodjeljivanje).

skupovni_izraz: *skupovni_operand*
  { *skupovni_operator skupovni_operand* }
skupovni_operand: *skup* |
  ime_skupovne_varijable |
  (*skupovni_izraz*)
skupovni_operator: [-&|]

Značenje je skupovnih operatora sljedeće:

	unija skupova
–	razlika skupova
&	presjek skupova

Evaluiranje je skupovnoga izraza slijeva nadesno, uz sljedeći prioritet izvršavanja pojedinih operacija i izračunavanja:

(1) podizraz u zagradi
(2) presjek
(3) unija ili razlika

Nad skupovima su definirane relacije. Relacije < i <= imaju značenje podskupa, a > i => nadskupa.

››› 9.6 Relacije sa skupovima

```
>>> S1 = {1, 2, 3, 4};  S2 = {2, 4}
>>> S1 == S2          False
>>> S1 < S2           False
>>> S2 < S1           True
```

PRIPADNOST SKUPU I ITERIRANJE

Pripadnost podatka *x* skupu *S* određeno je izračunavanjem relacijskog izraza *x* in *S*. Na primjer:

```
>>> S1 = {1, 2, 3, 4};  S2 = {2, 4}
>>> S2 in S1   False  >>> 2 in S1   True
```

Proširena je sekvenca podataka, pa je sada:

sekvenca_podataka : *string*| *lista* |
  n-torka | *skup*

```
>>> for x in S1 : print( x, end = ' ' )
1 2 3 4
>>> S = {'jedan','dva','tri','četiri'}
>>> for x in S : print( x, end = ' ' )
dva četiri jedan tri
```

FUNKCIJE NAD SKUPOVIMA

Nad skupovima su definirane tri standarne funkcije:

`len()`, `min()` i `max()`

›››9.7 len(), min() i max()

```
>>> len ({1, 30, -2})                3
>>> min ({1, 30, -2})               -2
>>> min ({1, 'Python', (1,2)})       1
>>> min (1,  'Python', (1,2))        1
>>> min ([1, 'Python', (1,2)])       1
>>> max ({1, 'Python', (1,2)})    (1, 2)
>>> max (1,  'Python', (1,2))     (1, 2)
>>> max ([1, 'Python', (1,2)])    (1, 2)
```

METODE NAD SKUPOM

Nad skupom, klasom `set`, definirane su sljedeće metode:

›››9.8 Metode nad skupom

```
>>> Set_Methods = [x for x in dir(set)
                   if x[0] != '_' ]
>>> print( * Set_Methods )

add clear copy difference
difference_update   discard
intersection intersection_update
isdisjoint issubset issuperset pop
remove   symmetric_difference
symmetric_difference_update union
update
```

U nastavku dajemo opis metoda koje se češće koriste.

. add (*element*)

Metoda koja dodaje nepromjenljivi element skupu.

```
>>> boje = {"crveno", "zeleno"};  boje
{'crveno', 'zeleno'}
>>> boje. add( "žuto" );  boje
{'crveno', 'žuto', 'zeleno'}
>>> boje. add( ["crno", "bijelo"] )
TypeError: unhashable type: 'list'
```

Naravno, element će biti dodan samo ako već nije sadržan u skupu. U protivnom, poziv metode nema učinka.

. clear ()

Svi će se elementi ukloniti iz skupa.

```
>>> brojke = { 1, 2, 3, 4, 5 };  brojke
                          {1, 2, 3, 4, 5}
>>> brojke. clear();  brojke      set()
```

. copy ()

Vraća kopiju objekta.

```
>>> X = {1, 2, 3}; Y = X.copy(); X, Y
({1, 2, 3}, {1, 2, 3})
>>> id(X), id(Y); X.clear(); Y
(2552475027936, 2552475029280)
{1, 2, 3}
>>> X = {1, 2, 3}; Y = X; X, Y
({1, 2, 3}, {1, 2, 3})
>>> id(X), id(Y); X.clear(); Y
(2552475028384, 2552475028384)
set()
>>>
```

Naredbom Y = X samo se stvara pokazivač, tj. drugo ime, na istu strukturu podataka (isti objekt).

. difference ()

Ova metoda vraća razliku dva ili više skupova kao novi skup, a izvorni skup ostaje nepromijenjen.

```
>>> A = {"a", "b", "c", "d", "e"}; \
    B = {"b", "c"}; C = {"c", "d"}
>>> A. difference( B )  {'e', 'a', 'd'}
>>> A. difference( B ).difference ( C )
{'e', 'a'}
```

Umjesto metode `difference()` može se rabiti operator `"-"`:

```
>>> A - B              {'a', 'd', 'e'}
>>> A - B - C              {'a', 'e'}
```

. difference_update ()

Metoda `difference_update()` uklanja sve elemente drugog skupa iz prvog skupa.

`A.difference_update(B)` je isto što i `A = A -B` ili `A -= B`.

```
>>> A = {"a", "b", "c", "d", "e"}; \
    B = {"b", "c"}
>>> A. difference_update( B ); A
{'a', 'd', 'e'}
>>> A = {"a", "b", "c", "d", "e"}; \
    B = {"b", "c"}
>>> A -= B             {'a', 'd', 'e'}
```

. discard (*el*)

el će biti uklonjen iz skupa ako je sadržan u njemu, inače, ništa se neće poduzeti.

```
>>> A = {"a", "b", "c", "d", "e"}
>>> A. discard ( "c" ); A
{'a', 'b', 'd', 'e'}
>>> A. discard ( "c" ); A
{'a', 'b', 'd', 'e'}
```

. remove (*element*)

Djeluje poput **discard()**, ali ako *element* nije član skupa, dojavit će se KeyError.

```
>>> A = {"a", "b", "c", "d", "e"}
>>> A. remove ( "c" ); A
{'a', 'b', 'd', 'e'}
>>> A. remove ( "c" ); A
KeyError: 'c'
```

. union (*s*)

Ova metoda vraća uniju dva skupa kao novi skup, tj. sve elemente koji se nalaze u bilo kojem skupu. Umjesto ove metode može se rabiti operator „|":

```
>>> A = {"a", "b", "c", "d", "e"}
>>> B = {"a", "b", "e", "f"}
>>> A. union ( B ); A | B
{'f', 'a', 'b', 'c', 'd', 'e'}
{'f', 'a', 'b', 'c', 'd', 'e'}
```

. update (*s*)

Ova je metoda ekvivalentna metodi union(*s*), tj. vraća uniju dva skupa kao novi skup.

```
>>> S = {1,2,3}; S              {1, 2, 3}
>>> S.update ({2,3,4}); S  {1, 2, 3, 4}
```

. intersection (*s*)

Vraća presjek skupa instance i skupa *s* kao novi skup. Drugim riječima, vraća se skup sa svim elementima koji su sadržani u oba skupa. Umjesto ove metode može se rabiti operator „&":

```
>>> X = {1, 2, 3, 4, 5, 6}
>>> Y = {2, 4, 6, 8}
>>> X.intersection(Y); X & Y
{2, 4, 6}
{2, 4, 6}
```

. isdisjoint ()

Ova metoda je logička funkcija (relacija) koja vraća True ako su dva skupa disjunktna (nemaju zajedničkih elemenata).

```
>>> x = {"a", "b", "c"}
>>> y = {"c", "d", "e"}
>>> x. isdisjoint(y)              False
>>> y = {"d", "e", "f"}
>>> x.isdisjoint(y)                True
```

. issubset ()

x.issubset(y) vraća True, ako je x podskup od y. Može se koristiti relacija "<=" sa značenjem „podskup od" ili "<" sa značenjem „pravi podskup od".

```
>>> x = {"a", "b", "c", "d", "e"}
>>> y = {"c", "d"}; x.issubset(y); x<=y
(False, False)
>>> y <= x     True    >>> y < x    True
>>> x <= x     True    >>> x < x   False
```

. issuperset ()

x.issuperset(y) vraća True, ako je x nadskup od y.

Može se koristiti relacija „>=" sa značenjem „nadskup od" ili „<" sa značenjem „pravi nadskup od".

```
>>> X = {1, 2, 3, 4, 5}; Y = {4, 5}
>>> X.issuperset(Y); X >= Y; X > Y
(True, True, True)
```

. pop ()

pop() uklanja i vraća slučajno izabrani element skupa. Metoda vraća KeyError ako je skup prazan.

```
>>> x = {"a", "b", "c", "d", "e"}
>>> x. pop()    'e' >>> x. pop()    'a'
```

„ZAMRZNUTI" SKUPOVI

Funkcija frozenset() vraća nepromjenjivi objekt „zamrznutog" skupa (koji je poput objekta klase set(), samo nepromjenljiv). Kažemo da su to konstante klase frozenset(). Pogledajmo primjere:

```
>>> A = [1, 2, 3]; B = frozenset (A); B
frozenset({1, 2, 3})
>>> type (B)        <class 'frozenset'>
>>> C = frozenset ('abcd');  C
frozenset({'c', 'a', 'd', 'b'})
>>> D = frozenset ( {4, 5} );  D
frozenset({4, 5})
```

Zamrznuti skup se može pojaviti samo u izrazima na mjestima gdje se može pisati skup. Ako se piše u skupovnom izrazu, rezultat je tipa frozenset.

```
>>> E = B | {2, 3, 4, 5}
>>> E        frozenset({1, 2, 3, 4, 5})
>>> type (E)         <class 'frozenset'>
```

Metode koje mijenjaju sadržaj varijabli tipa `set()` nisu definirane nad `frozenset`.

SKUPOVI, *n*-TORKE I LISTE

Uvođenjem skupova moguće su konverzije u svim smjerovima između ove tri strukture podataka. Također su moguće konverzije zamrznutog skupa u nizove, kao i skup i obrnuto.

```
>>> tuple ( { 'Ncp', 'Abc' } )
('Abc', 'Ncp')
>>> A = ( 'Ncp', 'Af' )
>>> B = set ( A ); C = tuple ( B )
>>> A; C
('Ncp', 'Af')
('Af', 'Ncp')
>>> Vokali = set ('aeiouAEIOU')
>>> Vokali
{'a', 'u', 'i', 'O', 'E', 'o', 'A', 'U',
'e', 'I'}
>>> V = frozenset (Vokali);  V
frozenset({'a', 'u', 'i', 'O', 'E', 'o',
'A', 'U', 'e', 'I'})
```

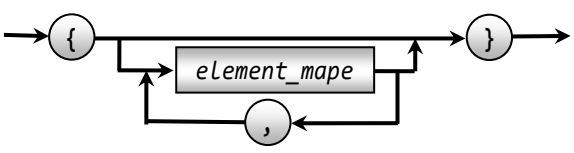
Pretvorbom liste (ili *n*-torke) koja ima dva ili više jednaka elementa u skup, duplikati će biti izbačeni. Na primjer:

```
>>> L = [10]*3 +2*[20] +2*[10]
>>> L       [10, 10, 10, 20, 20, 10, 10]
>>> L = set (L);  L              {10, 20}
>>> L = list (L); L              [10, 20]
```

Rječnik („mapa“)

Rječnik je posljednja standardna struktura podataka, odnosno klasa s imenom `dict`, Pythona. Mi ćemo je ponekad zvati „mapa“, jer je to struktura koja nadilazi pojam rječnika. Vidjet ćemo da se tu, općenito, radi o preslikavanju jednostavnih podataka i/ili struktura (domena) u jednostavne podatke i/ili strukture. Pravilo pisanja mape dano je sljedećim sintaksnim dijagramom:

mapa :



element_mape: *ključ* : *podatak*
ključ: *element_skupa*
podatak: *br_izraz*| *log_izraz*| *string*|
n-torka| *lista*| *skupovni_izraz*|
mapa

Prazna mapa (rječnik bez elemenata) piše se kao {}.

{ } je prazna mapa, a ne prazan skup! Prazan skup je `set ( )`.

Neprazna mapa sadrži jedan ili više elemenata mape odvojenih zarezom, napisanih prema danom pravilu. Mapa se može zamisliti kao skup u kojem je elementu skupa (nazvali smo ga ***ključ***) pridružen podatak dobiven izračunavanjem izraza bilo kojeg tipa.

Ključevi mape, kao i elementi skupa, imaju jedinstvene vrijednosti i služe kao „indeksi“ podataka koji su im pridruženi. Napomenimo da je uređenje ključeva poznato u teoriji algoritama (sortiranju) kao raspršeno („hash“) sortiranje. Međutim, to nam nije uopće bitno za rad s mapama.

››› 9.9 Mape

```
>>> {1: 'jedan', 2: 'dva', 3: 'tri'}
{1: 'jedan', 2: 'dva', 3: 'tri'}
>>> A = { 1, 2, 3 };  B = { 2, 3, 4, 5 }
>>> { 'A|B': A|B, 'A&B': A&B,
      'A-B': A-B  }
{'A|B': {1, 2, 3, 4, 5}, 'A&B': {2, 3},
'A-B': {1}}
```

PRIDRUŽIVANJE PODATAKA

Rječnik može biti pridružen izabranom imenu naredbom jednostavnog pridruživanja:

ime = *rječnik*

Izvršenjem te naredbe ime će postati varijabla (objekt) tipa (klase) `dict`. Na primjer, sa

```
>>> A = {}
>>> type (A)              <class 'dict'>
```

imenu `A` pridružen je prazan rječnik.

Osim eksplicitnog pridruživanja podataka rječniku, postoji još jedan način, prema pravilu:

ime_rječnika [*ključ*] = *podatak*

››› 9.10 Pridruživanje podataka

```
>>> A = {}
>>> B = ('', 'jedan', 'dva', 'tri')
>>> for i in range (1, 4): A[i] = B[i]
>>> A  {1: 'jedan', 2: 'dva', 3: 'tri'}
>>> Dan = {}
>>> Dan[1] = 'pon'; Dan[2] = 'uto'; \
    Dan[3] = 'sri'; Dan[4] = 'čet'; \
    Dan[5] = 'pet'; Dan[6] = 'sub'; \
    Dan[7] = 'ned'; Dan
```

```
{1: 'pon', 2: 'uto', 3: 'sri', 4:
'čet', 5: 'pet', 6: 'sub', 7: 'ned'}
```

PRIPADNOST MAPI I ITERIRANJE

Ako je *X* mapa, pristup elementima je sa:

X [*ključ*]

uz prepostavku da navedeni ključ postoji. Inače, dojavljuje se pogreška KeyError: <*ključ*>.

Pripadnost ključa *k* mapi *X* provjerava se na uobičajeni način, relacijom in, *k* in *X*.

I iteriranje je definirano na uobičajeni način, samo što sekvenci podataka dodajemo mapu, pa je sada konačno:

sekvenca_podataka : *string* | *lista* | *n-torka* | *skup* | *mapa*

Općenito uređenje ključeva mape nije onim redom kako je mapa proširivana (kao i kod skupa). Uz pretpostavku da je vježba 9.11 nastavak vježbe 9.10, evo nekoliko primjera:

››› 9.11 Element mape

```
>>> A[1]          'jedan'
>>> A[4]
... A[4]
KeyError: 4
>>> A[0] = 'nula';  A[0]          'nula'
>>> Dan[6], Dan[7]        ('sub', 'ned')
>>> for d in Dan : print (d, Dan[d])
1 pon
2 uto
3 sri
4 čet
5 pet
6 sub
7 ned
```

GENERIRANJE MAPE IZ SEKVENCE PAROVA

Mapa može biti generirana iz sekvence parova

mapa:

››› 9.12 Generiranje iz sekvence parova

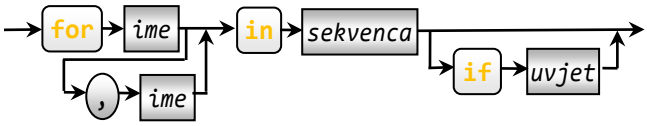
```
>>> dict (enumerate ('0123456789abcdef'))
{0: '0', 1: '1', 2: '2', 3: '3', 4: '4',
5: '5', 6: '6', 7: '7', 8: '8', 9: '9',
10: 'a', 11: 'b', 12: 'c', 13: 'd', 14:
'e', 15: 'f'}
>>> { c : chr(c) for c in
      range (945, 945+25) }
{945: 'α', 946: 'β', 947: 'γ', 948: 'δ',
949: 'ε', 950: 'ζ', 951: 'η', 952: 'θ',
953: 'ι', 954: 'κ', 955: 'λ', 956: 'μ',
957: 'ν', 958: 'ξ', 959: 'ο', 960: 'π',
961: 'ρ', 962: 'ς', 963: 'σ', 964: 'τ',
965: 'υ', 966: 'φ', 967: 'χ', 968: 'ψ',
969: 'ω'}
>>> C = tuple (zip (A, B));  C
((1, 'jedan'), (2, 'dva'), (3, 'tri'))
>>> dict (C)
{1: 'jedan', 2: 'dva', 3: 'tri'}
>>> D = dict(C)
>>> for d in D : print ( d, D[d] )
1 jedan
2 dva
3 tri
```

UVJETNO GENERIRANJE MAPE

Mape se, kao nizovi i skupovi, mogu uvjetno generirati prema pravilu:

uvjetno_generirana_mapa:

gen_mape:



Poseban slučaj uvjetnog generiranja mape jest:

```
{ a : b for a, b in sekvenca_parova }
```

koja je semantički ekvivalentna generiranju mape:

```
dict ( sekvenca_parova )
```

Na primjer:

```
>>> L = [(1,'jedan'), (2,'dva'),
         (3,'tri')]
>>> { x : y for x, y in L }
{1: 'jedan', 2: 'dva', 3: 'tri'}
>>> dict (L)
{1: 'jedan', 2: 'dva', 3: 'tri'}
```

METODE NAD MAPOM

Nad mapom, klasom dict, definirane su metode bez prefiksa „_” dobivene iz generirane liste:

```
>>> D_met = [ x for x in dir (dict)
                 if x[0] != '_' ]
>>> for s in D_met: print (s, end =' ')
clear copy fromkeys get items keys pop
popitem setdefault update values
```

U nastavku dajemo njihovo značenje. S ***D*** smo označili mapu, s ***k*** ključ i s ***p*** podatak. Neke ćemo metode pokazati nad mapom:

```
>>> D = { 1:'pon', 2:'uto', 3:'sri',
 4:'čet', 5:'pet', 6:'sub', 7:'ned' }
```

. clear()

Briše sve elemente iz mape.

. copy()

Vraća kopiju mape.

```
>>> Y = D.copy();  Y
{1: 'pon', 2: 'uto', 3: 'sri', 4: 'čet',
5: 'pet', 6: 'sub', 7: 'ned'}
```

. fromkeys(*k* [, *p*])

Vraća mapu sa specficiranim ključevima i podatkom. Ako je podatak izostavljen, zadano je None.

. get(*k* [, *p*])

Vraća podatak specificiranog ključa, ako je ***k*** in ***D***, inače ***p***, odnosno None ako je ***p*** izostavljeno.

. items()

Vraća listu parova svih ključeva i njihovih vrijednosti.

```
>>> D.items()
dict_items([(1, 'pon'), (2, 'uto'),
(3, 'sri'), (4, 'čet'), (5, 'pet'),
(6, 'sub'), (7, 'ned')])
```

. keys()

Lista svih ključeva mape. Najčešće je trebamo onda kad želimo ispis po sortiranim ključevima, kao što je prikazano u sljedećoj skripti za alfabetski ispis znakova sadržanih u nekom znakovnom nizu:

```
# brojač znakova
S = input ('Upiši znakovni niz:\n')
B = {}
for c in S :
  if c == ' ' : continue
  if c in B   : B[c] += 1
  else        : B[c]  = 1
A = B.keys();  A.sort()
for a in A : print ("%s -->%3d " %
             (a, B[a]), end = ' ')
>>>
```

```
Upiši znakovni niz:
Prebroj sve znakove ove rečenice!

! -->  1  P -->  1  a -->  1  b -->  1
c -->  1  e -->  7  i -->  1  j -->  1
k -->  1  n -->  2  o -->  3  r -->  3
s -->  1  v -->  3  z -->  1  č -->  1
```

Lista svih ključeva mape može se dobiti i sa:

```
list ( D )
```

. pop(*k* [, *p*])

Uklanja podatak s ključem ***k***. Ako ključ ne postoji, vraća ***p***, ako nije izostavljeno, inače dojavljuje:

```
KeyError: < k >
```

```
>>> D. pop (0)
...
KeyError: 0
>>> D. pop(0, 'ne postoji')
'ne postoji'
```

. popitem()

Uklanja stavku koja je zadnja umetnuta u mapu i vraća kao par (*ključ, vrijednost*). Ako je mapa bila prazna, dojavljuje pogrešku KeyError.

```
>>> D. popitem()          (7, 'ned')
>>> D. popitem()          (6, 'sub')
```

. setdefault(*k* [, *p*])

Vraća vrijednost sa zadanim ključem. Ako ključ ne postoji, umeće ga sa zadanom vrijednošću:

```
D[k] = p
```

ili, ako je ***p*** izostavljeno,

```
D[k] = None
```

Primjeri:

```
>>> D. setdefault (3, '123')      'sri'
>>> D. setdefault (6, 'sub')      'sub'
>>> D[6]                          'sub'
>>> D. setdefault (7)
>>> print (D[7])                   None
```

. update(*iterabilni*)

Ubacuje navedene stavke u rječnik. Navedene stavke mogu biti rječnik ili iterabilni objekt s parovima vrijednosti ključa.

```
>>> GR = { c: chr(c)
    for c in range (945, 945+25)}; GR
{945: 'α', 946: 'β', 947: 'γ', 948: 'δ',
949: 'ε', 950: 'ζ', 951: 'η', 952: 'θ',
```

```
953: 'ι', 954: 'κ', 955: 'λ', 956: 'μ',
957: 'ν', 958: 'ξ', 959: 'ο', 960: 'π',
961: 'ρ', 962: 'ς', 963: 'σ', 964: 'τ',
965: 'υ', 966: 'φ', 967: 'χ', 968: 'ψ',
969: 'ω'}
>>> GR.update ( { chr(c): c
    for c in range (945, 945+25) } )
>>> GR
{945: 'α', 946: 'β', 947: 'γ', 948: 'δ',
949: 'ε', 950: 'ζ', 951: 'η', 952: 'θ',
953: 'ι', 954: 'κ', 955: 'λ', 956: 'μ',
957: 'ν', 958: 'ξ', 959: 'ο', 960: 'π',
961: 'ρ', 962: 'ς', 963: 'σ', 964: 'τ',
965: 'υ', 966: 'φ', 967: 'χ', 968: 'ψ',
969: 'ω',
'α': 945, 'β': 946, 'γ': 947, 'δ': 948,
'ε': 949, 'ζ': 950, 'η': 951, 'θ': 952,
'ι': 953, 'κ': 954, 'λ': 955, 'μ': 956,
'ν': 957, 'ξ': 958, 'ο': 959, 'π': 960,
'ρ': 961, 'ς': 962, 'σ': 963, 'τ': 964,
'υ': 965, 'φ': 966, 'χ': 967, 'ψ': 968,
'ω': 969}
```

. values()

Vraća listu svih vrijednosti mape.

```
>>> D.values()
dict_values(['pon', 'uto', 'sri', 'čet',
'pet', 'sub', 'ned'])
>>> list (D.values())
['pon', 'uto', 'sri', 'čet', 'pet',
'sub', 'ned']
```

PRETVORBA MAPE U LISTU PAROVA

Mapa može biti generirana iz sekvence parova. Tako i obrnuto, lista parova može biti generirana iz mape sa

```
list ( D.items() )
```

Parove čine (*ključ*, *vrijednost*).

```
>>> D[7] = 'ned'
>>> D
{1: 'pon', 2: 'uto', 3: 'sri', 4: 'čet',
5: 'pet', 6: 'sub', 7: 'ned'}
>>> K = list (D)
>>> K                [1, 2, 3, 4, 5, 6, 7]
>>> Dani = list (D.items())
>>> Dani
[(1, 'pon'), (2, 'uto'), (3, 'sri'), (4,
'čet'), (5, 'pet'), (6, 'sub'), (7,
'ned')]
```

GOVORIMO PYTHONSKI

Struktura skupa daje nam posebne mogućnosti u programiranju. To se prije svega odnosi na rad sa znakovima i nizovima znakova. Primjenom skupova jednostavno se rješavaju problemi unije ili presjeka dvaju skupova znakova itd. Primjeri:

```
>>> A = {1, 2, 3}; B = {3, 4, 5}
>>> A | B                  {1, 2, 3, 4, 5}
>>> {*A, *B}              {1, 2, 3, 4, 5}
>>> A | B == {*A, *B}                True
>>> X +Y == [*X, *Y]                 True
>>> [X, Y]          [[1, 2, 3], [3, 4, 5]]
>>> A = { 1, 2, 3 };  B = { 4, 5, 6 }
>>> C = { *A, *B }
>>> C                  {1, 2, 3, 4, 5, 6}
>>>
```

Često će u rješavanju nekih problema trebati generirati skupove iz nekih drugih sekvenci i poslije određenih izračunavanja vratiti rezultat (podatke) u izvornu sekvencu i tamo ih, eventualno, sortirati. Na primjer, evo kako jednostavno možemo naći sve znakove koji se pojavljuju u nekom tekstu, ne smatrajući razmak znakom, potom u drugom programu pokazujemo kako ih prebrojati:

```
>>> s = input ('Upiši rečenicu ')
Upiši rečenicu Danas je lijepo vrijeme.
Konačno bez kiše!
>>> A = set (s) - {' '}
>>> A = list(A)
>>> A . sort()
>>> for x in A: print ( x, end = ' ' )
! . D K a b e i j k l m n o p r s v z č š
>>> # Prebrojavanje znakova
>>> s = input ('Upiši niz znakova\n');
>>> S = s.replace (' ', '')
Upiši niz znakova
Danas je 9. svibnja, Dan Europe!
>>> B = {}
>>> for c in S :
    if c not in B : B[c] = S.count (c)
>>> C = list (B.keys()); C.sort()
>>> for c in C :
    print (c, '->', B[c], end = ' ')
```

```
! -> 1 , -> 1 . -> 1 9 -> 1 D -> 2 E -> 1
a -> 4 b -> 1 e -> 2 i -> 1 j -> 2 n -> 3
o -> 1 p -> 1 r -> 1 s -> 2 u -> 1 v -> 1
```

Evo i programa koji iz generirane liste izbacuje brojeve koji se ponavljaju (Zadatak 9.1):

Lista_2.py

```
# Lista brojeva bez ponavljanja
from Moj_modul import *

n = 50
A = [randint(1, 100) for i in range (n)]
A.sort()
print( 'Ulazni podaci: ', NL*2, A, NL )

B = list( set( A ) );  B.sort()
print( 'Izbačeno je', n-len(B),
       'duplikata:', NL )
print( B )
```

```
>>>
Ulazni podaci:

 [8, 12, 12, 15, 16, 18, 19, 21, 22, 23,
24, 24, 25, 27, 30, 30, 33, 35, 36, 37,
37, 38, 38, 38, 41, 43, 46, 47, 49, 51,
60, 61, 66, 67, 67, 67, 69, 78, 78, 81,
82, 86, 90, 91, 93, 94, 95, 97, 98, 100]

Izbačeno je 9 duplikata:

[8, 12, 15, 16, 18, 19, 21, 22, 23, 24,
25, 27, 30, 33, 35, 36, 37, 38, 41, 43,
46, 47, 49, 51, 60, 61, 66, 67, 69, 78,
81, 82, 86, 90, 91, 93, 94, 95, 97, 98,
100]
```

Izvršenjem naredbe

```
B = list( set( A ) )
```

prvo je sa `set(A)` generiran skup koji ne sadrži duplikate učitane liste `A`. Potom je dobiveni skup transformiran u listu i pridružen varijabli `B`.

Na primjer, ako želimo generirati skup `k` različitih brojeva iz intervala brojeva od `1` do `n`, bez uporabe funkcije `sample()` iz modula `random`, možemo to učiniti uporabom skupa. Program je jednostavan:

```
>>> from random import randint
>>> S = set()
>>> while len (S) != 5 :
    S |= {randint (1, 39)}
>>> S                {33, 35, 8, 25, 15}
```

METODE MAPE

Sljedeći program dajemo kao pomoć za opis i prikaz parametara metoda mape.

dict_metode.py

```
# Metode rječnika (mape)
D_met = [ x for x in dir (dict)
          if x[0] != '_' ]
print ( '\nOpis metoda '
        'rječnika (mape)\n' )
for i, s in enumerate (D_met) :
  print ( i, s )
print( )

while 'izbor' :
  i = input ('Izaberi broj ispred '
             'metode '
             '(Enter za prekid): ')
  if not i : break
  i = eval (i)
  if (type(i) == int and
      i in range (0, len(D_met)) ):
    exec ("help (dict." +D_met[i] +")")
```

```
Opis metoda rječnika (mape)

0 clear
1 copy
2 fromkeys
3 get
4 items
5 keys
6 pop
7 popitem
8 setdefault
9 update
10 values

Izaberi broj ispred metode (Enter za
prekid): <Enter>
```

UGNJEŽĐENE MAPE

Mapa može sadržavati drugu mapu kao podatak. Tada kažemo da je to ugnježđena mapa. Na primjer, sljedeća mapa sadrži tri mape:

```
>>> Moje_mačke = {
    "maca1" :  { "ime"      : "Jojo",
                 "godište" : 2010 },
    "maca2" :  { "ime"      : "Jaun",
                 "godište" : 2010 },
    "maca3" :  { "ime"      : "Jacques",
                 "godište" : 2010 }  }
```

Pristup pojedinom podatku je sa:

```
>>> Moje_mačke["maca1"]
{'ime': 'Jojo', 'godište': 2010}
>>> Moje_mačke["maca3"]["ime"]
'Jacques'
```

SPAJANJE DVIJE MAPE

Ako su A i B dvije mape možemo ih spojiti u jednu mapu:

```
>>> A = { 'a' : 1, 'b' : 2 }
>>> B = { 'c' : 5, 'd' : 8, 'a' : 9}
>>> C = { **A, **B }
>>> # --> C = A.copy(); C.update (B)
>>> D = { **B, **A }
>>> # --> D = B.copy(); D.update (A)
>>> C  {'a': 9, 'b': 2, 'c': 5, 'd': 8}
>>> D  {'c': 5, 'd': 8, 'a': 1, 'b': 2}
```

Ako postoji zajednički ključ u A i B, vrijednost će biti ažurirana s vrijednošću mape koja je drugo navedena!

SORTIRANJE SADRŽAJA MAPE

Sadržaj mape može se sortirati uz pomoć standardne funkcije `sorted()`:

```
>>> # Kako sortirati rječnik po
>>> # vrijednosti
>>> X = {'a': 4,'b': 3,'c': 2 'd': 1}
>>> print ('X =', X)
>>> Xsort = sorted (X.items (),
            key = lambda x : x [1])
>>> print ('Xsort =', Xsort)

X = {'a': 4, 'b': 3, 'c': 2, 'd': 1}
Xsort = [('d', 1), ('c', 2), ('b', 3),
('a', 4)]
```

MAPE I INTERPRETATOR PYTHONA

Interpretator Pythona sve varijable programa pohranjuje u globalnom rječniku s imenom `vars()`. Uđimo u interaktivni mod i otipkajmo `vars()`:

```
>>> vars()
{'__name__': '__main__', '__doc__':
None, '__package__': None,
'__loader__': <class
'_frozen_importlib.BuiltinImporter'>,
'__spec__': None, '__annotations__':
{}, '__builtins__': <module 'builtins'
(built-in)>}
```

Prikazan je inicijalni sadržaj rječnika (mape) `vars()`. Izvršimo nekoliko naredbi i pogledajmo sadržaj:

```
>>> a = 10; b = 20; c = a; L = [1,2,3]
>>>
>>> vars()
{'__name__': '__main__', '__doc__':
None, '__package__': None,
'__loader__': <class
'_frozen_importlib.BuiltinImporter'>,
'__spec__': None, '__annotations__':
{}, '__builtins__': <module 'builtins'
(built-in)>, 'a': 10, 'b': 20, 'c': 10,
'L': [1, 2, 3]}
>>>
```

Dakle, `vars()` je prošireno sa `{'a': 10, 'b': 20, 'c': 10, 'L': [1, 2, 3]}`. Ključevi su imena varijabli s pridruženim vrijednostima odgovarajućeg tipa. S obzirom na to da je `vars()` rječnik, možemo mu promijeniti sadržaj dodajući novi element (varijablu i vrijednost) ili ukinuti ili promijeniti vrijednost postojeće varijable. Na primjer:

```
>>> vars()['a']  = 100
>>> vars()['A2'] = a**2
>>> print ( a, A2 )            100 10000
>>> del vars()['b']
>>> b
Traceback (most recent call last):
  File "<pyshell#20>", line 1, in
<module>
    b
NameError: name 'b' is not defined
>>>
```

U dijelu PROGRAMI pokazali smo kako se ovo svojstvo Pythona može iskoristiti u rješavanju nekih problema.

RIJETKO ZAPOSJEDNUTE MATRICE

Često u praksi imamo primjere rijetko zaposjednutih matrica. To su matrice formata $m \times n$, gdje je m broj redova, n broj stupaca, u kojima su definirane vrijednosti samo na određenim mjestima (i,j), $i \in m$, $j \in n$. Na primjer:

i\j	1	2	34	53
1	2,0			
2			3,1	
3				4,0
4		4,2		0,0

Implementacija je primjenom mapa vrlo jednostavna. Ključevi mape će biti parovi `(i,j)` kojima će biti pridružena vrijednost.

TABLICE

S mapama se mogu generirati mnoge tablice. Na primjer, tablica množenja:

```
>>> # Tablica_množenja.py
>>> T = {}; R = range (1, 11)
>>> for i in R :
    for j in R : T[(i, j)] = i * j
>>> T[(9,9)]              81
```

Ili, tablica funkcija `sin()` i `cos()` od `0` do `90` stupnjeva s korakom `10`:

```
>>> from math import pi, sin, cos
>>> sin_cos = {
    x : (round (sin(x*pi/180), 4),
         round (cos(x*pi/180), 4))
         for x in range (0, 91, 10) }
>>> tablica = """
print (" α    sin(α)    cos(α)")
print ("-" *22)
for α in sin_cos :
  print ("%2d%10.6f%10.6f" %
         (α, *sin_cos[α]) ) """
>>> exec (tablica)
 α    sin(α)    cos(α)
----------------------
 0  0.000000  1.000000
10  0.173648  0.984808
20  0.342020  0.939693
30  0.500000  0.866025
40  0.642788  0.766044
50  0.766044  0.642788
60  0.866025  0.500000
70  0.939693  0.342020
80  0.984808  0.173648
90  1.000000  0.000000
```

PRETVORBA MJERA

U sljedeća dva primjera pokazano je kako mapu možemo rabiti u pretvorbi nekih mjera.

```
>>> M = { 'cm' : (10,    'mm'),
          'dm' : (10,    'cm'),
           'm' : (10,    'dm'),
          'km' : (1000,   'm'),
           'h' : (3600, 'sec')   }
>>> # Pretvorba brzine iz m/sec u km/h
>>> v   = (10, 'm/sec') # v [km/h]?
>>> m   = (1 /M['km'][0], 'km');
>>> sec = (1 /M['h'][0], 'h')
>>> V = (v[0] *eval
        ('m[0]/sec[0]'),
          m[1] +'/' +sec[1])
>>> print ( *V )           36.0, 'km/h'
>>> # Pretvorba km u mm
>>> x = (1, 'km') # x ['mm']?
>>> while x[1] != 'mm' :
    y = M [x[1]];
    x = (x[0]*y[0], y[1])
>>> print ( * x )           1000000 mm
```

P R O G R A M I

HRVATSKO-ENGLESKO-FRANCUSKI BROJEVI 1 DO 10 (2)

Ovdje dajemo primjenom mape drugu verziju programa H_E_F.py iz prethodnog poglavlja.

H_E_F_2.py

```
H = ('jedan', 'dva',  'tri',   'četiri',
     'pet',   'šest', 'sedam', 'osam',
     'devet', 'deset', )
E = ('one',   'two',  'three', 'four',
     'five',  'six',  'seven', 'eight',
     'nine',  'ten'   )
F = ('un',    'deux', 'trois', 'quatre',
     'cinq',  'six',  'sept',  'huit',
     'neuf',  'dix',   )

A   = dict( zip( H, zip( E, F )))
B   = dict( zip( E, zip( H, F )))
C   = dict( zip( F, zip( H, E )))
HEF = { **A, **B, **C }
while "broj" :
  B = input ('Broj (H, E ili F) '
             ). lower()
  if not B : break
  if B in HEF :
    print (B, '-->', HEF[B])
  else : print ('nije broj')
```

POVRŠINA I OPSEG TROKUTA (2)

Dajemo program za računanje površine i opsega trokuta bez objašnjenja.

Trokut.py

```
# Površina i opseg trokuta s vrhovima
# A, B i C
d = lambda X, Y : round (((X[0] -
   Y[0])**2 +(X[1] -Y[1])**2) **0.5, 4)
T = {}; X = 'ABC'
for Y in X : T[Y] = eval (input (
          'Koordinate točke ' +Y +' '))
for Y in X : exec (Y +" = " +Y +"x, "
                +Y +"y = T['" +Y +"']")
P = round (abs (Ax*(By-Cy) +Bx*(Cy-Ay)
             +Cx*(Ay-By))/2.0, 4)
O = round (d(B,C) +d(A,C) +d(A,B), 4)
print ('O =', O, 'P =', P)
```

```
>>>
Koordinate točke A -3, -2
Koordinate točke B 1, -2
Koordinate točke C 1, 1
O = 12.0 P = 6.0
```

GENERIRANJE HAMMINGOVOGA NIZA

Članovi su Hammingovoga niza cijeli brojevi generirani prema pravilima:

1) 1 je član Hammingovoga niza.
2) Ako je X član Hammingovoga niza, tada su 2*X, 3*X i 5*X također članovi niza.

Evo programa koji generira sve članove Hammingovoga niza, ali u intervalu od 1 do N. Rješenje sadrži tri verzije generiranja liste.

Hammingov_niz.py

```
# Generiranje Hammingovog niza na
# intervalu od 1 do N
while 'N < 1' :
  N = int (input (
  'Generiram Hammingov niz do N, '
  'N >= 1 '))
  if N >= 1 : break
# 1. verzija
H = [1]; X = 1
while X <= N // 2 :
  if 2*X not in H : H += [2*X]
  if X <= N//3 and 3*X not in H :
     H += [3*X]
  if X <= N//5 and 5*X not in H :
     H += [5*X]
  X += 1;  H.sort()
  while X <= N and X not in H :
    X += 1
print ( H )
# 2. verzija
H = [1]; X = 1
while X <= N // 2 :
  Y = [k*X for k in [2,3,5]]
  for h in Y :
    if h not in H and h <= N : H += [h]
  X += 1;  H.sort()
  while X <= N and X not in H : X += 1
print ( H )

# 3.verzija
H = [False]*(N+1); X = 1; H[X] = 1
while X <= N // 2  :
  for k in [2,3,5] :
    i = k*X
    if i <= N : H[i] = i
  X += 1
  while X <= N and not H[X] : X += 1
while False in H : H.remove(False)
print ( H )
```

```
Generiram Hammingov niz do N, N >= 1
100
[1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15,
16, 18, 20, 24, 25, 27, 30, 32, 36, 40,
45, 48, 50, 54, 60, 64, 72, 75, 80, 81,
90, 96, 100]
```

GENERIRANJE HAMMINGOVOGA NIZA (2)

Evo četvrte verzije Hammingovog niza u kojoj koristimo skupove. Usporedite je s prethodne tri!

Hammingov_niz_2.py

```
# Generiranje Hammingovog niza na
# intervalu od 1 do n
while 'N < 1' :
  n = eval(input ('Generiram Hammingov'
      ' niz do n, n >= 1 '))
  if type (n) == int and n >= 1 : break
H = { 1 }; X = 1
while X <= n // 2 :
  H |= { 2*X } | ({ 3*X } if 3*X <= n
  else set()) | ({ 5*X }  if 5*X <= n
  else set())
  X += 1
  while X not in H : X += 1
H = list (H);  H.sort();  print ( H )
```

n-TI ČLAN FIBONACCIJEVOG NIZA (4)

Evo i četvrte verzije izračunavanja n-tog člana Fibonaccijevog niza primjenom mape u kojoj sljedeći član, i, izračunavamo zbrajanjem dva prethodna člana, i-1 i i-2,

Fibonacci_4.py

```
# n-ti član Fibonaccijevog niza
while 'n < 2' :
  n = eval ( input (
  'Zadaj cijeli (int) broj veći od 0 '))
  if type (n) == int and n > 0 : break
Fib = { 1: 1, 2: 1 }
for i in range (3, n+1) :
  Fib[i] = Fib[i-2] +Fib[i-1]
print ( str(n) +'-ti član Fibonaccijevog
niza je', Fib [n] )
```

Arapsko-rimsko-arapski rječnik

A_R_A.py

```
# Arapsko-rimsko-arapski rječnik
from pickle import *
try :
  f   = open ('ARA.dat', 'rb')
  ARA = load (f);  f. close()
except :
  RB  = lambda x, y='', z='' : (
    ('', x, 2*x, 3*x) if x == 'M' else
    ('', x, 2*x, 3*x, x+y, y, y+x,
         y+2*x, y+3*x, x+z) )
  M  = RB ('M'); C = RB ('C', 'D', 'M')
  X  = RB ('X', 'L', 'C')
  I = RB ('I', 'V', 'X');  ARA = {}
  for i in range (1, 4000) :
    s = [int (c) for c in "%04d" % i]
    r = M[s [0]] +C[s [1]] +X[s [2]] \
        +I[s [3]]
    a = str (i)
    ARA [a], ARA [r] = r, a
  f =  open ('ARA.dat', 'wb')
  dump (ARA, f)
```

Prevođenje arapskih brojeva u rimske i obrnuto (3)

ARA.py

```
# Prevođenje arapskih brojeva u rimske
# i obrnuto
from A_R_A import *
while 'input' :
  print(); a = input(
   'Upiši rimski ili arapski broj '
   ). upper()
  if not a : break
  if a in ARA : print (' ', a, '-->',
                       ARA[a])
  else        : print (' ', a,
     'nije rimski broj' if a.isalpha()
      else
       'arapski broj izvan domene'
        if a.isdigit()
      else 'nije rimski niti arapski '
           ' broj' )
```

```
>>>
Upiši rimski ili arapski broj 3999
  3999 --> MMMCMXCIX
Upiši rimski ili arapski broj MLI
  MLI --> 1051
Upiši rimski ili arapski broj 4000
  4000 arapski broj van domene
Upiši rimski ili arapski broj 0
  0 arapski broj van domene
Upiši rimski ili arapski broj 3888
  3888 --> MMMDCCCLXXXVIII
Upiši rimski ili arapski broj cix
  CIX --> 109
Upiši rimski ili arapski broj MILI
  MILI nije rimski broj
Upiši rimski ili arapski broj a1
  A1 nije rimski niti arapski broj
Upiši rimski ili arapski broj <Enter>
>>>
```

Izrazi s rimskim brojevima

U sljedećem ćemo programu pokazati kako možemo primijeniti modul **A_R_A.py** u izračunavanju cjelobrojnih izraza s rimskim i arapskim brojevima, s prikazom rezultata izračunavanja kao arapski i rimski broj. Na primjer:

```
3 * M  --> 3000 --> MMM
XV**II -->  225 --> CCXXV
```

Rimski_izrazi.py

```
# Izračunavanje izraza s rimskim
# brojevima
from A_R_A import *
for a in range (1, 4000) :
  rim = ARA [str(a)];  vars()[rim] = a
while 'izraz' :
  print (); E = input (
                'Upiši izraz ' ). upper()
  if not E : break
  try :
     R = str (eval (E))
     if R in ARA :
        print (
        '  --> '+ R +' -->', ARA[R] )
     else        :
        print ( R +' nije rimski broj' )
  except : print (
     'Leksička ili sintaksna pogreška!')
```

```
>>>
Upiši izraz I**2 +II**2 +III**2 +IV**2
+V**2
  --> 55 --> LV
Upiši izraz M % VII
  --> 6 --> VI
Upiši izraz (i + i)*100
  --> 200 --> CC
Upiši izraz MMM +CM +XC +IX
  --> 3999 --> MMMCMXCIX
Upiši izraz C**3
1000000 nije rimski broj
Upiši izraz M -(CM +V)
  --> 95 --> XCV
Upiši izraz <Enter>
```

KEMIJSKI SPOJEVI

Na Wikipediji,

https://hr.wikipedia.org/wiki/Relativna_molekulska_masa

možemo naći definiciju *relativne molekulske mase* kemijskog spoja, a to je zbroj relativnih atomskih masa koje čini formulsku jedinku ili molekulu spoja. Na primjer, relativna molekulska masa, M_r, vode (H_2O) računa se ovako:

$$M_r\ (H_2O) = 2 * A_r(H) + A_r(O)$$
$$= 2 * 1.008 + 16.0 = 18.016$$

Problem izračunavanja relativne molekularne mase kemijskih spojeva definiranih kemijskim formulama dobar je primjer za primjenu mape. Tekstualna datoteka PER-SUS.TXT sadrži periodni sustav elemenata, zapise u obliku:

```
  1 H     1.008 vodik     hydrogen
  2 He    4.003 helij     helium
...
112 Cn  277.000 ununbij   ununbium
```

Učitani su zapisi datoteke PER-SUS.TXT i oformljena je lista PS. Iz nje je generiran rječnik Psus. Ključevi su simboli kemijskih elemenata, a podaci njihove relativne molekularne mase.

Spojevi.py

```
Persus = open("PER-SUS.TXT","r"); PS = []
for el in Persus :
  PS.append (el[:-1].split())
Psus = {x[1]: float(x[2]) for x in PS }
vars().update (Psus); del Psus
```

Sintaksna analiza ulaznih formula realizirana je uz pomoć prepoznavača jezika sa svojstvima koji je upravljan tablicom prijelaza i akcija, [Dov2013].

Ukratko, tablica Tpa sadrži stanja (od 1 do 3) i prijelaze, znakove iz stringa 'Ssb()' u kojem 'S' označuje veliko slovo, 's' malo slovo, 'b' broj i 'c' ostale znakove koji se smiju pojaviti u formuli. Početno je stanje jednako 0. Ovisno o tekućem stanju i prijelazu prelazi se u naredno stanje i zvršava akcija definirana u tablici akcija, mapi Ta.

```
Tpa = {
  (0,'S'): (1, 1), (0,'('): (0, 4),
  (1,'S'): (1, 2), (1,'s'): (2, 1),
  (1,'b'): (3, 3), (1,'('): (0, 4),
  (1,')'): (1, 5),
  (2,'S'): (1, 2), (2,'('): (0, 4),
  (3,'b'): (3, 1), (3,'('): (0, 4),
  (3,')'): (1, 5), (3,'S'): (1, 2) }
Ta = {
  1: "Exp += c",      2: "Exp += '+' +c",
  3: "Exp += '*' +c",
  4: "Exp += '+' +c; b += 1",
  5: "Exp += c; b -= 1; Error = b < 0" }
while 'formula' :
  print()
  Spoj = input('Upiši kemijsku formulu ')
  if not Spoj : break
  q = b = 0; Exp = ''; Error = False
  for c in Spoj :
    if Error : break
    x = ('S' if c.isupper() else
         's' if c.islower() else
         'b' if c.isdigit() else c )
    if c == '+' : Exp += c; continue
    y = (q, x)
    if y in Tpa :
      q, a = Tpa[y]; exec (Ta[a])
    else        : Error = True; break
  Error = not Error and b > 0
  if not Error :
    try             :
      print(round(eval(Exp),3))
    except NameError:
      print('Nepoznat kem. el.')
    except SyntaxError:
      print ('Sintaksna pogreška')
  else          :
    print ('Sintaksa, zagrade')
```

Evo relativnih molekularnih masa glukoze C6H12O6 i amonijevoga željeznog sulfata FeSO4+(NH4)2SO4:

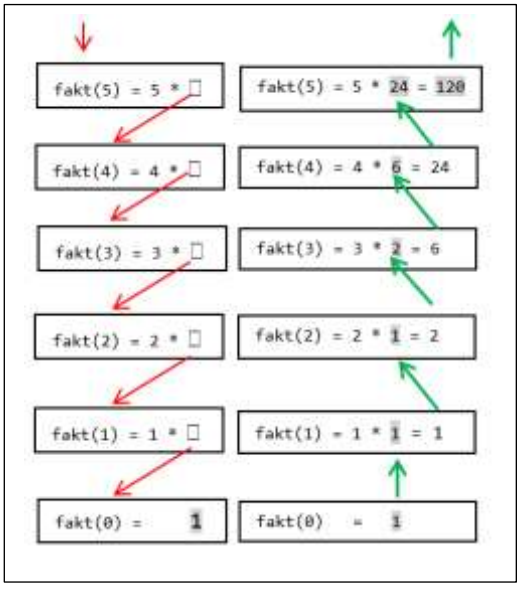
```
>>>
Upiši kem. formulu C6H12O6
180.156
Upiši kem. formulu FeSO4+(NH4)2SO4
284.074
```

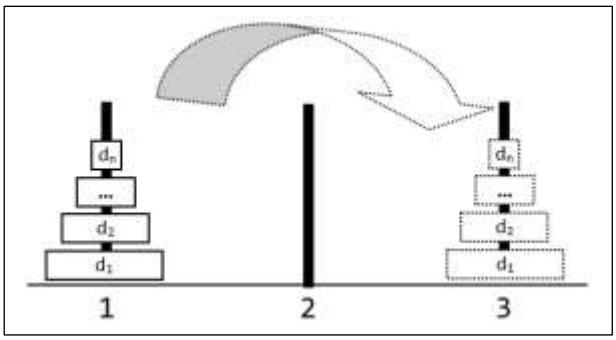
10.

POTPROGRAMI

Potprogrami su složene naredbe koje grupiraju niz naredbi i omogućuju njihovo „pozivanje" i izvršavanje iz nekoga drugoga mjesta programa, bez ograničenja broja pozivanja. U mnogim se jezicima za programiranje potprogrami dijele na procedure i funkcijske potprograme. U Pythonu imamo potprograme, ili funkcije, kako ih se najčešće naziva u dokumentaciji Pythona i literaturi. Ipak, ovisno o strukturi potprograma, mi ćemo razlikovati procedure i funkcijske potprograme.

U ovom smo poglavlju detaljno opisali značenje potprograma u programiranju. Dio GOVORIMO PYTHONSKI sadrži dosta analiza i preporuka, posebno o uporabi rekurzija. Dio PROGRAMI sadrži rješenja nekih problema s maksimalnom primjenom potprograma.





ENG_plural.py

```
# Množina engleskih imenica
def plural ( w ) :
    eq  = ['sheep', 'fish', 'deer', 'species',
      'aircraft', 'software', 'information']
    ex  = { 'person': 'people', 'datum': 'data',
      'mouse' : 'mice'  , 'bus'  : 'buses',
      'child' : 'children', 'food' : 'foods' }
    E = { 'y': 'ies', 'f': 'ves', 'fe': 'ves',
      'us': 'i', 'is': 'es', 'an': 'en',
      'on': 'a' }
    w1, w2 = w[-1], w[-2:]
    pl = lambda w : (
       ex[w]          if w in ex            else
       w              if w in eq            else
       w[:-2] +E[w2] if w2 in E            else
       w[:-1] +E[w1] if w1 in E            else
       w +'es'        if w1 in 'sxo' or
                         w2 in ['sh', 'ch'] else
       w.replace ('oo', 'ee')
                      if 'oo' in w else
       w +'s'                                )
    return pl ( w )
```

Uvod

Prije nego što prijeđemo na opis potprograma, neka nam rješenje sljedećega zadatka posluži kao dovoljan motiv za njihovu uporabu:

Zadatak 10.1

Izračunati razliku (broj dana) dvaju datuma iste godine.

Odmah se nameće sljedeće rješenje:

Raz_dat_1.py

```
# Razlika dvaju datuma iste godine.
from Moj_modul import *
while True : # Učitavam godinu
  G = Input ('Upišite godinu >1582 ')
  if Int(G) and G > 1582 : break
M = [0, 31, 28+PG(G), 31, 30, 31,
       30, 31, 31, 30, 31, 30, 31]
while 1 : # Učitavam prvi datum
  d1, m1 = Input( 'Prvi datum,
                   dan i mjesec ')
  if (Int(d1) and Int(m1) and
      1 <= m1 <= 12 and
      1 <= d1 <= M[m1]) : break
while 2 : # Učitavam drugi datum
  d2, m2 = Input( 'Drugi datum,
                   dan i mjesec ')
  if (Int(d2) and Int(m2) and
      1 <= m2 <= 12 and
      1 <= d2 <= M[m2]) : break
# redni brojevi dana prvog i drugog
# datuma
d1 += sum( M[:m1] )
d2 += sum( M[:m2] )
d   = abs( d1-d2 )
print ('Razlika je ', d, ' dan' +'a'
       if d%10 != 1 or d == 11 else '')
```

```
>>>
Upišite godinu >1582 2020
Prvi datum, dan i mjesec 1, 1
Drugi datum, dan i mjesec 31, 12
Razlika je  365  dana
>>>
Upišite godinu >1582 2019
Prvi datum, dan i mjesec 1, 1
Drugi datum, dan i mjesec 31, 12
Razlika je  364  dana
>>>
Upišite godinu >1582 2020
Prvi datum, dan i mjesec 1, 1
Drugi datum, dan i mjesec 1, 2
Razlika je  31  dana
```

Dekompozicijom programa uočit ćemo da je struktura rješenja postavljenoga problema:

1) Unos godine i ispravak broja dana drugoga mjeseca (za pr. god.)
2) Unos i provjera prvoga datuma
3) Unos i provjera drugoga datuma
4) Izračunavanje rednoga broja prvoga datuma
5) Izračunavanje rednoga broja drugoga datuma
6) Izračunavanje i ispis razlike

Dakle, program se sastoji od dva istovjetna dijela za unos i provjeru datuma i dva, također istovjetna, dijela za izračunavanje rednoga broja datuma:

```
while 1 : # Unos datuma
  d, m = Input (
         s +' datum, dan i mjesec ')
  if (Int(d) and Int(m) and
     1 <= m <= 12 and 1 <= d <= M[m]) :
     break

d += sum (M[:m])  # Izračunavanje
#                   rednog broja datuma
```

samo što su imena varijabli `d` i `m` zamijenjena sa `d1` i `m1`, odnosno `d2` i `m2`, i `s` je string koji za unos prvoga datuma ima vrijednost `'Prvi'`, a za drugi datum `'Drugi'`.

Prvi dio programa, unos i provjera datuma, predstavlja poseban dio programa - proceduru, koja prihvaća dvije ulazne vrijednosti, provjerava ih i, ako su korektne, vraća ih kao izlazne vrijednosti. Drugi dio također je procedura, ali, s obzirom da za dvije ulazne vrijednosti izračunava jednu - izlaznu, predstavlja posebnu vrstu potprograma - funkcijski potprogram.

Python, kao i većina jezika za programiranje, omogućuje izdvajanje takvih dijelova programa i njihovo definiranje kao posebnih cjelina: procedura i funkcijskih potprograma. Na primjer, dajemo rješenje postavljenoga zadatka primjenom potprograma:

Raz_dat_2.py

```
# Razlika dvaju datuma u istoj godini
from Moj_modul import *
M = [ 0, 31, 28, 31, 30, 31, 30,
         31, 31, 30, 31, 30, 31 ]
def datum (s = '') :
```

```
    while 1 :
      d, m = Input(
             s +' datum, dan i mjesec '
            +str(G)+'. god. ' )
      if (Int(d) and Int(m) and
          1 <= m <= 12      and
          1 <= d <= M[m]) : return d, m
def dan (d, m) : return sum( M[:m] ) +d
while True : # Učitavam godinu
  G = Input( 'Upišite godinu >1582 ' )
  if Int(G) and G > 1582 :
     M[2] += PG(G);  break
d1, m1 = datum ('Prvi');
d2, m2 = datum ('Drugi')
d      = abs ( dan (d1, m1)
               -dan (d2, m2) )
print ('Razlika je ', d, 'dan'
     +'a'*(d %100 == 11 or d %10 != 1))
```

```
>>>
Upišite godinu >1582 2020
Prvi datum, dan i mjesec 2020. god. 1, 1
Drugi datum, dan i mjesec 2020. god. 1,
2
Razlika je 31 dan
```

Funkcija `datum` odgovara dijelu za unos i provjeru datuma, a funkcija `dan` dijelu za izračunavanje rednoga broja datuma.

Uspoređujući programe **Raz_dat_1** i **Raz_dat_2** zaključilo bi se da je ovaj drugi pregledniji, bliži strukturi rješenja postavljenoga problema. Ne samo da su izbjegnuta nepotrebna ponavljanja i reduciran tekst programa, već su izdvojene cjeline koje predstavljaju rješenje pojedinih potproblema postavljenoga problema. Takve cjeline mogu se upotrijebiti kasnije, u rješavanju drugih problema. To nas podsjeća na standardiziranje dijelova i sklopova, na primjer u elektronici ili strojarstvu.

Primjena potprograma nije uvijek vezana za dijelove koji se ponavljaju, kao što je to bio slučaj u našem problemu, već općenito za dekomponiranje kompleksnih problema i postupno rješavanje.

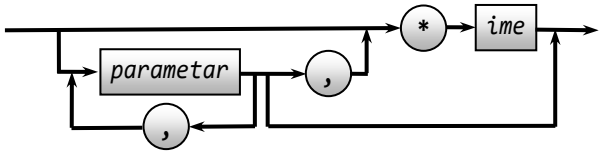
Procedure i funkcije

Prema osnovnoj sintaksnoj strukturi Pythona *program* je bio definiran kao *blok*. Dalje je *blok* bio definiran skupom primitivnih i složenih naredbi. Dosad opisane naredbe koristile su dio toga pravila. Bile su to primitivne i složene naredbe koje su se nizale jedna za drugom i činile "glavni program".

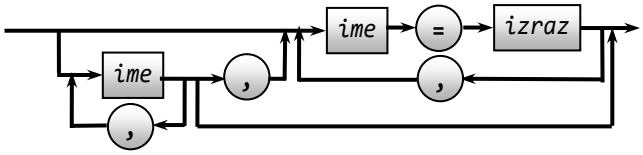
Karakteristika je tih naredbi, osim *LAMBDA funkcija*, da su bile izvršne. Ni potprogram, kojeg sada uvodimo, nije izvršni dio glavnog programa. Definiranje potprograma dano je sljedećim pravilima:

potprogram:
  `def` *ime_def* ([*parametri* | ***ime*]) :
    naredbe

ime_def : *ime*
parametri:

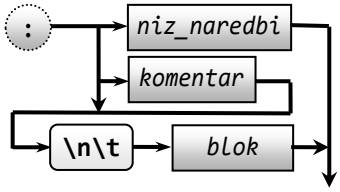


parametar:

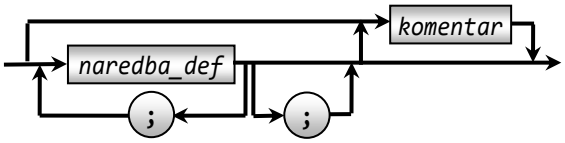


Sintaksnu kategoriju *naredbe* definirali smo ranije kao

naredbe:



niz_naredbi:



naredba_def : *naredba*| *naredba_RETURN*
naredba_RETURN :
  `return` [*izraz* {, *izraz*}]

iz čega zaključujemo da tijelo potprograma ima jednaku strukuturu kao naredbe unutar bilo koje složene naredbe (selekcije, *WHILE petlje*, iteracije itd.), uz dodatak *naredbe RETURN*.

SEMANTIKA

Potprogram koji ne sadrži *naredbu RETURN* ili je sadrži, ali bez izlaznih vrijednosti, nazivat ćemo *procedura*. Inače je *funkcijski potprogram* ili *funkcija*. Da bismo što bolje opisali semantiku potprograma, vratimo se uvodnom primjeru, programu **Raz_dat_2**, i zamijenimo dio programa

```
while True : # Učitavam godinu
  G = Input ('Upišite godinu >1582 ')
  if Int(G) and G > 1582 : M[2] += PG(G);
break
```

sa

```
def Unos () : # 1
  global G
  while True : # Učitavam godinu
    G = Input ('Upišite godinu >1582 ')
    if ok (G) :  M[2] += PG(G);  break
def god (g) : return Int(g) and g > 1582
Unos()
```

Pogledajmo naredbe koje čine potprograme datum i dan.

```
def datum (s = '') : # 2
   while 1 :
     d, m = Input (s +' datum, dan i '
           'mjesec ' +str(G)+'. god. ')
     if (Int(d) and Int(m) and
         1 <= m <= 12 and
         1 <= d <= M[m] : return d, m

def dan (d, m) : return sum (M[:m]) +d
```

PARAMETRI

Iz sintakse pisanja formalnih parametara potprograma imamo sljedeće slučajeve:

()	nema parametara
ime	rječnik parametara
ime	(normalna) ulazna varijabla, bilo kojeg tipa
ime=izraz	ulazna varijabla s inicijalno pridruženom vrijednošću
**ime*	niz proizvoljnog broja parametara *k*, *k*≥0.

Kontekstni aspekti pisanja formalnih parametara (v. sintaksu) su sljedeća:

1) Prvo se pišu normalni parametri (ako ih ima)
2) Potom se pišu varijable s inicijalno pridruženim vrijednostima
3) Parametar s proizvoljnim brojem imena piše se na kraju

Evo nekoliko primjera pravilno napisanih definicija potprograma:

```
>>> def A ()           : pass
>>> def B (x, y)       : pass
>>> def C (x=0, y=0)   : pass
>>> def D (x, y=1, *z) : pass
>>> def E (*a)         : pass
```

Ovdje smo rabili *naredbu PASS* koja ne čini ništa, a dovoljna je za potpuno definiranje potprograma. U sljedećem primjeru definicija potprograma X nije napisana prema pravilima:

```
>>> def X (x, y=1, z) : pass
SyntaxError:    non-default    argument
follows default argument
```

jer poslije predefiniranog parametra, y, ne može se pisati nepredefinirani parametar.

POZIVANJE POTPROGRAMA

Potprogram (funkcija) se poziva prema pravilu:

poziv_fun :
  ime_def ([*argument* {, *argument* }])
argument : [*ime* =]*izraz*

Potprogrami bez parametara

Ako potprogram nema parametara, poziva se bez argumenata.

```
>>> def A () : print (1234)
>>> A ()       1234
```

Fiksni broj parametara

U definiciji potprograma je jedan ili više parametara. Poziv potprograma mora sadržati jednak broj argumenata. Parametrima će biti redom pridružene vrijednosti argumenata:

```
>>> def xy (x, y) : print (x *y**2)
>>> xy (1, 2)                      4
>>> xy (2, 1)                      2
>>> xy (10, 2)                    40
```

Poziv može sadržati izraze s imenima parametara i pridruženim vrijednostima. Tada redosljed nije bitan:

```
>>> xy (y=10, x=2)                200
```

Predefinirani parametri

Mogu biti izostavljeni u pozivu potprograma. Tada ostaje predefinirana vrijednost. Inače, ako je navedeno u izrazu poziva, bit će s novom vrijednošću.

Varijabilni broj parametara

Sada ćemo predstaviti potprograme koji mogu uzeti proizvoljan broj argumenata. Zvjezdica „*" se koristi za definiranje promjenjivog broja argumenata. Znak

zvjezdice mora prethoditi identifikatoru varijable na popisu parametara.

```
>>> def X ( * x ) : print ( x ); print
(list(x))
>>> X ('srijeda', 19, 5, 2021, 'kiša!')
('srijeda', 19, 5, 2021, 'kiša!')
['srijeda', 19, 5, 2021, 'kiša!']
```

Iz prethodnog primjera doznajemo da se argumenti prosljeđeni pozivu potprograma X() prikupljaju u skup, kojem se može pristupiti kao „normalnoj" varijabli x unutar tijela potprograma. Ako se potprogram poziva bez ikakvih argumenata, vrijednost x je prazna *n*-torka. Ponekad je potrebno koristiti fiksne parametre praćene proizvoljnim brojem parametara u definiciji potprograma. To je moguće, ali fiksni parametri uvijek moraju prethoditi proizvoljnim parametrima (v. sintaksu). U sljedećem primjeru imamo pozicijski parametar „x", koji se uvijek mora navesti, nakon čega slijedi proizvoljan broj parametara:

```
>>> def Y (x, *y): print (x, y)
>>> Y ()
...
TypeError: Y() missing 1 required
positional argument: 'x'
>>> Y ('Zagreb')
Zagreb ()
>>> Y ('Zagreb', 'Osijek', 'Rijeka',
'Split')
Zagreb ('Osijek', 'Rijeka', 'Split')
>>>
```

„*" u pozivu potprograma

Može se pisati zvjezdica u pozivima potprograma, kao što smo upravo vidjeli u prethodnoj primjeru: semantika je u ovom slučaju „inverzna" zvjezdici u definiciji potprograma. Elementi liste ili *n*-torke će biti upareni s parametrima zaglavlja potprograma. Primjer:

```
>>> def f (x, y, z) : print (x, y, z)
>>> Y = ('jedan', 'un', 'one'); Z = [1,
2, 5]
>>> f ( *Y ); f ( * Z )
jedan un one
1 2 5
```

Ne treba spomenuti da je ovaj način pozivanja našeg potprograma ugodniji od:

```
>>> f (Y[0], Y[1], Y[2])
jedan un one
```

Kontekstni aspekt poziva potprograma na ovaj način je da duljina niza kao argumenta mora biti jednaka broju parametara.

Proizvoljna imena kao parametri

Postoji i mehanizam za proizvoljan broj parametara ključne riječi. Da bismo to učinili, koristimo oznaku dvostruke zvjezdice „**":

```
>>> def X (**x): print (x)
>>> X ()
{}
>>> X (de = 'njemački',  en =
                          'engleski',
       fr = 'francuski', hr =
                          'hrvatski')
{'de': 'njemački', 'en': 'engleski',
 'fr': 'francuski', 'hr': 'hrvatski'}
>>>
```

Evo još značenja proizvoljnih imena kao parametara:

```
>>> def Y (**y) :
    print ("Pjeva " +y['pjevač'])
>>> Y (pjevač = 'Brassens')
Pjeva Brassens
```

```
>>> def X (**x) :
    print ( x['a']**2 +x['b']**2 )
>>> X (a = 3, b = 4)                  25
```

Dvostruka zvjezdica u pozivu potprograma

U sljedećem primjeru je demonstrirana upotreba ** u pozivu potprograma:

```
>>> def f (a, b, x, y) :
    print (a, b, x, y)
>>> Y = {'a' : 'A', 'b' : (1, 2),
        'x' : 'X', 'y' : (-1, 3)}
>>> f ( **Y )          A (1, 2) X (-1, 3)
```

ili, u kombinaciji s „*":

```
>>> A = ('A', (1,2))
>>> X = {'x' : 'X', 'y' : (-1, 3)}
>>> f ( *A, **X )     A (1, 2) X (-1, 3)
>>> def B (x, y=1, z) : print ( x*y*z )
SyntaxError: non-default argument follows
default argument
>>> def B (x, y=1, *z) : print ( x*y*z)
>>> B (2, 2, 10)  (10, 10, 10, 10)
>>> def m (*x) : return min (x)
>>> m (1, -1, 2, -2)                  -2
```

Izvršenje programa uvijek počinje naredbama glavnoga programa. Procedure i funkcijski potprogrami složene su naredbe koje će biti izvršene samo ako su „pozvane" iz glavnoga programa, odnosno nekoga drugoga potprograma. Poziv procedure je primitivna naredba, a poziv funkcije je dio izraza.

Kontekstni su aspekti pisanja aktualnih argumenata:

1) Redoslijed aktualnih argumenata mora odgovarati redoslijedu formalnih parametara navedenih u zaglavlju definicije potprograma.
2) Ako je formalni argument deklariran kao ulazno-izlazni, aktualni argument mora biti ime varijable.
3) Izrazi kao aktualni argumenti mogu stajati samo na mjestima formalnih argumenata koji su deklarirani kao ulazni.

IZLAZAK IZ POTPROGRAMA

"Normalni" će izlazak iz potprograma biti u slučaju izvršenja niza naredbi unutar potprograma i dosezanja kraja bloka. Osim toga, imamo *naredbu RETURN* koja vraća niz vrijednosti (izraza) bilo kojeg tipa:

```
return [ izraz {, izraz } ]
```

Naredba RETURN može biti napisana u bilo kojem dijelu bloka potprograma. Ako je napisana na kraju bloka bez izraza, nema posebno značenje i može biti izostavljena. Potprogram koji ne sadrži *naredbu RETURN* ili sadrži jednu ili više *naredbi RETURN* bez izraza je procedura, a ako sadrži jednu ili više *naredbi RETURN* s izrazima, onda je funkcija. Evo primjera logičke funkcije koja vraća vrijednost `True` ako je argument prost broj, `False` ako nije:

```
# prim.py
def prim ( n ):
  for i in range(2, int(n **0.5) +1) :
    if n % i == 0 : return False
  return True
```

10.1 Definiranje i pozivanje potprograma

```
>>> def A (x, y) :
    print( 'x =', x, ' y =', y )
>>> A (1, 2)                    x = 1  y = 2
>>> A (20, 1.56)              x = 20  y = 1.56
>>> A ('prvi', 2.)          x = prvi  y = 2.0
>>> a = 99; b = -1.5; A (a, b)
x = 99  y = -1.5
>>> def B (x, y = 0) :
    print( 'x =', x, ' y =', y )
>>> B (20)                      x = 20  y = 0
>>> B (0, 15)                   x = 0  y = 15
>>> B (1, 'abc')              x = 1  y = abc
>>> def C (x = 0, y = 0) :
    print( 'x =', x, ' y =', y )
>>> C ()                        x = 0  y = 0
>>> C (5)                       x = 5  y = 0
>>> C ('jedan', 'dva')
x = jedan  y = dva
>>> C (y=30)                    x = 0  y = 30
>>> def D (x, *y) :
    print( 'x =', x, ' y =', y )
>>> D (12, 13)              x = 12  y = (13,)
>>> D (11)                    x = 11  y = ()
>>> D ('n-torka', 1, 2, 3, 4)
x = n-torka  y = (1, 2, 3, 4)
>>> def E (*x, y) :
    print( 'x =', x, ' y =', y )
>>> E (1)
...
TypeError: E() missing 1 required
keyword-only argument: 'y'
>>> E (1, 2)
...
TypeError: E() missing 1 required
keyword-only argument: 'y'
```

Standardne funkcije `min()` i `max()` primjeri su funkcija s proizvoljnim brojem argumenata.

```
>>> def m (*a) : print ( min (*a) )
>>> m (1, 2, -3)                    -3
```

Izvršenje programa uvijek počinje naredbama glavnoga programa. Procedure i funkcijski potprogrami složene su naredbe koje će biti izvršene samo ako su „pozvane" iz glavnoga programa, odnosno nekoga drugoga potprograma. Poziv procedure je primitivna naredba. U osnovnoj sintaksnoj strukturi to je ***poziv_procedure*** i dio je sintaksne kategorije ***naredba***. Kontekstni su aspekti pisanja aktualnih argumenata sljedeći:

1) Redoslijed aktualnih argumenata mora odgovarati redoslijedu formalnih parametara navedenih u zaglavlju definicije potprograma.
2) Na mjestu formalnog argumenta s proizvoljnim brojem ulaznih argumenata možemo izostaviti argument, napisati jedan ili više argumenata.

10.2 Pozivanje potprograma

```
>>> def min_max (*a) :
    return min (*a), max (*a)
```

```
>>> m, M = min_max ( {1, 10, -5, 32} )
>>> m, M                (-5, 32)
>>> print( min_max ( {1, 10, -5, 32} ) )
(-5, 32)
>>> print(
    min_max (100, 3.42, 15.25, 101) )
(3.42, 101)
>>> def test (*x) : return x
>>> test (1,2)          (1, 2)
>>> test ([1,2])        ([1, 2],)
>>> test ('abc')        ('abc',)
>>> def test2 (x, *args) :
    print ( x, args )
>>> test2 (1, 2)          1 (2,)
>>> def test3 (*args, y) :
    print( args, y )
SyntaxError: invalid syntax
>>> def test4 (x, y, *args) :
    print( x, y, args )
>>> test4 (1, 2)                      1 2 ()
>>> test4 (1)
TypeError:  test4()  takes  at  least  2
arguments (1 given)
```

DOSEG DEFINIRANOSTI IMENA

Imena upućuju (pokazuju ili referiraju) na objekt u memoriji. To znači da se imenu pridružuje identifikacija objekta (instance). Na primjer,

```
>>> type (print)
<class 'builtin_function_or_method'>
>>> id (print)              2128164454264
>>> Ispis = print
>>> id (Ispis)              2128164454264
```

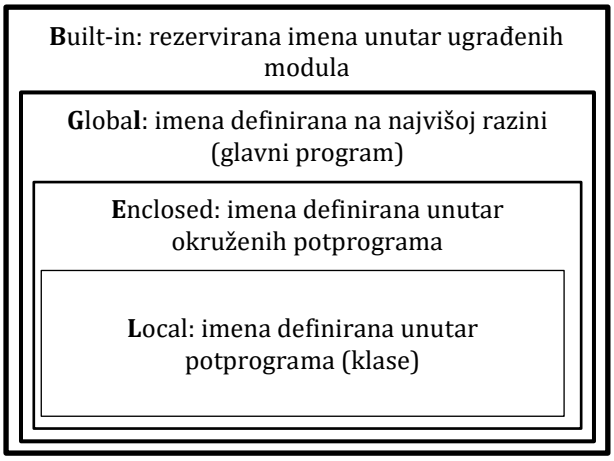
Za razliku od drugih jezika, poput C, C ++, JAVA i Pascal, u kojima su imena varijabli statična, tj. deklariraju se i definira tip prije svoje uporabe, u Pythonu se uvode svojim prvim pridruživanjem objekta (instance neke klase) na kojeg će se referirati.

Doseg je djelokrug definiranja, hijerarhijski redoslijed u kojem se moraju pretraživati prostori imena kako bi se dobila preslikavanja imena u objekt (varijable). To je kontekst u kojem postoje varijable s određenim sadržajem i imaju svoj vijek trajanja.

LEGB pravilo

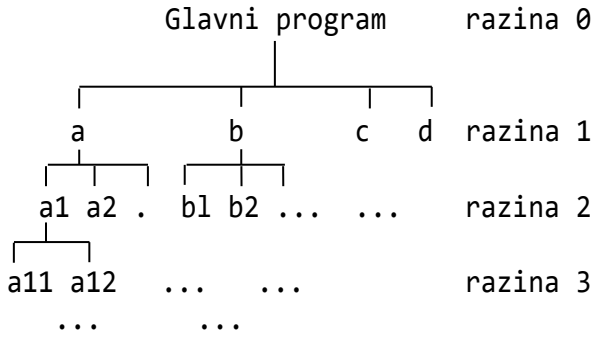
LEGB pravilo koristi se za određivanje redoslijeda kojim će se pretraživati prostori definiranosti pojedinih imena radi razlučivosti dosega. Dosezi su dolje navedeni u smislu hijerarhije (od najviše do najniže / od najuže do najšire):

- Local (L) : definirano unutar funkcije / klase
- Enclosed (E) : definirano unutar zatvorenih funkcija (koncept ugniježđene funkcije), okružen (ograđen)
- Global (G) : definirano na najvišoj razini
- Bild-in (B) : rezervirana imena u ugrađenim modulima



KONTEKSTNI ASPEKTI POZIVANJA POTPROGRAMA

Hijerarhijskom strukturom programa definirani su tekstualni dosezi potprograma, a istodobno i kontekstni aspekti ili pravila pozivanja potprograma. Ako strukturu programa općenito predočimo stablom:



gdje čvorovi predstavljaju potprograme, pravila su pozivanja sljedeća:

1) Iz glavnoga programa moguće je pozivati samo potprograme razine 1.

2) Iz potprograma $X_{i,j}$, na razini `r`, moguće je pozvati sve potprograme razine `r+1` definirane unutar potprograma $X_{i,j}$, potprograme $X_{i,1}$ do $X_{i,j-1}$ (definirane na razini `r` unutar istoga potprograma X_i), potprograme razine `r-1`, `r-2`, ... , `p` i prethodno definirane potprograme na istoj razini u podstablima unutar kojih je definiran potprogram $X_{i,j}$.

```
# Pozivanje_potprograma.py
0.
""" naredbe glavnog programa """
def A () :
    1.
    print( 'A ' )
    def A_1 () :
        2.
        print( 'A_1' )
    def A_2 () :
        2.
        print( 'A_2' )
        def A_2_1 () :
            3.
            print( 'A_2_1' )
        A_2_1 ()
    A_1 ();  A_2 ()

A ()

""" naredbe glavnog programa """
```

```
>>>
A
A_1
A_2
A_2_1

>>> A ()
A
A_1
A_2
A_2_1

>>> A_1()
...
NameError: name 'A_1' is not defined
>>>
```

GLOBALNE I LOKALNE VARIJABLE

Neka je definirana procedura P:

```
def P (): print( s )
```

Pogledajmo što će biti ispisano poslije definiranja stringa s i poziva procedure P():

```
s = "Ines voli mačke.";  P ()
Ines voli mačke.
```

print(s) jedina je naredba unutar procedure P(). S obzirom na to da varijabla s nije definirana unutar procedure, bit će rabljena globalna varijabla **s**, pa je ispisan string "Ines voli mačke.". Pitanje je što će se dogoditi ako promijenimo vrijednost varijable **s** unutar procedure P(). Na primjer,

```
def P ():
    s = "Ja također!"
    print (s)
s = "Ines voli mačke."; P (); print (s)

Ja također!
Ines voli mačke.
```

Prvo je varijabla **s** bila definirana unutar procedure i ispisan njezin sadržaj. Vidimo da je to različito od sadržaja varijable **s** poslije povratka iz procedure. Varijabla s unutar procedure je lokalna za proceduru P(). Njezino značenje vrijedi samo unutar procedure. Značenje varijable **s** izvan procedure je globalno i nije promijenjeno.

Na primjer, analizirajmo dosege varijabli x i y u sljedećoj skripti:

```
# Primjer_10_1.py
def A () :
  def B () :
    print ('-> B, x =', x)
    print ('-> B, y =', y)
  print ('-> A, x =', x)
  y = 222;  B()

x = 111
print ('gl. program, x =', x)
A ()
```

```
>>>
gl. program, x = 111
-> A, x = 111
-> B, x = 111
-> B, y = 222
```

Rezervirana riječ global

Pravilo pisanja naredbe GLOBAL dano je sa

```
global ime { , ime }
```

Značenje je deklaracije global da vrijedi za cijeli trenutni blok koda. To znači da se navedena imena trebaju

tumačiti kao globalna. Imena navedena u naredbi GLOBAL ne smiju se definirati kao formalni parametri ili kao kontrolne varijable FOR petlje, definicije klase, definicija potprograma ili ime modula.

Analizirajmo dosege nekoliko varijabli u sljedećoj skripti, u kojoj imamo neka imena deklarirana kao globalna:

```
# Primjer_10_2.py
def A () :
  global X
  print ('-> A, X =', X)
  if X < 12 : B ()
def B () :
    global X;  print ('-> B')
    X += 1;  A ()
X = 10;  A()
def C () :
  print ('-> C, x =', x)
  print ('      M =', M)
  print ('      N =', N)
  # x += 10
  M.append (30);  M.sort()
  N[1] = 31
  # M = 1234
x = 1;  M = [0, 31, 29, 31];  N = {}
print ('gl. program, x =', x)
C()
print ('gl. program, x =', x)
print ('             M =', M)
print ('             M =', M)
def D () :
  global y
  def E () :
    global y
    print ('-> E, y =', y)
    y *= 5
  print ('-> D, y =', y)
  y += 10;  E()
y = 10
print ('gl. program, y =', y)
D ();  print ('gl. program, y =', y)

def F () :
  global z
  z = 125
F ();  print ('z =', z)
>>>
-> A, X = 10
-> B
-> A, X = 11
-> B
-> A, X = 12
gl. program, x = 1
-> C, x = 1
      M = [0, 31, 29, 31]
      N = {}
gl. program, x = 1
             M = [0, 29, 30, 31, 31]
             M = [0, 29, 30, 31, 31]
gl. program, y = 10
-> D, y = 10
-> E, y = 20
gl. program, y = 100
```

LOKALNI, ZATVORENI, UGRAĐENI I GLOBALNI DOSEG

Lokalni doseg odnosi se na varijable definirane u trenutnom potprogramu. Uvijek će potprogram prvo tražiti ime varijable u svom lokalnom dosegu. Samo ako ga tamo ne pronađe, provjeravaju se vanjski dosezi. Pogledajmo dva segmenta programa:

```
#1
S = 'globalna S varijabla'
def Unutarnja ():
    print (S)
Unutarnja()

globalna S varijabla

#2
S = 'globalna S varijabla'
def Unutarnja ():
    S = 'lokalna S varijabla'
    print (S)
Unutarnja()

lokalna S varijabla
```

U segmentu #1 procedura `Unutarnja()` ispisala je sadržaj varijable `S` iz globalnog dosega, a u segmentu #2 iz lokalnog dosega.

Za zatvoreni doseg moramo definirati vanjski potprogram koji zatvara unutarnji potprogram, komentirati lokalnu varijablu `S` unutarnjeg potprograma i uputiti na `S` pomoću `nonlocal` rezervirane riječi. Drugim riječima, rezervirana riječ `nonlocal` koristi se u slučaju ugniježđenih potprograma. Djeluje slično globalnoj, ali umjesto globalne, ova rezervirana riječ deklarira varijablu koja ukazuje na varijablu vanjskog potprograma koju zatvara, u slučaju ugniježđenih potprograma. Pogledajmo primjer:

```
# zatvoreni_doseg.py
S = 'globalna S varijabla'
def Vanjsko ():
    S = 'vanjska S varijabla'
```

```
    def Unutarnje ():
        # S = 'unutarnja S varijabla'
        nonlocal S
        print (S)
    Unutarnje()
Vanjsko ()
print   (S)
```

```
vanjska S varijabla
globalna S varijabla
```

Kada se izvrši `Vanjsko()`, izvršava se `Unutarnje()` i shodno tome funkcije ispisa koje ispisuju vrijednost priložene varijable S. Ispis traži varijablu u lokalnom dosegu unutarnjeg potprograma, ali je tamo ne nalazi. Budući da se S deklarira u **nonlocal** rezerviranoj riječi, to znači da se varijabli S treba pristupiti iz vanjske funkcije (tj. vanjskog dosega).

Da rezimiramo, varijabla S nije pronađena u lokalnom dosegu, pa se traže širi dosezi. Nalazi se i u zatvorenom i u globalnom dosegu. No, prema LEGB hijerarhiji, uzeta je u obzir varijabla dosega iako je imamo definiranu u globalnom dosegu.

Konačna provjera može se izvršiti uvozom `pi` iz matematičkog modula i komentiranjem globalnih, zatvorenih i lokalnih `pi` varijabli kao što je prikazano u nastavku:

```
from math import pi
# pi = 'globalna pi varijabla'
def Vanjsko ():
    # pi = 'vanjska pi varijabla'
    def Unutarnje ():
        # pi = 'unutarnja pi varijabla'
        print (pi)
    Unutarnje ()
Vanjsko ()
3.141592653589793
```

Ovdje `pi` nije definirano kao lokalna, zatvorena niti globalna, već kao ugrađena varijabla i ima doseg u cijeloj skripti. Doseg u cijeloj skripti imaju liste i mape. Ponašaju se kao globalne varijable. Na primjer:

```
>>> L = [1, 2, 3]
>>> def X () : print ( *L ); L [0] = 10
>>> X ()                                1 2 3
>>> L                             [10, 2, 3]

>>> D = {1: 10, 2: 20}
>>> def Y () :
    def Z () : D [2] = '+++'
    Z (); D[1] = '***'
>>> Y();  D          {1: '***', 2: '+++'}
```

Na primitivne ili složene varijable drugih tipova se može referirati (mogu biti u izrazima):

```
>>> a = 10
>>> def A () : print (a)
>>> A ()                                10
```

Međutim, ne smije im se mijenjati vrijednost. Tada postaju lokalne:

```
>>> def B () : print (a); a = 20
>>> B ()
...
UnboundLocalError: local variable 'a'
referenced before assignment
```

LAMBDA FUNKCIJA

Sada se vratimo *LAMBDA funkciji* i proširimo joj značenje: *LAMBDA funkcija* može se definirati kao funkcijski potprogram koji sadrži samo izraz u svojoj definiciji naredbe. Parametri *LAMBDA funkcije* se definiraju na jednak način kao i parametri potprograma. Pozivanje *LAMBDA funkcije* jednako je pozivu funkcijskog potprograma. Uvijek vraća rezultat izračunavanja izraza sadržanog u svojoj definiciji.

```
>>> F = lambda x, y = 0, *z : \
                x + y + max (z)
>>> F (1,20, 2,10,3)                    31
```

Poziv može biti sa '*' ili '**':

```
>>> f = lambda x, y, z : print (x, y, z)
>>> Y = ('jedan', 'un', 'one'); \
    Z = [1, 2, 5]
>>> f ( *Y ); f ( *Z )
jedan un one
1 2 5
>>> f = lambda a, b, x, y : \
        print (a, b, x, y)
>>> Y = {'a' : 'A', 'b' : (1,2),
         'x' : 'X', 'y' : (-1, 3)}
>>> f ( **Y )          A (1, 2) X (-1, 3)
```

Funkcije `eval()` i `exec()`

Sada možemo dati potpuno značenje funkcija `eval()`: i `exec()`. Sintaksa je:

```
eval ( izraz [, globalno [, lokalno]] )
```

Kao što smo definirali još u prvom poglavlju, *izraz* je string koji sadrži izraz bilo kojeg tipa, napisan bez sintaksnih pogrešaka. Argumenti globalno i lokalno su rječnici (mape) koje sadrže imena varijabli i njihove vrijednosti koje će biti pridružene imenima varijabli u izrazu.

```
>>> a = 10; eval ('a **2')          100
>>> eval ('a **2', {'a' : 15})      225
>>> def X (a, b) :
    global c; print (a, b);  c = 10
    print (eval ("a +b +c", {'c' : 30},
                {'a' : 10, 'b' : 20} ))
>>> X (1, 2)
1 2
60
>>> c       10
```

Kontekstni je aspekt pisanja globalnih i lokalnih varijabli da moraju biti predefinirane sve varijable u izrazu. U suprotnom se dojavljuje pogreška:

```
...
NameError: name
```

I funkcija `exec()` ima šire značenje. Piše se prema pravilu:

```
exec ( naredbe [, globalno [, lokalno]] )
```

gdje *globalno* i *lokalno* imaju jednako značenje kao kod funkcije `eval()`.

```
>>> exec ( "for x in L: print (x)",
          { 'L' : [1, 2, 3] } )
1
2
3
>>> L
...
    L
NameError: name 'L' is not defined
```

GOVORIMO PYTHONSKI

PARAMETRI ILI ARGUMENTI?

U praksi se često umjesto parametar koristi i termin argument. Značenje je jednako: informacija koja će biti prenesena potprogramu. S perspektive značenja potprograma ipak ih možemo razdvojiti, pa će parametri biti imena navedena između zagrada u definiciji potprograma, a argumenti vrijednosti koje se prenose u potprogram prilikom njegovog poziva.

UPORABA POTPROGRAMA

Iz prethodnih razmatranja slijedi da ćemo potprograme koristiti u postupnom razvoju programa, grupiranju naredbi koje čine rješenje nekoga potproblema i koje mogu biti pozvane više puta, s različitim ulaznim vrijednostima.

Općenito ćemo funkcijske potprograme koristiti kao procedure kad ne postoje izlazne vrijednosti, kad treba izvršiti određenu akciju izvršavanjem niza naredbi i/ili kad su sve izlazne vrijednosti sadržane u globalnim varijablama.

Funkcijske ćemo potprograme koristiti za samo jednu izlaznu vrijednost. Na primjer, umjesto *LAMBDA funkcije* `Prikazi()`:

```
Prikazi = lambda : print ( T +S[ :4] +NLT
                + S[4:7] +NLT +S[8:11] )
```

u programu `Križić_kružić.py` može se definirati funkcijski potprogram:

```
def Prikazi (S) : return (T +S[ :4] +NLT
                          +S[4:7] +NLT +S[8:11] )
```

koji će biti pozvan sa:

```
print (Prikazi(S))
```

Ili, za izračunavanje *n*-tog člana Fibonaccijevog niza možemo definirati funkciju:

```
def fib (n): # n-ti član Fibonac. niza
    a, b = 1, 1
    for i in range (n): a, b = b, a + b
    return a
```

Osim toga, procedure ćemo koristiti za grupiranje nekoliko akcija koje će se moći aktivirati i izvršiti pozivom iz nekoga niza naredbi, a funkcijske potprograme u složenijem izračunavanju vrijednosti, kad se izlazna vrijednost dobiva izvršenjem nekoliko naredbi, ili ako su argumenti funkcije strukturiranoga tipa (npr. nizovi ili skupovi).

Na svim mjestima u sintaksnim strukturama gdje se pojavljivala funkcija može stajati poziv funkcijskog potprograma. To će biti u izrazima bilo kojeg tipa. Ili, na primjer, u funkciji `filter()` i `map()`. Također je važna autonomnost koda i mogućnost poboljšanja. Na primjer, umjesto funkcijskog potprograma

```
def dan (d, m) :
    for m in M[:m] : d += m
    return d
```

može se napisati jednostavniji kôd:

```
def dan (d, m) : return d +sum(M[:m])
```

Sada se treba vratiti na prethodna poglavlja, analizirati programe i prepoznati u njima cjeline koje mogu biti zamijenjene procedurom ili funkcijskim potprogramom. Možda ćemo neke funkcijske potprograme dodati našem radnom modulu **Moj_modul.py**. Na primjer, iz dijela programa **baza_2_do_16.py** za pretvorbu cijelog broja *n*, većeg ili jednokog nuli, u bazu 2 do 16, može se napisati funkcijski potprogram:

```
# N_u_b.py
# Pretvorba broja N bazu b (od 2 do 16)
def N_u_b (N, b) :
  Z = '0123456789abcdef'
  if not b in range (2, 17) :
     return False
  k = N;  Nb = ''
  while k > 0 :
     k, i = divmod (k, b); Nb = Z[i] +Nb
  return Nb
>>> N_u_b (12345, 7)              '50664'
>>> N_u_b (99999999999, 15)  '2904239969'
>>> N_u_b (12345, 20)              False
```

POTPROGRAMI I REKURZIJE

S pojmom rekurzije susreli smo se kad smo uveli *LAMBDA funkcije*. Znamo da je to svojstvo objekta da se definira i pomoću samoga sebe (ili, da sudjeluje u svojoj definiciji), kao na donjoj slici, [Wir1976].

Rekurzivna slika.

Kažemo da su funkcije i procedure koje pozivaju same sebe *rekurzivne*. To je *izravna rekurzija*. Rekurzija može biti i *neizravna*, a to će se dogoditi ako, na primjer, potprogram A poziva potprogram B, a potprogram B sadrži poziv potprograma A.

Definicija rekurzivne funkcije (procedure) sastoji se od dva dijela: (1) izlaza (ili "sidrišta") i (2) induktivnog koraka u kojem su vrijednosti funkcije (ili parametri) definirani uz pomoć prethodne vrijednosti.

Obično se kao „školski primjer" rekurzije navodi izračunavanje faktorijela cijelog broja *n*, $n \geq 0$:

$$n! = \begin{cases} 1 & \text{za } n=0 \\ 1 \cdot 2 \cdot \ldots \cdot n & \text{za } n>0 \end{cases}$$

Takvu smo funkciju za izračunavanje faktorijela broja *n* znali napisati kao *LAMBDA funkciju*,

```
>>> f = lambda n : (
        'nije definirano!' if n < 0
        else         1 if n in [0,1]
        else n*f(n-1)
```

Sada je možemo napisati i kao funkcijski potprogram:

```
def fakt (n)  :
  if   n < 0  : return ('nije '
                        'definirano!' )
  elif n == 0 : return 1
  else        : return n*fakt(n-1)
```

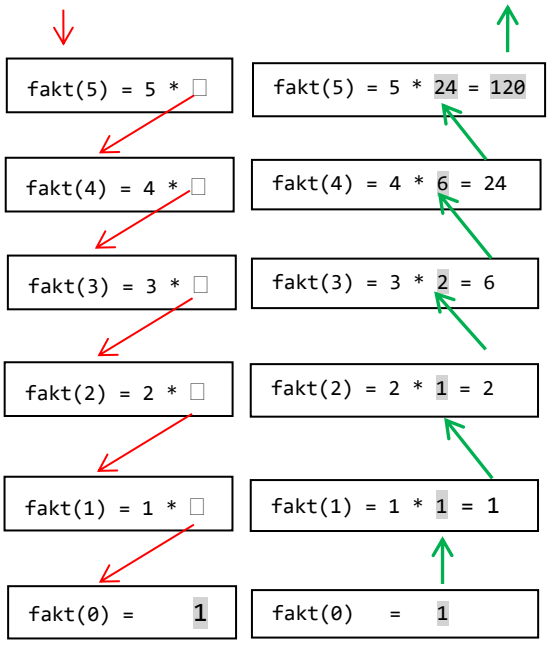
ili

```
>>> def fakt (n)    : return (
   'nije definirano!' if n < 0
    else               1 if n in [0,1]
    else     n*fakt(n-1) )
```

Pozivom tih funkcija za jednaki argument dobivamo jednak rezultat:

```
>>> f(30); fakt(30)
265252859812191058636308480000000
265252859812191058636308480000000
```

Kako se izračunavaju ove rekurzivne funkcije? Na primjer, za izračunavanje faktorijela broja 5 imali bismo:



Konačno se, s `fakt(0)`, rekurzija prekida i s vrijednošću `1` vraća se sve do prvog poziva i izračunavanja vrijednosti `fakt(5)`.

Kada rabiti rekurzije?

Postavlja se pitanje: Kada rabiti rekurzije? Odgovor na postavljeno pitanje mogao bi biti: onda kad definicija problema sadrži rekurzije, kao što je, na primjer, rekurzivna funkcija za izračunavanje faktorijela broja *n* iz prethodnog primjera, a takva je i funkcija $f(x)$ definirana po segmentima kao:

$$f(x)=\begin{cases} x+10 & ako\ je\ x \geq 100 \\ f(x+10) & ako\ je\ x < 100 \end{cases}$$

```
>>> def F (x) : return (
    x +10 if x >= 100 else F (x +10) )
>>> F(-12.5)  117.5 >>> F(7.5)    117.5
```

U sljedećem se primjeru mladi programer oduševio rekurzijama i primijenio ih je kao neizravne u svom programu:

```
# Tik_Tak.py
def Tak (n) :
  print ('tak', end = ' ')
  if n >= 0 : Tik (n-1)
def Tik (n) :
  print ('tik', end = ' ')
  if n >= 0 : Tak (n-1)
N = 20; Tik (N)
```

Program ispisuje `N` puta `tik tak`. Može li se napisati samo jedna nerekurzivna procedura koja će raditi isto? Može, u Pythonu je to jednostavno:

```
N = 20;  print ('tik tak ' *(N //2 +1))
```

Praksa pokazuje da programeri početnici ili bježe od rekurzija ili, kad misle da su je potpuno shvatili, primjenjuju je na svim mjestima gdje je moguće. Na primjer, netko bi smatrao da je problem nalaženja prim brojeva najbolje rješavati rekurzijom:

Prim_brojevi_2.py

```
# Generiranje prim brojeva rekurzivno
from Moj_modul import *
def prim (n) :
  P = []
  for i in range (2, n+1) :
    p = True
    for j in prim (int(i**0.5)) :
      p = p and (i == j or i % j)
      if not p : break
    if p : P.append (i)
  return P
while 'n < 2' :
  N = Input ('Zadaj cijeli broj > 1 ')
  if Int (N) and N > 1 : break
PB = prim (N);  print (* PB)
```

```
>>>
Zadaj cijeli broj > 1 100
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
53 59 61 67 71 73 79 83 89 97
```

Je li vam jasan postupak? Morat ćete priznati da je prilično zamršen. A ne radi baš efikasno, posebno za `n > 5000`.

S druge strane, ni probleme koji u svojoj definiciji sadrže rekurzije nije uvijek preporučljivo rješavati na taj način. Najbolji primjer za to jest izračunavanje `i`-tog člana Fibonaccijevog niza, označimo ga s a_i:

$$a_i = \begin{cases} 1 & za\ i=1,2 \\ a_{i-2} + a_{i-1} & za\ i>2 \end{cases}$$

iz čega se jednostavno napiše rekurzivna funkcija:

```
def fib (n) :
  if n < 3  : return 1
  else      : return fib(n-2) +fib(n-1)
```

Sve pet! Ali, kao što smo pokazali u 3. poglavlju, koristeći rekurzivnu *LAMBDA funkciju*, vrijeme izračunavanja znatno se povećava za `n>40`.

Rekurziju ćemo primjenjivati i u definiciji neke sintaksne kategorije. Na primjer, sintaksu cjelobrojnog izraza, *izraz*, na najvišoj razini najbolje je definirati rekurzivno:

```
izraz     : cjel_opr                | (0)
            predznak izraz          | (1)
            izraz operacija izraz   | (2)
            ( izraz )                 (3)
predznak  : { + | - }
cjel_opr  : cijeli_broj | cjel_var |
            cjel_fun
operacija : + | - | * | // | % | **
```

gdje su *cjel_var* i *cjel_fun* cjelobrojna varijabla i cjelobrojna funkcija. U (1), (2) i (3) imamo rekurzivne pozive. Izlaz je (0). Na primjer, primijenili smo to u programu koji generira jezik zagrada:

Zagrade.py

```
# JEZIK ZAGRADA R -> ()| (R)| R()| ()R
def gen_zag (L, D, R):
  # L - lijeva; D - desna; R - rečenica
  global W
  if L == D == 0 : W.append (R)
```

```
  if L >  0      : gen_zag (L -1, D +1,
                            R +'(')
  if D >  0      : gen_zag (L,    D -1,
                            R +')')
def s ( b ) : return (
 'e' if b % 10 in range (2, 5) else
 'a' )
n = int (input ('Koliko zagrada? '))
W = []
gen_zag (n, 0, '')
d = len (W)
print (n, 'zagrad'  +s (n),
       d, 'rečenic' +s(d))
```

```
Koliko zagrada? 3
3 zagrade 5 rečenica
>>> print ( *W )
((()))  (()())  (())()  ()(())  ()()()
```

Bez sumnje, ako pišemo rekurzivne funkcije (procedure) u našim programima, može se dobiti kompaktan kôd. Ali, postoje dvije zamke u uporabi rekurzija! To su maksimalna dubina rekurzije i vrijeme izvršavanja.

Maksimalna dubina rekurzije

Pri svakom pozivu rekurzivne funkcije generira se dio programskog koda koji će biti izvršen po dosezanju izlaza iz rekurzije. No, postoji ograničenje dubine rekurzije! Na primjer, za funkciju `F()` je to `1021` puta:

```
>>> F (-10000)
Traceback ...
f(x +10))
[Previous line repeated 1021 more times]
  ...
return (x +10    if x >= 100 else
RecursionError: maximum recursion depth
exceeded in comparison
```

Za izračunavanje faktorijela broja `1025` rekurzivnom funkcijom, `fakt(1025)`, bilo bi dojavljeno:

```
RuntimeError: maximum recursion depth
exceeded in cmp
```

Kontrola ulaznih podataka i rekurzije

Dijelovi programa za unos podataka najčešće imaju strukturu:

```
while True :
  "Unos podataka"
  if "Podaci u domeni definiranosti" :
break
```

Ako je unos podataka dio potprograma, kao u potprogramu za unos datuma (dana i mjeseca):

```
def datum (s = '') :
   while 1 :
     d, m = Input (s +' datum, dan i '
           'mjesec ' +str(G)+'. god. ')
     if (Int(d) and Int(m) and
         1 <= m <= 12 and
         1 <= d <= M[m] ) : return d, m
```

može se izbaciti *WHILE petlja* i koristiti rekurzija:

```
def datum (s = '') :
  d, m = Input (s +' datum, dan i '
        'mjesec ' +str(G)+'. god. ')
  if (Int(d) and Int(m) and
      1 <= m <= 12 and
      1 <= d <= M[m] ) : return d, m
  datum (s)
```

To je rekurzija zdesna koja u ovakvoj strukturi ima značenje „vrati se na početak potprograma” (kao *naredbe GOTO* u nekim jezicima). Jedino se postavlja pitanje: Je li takav dio programa pregledniji od prvobitnog?

JEDNADŽBA S JEDNOM NEPOZNANICOM

Općenito je jednadžba s jednom nepoznanicom (jednadžba pravca):

```
a*x + b = c*x + d
```

iz čega slijedi:

```
(a-c)*x +(b-d) = 0  -> x = - (b-d)/(a-c)
```

Da bismo dobili `(a-c)` i `(b-d)` zamijenit ćemo `x` u evauliranju izraza sa `1j`. Rezultat će biti kompleksni broj `c`. Tada je rješenje:

```
x = -c.real/c.imag
```

pa možemo definirati funkciju `X()` u kojoj ćemo preuređenu zadanu jednadžbu izjednačiti s nulom i varijablu `x` u funkciji `eval()` zamijeniti s globalnom varijablom `x = 1j`:

X.py

```
def X (eq):
    eq = eq.replace ("=","-(")+")"
    print (eq)
    c  = eval (eq, {'x': 1j})
    print (c)
    return -c.real /c.imag
eq = input ('Zadaj jednadžbu ')
x = X ( eq )
print ( 'x =', x )
```

```
>>>
Zadaj jednadžbu -3*x +6 = 2*x -2
-3*x +6 -( 2*x -2)
(8-5j)
x = 1.6
```

FUNKCIJA S REALNIM INKREMENTOM

Funkcija `range()` generira cjelobrojni niz brojeva, s cjelobrojnim inkrementom. Definirajmo funkciju `frange()` koja će imati realne vrijednosti granica intervala i realni inkrement (korak). Ovdje prvi put koristimo *naredbu YIELD* („prinos"). Pravilo pisanja je:

```
yield izraz
```

Značenje je poput *naredbe RETURN* s jedinom razlikom što vraća generirani niz vrijednosti. Kada se funkcija pozove i nit izvršenja pronađe rezerviranu riječ `yield` u funkciji, izvršavanje funkcije zaustavlja se na toj samoj liniji i vraća objekt generatora natrag pozivatelju.

frange.py

```
# + Moj_modul.py
def frange (arg0, arg1 = None,
                  arg2 = None):
  """
  Generator aritmetičkog niza s realnim
  brojevima. Sintaksa je slična funkciji
  range():
     frange ( [start, ] stop [, inc] )
  gdje su: start, stop i inc
  realni izrazi (inc != 0)   """
  start = 0.0;  inc   = 1.0
  if arg2 is not None :
     start = arg0; stop = arg1
     inc = arg2
  elif arg1 is not None :
     start = arg0; stop = arg1
   else                   : stop = arg0
   while (start <= stop -inc) if inc>0\
     else (start >= stop -inc ) :
     yield round (start, 2) # generator
     start += inc
```

```
>>>
>>> frange ( 5 )
<generator object gfrange at
0x0000029222CE2308>
>>> list ( frange (5) )
[0.0, 1.0, 2.0, 3.0, 4.0]

>>> ARG = [ (5, ), (1, 5, 0.5),
            (5, 0, -0.5)]
>>> for arg in ARG : exec (
    "print ( list (frange " +str(arg)
   +") )" )
[0.0, 1.0, 2.0, 3.0, 4.0]
[1, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5]
[5, 4.5, 4.0, 3.5, 3.0, 2.5, 2.0, 1.5,
1.0, 0.5]

>>> for arg in ARG :
    exec ("for x in frange " +str(arg)
         +" : print ( x, end = ' ' )" )
    print ()
0.0 1.0 2.0 3.0 4.0
1 1.5 2.0 2.5 3.0 3.5 4.0 4.5
5 4.5 4.0 3.5 3.0 2.5 2.0 1.5 1.0 0.5
>>> from math import *
>>> for s in frange (0, 5.5, 0.5) :
    x = radians (s)
    print ( ("%4.1f" +"%10.6f"*2)
          % (s, sin(x), cos(x)) )
 0.0  0.000000  1.000000
 0.5  0.008727  0.999962
 1.0  0.017452  0.999848
 1.5  0.026177  0.999657
 2.0  0.034899  0.999391
 2.5  0.043619  0.999048
 3.0  0.052336  0.998630
 3.5  0.061049  0.998135
 4.0  0.069756  0.997564
 4.5  0.078459  0.996917
 5.0  0.087156  0.996195
```

FUNKCIJE `filter()` I `reduce()`

U sedmom smo poglavlju definirali funkcije `filter()` i `reduce()`. Kao uvjet „filtriranja" i reduciranja koristili smo *LAMBDA funkciju*. Umjesto nje može se rabiti funkcijski potprogram. Na primjer, kao u nalaženju prim brojeva u intervalu od 2 do n i ispisu njihova umnoška:

```
# prim_brojevi.py
from functools import *
def f1 (x)    : return x == i or x % i
def f2 (x, y) : return x * y
n  = eval (input ('Prim brojevi do '))
PB = list (range (2, n+1))
for i in range (2, int(n**0.5) +1) :
    PB = list (filter (f1, PB))
print (* PB)
print (reduce (f2, PB), '\n')
```

```
Prim brojevi do 50
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
614889782588491410
```

P R O G R A M I

Poslije uvođenja potprograma znatno se proširuju mogućnosti programiranja, pa će ih i rješenja problema - programi - u preostalom dijelu knjige gotovo uvijek sadržavati. Ovdje dajemo nekoliko primjera programa koji u dovoljnoj mjeri ilustriraju uporebu potprograma i obuhvaćaju gotovo sve dosad opisane naredbe, tipove i strukture podataka.

SLAGALICA

Igra „slagalica“ stara je igra koju je osmislio Sam Loyd još u devetnaestom stoljeću. Bio je to kvadratni okvir koji je sadržavao 4x4 polja od kojih je petnaest polja bilo popunjeno kvadratnim pločicama, označenim s 1 do 15, a jedno polje je prazno i služi za premještanje pločica.

Ovdje dajemo program za slagalicu 3x3, sa skromnim grafičkim mogućnostima. „Pločice“ su označene brojevima od 1 do 8, a „pomičemo“ ih otipkavanjem onih brojeva za koje je moguć pomak, mapa Q. Inicijalno su brojevi postavljeni slučajnim rasporedom, s praznim poljem na poziciji (3, 3). Cilj je pomicanjem pločica dobiti uređenje od 1 do 8.

Slagalica.py

```
from random import *
"""
123
456
78
"""
Q = { 0 : (1, 3),        1 : (0, 2, 4),
      2 : (1, 5),        3 : (0, 4, 6),
      4 : (1, 3, 5, 7),  5 : (2, 4, 8),
      6 : (3, 7),        7 : (4, 6, 8),
      8 : (5, 7) }

def Prikazi () :
  s = ''.join (S)
  print (' ' +s[:3], '\n', s[3:6],
         '\n', s[6:])
  return s == C+' '
C  = '12345678'; S = sample (C, 8) +[' ']
Ok = Prikazi (); q = S.index (' ')
while not Ok :
  x = input ('Pomičem broj? ')
  if x in S :
    y = S.index (x)
    if y in Q[q] :
      S[q] = x;  S[y] = ' '; q = y
    Ok = Prikazi ()
```

ISPIS BROJA OD 1 DO 99 SLOVIMA

Sljedeći program ispisuje slovima zadani broj iz intervala [1, 99].

Slovima.py

```
def slovima ( n ) :
  if n not in range (1, 100) :
     return ('Broj nije iz intervala '
             '[1, 99]' )
  J = ['','jedan','dva','tri','četiri',
       'pet','šest','sedam','osam',
       'devet','deset']
  T = ['', 'jeda'] +J[2:4] \
     +['četr', 'pet', 'šes'] +J[7:]
  D = J[:4] +['četr', 'pe', 'šez'] \
     +J[7 :9] +['deve']
  d = n //10;  j = n %10
  return J[n]          if n <= 10 else\
         T[j] +'naest' if n <  20 else\
         D[d] +'deset' +J[j]
while True :
  N = input('\nZadaj broj iz intervala'
            ' [1, 99] ')
  if not N : break
  try    : N = eval( N ); print( N,
                '-->', slovima (N) )
  except : pass
```

```
>>>
Zadaj broj iz intervala [1, 99] 101
101 --> Broj nije iz intervala [1, 99]

Zadaj broj iz intervala [1, 99] deset

Zadaj broj iz intervala [1, 99] 10
10 --> deset

Zadaj broj iz intervala [1, 99] 16
16 --> šesnaest

Zadaj broj iz intervala [1, 99] 44
44 --> četrdesetčetiri

Zadaj broj iz intervala [1, 99] 66
66 --> šezdesetšest

Zadaj broj iz intervala [1, 99] <Enter>
>>>
```

IGRA ČETVORKE

Nekad, dok nije bilo mobitela, među djecom je bila popularna igra četvorke. Ploča s 8 stupaca i 6 redova popunjavala se plavim i crvenim gumbima, od dna

prema gore. Pobjednik je onaj koji postavi svoja četiri gumba jednog do drugog po horizontali, vertikali ili dijagonali. Sada još nemamo grafičke mogućnosti prikaza (ista igra je dana u četrnaestom poglavlju), ali je dajemo da se vidi kako je riješen problem četiri gumba jedan do drugog.

Igra_četvorke.py

```
# Igrajmo se četvorke
M = 7;  N = 8;  St = [0]*N
L = '\n' +'+' +'---+' *N;  T = {}
for m in range (M) : T[m] = [' ']*N
def Ispis () :
  print ('  0', end = ' ')
  for n in range (1, N) :
      print ('  ' +str(n), end = ' ')
  print (L)
  for i in range (M) :
    print ('|', end = ' ')
    for j in range (N) :
      print (T[i][j] +' |', end = ' ')
    print (L)

def dom_M (m) : return m in range (M)
def dom_N (n) : return n in range (N)

def Izgradi_Niz (i, j) :
  global S
  S = ''
  for k in range (j-3, j+4) :
    if dom_N (k) : S += T[i][k]
  S += ' '
  for k in range (i-3, i+4) :
    if dom_M (k) : S += T[k][j]
  S += ' '
  for k in range (3, -4, -1) :
    if dom_M (i+k) and dom_N (j-k) :
       S += T[i+k][j-k]
  S += ' '
  for k in range (-3, 4) :
      if dom_M (i+k) and dom_N (j+k) :
         S += T[i+k][j+k]
Ispis()
Pop = 0;  Pobjeda = False;  I = 'X'
while not Pobjeda and Pop < M*N :
  while 1 :
    print (I +'?', end = ' ')
    j = int (input ())
    if dom_N(j) and St[j] < M : break
  Pop += 1;  St[j] += 1
  i = M -St[j]; T[i][j] = I
  Ispis();      Izgradi_Niz (i, j)
  Pobjeda = I*4 in S
  if Pobjeda : print(
     'POBJEDA ' +I +'!'); break
  I = 'O' *(I=='X') or 'X'*(I=='O')
if not Pobjeda : print ('NERIJEŠENO!')
```

Permutacije

Permutacija je niz koja sadrži točno jednom svaki element iz nekog konačnog skupa. Koncept niza se razlikuje od koncepta skupa po tome što u skupu nije određen redoslijed elemenata, dok su u nizu točno određeni prvi, drugi, itd. elementi. U priloženom programu ispisuju se permutacije brojeva liste od 0, 1,..., n.

Permutacije.py

```
def perm (L):
  # Lista svih permutacija liste L
  if len(L) <= 1: return [L]
  r = []
  for i in range(len(L)):
    s = L[:i] + L[i+1:]
    p = perm (s)
    for x in p:
      r.append(L[i:i+1] + x)
  return r
L = [0, 1, 2]
P = perm (L)
for X in P : print ( *X )
```

```
>>>
0 1 2
0 2 1
1 0 2
1 2 0
2 0 1
2 1 0
>>> len (P)      6
```

Problem n-te permutacije N elemenata jednak je P[n], 0<=n<N!. Posljednja permutacija jednaka je P[-1].

Faktorijel (3)

Evo još jednog programa za izračunavanje faktorijela broja n, n≥0. Koristimo značenje funkcije reduce().

Faktorijel_3.py

```
from functools import *
while 1 :
  n = eval (input (
      'Faktorijel broja n, n>=0: '))
  if type(n) == int and n >= 0 : break
def fac (n) :
  F = 2*[1] +list (range (2, n+1))
  return reduce (lambda x, y: x*y, F)
print ( fac(n) )
```

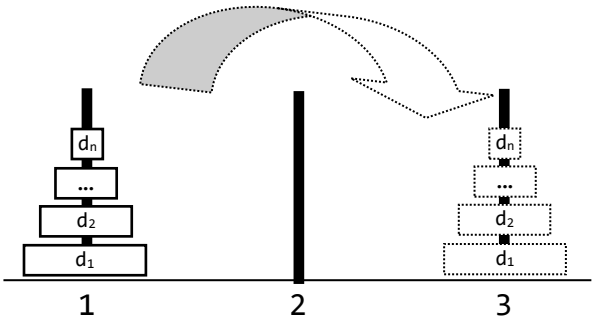
```
>>>
Faktorijel broja n, n>=0: 20
2432902008176640000
```

TORNJEVI HANOJA

Tornjevi ili kule Hanoia je drevna igra,

https: //en.wikipedia.org/wiki/Tower_of_Hanoi

u kojoj je potrebno preseliti jedan toranj sačinjen od n diskova d_1, d_2, ..., d_n, različitih promjera smještenih u stogu (1) tako da je disk s najvećim promjerom, d_1, na dnu, a disk s najmanjim promjerom, d_n, na vrhu stoga. Zadatak je premjestiti toranj (diskove) na drugi stog (3) koristeći pritom pomoćni stog (2).



U svakom koraku premješta se disk s vrha jednog stoga na vrh drugoga ali uz uvjet da je promjer diska, d_i, koji se premješta manji od promjera diska, d_j, na vrhu stoga na koji se premješta, ako stog nije prazan. Zadatak je uspješno obavljen ako su stogovi (1) i (2) prazni, a stog (3) sadrži diskove d_1, d_2, ..., d_n, u istom uređenju kao stog (1) na početku.

Postoji nekoliko rješenja: iterativno, rekurzivno i grafičko i rekurzivno. Ovdje smo se odlučili za prikaz rekurzivnog rješenja jer najbliži strukturi algoritma za rješenje tog problema.

Ako je n=1, rješenje je trivijalno: prebaciti disk sa stoga (1) na stog (3). Ako je n>1, rješenje je opisano slijedećim rekurzivnim pravilom:

1. Premjestiti n-1 diskova sa stoga (1) u stog (2) koristeći stog (3) kao pomoćni.
2. U stogu (1) je ostao najveći disk. Premjestiti ga u stog (3).
3. Premjestiti n-1 diskova sa stoga (2) u stog (3) koristeći stog (1) kao pomoćni.

Evo kompletnog programa u kojem smo definirali tri stoga i dodali prikaz premještanja diskova u svakom koraku

Hanoi.py

```
Z = '='
def Prikazi ():
  def s (x) :
    r = (N -len(x)) //2
    return ' '*r +x +' '*r
  for k in range (n, 0, -1) :
    N = 2*n+1
    A = B = C = ' '*n +'|' +' '*n
    if k <= len (D[0]) :
       A = s (D[0][k-1])
    if k <= len (D[1]) :
       B = s (D[1][k-1])
    if k <= len (D[2]) :
       C = s (D[2][k-1])
    print (A, B, C)
  print ()

def Prenesi (n, _1, _3, _2):
  global i
  if n == 1:
    X = D[_1].pop(); D[_3].append(X)
    Prikazi ();  i += 1
  else:
    Prenesi (n-1, _1, _2, _3)
    Prenesi (  1, _1, _3, _2)
    Prenesi (n-1, _2, _3, _1)

n = int (input ('Broj diskova? '))
D = [ [(Z*i +'|' +Z*i) for i in
       range (n, 0, -1)], [], [] ]
i = 0
print ()
Prikazi()
Prenesi (n, 0, 2, 1)
print ('\n', i, 'premještanja')
```

```
>>>
Broj diskova? 2

 =|=     |      |
==|==    |      |

  |      |      |
==|==   =|=     |

  |      |      |
  |     =|=   ==|==

  |      |     =|=
  |      |    ==|==

3 premještanja
```

HRVATSKO-ENGLESKO-FRANCUSKI BROJEVI 1 DO 10 (3)

Evo još jedne verzije poznatog programa za prevođenje brojeva od 1 do 10 u tri jezika.

Hr_En_Fr_3.py

```
H = ( 'jedan', 'dva', 'tri', 'četiri',
      'pet', 'šest', 'sedam', 'osam',
      'devet', 'deset' )
E = ( 'one', 'two', 'three', 'four',
      'five', 'six', 'seven', 'eight',
      'nine', 'ten')
F = ( 'un', 'deux', 'trois', 'quatre',
      'cinq', 'six', 'sept', 'huit',
      'neuf', 'dix' )
w   = input( 'Prevodim broj? ' )
def prevedi (X, Y, Z) :
    i = X.index(w); print (Y[i], Z[i])
if   w in H : prevedi (H, E, F)
elif w in E : prevedi (E, H, F)
elif w in F : prevedi (F, H, E)
else        :
  print( 'Ne postoji broj ' +w,
              'niti u jednom jeziku!')
```

```
>>>
Prevodim broj? devet
nine neuf
```

MNOŽINA ENGLESKIH IMENICA

Prilažemo jednostavan program za tvorbu množine engleskih imenica koji obuhvaća dio engleskih imenica koje tvore „nepravilnu" množinu. Lista eq sadrži imenice koje imaju jednaku množinu, rječnik ex su imenice koje imaju „nepravilnu" množinu i rječnik E sadrži prijevode nekih sufiksa. Za imenice koje se ne nalaze u eq, ex i E, množina se tvori dodavanjem sufiksa `'s'`.

ENG_plural.py

```
# Množina engleskih imenica

def plural ( w ) :
  eq  = ['sheep', 'fish', 'deer',
         'species', 'aircraft',
         'software', 'information']

  ex  = { 'person': 'people',
    'datum': 'data', 'mouse': 'mice',
    'bus'  : 'buses','child': 'children',
    'food' : 'foods' }

  E = { 'y': 'ies',  'f': 'ves',
       'fe': 'ves', 'us': 'i',
       'is': 'es', 'an': 'en', 'on': 'a' }

  w1,w2 = w[-1],w[-2:]; pl = lambda w : (
        ex[w]          if w in ex
  else  w              if w in eq
  else  w[:-2] +E[w2] if w2 in E
  else  w[:-1] +E[w1] if w1 in E
  else  w +'es'        if w1 in 'sxo' or
                          w2 in ['sh','ch']
    else w.replace ('oo', 'ee') if 'oo' in w
    else w +'s' )
  return pl ( w )
Lx = [
  'boat', 'house', 'cat', 'river', 'bus',
  'wish', 'pitch', 'box', 'penny', 'spy',
  'baby', 'city', 'daisy', 'woman', 'man',
  'child', 'tooth', 'foot', 'leaf',
  'mouse', 'goose', 'half', 'software',
  'information', 'knife', 'wife', 'life',
  'elf', 'loaf', 'potato', 'tomato',
  'cactus', 'focus', 'fungus', 'nucleus',
  'syllabus', 'analysis', 'diagnosis',
  'oasis', 'thesis', 'crisis',
  'phenomenon', 'criterion', 'datum',
  'sheep', 'fish', 'deer', 'species',
  'aircraft' ]
Lx.sort(); print( )
print( "jednina     množina     " *2 )
print( '-' *22 *2 )
i = 0
for x in Lx :
   if i > 0 and i % 2 == 0 : print ()
   y  = x.lower()
   PL = plural ( y )
   print( "%-11s %-11s" % (y, PL),
          end = ' ')
   i += 1
```

```
>>>

jednina     množina     jednina     množina
-----------------------------------------
aircraft    aircraft    analysis    analyses
baby        babies      boat        boats
box         boxes       bus         buses
cactus      cacti       cat         cats
child       children    city        cities
crisis      crises      criterion   criteria
daisy       daisies     datum       data
deer        deer        diagnosis   diagnoses
elf         elves       fish        fish
focus       foci        foot        feet
fungus      fungi       goose       geese
half        halves      house       houses
information information knife       knives
leaf        leaves      life        lives
loaf        loaves      man         men
mouse       mice        nucleus     nuclei
oasis       oases       penny       pennies
phenomenon  phenomena    pitch       pitches
potato      potatoes    river       rivers
sheep       sheep       software    software
species     species     spy         spies
syllabus    syllabi     thesis      theses
tomato      tomatoes    tooth       teeth
wife        wives       wish        wishes
woman       women
```

FRANCUSKI GLAGOLI

Ovo je kutak za „Francuze“. Ideja je nastala prije 12 godina dok smo „prelistavali“ rječnik s 12000 glagola

Bescherelle *La conjugaison pour tous*
HATIER – Paris 1997

Analizirali smo grupe nastavaka za pojedine glagole i izdvojili smo njih stotinjak najčešćih u praksi.

Pokazat ćemo kako se može napisati program za konjugaciju u tvorbi prezenta, imperfekta i futura. Dodali smo i particip perfekta.

Složeni perfekt (*passé composé*) je prošlo vrijeme koje se tvori od pomoćnog glagola ***avoir*** ili ***être*** u prezentu i participa perfekta (*participe passé*) glagola koji se mijenja. Većina glagola gradi ovo vrijeme koristeći pomoćni glagol *avoir*.

Pregled nastavaka triju glagola dan je u sljedećoj tablici. Nastavci imperfekta i futura dani su s ključevima `'-I'` i `'-F'` u mapi **s** programa. Particip perfekta ima oznaku A za tvorbu s glagolom *avoir*, E s *être*.

glagol	***être***	***avoir***	***aller***
hrvatski	*biti*	*imati*	*ići*
prezent	*suis, es, est; sommes, êtes, sont*	*ai, as, a; av-07; ont*	*vais, vas, va; all-07;*
imperfekt	*ét-*	*av-*	*all-*
aorist	*f-7*	*e-7*	*all-6*
futur	*ser-*	*aur-*	*ir-*
konjuktiv prezenta	*so-!3*	*a-!2*	*aill-!1*
konjuktiv imperfekta	*f-#3*	*e-#3*	*all-#2*
particip prošli	*A été*	A *eu*	E *allé*

Iz leksikona koji sadrži stotinjak glagola izabrali smo njih 15, lista `Verbes`, koji u dovoljnoj mjeri prikazuju kako se generiraju konjugacije. Zaglavlje svakog glagola sadrži infinitiv, particip prošli i prijevod na hrvatski. Dodali smo varijablu A koja sadrži posebne znakove francuske abecede ako želite pogledati leksikon za izabranu riječ sa `Lx['riječ']` u interaktivnom modu.

FR_verbes.py

```
from pickle  import *

A = "âéèêîûü"
global Lx

def IMPORT () : # uvoz leksikona
    global Lx
    Add = open ('FR-verbes.TxT', 'r',
                encoding = 'UTF-8')
    ADD = []
    for line in Add :
        ADD.append (line[:-1].split('\t'))

    Lx = {}
    for W in ADD :
        if W : Lx[W[0]] = tuple (W[1:])
    Add.close ()

    f = open ('FR-verbes.dat', 'wb')
    dump (Lx, f)
    f. close()

IMPORT ()

def Generiraj ( Y, S = '' ) :
    Y = Y. replace (', ', " ")
    Y = Y. replace ('; ', " ")
    Y = Y. split ()
    V = []
    for x in Y :
      if '-' in x :
        i = x.find ('-'); k = x[i+1:]
        if k == '' : k = S
        for y in s[k] :
          V. append (x[ :i] +y)
      else :
          V. append (x)
    return V

def Vremena ( v ) :
    Y = Lx [ v ] [2] # prezent
    print (* Generiraj ( Y ) )

    Y = Lx [ v ] [3] # imperfekt
    print (* Generiraj ( Y, '-I' ) )

    Y = Lx [ v ] [5] # futur
    print (* Generiraj ( Y, '-F' ))

s = {
  # imperfekt
  '-I' : ('ais', 'ais', 'ait', 'ions',
          'iez', 'aient'),
  # futur
  '-F' : ('ai', 'as', 'a', 'ons', 'ez',
          'ont'),
  # prezent
 '01' : ('s', 's', 't'),
 '02' : ('s', 's', ''),
 '03' : ('e', 'es', 'e'),
 '04' : ('x', 'x', 't'),
 '05' : ('is', 'is', 'it'),
 '06' : ('ons', 'ez', 'ent'),
 '07' : ('ons', 'ez'),
 '08' : ('yons', 'yez', 'ient'),
 '09' : ('mes', 'tes', 'rent'),
```

```
    # aorist
    '4' : ('is', 'is', 'ît'),
    '5' : ('is', 'is', 'it', 'îmes', 'îtes',
           'irent'),
    '6' : ('ai', 'as', 'a', 'âmes', 'âtes',
           'èrent'),
    '7' : ('us', 'us', 'ut', 'ûmes', 'ûtes',
           'urent'),
    '8' : ('ûs', 'ûs', 'ût', 'ûmes', 'ûtes',
           'ûrent'),
    '9' : ('ins', 'ins', 'int', 'înmes',
           'întes', 'inrent'),
   # konjuktiv prezenta
   '!1' : ('e','es','e','ions','iez','ent'),
   '!2' : ('e','es','e','ions','iez', ent'),
   '!3' : ('is','is','it','yons','yez','ient'),
   '!4' : ('ie','ies','ie','yions','yiez',
           'ient'),
   '!5' : ('ne','nes','ne','ions','iez','nent'),
   # konjuktiv imperfekta
   '#1' : ('isse', 'isses', 'ît', 'issions',
           'issiez', 'issent'),
   '#2' : ('asse', 'asses', 'ât', 'assions',
           'assiez', 'assent'),
   '#3' : ('usse', 'usses', 'ût', 'ussions',
           'ussiez', 'ussent'),
   '#4' : ('ûsse', 'ûsses', 'ût', 'ûssions',
           'ûssiez', 'ûssent'),
   '#5' : ('insses', 'insses', 'înt',
           'inssions', 'inssiez', 'inssent')  }
Verbes = ['aller', 'avoir', 'boire', 'devoir',
  'dire', 'être', 'lire', 'pouvoir', 'prendre',
  'savoir', 'sortir', 'venir', 'vivre', 'voir',
  'vouloir']
for x in Verbes :
    print ( x, Lx[x][0], Lx[x][-2:],
            Lx[x][1])
    Vremena (x); print ()
```

```
>>>
aller  ('E ', 'allé') ići
vais vas va allons allez vont
allais allais allait allions alliez allaient
irai iras ira irons irez iront

avoir  ('A', 'eu ') imati
ai as a avons avez ont
avais avais avait avions aviez avaient
aurai auras aura aurons aurez auront

boire  ('A', 'bu') piti
bois bois boit buvons buvez boivent
buvais buvais buvait buvions buviez buvaient
boirai boiras boira boirons boirez boiront

devoir  ('A', 'dü/due, dus/dues') morati, dugovati
dois dois doit devons devez doivent
devais devais devait devions deviez devaient
devrai devras devra devrons devrez devront

dire  (‘A’, 'dit') reći
dis dis dit disons dites disent
disais disais disait disions disiez disaient
dirai diras dira dirons direz diront

être  ('A', 'été') biti
suis es est sommes êtes sont
étais étais était étions étiez étaient
serai seras sera serons serez seront

lire  ('A', 'lu') čitati
lis lis lit lisons lisez lisent
lisais lisais lisait lisions lisiez lisaient
lirai liras lira lirons lirez liront

pouvoir  ('A', 'pu') moći
peux peux peut pouvons pouvez peuvent
pouvais pouvais pouvait pouvions pouviez pouvaient
pourrai pourras pourra pourrons pourrez pourront

prendre  ('A', 'pris') uzeti
prends prends prend prenons prenez prennent
prenais prenais prenait prenions preniez prenaient
prendrai prendras prendra prendrons prendrez
prendront

savoir  ('A', 'su') znati
sais sais sait savons savez savent
savais savais savait savions saviez savaient
saurai sauras saura saurons saurez sauront

sortir  ('E', 'sorti') izaći
sors sors sort sortons sortez sortent
sortais sortais sortait sortions sortiez sortaient
sortirai sortiras sortira sortirons sortirez
sortiront

venir  ('E ', 'venu') doći
viens viens vient venons venez viennent
venais venais venait venions veniez venaient
viendrai viendras viendra viendrons viendrez
viendront

vivre  ('A', 'vécu') živjeti
vis vis vit vivons vivez vivent
vivais vivais vivait vivions viviez vivaient
vivrai vivras vivra vivrons vivrez vivront

voir  ('A', 'vu') vidjeti
vois vois voit voyons voyez voient
voyais voyais voyait voyions voyiez voyaient
verrai verras verra verrons verrez verront

vouloir  ('A', 'voulu') željeti
veux veux veut voulyons voulyez voulient veulent
voulais voulais voulait voulions vouliez voulaient
voudrai voudras voudra voudrons voudrez voudront

falloir  ('A', 'fallu') trebati
il faut
il fallait
il faudra

>>> # Il faut voyager, kaže Georges Moustaki

https://www.youtube.com/watch?v=AHzBjY-sOyA

>>> A         'âéèêîûü'
>>> Lx['être']
('', 'biti', 'suis, es, est; sommes, êtes, sont',
'ét-', 'f-7', 'ser-', 'so-!3', 'f-#3', 'A', 'été')
```

11.

KLASE I OBJEKTI

Python je od inačice 3.0 objektno orijentirani programirni jezik, ali smo namjerno izbjegavali bavljenje objektno orijentiranim programiranjem (OOP) u prethodnim poglavljima ove knjige. Preskočili smo to jer smo uvjereni da je lakše i zabavnije započeti učenje Pythona, a da ne moramo znati sve detalje objektno orijentiranog programiranja. Iako smo izbjegavali OOP, on je uvijek bio prisutan u vježbama i primjerima. Koristili smo objekte i metode iz klase bez potpunog objašnjavanja njihove OOP pozadine.

Gotovo sve u Pythonu je objekt, sa svojim svojstvima i metodama. Klasa je poput konstruktora objekata ili "nacrta" za stvaranje objekata. Svi su primitivni i složeni podaci Pythona objekti određenih klasa: int, float, str, list, tuple, set i dict.

U ovom ćemo poglavlju pokazati kako definirati vlastite klase i metode, te kako pisati objektno orijentirane programe u Pythonu utemeljene na četiri fundamentalna koncepta objektno orijentiranog programiranja.

```
class ime_klase [( [osnovna_klasa ])] :
      [ atributi_klase ] blok_klase

blok_klase :
    [konstruktor][blok]
```

```
konstruktor :
    def __init__ (self [, parametri] ) :
        blok
self : ime
```

Prvi parametar svake metode, `self`, ime je koje se odnosi na pozivajući objekt. U literaturi je to najčešće `self`. No, možete dati bilo koje ime, a ne nužno "`self`". □

```
class RECORD :
  def __init__ ( s, x, y ):
    for i in range (len (x)) : exec (
        's.' +x[i] +' = ' +'str(y[i])')

>>> atr = ('id', 'kat_br', 'naziv', 'nc',
           'vpc', 'kol', 'JM'),
>>> A  = [1, "112/370", "AMORTIZER",
          247.65, 310.86, 0.00, "kom"]
>>> ART = RECORD (art, A); ART.naziv
>>> 'AMORTIZER'
```

Uvod

Mnogi računalni znanstvenici i programeri smatraju OOP (objektno orijentirano programiranje) modernom programskom paradigmom. Međutim, prvi programirni jezik koji je koristio objekte bio je Simula 67. Kao što naziv govori, Simula 67 predstavljena je još 1967. godine. Veliki napredak za objektno orijentirano programiranje dogodio se tek s programskim jezikom Smalltalk u 1970-ima.

Python je objektno orijentirani jezik još od vremena kada postoji. Zbog toga je stvaranje i korištenje klasa i objekata izravno jednostavno.

PREGLED TERMINOLOGIJE *OOP*

Ovo će vam poglavlje pomoći da postanete stručnjak za korištenje Pythonove objektno orijentirane programske podrške. Ako nemate nikakvih prethodnih iskustava s objektno orijentiranim programiranjem, evo osnovne terminologije:

Klasa je tip (struktura) podataka koja objedinjuje podatke ili svojstva i aktivnosti nad njima, funkcije i procedure, nazvane jednim imenom „metode".

Objekt („object") je instanca klase, ili varijabla tipa podataka definiranog klasom. Veza između objekata i klasa jednaka je vezi između varijable i tipa podataka. Objekti su stvarni entiteti. Kada se program izvršava, objekti zauzimaju dio memorije za svoje interne reprezentacije.

Klasa je korisnički definirani prototip za objekt koji definira skup atributa koji karakteriziraju bilo koji objekt klase. Atributi su članovi podataka (varijable klase i varijable instance) i metode kojima se pristupa točkovnim zapisom.

- Instanca - Pojedinačni objekt određene klase.
- Varijabla klase - varijabla koju dijele sve instance klase. Varijable klase definirane su unutar klase, ali izvan bilo koje metode klase. Varijable klase ne koriste se tako često kao varijable instance.
- Varijabla instance - varijabla koja je definirana unutar metode i pripada samo trenutnoj instanci klase.
- Član podataka - varijabla klase ili varijabla instance koja sadrži podatke povezane s klasom i njezinim objektima.
- Instanciranje - Stvaranje instance klase.
- Objekt - jedinstvena instanca podatkovne strukture koja je definirana njezinom klasom. Objekt sadrži i članove podataka (varijable klase i varijable instance) i metode.
- Metoda - posebna vrsta funkcije koja je definirana u definiciji klase.
- Nasljeđivanje - prijenos karakteristika klase u druge klase koje su iz nje izvedene.
- Preopterećenje funkcije - dodjela više od jednog ponašanja određenoj funkciji. Izvedena operacija razlikuje se ovisno o vrstama uključenih objekata ili argumenata.
- Preopterećenje operatora - dodjela više od jedne funkcije određenom operateru.

Mi radimo s klasama i objektima od početka ove knjige. Svaki je element u programu Pythona objekt klase. Brojevi, znakovni nizovi, n-torke, liste, skupovi i rječnici objekti su odgovarajuće ugrađene klase. Čak i funkcija definirana pomoću rezervirane riječi `def` pripada klasi funkcija.

```
>>> type(20)                <class 'int'>
>>> type("Python")          <class 'str'>
>>> type([1, 2, 3])       <class 'list'>
>>> type((1, 2, 3))      <class 'tuple'>
>>> type({'žuto', 'crno'})
<class 'set'>
>>> type({0: False, 1: True})
<class 'dict'>
>>> type(abs) # ugrađena funkcija
<class 'builtin_function_or_method'>
>>> f = lambda x : x**2;
>>> type(f)           <class 'function'>
>>> def x_2 (x) : return x**2
>>> type(x_2)         <class 'function'>
```

Jedna od najčešće rabljenih klasa u ovoj knjizi i primjerima programa je klasa `list`. Ta klasa sadrži mnoštvo metoda za izradu lista, sortiranje, pristup i promjenu elemenata ili za uklanjanje elemenata. Na primjer:

```
>>> X = [3, 6, 9]; print( *X )     3 6 9
>>> X[1] = 99; print( *X )        3 99 9
>>> X.append(42); print( *X ) 3 99 9 42
```

```
>>> X.sort(); print( *X )      3 9 42 99
>>> y = X.pop(); print( y, *X )
99 3 9 42
>>> del X; print( *X )
...
NameError: name 'X' is not defined
>>> Y = list( range( 9, -1, -1 ) )
>>> print( *Y ); Y.sort(); print( *Y )
9 8 7 6 5 4 3 2 1 0
0 1 2 3 4 5 6 7 8 9
```

Ovdje imena X i Y označuju dvije instance (primjerka ili objekta) klase `list`. `pop()`, `append()` i `sort()` su metode klase `list`. Njihovo nam je značenje poznato još od sedmog poglavlja.

Nije nam potrebno objašnjenje kako je Python interno implementirao liste. Ne trebaju nam ove informacije, jer nam klasa `list` pruža sve potrebne metode za neizravni pristup podacima. To znači da su detalji "učahurenja" (enkapsulacije) sadržani u "čahuri" (kapsuli), što ćemo saznati kasnije.

Drugi primjer, klasa cijelih brojeva, `int`, sadrži veliki broj funkcija i metoda:

```
>>> dir (int)
['__abs__',     '__add__',     '__and__',
'__bool__',    '__ceil__',    '__class__',
'__delattr__', '__dir__', '__divmod__',
'__doc__',     '__eq__',      '__float__',
'__floor__',       ...,         '__xor__',
'bit_length',                 'conjugate',
'denominator',   'from_bytes',    'imag',
'numerator', 'real', 'to_bytes']
```

Pregled značenja svih metoda dobit ćemo sa:

```
>>> help (int)
Help on class int in module builtins:

class int(object)
 |  int(x=0) -> integer
 |  int(x, base=10) -> integer
...
 |  Methods defined here:
 |  __abs__(self, /)
 |      abs(self)
...
 |  real
 |     the real part of a complex number
```

Pogledajmo, na primjer, kako su definirane metode (funkcije) `__abs__` i `__add__`:

```
>>> help (int.__abs__)

Help on wrapper_descriptor:

__abs__(self, /)

    abs(self)

>>> (-12).__abs__()                12

>>> help (int.__add__)
Help on wrapper_descriptor
__add__(self, value, /)
Return self+value.
>>> (12).__add__(15)               27
```

U nastavku ćemo poglavlja pokazati kako možemo definirati vlastite klase i njihove objekte.

Klase

Klasa je, na najvišoj razini, definirana kao

klasa *:*
class ime_klase [([*osnovna_klasa*])] :
    [*atributi_klase*] *blok_klase*

blok_klase *:*
    [*konstruktor*] [*blok*]

Klasa obično uključuje sljedeće članove:

1. Konstruktor
2. Atributi instance
3. Atributi klase
4. Metode (postupci)

U nastavku poglavlja ćemo ih detaljno opisati.

PRAZNA KLASA

Iz dane se sintakse mogu napisati prazne klase:

```
>>> class X : pass
>>> class Y : 10
>>> class Z : "I ovo je prazna klasa"
```

Svaka klasa sadrži standardne metode koju ima i prazna klasa:

```
>>> dir (X) # dir (Y) i dir (Z)
['__class__', ...,
'__str__', '__subclasshook__',
'__weakref__']
```

Objekti

Ako je, na primjer, definirana prazna klasa osoba sa:

```
>>> class osoba : pass
>>> type(osoba)          <class 'type'>
```

tada objekte te klase možemo definirati kao što bismo to učinili za bilo koju ugrađenu klasu. Primjerice:

```
>>> A = osoba ();  B = osoba ()
>>> type (A)   <class '__main__.osoba'>
>>> type (B)   <class '__main__.osoba'>
>>> dir  (A) # = dir (B) = dir (osoba)
['__class__', ..., '__weakref__']
>>> id(A), id(B)
(1900426481040, 1900426191056)
>>> A = osoba(); B = A
>>> A.ime = 'Darko'; B.ime      'Darko'
>>> B.god = 41;  A.god             41
>>> A_ = set( dir(A))
>>> osoba_ = set( dir(osoba))
>>> A_ -osoba_             {'god', 'ime'}
```

KONSTRUKTOR

Konstruktor je metoda klase koja se automatski poziva svaki put kada se inicira novi objekt klase. Konstruktor je potprogram s imenom __init__(), napisan prema pravilu:

```
konstruktor :
  def __init__ (self [, parametri] ) :
      blok
self : ime
```

Prvi parametar svake metode, self, ime je koje se odnosi na pozivajući objekt. U literaturi je to najčešće self. *No, možete dati bilo koje ime, a ne nužno "*self*".* □

Sljedeći primjer definira konstruktor.

```
class osoba :
    def __init__ (self): # konstruktor
        print ('Pozvan je konstruktor')
```

Sada, kad god kreiramo objekt klase osoba, pozvat će se metoda __init__ (), kao što je prikazano u nastavku.

```
>>> p1 = osoba()  Pozvan je konstruktor
>>> p2 = osoba()  Pozvan je konstruktor
```

Atributi instance

Atributi instance su atributi ili svojstva pridružena instanci klase. Definiraju se u konstruktoru. Sljedeći primjer definira ime atributa instance i dob u konstruktoru:

```
class osoba :
  def __init__ (self): # atr. instance
    self.ime = "Nepoznato"
    self.dob = 0
```

Atributu instance može se pristupiti pomoću notacije točaka:

```
ime_objekta . ime_atributa
```

kao što je prikazano u nastavku.

```
>>> O1 = osoba (); O1.ime     Nepoznato
>>> O1. dob                   0
```

Vrijednost atributa može se pridružiti pomoću notacije točaka:

```
>>> O2     = osoba ()
>>> O2.ime = "Igor"; O2.ime      'Igor'
>>> O2.dob = 37; O2.dob             37
```

No, bolje je vrijednosti atributa objekta pridruživati unutar konstruktora. Na primjer, sljedeći konstruktor uključuje parametre Ime i Dob, osim self parametra.

```
class osoba :
  def __init__ (self, Ime, Dob):
    # konstruktor
    self.ime = Ime # atribut instance
    self.dob = Dob # atribut instance
```

Sada možemo zadati vrijednosti atributa kreirajući instancu. Primjerice:

```
>>> p2 = osoba ("Igor", 37); p2.ime
'Igor'
>>> p2.dob                          37
```

Ne morate navesti vrijednost ***self*** *parametra. Bit će dodijeljen interno u Pythonu. Također možete postaviti zadane vrijednosti, na primjer, atribute (kao i kod parametara potprograma).* □

Sljedeći kôd postavlja zadane vrijednosti parametara konstruktora. Umjesto self izabrali smo ime „_“.

```
class osoba :
    def __init__ (_,
        Ime = "***", Dob = 10) :
        _. ime = Ime;  _. dob = Dob
```

Dakle, ako se vrijednosti ne daju pri kreiranju objekta, njima će se pridružiti predefinirane vrijednosti parametara:

```
>>> X = osoba ();  X.ime, X.dob
('***', 10)
>>> Y = osoba ('Mirko', 70)
>>> Y.ime, Y.dob            ('Mirko', 70)
>>> Z = osoba (Dob = 79,
               Ime = 'Georges')
>>> Z.ime, Z.dob          ('Georges', 79)
```

ATRIBUTI KLASE

Atributi klase razlikuju se od atributa instance. Atribut čija je vrijednost jednaka za sve instance klase naziva se atributom klase. Atributi klase definirani su na razini klase, a ne unutar konstruktorske metode `__init__()`. Za razliku od atributa instance, atributima klase pristupa se pomoću naziva klase.

```
class čovjek : tko_si = 'Živi čovjek!'
```

Klasa `čovjek` uključuje atribut klase s imenom `tko_si`. Ovom atributu može se pristupiti pomoću naziva klase, kao što je prikazano u nastavku.

```
>>> čovjek().tko_si       'Živi čovjek!'
```

Svaki objekt klase `čovjek` može imati ovaj atribut klase kojem se pristupa pomoću

ime_objekta . ime_atributa.

```
>>> Č = čovjek ();  Č
<__main__.čovjek object at
0x00000290F41050A0>
>>> Č. tko_si             'Živi čovjek!'
```

Promjena atributa klase pomoću naziva klase odražavat će se na sve instance klase.

```
>>> čovjek. ime = 'Duje'
>>> Č = čovjek ()
>>> print ('Tko si?       ',
Č.tko_si, '\nKako se zoveš?', Č.ime)
Tko si?        Živi čovjek!
Kako se zoveš? Duje
>>> X = čovjek ()
>>> X . tko_si            'Živi čovjek!'
>>> X . ime                       'Duje'
```

Međutim, promjena atributa klase pomoću instance neće se odraziti drugdje. To će utjecati samo na tu posebnu instancu.

```
>>> X. ime = 'Zdravko'; X. ime  'Zdravko'
>>> Y = čovjek ();  Y. ime      'Duje'
```

Pogledajmo sljedeći primjer:

```
>>> class student :
    broj = 0
    def __init__(self) :
        student.broj += 1
```

Ovdje je `broj` atribut klase `student`. Kad god se stvori novi objekt, vrijednost `broj` povećava se za 1. Sada se može pristupiti atributu `broj` nakon kreiranja objekata, kao što je prikazano u nastavku:

```
>>> Prvi  = student(); Prvi.broj      1
>>> Drugi = student(); Drugi.broj     2
```

Ugrađeni atributi klase

Svaka Pythonova klasa sadrži sljedeće ugrađene atribute koji mogu biti rabljeni s točka-operatorom slično drugim atributima:

`__dict__` - rječnik koji sadrži imena klase.
`__doc__` - dokumentacija klase ili je nema, ako je izostavljena
`__name__` - ime klase
`__module__` - ime modula u kojem je definirana klasa. Ovaj je atribut "__main__" u interaktivnom modu.

Za gornju klasu pokušajmo pristupiti svim tim atributima:

```
# Radnik.py
class Radnik:
  'Osnovna klasa za sve radnike'
  Broj = 0
  def __init__ (self, Ime, Plaća):
      self.ime     = Ime
      self.plaća   = Plaća
      Radnik.Broj += 1
  def PrikažiBroj (self):
      print ("Ukupno radnika %d" %
             Radnik.Broj)
  def PrikažiRadnika (self):
      print ("Ime   : ", self.ime)
      print ("Plaća : ", self.plaća)
print ("Radnik.__doc__:", Radnik.__doc__)
print ("Radnik.__name__:",
       Radnik.__name__)
while 'Unos' :
  R = input ('Ime radnika ')
  if not R : break
  P = eval (input ('Plaća     '))
```

```
    Y = Radnik (R, P)
    Radnik. PrikažiRadnika (Y)
    Radnik. PrikažiBroj    (Y)
>>>
Radnik.__doc__: Osnovna klasa za sve
radnike
Radnik.__name__: Radnik
Ime radnika Ines
Plaća       10200
Ime    :  Ines
Plaća  :  10200
Ukupno radnika 1
Ime radnika <Enter>
```

METODE KLASE

Pomoću rezervirane riječi `def` može se definirati koliko god metoda trebamo u klasi. Svaka metoda mora imati prvi parametar, općenito s imenom `self`, koji se odnosi na pozvanu instancu. U primjeru `Radnik.py` metode `PrikažiBroj` i `PrikažiRadnika` su u klasi `Radnik`.

Kao što možete vidjeti, atributima instance može se pristupiti pomoću `self` parametra. Metode klase mogu se pozvati koristeći instancu, kao što slijedi:

```
>>> Z = Radnik('Student', 3333)
>>> Z. PrikažiRadnika()
Ime    :  Student
Plaća  :  3333
```

Privatni, javni i zaštićeni članovi klase

Privatnim članovima klase uskraćuje se pristup iz okoline izvan klase. S njima se može rukovati samo iz klase.

Javnim članovima (uglavnom metodama deklariranim u klasi) pristupa se izvan klase. Objekt iste klase dužan je pozvati javnu metodu. Ovaj raspored varijabli privatnih instanci i javnih metoda osigurava princip kapsulacije podataka.

Zaštićeni članovi klase dostupni su unutar klase i u njezinim podklasama. Nijednom drugom okruženju nije mu dopušteno pristupiti. To omogućava da podklasa nasljeđuje određene resurse nadklase.

Python nema mehanizam koji učinkovito ograničava pristup bilo kojoj varijabli ili metodi objekta. Python propisuje konvenciju prefiksa imena varijable/metode jednim ili dvostrukim podvlakama kako bi oponašao zaštićenih i privatnih pristupnih odredišta (*specifiers*).

Svi su članovi Pythonove klase prema zadanim postavkama javni. Bilo kojem članu može se pristupiti izvan klase. Na primjer, ako je klasa `Radnik` definirana kao

```
class Radnik:
  def __init__ (self, Ime, Plaća):
      self.ime   = Ime
      self.plaća = Plaća
```

Možete pristupiti atributima klase `Radnik` i također mijenjati njihove vrijednosti, kao što je prikazano u nastavku.

```
>>> X = Radnik ("Zdravko", 5000)
>>> X . plaća                    5000
>>> X . plaća = 7790
>>> X . plaća                    7790
```

Pythonova konvencija da bi se varijabla instance učinila zaštićenom je dodavanje prefiksa „_“ (jedna podvlaka). Na taj joj se način onemogućuje pristup, osim ako nije iz neke podklase.

```
class Radnik:
  def __init__ (self, Ime, Plaća):
    self._ime   = Ime   # zaštićen atr.
    self._plaća = Plaća # zaštićen art.
```

U stvari, to ne sprječava da varijable instance pristupaju ili mijenjaju instancu. I dalje možete izvršavati sljedeće operacije:

```
>>> X = Radnik ("Marko", 9000)
>>> X . _plaća                   9000
>>> X . _plaća = 10000
>>> X . _plaća                  10000
```

Ovdje bi se odgovorni programer suzdržao od pristupanja i izmjene varijabli instance s prefiksom _ izvan svoje klase. Slično tome, dvostruka podvlaka „_“ s prefiksom varijabli čini je privatnom, s nemogućnošću da je dirate izvan klase. Svaki pokušaj da se to učini rezultirat će sa `AttributeError`:

```
class Radnik:
  def __init__ (self, Ime, Plaća):
    self.__ime   = Ime   # privatni atr.
    self.__plaća = Plaća # privatni art.
```

```
>>> Y = Radnik ("Mili", 15000)
>>> Y . __plaća
AttributeError: 'Radnik' object has no
attribute '__plaća'
```

Python upravlja imenima privatnih varijabli. Svaki član s dvostrukom podvlakom bit će promijenjen u:

```
_objekt . _klasa__variabla
```

Ako je to potrebno, i dalje mu se može pristupiti izvan klase, ako baš moramo, ali se ne preporučuje:

```
>>> R = Radnik ("Dražen", 3500)
>>> R . _Radnik__plaća                3500
>>> R . _Radnik__plaća += 500
>>> R . _Radnik__plaća                4000
```

NASLJEĐIVANJE

Često nailazimo na različite proizvode koji imaju osnovni model i napredni model s dodatnim značajkama iznad i ispod osnovnog modela. Pristup OOP-a za modeliranje softvera omogućava proširivanje mogućnosti postojeće klase za izgradnju nove klase, umjesto da se gradi ispočetka. U OOP terminologiji ta se karakteristika naziva nasljeđivanje, postojeća klasa naziva se osnovna ili roditeljska klasa, dok se nova klasa naziva podređena ili podklasa.

Nasljeđivanje dolazi na vidjelo kada nova klasa ima odnos „je" s postojećom klasom.

Pas je životinja. I mačka je životinja. Dakle, životinja je osnovna klasa, dok su psi i mačke naslijeđeni. Četverokut ima četiri stranice. Pravokutnik je četverokut, pa je tako i kvadrat. Četverokut je osnovna klasa (koja se također naziva roditeljska ili nadređena klasa), dok su pravokutnik i kvadrat naslijeđene klase - koje se nazivaju i podređene klase.

Podređena (child) klasa nasljeđuje definicije podataka i metode od nadređene klase. To olakšava ponovnu upotrebu značajki koje su već dostupne. Podređena klasa može dodati još nekoliko definicija ili redefinirati metodu osnovne klase.

Ova se značajka naziva višestruko nasljeđivanje. Izuzetno je korisna u izgradnji hijerarhije klasa za objekte u sustavu. Također je moguće dizajnirati novu klasu koja se temelji na više postojećih klasa. U nastavku je prikazan opći mehanizam uspostavljanja nasljeđivanja:

```
class roditelj :
    naredbe
class dijete (roditelj) :
    naredbe
```

Pri definiranju klase djeteta, ime nadređene klase stavlja se u zagrade ispred njega, što označava odnos između njih. Atributi i metode instance definirane u roditeljskoj klasi naslijedit će objekt podređene klase. Da bismo to predočili, pogledajmo sljedeći primjer. Prvo se definira klasa `četverokut` koja se koristi kao osnovna klasa klase četverokuta:

```
>>> class četverokut :
    def __init__ (s, a, b, c, d):
      s.a = a; s.b = b; s.c = c; s.d = d
    def opseg (s) :
      O = s.a +s.b +s.c +s.d
      m = max( s.a, s.b, s.c, s.d )
      print(
        ("O = " +str(O)) if O-m > m else
         "nije četverokut!" )
```

Klasa `četverokut` ima četiri stranice kao varijable instance i metodu `opseg()` za izračunavanje i ispis opsega ako odnos stranica zadovoljava uvjet četverokuta. Konstruktor, metoda `__init__()`, u kojoj smo umjesto uobičajenog imena `self` koristili ime `s`, prima četiri parametra i dodjeljuje ih četirima varijablama instance. Da bismo testirali klasu, definirajmo njezine objekte `X` i `Y` i pozovimo metodu `opseg()`:

```
>>> X = četverokut (10, 15, 20, 25)
>>> X.opseg()                          O = 70
>>> Y = četverokut (10, 15, 20, 55)
>>> Y.opseg()              nije četverokut!
```

Funkcija super()

Funkcija `super()` čini nasljeđivanje klasa upravljivijim i proširivijim. Funkcija vraća privremeni objekt koji omogućuje referencu na roditeljsku klasu pomoću ključne riječi `super`. Funkcija `super()` ima dva glavna slučaja upotrebe:

- Da se izričito izbjegne korištenje klase super (roditelj).
- Omogućiti više nasljeđivanja.

Budući da su suprotne strane pravokutnika jednake, potrebne su nam samo dvije susjedne strane kako bismo izgradili njegov objekt. Dakle, ostala dva parametra metode `__init__()` postavljena su na `None`. Metoda `__init__()` prosljeđuje parametre

konstruktoru svoje osnovne (četverostrane) klase koristeći funkciju `super()`. Objekt se inicijalizira sa c i d postavljenim na None. Nasuprotne strane izjednačene su s konstruktorom klase pravokutnika. Treba imati na umu da je ona automatski naslijedila metodu `opseg()`, stoga je nema potrebe redefinirati.

```
>>> class pravokutnik (četverokut) :
      def __init__ (s, a, b) :
        super().__init__ (a, b, a, b)

>>> P = pravokutnik (30, 20)
>>> P.opseg()                          O = 100
```

Možemo definirati i klasu `kvadrat` kao podklasu klase `pravokutnik`:

```
>>> class kvadrat (pravokutnik) :
    def __init__ (self, a) :
      super().__init__ (a, a)
>>> Q = kvadrat (50); Q.opseg()   O = 200
```

POLIMORFIZAM

Iz prethodnih primjera vidimo kako se resursi osnovne klase ponovno koriste tijekom konstruiranja naslijeđene klase.

Međutim, ako je potrebno, možemo izmijeniti funkcionalnost bilo koje metode osnovne klase. U tu svrhu naslijeđena klasa sadrži novu definiciju metode (s istim imenom i oznakom koji su već prisutni u osnovnoj klasi). Naravno, objekt nove klase imat će pristup objema metodama, ali onaj iz vlastite klase imat će prednost kada bude pozvan. To se naziva polimorfizam (preobličenje ili višeobličje).

Na primjer, prvo ćemo definirati novu metodu pod nazivom `površina()` u klasi `pravokutnik` i koristiti je kao bazu za klasu `kvadrat`:

```
>>> class pravokutnik (četverokut) :
    def __init__ (s, a, b) :
      super().__init__ (a, b, a, b)

    def površina (s) : return s.a *s.b
>>> X = pravokutnik (30, 25)
>>> X.površina()                        750
```

Dodajmo metodu `površina()` klasi kvadrat koja nasljeđuje klasu pravokutnik. Metoda `površina()` se preobličuje za primjenu formule za površinu kvadrata kao kvadrat njegovih stranica:

```
>>> class kvadrat (pravokutnik) :
    def __init__ (s, a) :
      super().__init__ (a, a)

    def površina (s) : return s.a **2
>>> Y = kvadrat (50)
>>> Y.opseg()                        O = 200
>>> Y.površina()                      2500
```

BRISANJE ATRIBUTA I OBJEKATA

Bilo koji atribut objekta neke klase ili sam objekt mogu biti obrisani naredbom DEL. Primjeri:

```
>>> class T :
    def __init__ (s, x, y):
        s.x = x; s.y = y
>>> A = T (1, 2); A.x, A.y          (1, 2)
>>> del A.y;  A.y
AttributeError: 'T' object has no
attribute 'y'
>>> B = T (5, 1); B.x, B.y          (5, 1)
>>> A = B;  A.x, A.y                (5, 1)
>>> del B
>>> B.x
NameError: name 'B' is not defined
>>> A.x, A.y                        (5, 1)
```

GOVORIMO PYTHONSKI

OBJEKTNO ORIJENTIRANO PROGRAMIRANJE

Objektno orijentirano programiranje ili kraće *OOP* je jedan od mogućih pristupa programiranju računala. Za razliku od ostalih pristupa, u kojima je težište na akcijama koje se vrše na podatkovnim strukturama, ovdje je težište na projektiranju aplikacije kao skupa objekata koji izmjenjuju poruke između sebe. Mi imamo privilegiju raditi u Pythonu koji je 100% objektno orijentirani jezik i to činimo od početka ove knjige. No, prije nego što nastavimo, evo malo povijesti vezano za priču o objektnom programiranju.

„Svakako, nije svaki dobar program objektno orijentiran i nije svaki objektno orijentiran program dobar." (Bjarne Stroustrup, danski informatičar, najpoznatiji po stvaranju i razvoju široko korištenog programskog jezika C ++) .

„Objektno orijentirano programiranje izuzetno je loša ideja koja je mogla nastati samo u Kaliforniji." (Edsger Dijkstra, (nizozemski računalni znanstvenik, 1930.-2002.). Dijkstra je također rekao: „... ono što društvo pretežno traži je zmijsko ulje. Naravno, zmijsko ulje ima najupečatljivija imena - inače ne biste ništa prodali – poput „Strukturirane analize i dizajna", „Softversko inženjerstvo", „Modeli zrelosti", „Upravljački informacijski sustavi", „Integrirana okruženja za podršku projektima", „Orijentacija prema objektima" i „Reinženjering poslovnih procesa".

Sve gore navedeno Dijkstra je izrekao sedamdesetih godina prošloga stoljeća. A on je bio taj koji je u svojoj knjizi „A Discipline of Programming", [Dij1976], definirao jedan mini jezik koji je imao strukturu polja koja je, u današnjoj terminologiji, bila klasa!

Mi ćemo se suzdržati u procjeni točnosti Dijkstrinih stavova. Jedino ćemo napomenuti da je Dijkstra u svojoj knjizi, [Dij1976], smislio jezik koji je ipak sadržavao koncepte objektno orijentiranog jezika. Predprocesor tog jezika, nazvan DDH, realiziran je u Pythonu s generiranjem objektnog koda također u Pythonu, [Dov2013]. Modul `arr.py` sadrži klasu `array` koja predstavlja implementaciju strukture polja iz Dijkstrinog jezika u Pythonu s izravnim prevođenjem. Sve smo to obradili u petnaestom poglavlju.

Pretpostavimo da trebamo izračunati opseg i površinu trokuta zadanog s tri točke `A(Ax,Ay)`, `B(Bx,By)` i `C(Cx,Cy)`.

```
A = Ax, Ay = eval (input (
          'Zadaj koordinate točke A '))
B = Bx, By = eval (input (
          'Zadaj koordinate točke B '))
C = Cx, Cy = eval (input (
          'Zadaj koordinate točke C '))
```

Površina trokuta sa zadanim vrhovima može se dobiti koristeći formulu:

```
P = ( Ax*(By-Cy) +Bx*(Cy-Ay)
     +Cx*(Ay-By))/2
```

a opseg trokuta sa zadanim vrhovima može se dobiti koristeći formulu:

```
O = a +b +c
```

gdje su: `a` stranica nasuprot vrha `A`, `b` nasuprot vrha `B` i `c` nasuprot vrha `C`, a mogu se dobiti

```
d = lambda X, Y : round ( ((X[0]
  -Y[0])**2 +(X[1] -Y[1])**2) **0.5, 4 )
```

pa možemo napisati prvu verziju programa:

```
# Trokut_1.py
# Površina i opseg trokuta s vrhovima
# A, B i C
from Moj_modul import *
d = lambda X, Y : round (
    ((X[0] -Y[0])**2
    +(X[1] -Y[1])**2) **0.5, 4 )
A = Ax, Ay = Input('Koordinate točke A ')
B = Bx, By = Input('Koordinate točke B ')
C = Cx, Cy = Input('Koordinate točke C ')
def P () : return abs(
           Ax*(By-Cy) +Bx*(Cy-Ay)
          +Cx*(Ay-By))/2
def O () : return (
           d(B,C) +d(A,C) +d(A,B) )
print ('O =', O(), 'P =', P())
```

```
>>>
Koordinate točke A -1, -2
Koordinate točke B 0, 2
Koordinate točke C 3, 1
O = 12.2854 P = 6.5
```

Pogledajmo drugu verziju:

```
# Trokut_2.py
# Površina i opseg trokuta s vrhovima
# A, B i C
d = lambda X, Y : \
    round (((X[0] -Y[0])**2
           +(X[1] -Y[1])**2) **0.5, 4)
T = {}; X = 'ABC'
for Y in X : T[Y] = eval (input (
  'Koordinate točke ' +Y +' '))
for Y in X : exec (
  Y +" = " +Y +"x, " +Y +
  "y = T['" +Y +"']" )
def P () :
  return abs (Ax*(By-Cy) +Bx*(Cy-Ay)
             +Cx*(Ay-By))/2.0
def O () : return d(B,C) +d(A,C) +d(A,B)
print ('O =', O(), 'P =', P())
```

```
>>>
Koordinate točke A -1, -2
Koordinate točke B 0, 2
Koordinate točke C 3, 1
O = 12.2854 P = 6.5
```

No, možemo razmišljati i na sljedeći način: Trokut je objekt koji pripada klasi svih trokuta. Opis te klase sadržavat će tri stranice (`a`, `b` i `c`).

```
# Trokut_3.py
# Površina trokuta s vrhovima A, B i C
class Trokut:
  def __init__ (o, A, B, C) :
    d = lambda X, Y : round (
        ((X[0] -Y[0])**2
        +(X[1] -Y[1])**2) **0.5, 4)
    o.A, o.B, o.C = A, B, C
    o.Ax, o.Ay    = o.A
    o.Bx, o.By    = o.B
    o.Cx, o.Cy    = o.C
    o.a, o.b, o.c = ( d(o.B, o.C),
         d(o.A, o.C), d(o.A, o.B) )
    o.O = o.opseg()
    o.P = o.površina()
  def opseg (o)    : return round (
      o.a +o.b +o.c, 4)
  def površina (o) :
      s = o.O /2; return round ((s*(s-
      o.a)*(s-o.b)*(s-o.c))**0.5, 4)
A = eval (input ('Koordinate točke A '))
B = eval (input ('Koordinate točke B '))
C = eval (input ('Koordinate točke C '))

T = Trokut(A, B, C)
print ('O =', T.O, 'P =', T.P)
'     ili     '
print ('O =', T.opseg(), 'P =',
       T.površina())
```

```
>>>
Koordinate točke A -1, -2
Koordinate točke B 0, 2
Koordinate točke C 3, 1
O = 12.2854 P = 6.5
O = 12.2854 P = 6.5
>>> type (T)  <class '__main__.Trokut'>
>>> T. A        (-1, -2)
>>> T. P        6.5
```

Ušli smo u svijet klasa i objekata čime nam se otvorila nova dimenzija u pisanju naših programa! No, prije toga evo nekoliko analiza i preporuka „objektnog govora".

Dosad smo namjerno izbjegavali bavljenje objektno orijentiranim programiranjem (OOP) u prethodnim poglavljima ove knjige. Preskočili smo OOP, jer smo uvjereni da je lakše i zabavnije započeti učenje Pythona, a da ne moramo znati sve detalje objektno orijentiranog programiranja.

Iako smo izbjegli OOP, ono je uvijek bilo prisutno u vježbama i primjerima našeg tečaja. Koristili smo objekte i metode iz klase bez potpunog objašnjavanja njihove OOP pozadine. U ovom ćemo poglavlju zaključiti ono što je dosad nedostajalo. Donijet ćemo uvod u principe objektno orijentiranog programiranja općenito i specifičnosti OOP pristupa Pythona. OOP je jedan od najmoćnijih alata Pythona, ali bez obzira na to ne morate ga koristiti, tj. možete i bez njega pisati moćne i učinkovite programe.

REDOSLJED METODA UNUTAR KLASE

Na primjeru klase `Točka` rezimirajmo kako izgleda struktura njezine definicije gdje koristimo ime "_" za *self*. Također prikazujemo da redosljed metoda unutar klase nije bitan.

Točka.py

```
# Klasa Točka
class Točka:
  # [dokument]
  # [atributi klase]

  def d ( _ ) :
    return round(
           (_.x**2 +_.y**2)**0.5, 2)

  def označi ( _, s = '' ) :
    return (s +'(' +str(_.x) +', '
           +str(_.y) +')' )

  def __init__ (_, t, ime = '') :
    _.x, _.y = t
    _.oznaka = _.označi (ime)
X = eval (input ('Koordinate točke > '))
A = Točka (X, 'A')
print (A.oznaka + ', d =', A.d())
```

```
>>>
Koordinate točke > 2, 3
A(2, 3), d = 3.61
```

Metode klasa standardnih primitivnih i složenih podataka

U sljedećem smo programu pokazali metode klasa standardnih primitivnih i složenih podataka Pythona i opis njihova značenja. n-torka `Klase` sadrži imena klasa.

```
# Metode_klasa.py
def Metode ( X ) :
  print (); DIR = eval ("dir(" +X +")")
  return [x for x in DIR if x[0] != '_' ]
```

```
Klase = ('int','float','complex','bool',
  'str', 'tuple', 'list', 'set', 'dict' )
while 'klasa' : # izbor klase
  print ('\nIzaberi klasu: \n')
  for k in range (len (Klase)) :
     print (k, Klase[k])
  print ()
  i = input ('broj? ')
  if not i : break
  i = eval (i)
  if (type (i) != int or type(i) == int
    and i not in range(0, len(Klase))):
    print ('Van domene!'); continue
  Klasa = Klase[i];  M = Metode (Klasa)
  print ('\nOpis metoda klase '
         +Klasa +'\n')
  for i, s in enumerate (M): print (i, s)
  print ()
  while 'metoda' : # izbor metode
    i = input (
        'Izaberi broj ispred metode '
       +'(Enter za prekid): ')
    if not i : break
    i = eval (i)
    if (type (i) == int and
        i in range (0, len(M) )):
      exec( "help (" + Klasa +"."
           + M[i] + ")")
```

```
>>>
Izaberi klasu:
0 int
1 float
2 complex
3 bool
4 str
5 tuple
6 list
7 set
8 dict
broj? 0
Opis metoda klase int

0 as_integer_ratio
1 bit_length
2 conjugate
3 denominator
4 from_bytes
5 imag
6 numerator
7 real
8 to_bytes

Izaberi broj ispred metode (Enter za
prekid): 6
...
numerator
    the numerator of a rational number
    in lowest terms
```

POLIMORFIZAM

Evo jednog jednostavnog primjera polimorfizma.

```
# Polimorfizam.py
class mačka :
    def mjauče (self):
        print ("mačka može mjaukati")
    def laje (self):
        print ("mačka ne može lajati")
class pas :
    def mjauče (self):
        print ("pas ne mjauče")
    def laje (self):
        print ("pas može lajati")

# zajednička svojstva
def mjauče_li ( x ):  x.mjauče ()
def laje_li   ( x ):  x.laje   ()

#instanciranje objekata
Jojo = mačka ()
Noa  = pas   ()

# testitanje
mjauče_li ( Jojo )
mjauče_li ( Noa )
laje_li   ( Jojo )
laje_li   ( Noa )
```

```
>>>
mačka može mjaukati
pas ne mjauče
mačka ne može lajati
pas može lajati
```

DEFINICIJA STRUKTURE SLOGA

Slog (record) kao složena struktura podataka bila je poznata u jezicima Pascalu i Delphiju. Objedinjavala je sve proste i složene tipove podataka tih jezika čija su se imena pojavljivala kao "atributi" unutar definicije sloga. Evo primjera klase i jednog objekta:

```
class RECORD :
  def __init__ ( s, x, y ):
    for i in range (len (x)) : exec (
        's.' +x[i] +' = ' +'str(y[i])')
>>> atr = ('id', 'kat_br', 'naziv', 'nc',
          'vpc', 'kol', 'JM'),
```

```
>>> A   = [1, "112/370", "AMORTIZER",
           247.65, 310.86, 0.00, "kom"]
>>> ART = RECORD (art, A); ART.naziv
'AMORTIZER'
>>> ART.vpc     310.86
```

P R O G R A M I

Površina i opseg trokuta (3)

Dajemo još jednu inačicu programa za računanje površine i opsega trokuta koja koristi dvije klase. Točke trokuta kao objekti klase točka prenose se kao argumenti za definiranje trokuta kao objekta klase trokut.

Trokut.py

```
# KLASE (točka i trokut)
from Moj_modul import *
class točka:
  def __init__ (_, t, ozn = '') :
    _.x, _.y = t
    _.ozn = (ozn +'(' +str(_.x)
               +', ' +str(_.y) +')')
  def d (_) :
    return sqrt (_.x**2 +_.y**2)
  def set_ozn (_, s = '') :
    _.ozn = ( s +'(' +str(_.x) +', '
                +str(_.y) +')' )
class trokut:
  def __init__ (o, A, B, C) :

    d = lambda X, Y : (
      round (((X.x -Y.x)**2 +
              (X.y -Y.y)**2) **0.5, 4))
    o.A, o.B, o.C = A, B, C
    o.a, o.b, o.c = (d(o.B, o.C),
         d(o.A, o.C), d(o.A, o.B))
    o.O = o.opseg(); o.P = o.površina()
   def opseg(o):
     return round (o.a +o.b +o.c, 4)
   def površina(o):
     s = o.opseg () /2
     return round ( (s*(s-o.a)
          *(s-o.b)*(s-o.c))**0.5, 4)
A = Input ('Koordinate točke A ')
A = točka (A, 'A')
B = Input ('Koordinate točke B ')
B = točka (B, 'B')
C = Input ('Koordinate točke C ')
C = točka (C, 'C')
T = trokut(A, B, C)
print ( 'O =', T.O, 'P =', T.P )
```

```
>>>
Koordinate točke A -1, -2
Koordinate točke B 0, 2
Koordinate točke C 3, 1
O = 12.2854 P = 6.5
>>>
>>> A.ozn                    'A(-1, -2)'
>>> A.d()          (2.23606797749979+0j)
>>> B.d().real                      2.0
```

Periodni sustav elemenata (2)

Na primjeru periodnog sustava elemenata pokazat ćemo kako se simboli kemijskih elemenata mogu „proglasiti" objektima. Imena tih objekata bit će jednaka simbolima elemenata, a atributi će im biti:

- rb - redni broj,
- m - relativna atomska masa,
- hn - hrvatski naziv i
- ln - latinski naziv

PSE.py

```
class PSE :
  def __init__ (s, S, RB, Rm, Hn, Ln) :
    s._ = (RB, S, Rm, Hn, Ln)
    s.rb, s.m, s.hn, s.ln  = ( RB, Rm,
                               Hn, Ln )
# UČITAJ PERIODNI SUSTAV
Dat = 'Per-SUS.TXT';  PS0 = []
try  :
  for line in open (Dat, 'r') :
      PS0.append (line[:-1])
except :  print (
  'NE POSTOJI DATOTEKA', Dat);  quit()

# 'RAZBIJ' PO STUPCIMA
ps = []
for x in PS0 : ps.append (x.split())
# Oformi objekte kemijskih elemenata
for i in range (len(ps)):
  Y  = ps[i]
  exec( Y[1] +" = PSE (Y[1], int (Y[0]),
        float(Y[2]), Y[3], Y[4] )" )
```

```
>>> H.__dict__
{'_':   (1,   'H',   1.008,   'vodik',
'hydrogen'), 'rb': 1, 'm': 1.008, 'hn':
'vodik', 'ln': 'hydrogen'}
>>> H.m    1.008      >>> H.hn   'vodik'
```

MJENJAČNICA (2)

Priloženi program mjenjačnice razlikuje se od programa Mjenjačnica.py danog u osmom poglavlju u definiciji pojedinih valuta kao objekte klase RECORD. Valute imaju atribute dane u *n*-torci atr, pa je formula za pretvorbu iznosa iz jedne u drugu valutu preglednija. Program radi s aktualnom tečajnom listom koju treba sa stranice HNB upamtiti pod nazivom TL.txt, kao što je objašnjeno u osmom poglavlju. Dijelove koje nismo promijenili dali smo kao komentar.

Mjenjačnica_2.py

```
from Moj_modul import *

class RECORD :
  def __init__ ( s, x, y ):
    for i in range (len (x)) :
      exec ('s.' +x[i] +' = '
            +'str(y[i])')

# UČITAJ TEČAJNU LISTU U TL

# OFORMI TEČAJNU LISTU

# 'RAZBIJ' TEČAJNU LISTU

# ISPIŠI TEČAJNU LISTU

# MJENJAČNICA
while True :
  try :
    s = input('Redni broj ulazne valute,'
    ' iznos i redni broj izlazne valute')
    if not s : break
    atr = ('val', 'šif', 'par',
           'kup', 'sre', 'pro')
    i, X, j = eval (s)
    if (0 <= i < len(TL) and
        0 <= j < len(TL) ) :
      # UL, IZ - ulazna/izlazna lista
      UL = RECORD (atr, TL[i])
      IZ = RECORD (atr, TL[j])
      print ( X, UL.val, '=',
         round (X *1/int (UL.par)
         *eval (UL.kup +'/' +IZ.pro), 2),
         IZ.val )
    else : print ('greška, redni broj '
                  'valute?' )
  except : print ('POGREŠKA PRI UNOSU '
                  'PODATAKA!' ); continue
```

```
>>>
Tečajna lista na dan:  27.05.2021.

RB Val Šif Par  Kupovni  Srednji  Prodajni
-----------------------------------------
...
13 EUR 978 001  7.483419 7.505937 7.528455
...
Redni broj ulazne valute, iznos i redni
broj izlazne valute 13, 1000, 0
1000 EUR = 7483.42 HRK
```

ZBRAJANJE VEKTORA

U trećem smo poglavlju u programu za izračunavanje rezultante dviju sila, **Rezultanta.py**, problem riješili svodeći ga na zbrajanje dvaju vektora, interpretiranih kao kompleksni brojevi. Evo rješenja primjenom klase vektor u kojoj su vektori objekti s atributima x i y (komponente vektora po osi x i y). Priloženi program pokazuje kako se mogu koristiti ugrađene metode __str__ i __add__. Uz pomoć metode _add_ definirali smo operaciju zbrajanja nad vektorima.

Vektor.py

```
from Moj_modul import Input
class vektor:
  def __init__ ( _1, a, b ):
    _1.x = a;  _1.y = b
  def __str__ ( _1 ):
    return '(%d, %d)' % (_1.x, _1.y)

  def __add__ ( _1, _2 ):
    return vektor(_1.x + _2.x,
                  _1.y + _2.y)

v1 = Input ('x, y prvog  vektora ')
v2 = Input ('x, y drugog vektora ')
v1 = vektor(*v1);  v2 = vektor(*v2)
print ("v1 =", v1, "v2 =", v2, '\n' +
       "v1 + v2 =", v1 + v2)
```

```
>>>
x, y prvog  vektora 1, 1
x, y drugog vektora -2, 10
v1 = (1, 1) v2 = (-2, 10)
v1 + v2 = (-1, 11)
```

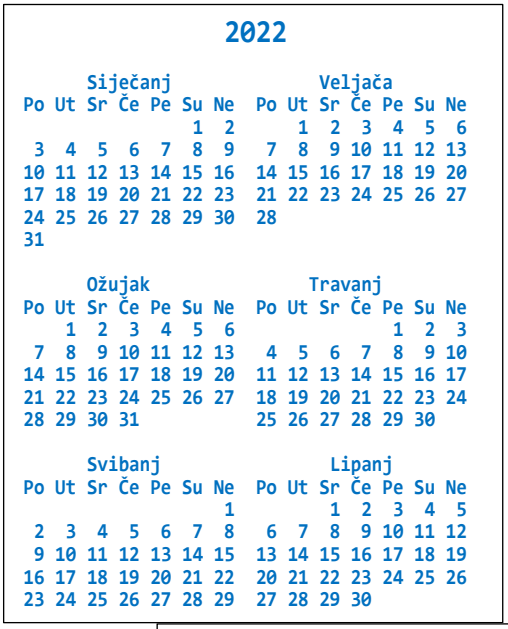
12.

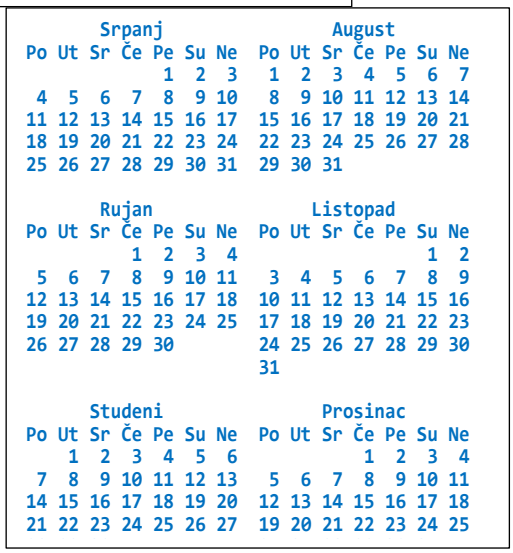
MODULI

Python dopušta grupiranje potprograma, podataka i klasa u posebne cjeline – module i pakete. To su kompilacijske cjeline koje se mogu uključiti u bilo koji program ili drugi modul.

Postoji veliki broj standardnih modula. Neke od njih smo već koristili. Ovdje ćemo opisati još desetak standardnih modula.

Mogu se definirati i vlastiti moduli. Već odavno imamo svoj modul, `Moj_modul.py`*, koji je dobar primjer kako definirati vlastiti modul.*





Uvod

Modul se može promatrati kao biblioteka kodova. Sadrži skup funkcija koje možemo uključiti u svoju aplikaciju.

Pythonova standardna biblioteka programa sadrži preko dvije stotine modula čiji broj varira od jedne do druge inačice. Dakako, ne preporučuju se svi ti moduli za one koji uče Python, jer su mnogi namijenjeni za posebne uporabe, uglavnom za programere koji rade na samom Pythonu.

Neki drugi moduli ostaci su starijih Pythonovih inačica zamijenjenih suvremenijim alternativama, a zadržani su uglavnom zbog kompatibilnosti sa starim kodom. Neke od njih smo koristili u prethodnim poglavljima:

.

math	Dodatne matematičke funkcije (`sin`, `cos`, `sqrt`, ...) i konstante (`e`, `pi`, ...).
cmath	Matematičke funkcije za rad s kompleksnim brojevima.
itertools	Funkcije za stvaranje iteratora učinkovitih petlji .
picle	Pretvorba objekata Pythona u nizove bajtova i natrag.
random	Generator slučajnih brojeva.
shelve	Pohrana podataka u "police".
string	Podaci i operacije sa stringovima.

Ovdje ćemo navesti samo dio standardnih modula koji se preporučuju za najčešću uporabu. Dani su u sljedećoj tablici, u alfabetskom uređenju njihovih imena.

Ime	*Opis*
calendar	Modul za rad s kalendarima, uključujući neke emulacije UNIX programa.
collections	Zbirka tipova podataka.
datatime	Funkcije za rad s datumom i vremenom.
json	Enkodiranje i dekodiranje u JSON format podataka.
os.path	Funkcije za manipulaciju i testiranje putova datoteka.
re	Operacije s regularnim izrazima.
urllib	Obrada URL zahtjeva.
webbrowser	Kontroler web preglednika.

U nastavku poglavlja opisujemo ukratko ove module. S druge strane, već smo rekli da je proučavanje Pythona veliki izazov, pa detaljnijije opise prepuštamo vama.

Operativni sistem

Postoje dva Pythonova standardna modula koja se odnose na vezu s operativnim sustavom, `sys` i `os`. Ovdje ćemo opisati samo `os`, njegovu podklasu `os.path`.

os.path

Ovaj modul sadrži neke korisne funkcije na imenima staza. Parametri puta su stringovi ili bajtovi. Funkcije se koriste u različite svrhe, poput spajanja, normaliziranja i dohvaćanja imena staza u Pythonu. Sve ove funkcije kao svoje parametre prihvaćaju ili samo bajtove ili samo string objekte. Rezultat je objekt iste vrste ako se vrati put ili ime datoteke. Kako postoje različite verzije operativnog sustava, tako postoji i nekoliko verzija ovog modula u standardnoj biblioteci. Slijede neke najčešće rabljene funkcije koje imaju samo jedan parametar *put* (*path*) tipa `str` i uz pretpostavku da smo uvezli `os.path` sa:

```
>>> from os.path import *
```

basename (*put*)

Koristi se za vraćanje osnovnog imena datoteke sa zadane staze.

```
>>> basename ("/PODACI/test.txt")
'test.txt'
```

dirname (*put*)

Vraća ime direktorija (foldera) sa zadane staze, bez imena staze.

```
>>> dirname ("/PODACI/test.txt")
'/PODACI'
```

isabs (*put*)

Logička funkcija koja vraća `True` ako je put apsolutan, inače `False`. U Windowsima put počinje s „\“ nakon eventualnog slova pogona.

```
>>> isabs ("/PODACI/test.txt")
True
```

isdir (*put*)

Logička funkcija koja specificira pripada li put direktoriju ili ne.

```
>>> isdir ("C:\\Users\Public")
True
```

isfile (*put*)

Logička funkcija koja provjerava postoji li put ili ne.

```
>>> isfile ("C:\\Users\Public")
False
>>> isfile ("C:\\Python39\zvijezda.py")
True
```

normcase (*put*)

Ova funkcija normalizira slučaj navedenog puta. U sustavu Windowsa pretvara put u mala slova, a kosu crtu u obrnutu.

```
>>> normcase("C":/Python39/zvijezda.py")
'c:\\python39\\zvijezda.py'
```

normpath (*put*)

Ova funkcija normalizira nazive staza sažimanjem suvišnih separatora i referencija na višoj razini, tako da `A//B`, `A/B/` i `A/./B` postanu `A/B`. U sustavu Windowsa pretvara kose crte prema natrag u kose crte.

```
>>> normpath("C:\Python39\zvijezda.py")
'C:\\Python39\\zvijezda.py'
```

Regularni izrazi

Povijest regularnih izraza dio je rane povijesti formalnih jezika i dugo su vremena bili predmetom teorijskih istraživanja vezanih za regularne skupove (jezike). Matematičar Stephen Kleene je 1950-ih opisao ove modele koristeći matematičku notaciju zvanu regularni skupovi.

Napisane su mnoge knjige o primjeni regularnih izraza, što opet ukazuje na njihovu sve veću važnost. Često se regularni izrazi koji se koriste u analizi (prepoznavanju) nizova znakova u nekom ulaznom nizu (tekstu) nazivaju uzorak (ili *pattern*). Realizacija programa za pretraživanje ulaznog niza upravljana regularnim izrazom (RE) naziva se motor (*engine*) regularnih izraza. U biti je to konačni automat, odnosno, konačni prepoznavač., [Dov2012a] i [Dov2012b].

Značenje regularnih izraza je sparivanje određenih nizova (riječi), u skladu s određenim pravilima. Koriste ih mnogi programi za uređivenje teksta, programi za pretragu i manipuliranje nizovima. Može se reći da se danas „regularni izraz", koji se još naziva „uzorak" (engl. *pattern*), rabi za opis (označivanje) skupa nizova znakova bez davanja njegova precizna značenja. Na primjer, skup koji sadrži četiri niza Mirko, Miro, Marko i Maro može se opisati regularnim izrazom ili uzorkom M(i|a)r(k?)o. Ovdje su "|" i "?" metaznakovi koji imaju značenje "+" i "*".

SINTAKSA

Počnimo opisom najjednostavnijih regularnih izraza: sparivanjem znakova. Mnogi znakovi jednostavno sparuju sami sebe. Na primjer, regularni izraz `123` sparit će egzaktno niz `123`. Ako niz sadrži slova, velika ili mala, sparivanje će također biti jednoznačno. Na primjer, regularni izraz `Python` sparit će niz `Python`, ali neće `PYTHON` niti `python`. Ako želimo da spari i te nizove, koristit ćemo opciju (*case-insensitive*) u kojoj je značenje velikih i malih slova engleskog alfabeta jednako.

Postoji izuzetak u primjeni tog pravila. Neki znakovi ASCII skupa znakova imaju posebno značenje i ne sparuju sami sebe. Nazivamo ih *meta-simboli* . To su:

```
.  ^  $  *  +  ?  {  }  [  ]  \  |  (  )
```

Na primjer, (i) imaju značenje kao i u osnovnoj definiciji ili pravilima pisanja regularnih izraza. Označuju podizraz („blok") regularnog izraza.

Ako neterminale označimo velikim kosim slovima, gdje je *R* startni simbol, općenito se pisanje regularnih izraza može prikazati sljedećim skupom produkcija:

```
R → S| B| Z| \Y| [N]| [-N]| [N-]| (R)|
    RK| RA| PR| RR

S → a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|
    p|q|r|s|t|u|v|w|x|y|z|
    A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|
    Q|R|S|T|U|V|W|X|Y|Z|SS

B → 0|1|2|3|4|5|6|7|8|9|BB

Z → -| _| "| #| $| %| =| @| .

Y → [| ]| \| ^| $| .| ||| ?| *| +|
    (| )| {| }| d| n| w| E| Q| W

N → S-S| B-B| NN| X          P → ^| ?

X → S| B| Z| Y          K → *| ?| +| $

A → {B}| {B,B}| {,B}| {B,}
```

META ZNAKOVI

Meta znakovi su znakovi s posebnim značenjem. Opisani su u sljedećoj tablici.

Znak	Opis	Primjeri
[]	Skup znakova.	"[a-f]"
\	Signalizira posebnu sekvencu (može se koristiti i za ESC sekvence)	"\d" "\w"
.	Bilo koji znak (osim znaka za novu liniju)	"Python."
^	Početak stringa sa.	"^Python"
$	Završetak stringa sa.	"kraj$"
*	Nula ili više pojava.	"a*"
+	Jedna ili više pojava.	"ai+"
{}	Navedeni broj pojava.	"al{2}"
|	Ili.	"true|false"
()	Grupa više mogućnosti.	

POSEBNE SEKVENCE

Posebna sekvenca je \ iza kojeg slijedi jedan od znakova koji ima posebno značenje, dan u sljedećoj tablici.

Znak	Opis	Primjeri
\A	Vraća podudaranje ako su navedeni znakovi na početku niza.	"\AThe"
\b	Vraća podudaranje gdje su navedeni znakovi na početku ili na kraju riječi ("r" na početku osigurava da se niz tretira kao "sirovi niz").	r"\bain" r"ain\b"
\B	Vraća podudaranje u kojem su navedeni znakovi, ali NE na početku (ili na kraju) riječi ("r" na početku osigurava da se niz tretira kao "sirovi niz").	r"\Bain" r"ain\B"
\d	Vraća podudaranje gdje niz sadrži znamenke (brojke od 0 do 9).	"\d"
\D	Vraća podudaranje u kojem niz NE sadrži znamenke.	"\D"
\s	Vraća podudaranje gdje niz sadrži razmak.	"\s"
\S	Vraća podudaranje u kojem niz NE sadrži razmak.	"\S"
\w	Vraća podudaranje u kojem niz sadrži bilo koje znakove riječi (znakovi od a do Z, znamenke od 0 do 9 i donji znak _)	"\w"
\W	Vraća podudaranje u kojem niz NE sadrži znakove riječi.	"\W"
\Z	Vraća podudaranje ako su navedeni znakovi na kraju niza.	"ing\Z"

SKUP

Skup je skup znakova unutar para uglatih zagrada [] s posebnim značenjem.

Skup	Opis
[arn]	Vraća podudaranje gdje je prisutan jedan od navedenih znakova (a, r ili n)
[a-f]	Vraća podudaranje bilo kojeg malog slova, abecedno između a i f.
[^arn]	Vraća podudaranje za bilo koji znak OSIM a, r i n.
[0123]	Vraća podudaranje gdje je prisutna bilo koja od navedenih znamenki (0, 1, 2 ili 3)
[0-9]	Vraća podudaranje za bilo koju brojku između 0 i 9.
[0-5][0-9]	Vraća podudaranje za bilo koji broj od 00 do 99.
[a-zA-Z]	Vraća podudaranje za bilo koje malo ili veliko slovo ASCII uređenja.
[+]	U skupovima, +, *, ., |, (), $, {} nema posebno značenje, pa [+] znači: vratiti podudaranje za bilo koji + znak u nizu.

MODUL re

Sve metode ovog modula poslije njegovog uvoza možemo dobiti na uobičajeni način, izvršenjem *naredbe DIR*:

```
>>> import re
>>> dir (re)
['A', 'ASCII', ..., 'compile', 'copyreg',
'enum', 'error', 'escape', 'findall',
'finditer', 'fullmatch', 'functools',
'match', 'purge', 'search', 'split',
'sre_compile', 'sre_parse', 'sub',
'subn', 'template']
```

Opisat ćemo samo funkcije modula re koje nam omogućuju traženje niza za podudaranje (uparenje):

- findall
- search
- split
- sub

findall

Vraća listu koja sadrži sva podudaranja. Ako ih nema, vraća praznu listu.

```
>>> import re
>>> T = (
"Zagreb je glavni grad Hrvatske" )
>>> x = re.findall("Zagreb", T)
>>> print (x)            ['Zagreb']
>>> x = re.findall("a", T)
>>> print (x)  ['a', 'a', 'a', 'a']
>>> x = re.findall("Split", T)
>>> print (x)                    []
```

search

Vraća objekt podudaranja ako se nalazi bilo gdje u nizu. Inače, vraća None.

```
>>> x = re.search("\s", T)
>>> print (x)
<re.Match object; span=(6, 7),
match=' '>
```

Ako postoji više od jednog podudaranja, kao u prethodnom primjeru, vratit će se samo prvo.

split

Vraća listu gdje je niz podijeljen pri svakom podudaranju.

```
>>> x = re.split("\s", T); print (x)
['Zagreb',   'je',   'glavni',   'grad',
'Hrvatske']
```

Ovo ne treba miješati s funkcijom `split()` nad listama, jer značenje nije jednako. Na primjer:

```
>>> print (
    "Tri   razmaka". split (' ') )
['Tri', '', '', 'razmaka']
```

sub

Zamjenjuje jedno ili više podudaranja znakovnim nizom.

```
>>> X = "Tri   razmaka"
>>> x = re.sub("\s", "*", X); print (x)
Tri***razmaka
>>> re.sub("\s", "*", X, 2)
'Tri** razmaka'
```

Zbirke podataka

Poseban Pythonov standardni modul `collections` (zbirke) omogućuje spremanje posebnih tipova podataka. Spremnik je objekt koji se koristi za spremanje različitih objekata i pružanje načina za pristup sadržanim objektima i iteraciju nad njima. Neki od ugrađenih spremnika su n-torka, lista, rječnik (mapa) itd.

MODUL collections

Ovaj modul implementira neke „lijepe“ strukture podataka koje će vam pomoći u rješavanju različitih problema iz stvarnog života.

```
>>> import collections; dir (collections)
['ChainMap', 'Counter', 'OrderedDict',
'UserDict', 'UserList', 'UserString', ...
'defaultdict', 'deque', 'namedtuple']
```

Opisat ćemo samo neke metode.

Counter

`Counter` (brojač) je podklasa rječnika `dict` koja pomaže brojati njezine objekte . Unutar njega elementi se pohranjuju kao ključevi rječnika, a brojevi se pohranjuju kao vrijednosti koje mogu biti nula ili negativa.

Objekti brojača imaju metodu koja se naziva `elements` koja vraća iterator preko elemenata koji se ponavljaju. Elementi se vraćaju proizvoljnim redoslijedom.

```
>>> c = Counter(a=4, b=2, c=0, d=-2)
>>> list (c.elements())
['a', 'a', 'a', 'a', 'b', 'b']
```

`most_common()` je funkcija koja vraća najčešće elemente i njihov broj do onih koji se najmanje pojavljuju i njihov broj.

```
>>> Counter('abracadabra').most_common(3)
[('a', 5), ('r', 2), ('b', 2)]
```

defaultdict

`defaultdict` je objekt poput rječnika, koji sadrži sve metode koje nudi rječnik, ali uzima prvi argument (*default_factory*) kao zadani tip podataka za rječnik. Korištenje objekta defaultdict brže je od rada s `dict.set_default` metodom.

```
>>> s = [('yellow', 1), ('blue', 2),
         ('yellow', 3), ('blue', 4),
         ('red', 1)]
>>> d = defaultdict (list)
>>> for k, v in s: d[k].append(v)
>>> d.items()
[('blue', [2, 4]), ('red', [1]),
('yellow', [1, 3])]
```

U ovom primjeru može se vidjeti da čak i ako nema ključa `defaultdict`, u objektu se on automatski stvara prazna lista. `list.append` zatim pomaže dodavanju vrijednosti listi.

namedtuple

Imenovane *n*-torke pomažu u značenju svakog elementa u *n*-torci i omogućuju nam kodiranje uz bolju čitljivost i samo-dokumentiranje koda. Može ih se koristiti na bilo kojem mjestu gdje imamo *n*-torke. Pravilo pisanja je:

```
ime_n = namedtuple ( 'ime_n', atr, atr
                     {, atr} )
atr   : string
```

Korištenje `namedtuple` puno je kraće od definiranja klase. Na primjer:

```
>>> from collections import namedtuple
>>> Auto = namedtuple ('Auto',
   ('tip', 'boja',  'godina', 'km'))
>>> X = Auto ('Octavia', 'crna', 2014,
              169925)
>>> X.tip                          'Octavia'
>>> X.boja                            'crna'
>>> X.godina                            2014
>>> X.km                              169925
```

Vrijednosti atributa imenovane *n*-torke su nepromjenjive:

```
>>> X.km = 175000
AttributeError: "can't set attribute"
```

U sljedećem primjeru ćemo stvoriti imenovani par koji će prikazati informacije o zadržavanju bodova.

```
>>> from collections import namedtuple
>>> Point = namedtuple (
   'Point', ['x', 'y'] ) # Atributi
>>> p = Point (10, y=20) # Kreiranje
>>> p                    Point(x=10, y=20)
>>> p.x + p.y                          30
>>> p[0] + p[1] # Normalni pristup     30
>>> x, y = p    # Raspakiranje n-torke
>>> x                                  10
>>> y                                  20
```

OrderedDict

`OrderedDict` je također podrazred rječnika, ali za razliku od rječnika, pamti redoslijed umetanja ključeva. Pogledajmo sljedeći primjer:

OrderedDict.py

```
# Demonstracija rada OrderedDict
from collections import OrderedDict
print ("Ovo je dict:\n")
d = {}
d['a'] = 1; d['b'] = 2; d['c'] = 3
d['d'] = 4
for ključ, vrijednost in d.items():
    print (ključ, vrijednost)
print ("\nOvo je OrderedDict:\n")
od = OrderedDict()
od['a'] = 1; od['b'] = 2; od['c'] = 3
od['d'] = 4
for ključ, vrijednost in od.items():
    print (ključ, vrijednost)
```

```
>>>
Ovo je dict:

a 1
b 2
c 3
d 4

Ovo je OrderedDict:

a 1
b 2
c 3
d 4
```

Tijekom brisanja i ponovnog umetanja istog ključa umetnut će ključ kao zadnji kako bi se održao redoslijed umetanja ključa.

```
# Ukidanje i umetanje istog elementa
od.pop ('a');  od['a'] = 1
print('\nPoslije ukidanja i umetanja:')
for ključ, vrijednost in od.items():
    print (ključ, vrijednost)
```

```
>>>
Poslije ukidanja i umetanja:
b 2
c 3
d 4
a 1
```

Metoda ChainMap

Metoda **ChainMap** spaja više rječnika u jednu jedinicu i vraća listu rječnika. Sintaksa je:

```
class collections. ChainMap ( d {, d} )
d    : ime_rječnika
```

Sljedeći program demonstrira rad s `ChainMap`:

ChainMap.py

```
from collections import ChainMap
d1 = {'a': 1, 'b': 2}
d2 = {'c': 3, 'd': 4}
d3 = {'e': 5, 'f': 6}
# Definicija ChainMap
c = ChainMap(d1, d2, d3);  print (c)
```

```
>>>
ChainMap({'a': 1, 'b': 2}, {'c': 3,
'd': 4}, {'e': 5, 'f': 6})
```

Pristup ključevima i vrijednostima

Vrijednostima iz ChainMap-a može se pristupiti pomoću imena ključa. Također im se može pristupiti pomoću metode keys() i values().

ChainMap_2.py

```
from collections import ChainMap
d1 = {'a': 1, 'b': 2}
d2 = {'c': 3, 'd': 4}
d3 = {'e': 5, 'f': 6}
# Definiranje ChainMap
c = ChainMap(d1, d2, d3)
# Pristup vrijednostima s imenom ključa
print (c['a'])
# Pristup vrijednostima s values()
print (c.values())
# Pristup ključevima s keys()
print (c.keys())
```

```
>>>
1
ValuesView(ChainMap({‘a’: 1, ‘b’: 2},
{‘c’: 3, ‘d’: 4}, {‘e’: 5, ‘f’: 6}))
KeysView(ChainMap({‘a’: 1, ‘b’: 2},
{‘c’: 3, ‘d’: 4}, {‘e’: 5, ‘f’: 6}))
```

Dodavanje novog rječnika

Novi rječnik može biti dodan uporabom metode new_child(). Novo dodani rječnik dodaje se na početak ChainMap.

ChainMap_3.py

```
# ChainMap i new_child()
import collections
# inicijalizacija mapa
dic1 = { 'a' : 1, 'b' : 2 }
dic2 = { 'b' : 3, 'c' : 4 }
dic3 = { 'f' : 5 }
# inicijalizacija ChainMap
chain = collections.ChainMap (
        dic1, dic2 )
print ("ChainMap sadržaj: ")
print (chain)
# dodavanje novog rječnika
chain1 = chain.new_child (dic3)
print ("Prikaz novog ChainMap: ")
print (chain1)
```

```
>>>
ChainMap sadržaj:
ChainMap({'a': 1, 'b': 2}, {'b': 3,
'c': 4})
Prikaz novog ChainMap:
ChainMap({'f': 5}, {'a': 1, 'b': 2},
{'b': 3, 'c': 4})
```

Metoda deque

Metoda **deque** (dvostruko završen red) optimizirana je lista za brže dodavanje i izlazak s obje strane niza. Sintaksa je:

```
class collections . deque ( lista )
```

Ova funkcija ima listu kao argument.

```
>>> from collections import deque
>>> # Declaring deque
>>> de = deque([1, 2, 3])
>>> print (de)          deque([1, 2, 3])
```

Umetanje elemenata

Elementi se u deque mogu umetnuti s oba kraja. Za umetanje elemenata s desne strane koristi se metoda append(), a za umetanje elemenata s lijeve strane appendleft().

```
>>> de.append (4);       print (de)
deque([1, 2, 3, 4])
>>> de.appendleft (6);   print (de)
deque([6, 1, 2, 3, 4])
```

Uklanjanje elemenata

Elementi se također mogu ukloniti iz strukture deque s oba kraja. Za uklanjanje elemenata s desne strane koristi se metoda pop(), a za uklanjanje elemenata s lijeve strane metoda popleft().

```
>>> de.pop()                          4
>>> de                deque([6, 1, 2, 3])
>>> de.popleft()                      6
>>> de                   deque([1, 2, 3])
```

Datum i vrijeme

Dva su standardna modula koja imaju veliki broj funkcija i metoda za prikaz kalendara, rad s datumima i vremenom. To su calendar i datetime.

MODUL calendar

Modul **calendar** koji obrađuje operacije povezane s kalendarom. Ovaj modul omogućuje izlazne kalendare poput programa i pruža dodatne korisne funkcije povezane s kalendarom. Funkcije i klase definirane u modulu `calendar` koriste idealizirani kalendar, trenutni gregorijanski kalendar produžen je na neodređeno vrijeme u oba smjera. Prema zadanim postavkama, ti kalendari imaju prvi dan u tjednu ponedjeljak, a posljednji nedjelju (Europska konvencija).

```
>>> import calendar; dir (calendar)
['Calendar', ..., 'MONDAY', 'SATURDAY',
'SUNDAY', 'THURSDAY', 'TUESDAY',
'TextCalendar', 'WEDNESDAY', ...,
'calendar', 'datetime', ..., 'week',
'weekday', 'weekheader']
```

Opisat ćemo nakoliko važnijih funkcija ovog modula.

month (*godina, mjesec*)

```
>>> import calendar
>>> print (calendar. month (2021, 6))
```

```
     June 2021
Mo Tu We Th Fr Sa Su
    1  2  3  4  5  6
 7  8  9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30
```

calendar (year, w=1, l=1, c=2, m=3)

Dobivanje kalendara m-stupaca za cijelu godinu kao niz s više linija. Značenje parametara je:

year - godina kalendara
w - širina kolona podataka
l - broj linija koje će koristiti svaki tjedan
c - broj praznina između kolona mjeseci
m - broj mjeseci u redu

Calendar

Klasa **Calendar** modula **calendar** kreira objekt koji se može koristiti za formatiranje.

```
>>> import calendar
>>> dir (calendar.Calendar)
['__class__', .., 'firstweekday',
'getfirstweekday', 'itermonthdates',
'itermonthdays', 'itermonthdays2',
'itermonthdays3', 'itermonthdays4',
'iterweekdays', 'monthdatescalendar',
'monthdays2calendar', 'monthdayscalendar',
'setfirstweekday', 'yeardatescalendar',
'yeardays2calendar', 'yeardayscalendar'],
'fset', 'getter', 'setter']
>>>
```

Postoji samo jedan parametar, prvi tjedan, čija je vrijednost prema zadanim postavkama postavljena na `0` za `'PONEDJELJAK'`, 6 za `'NEDJELJU'`.

iterweekdays()	Vraća dan u tjednu, koji je trenutno postavljen kao prvi dan. Za ponedjeljak je '0'.
itermonthdates (*godina, mjesec*)	Vraća iterator koji prikazuje sve dane u mjesecu i neke dane u prethodnom i sljedećem mjesecu koji su potrebni za završetak tjedna.
itermonthdays2 (*godina, mjesec*)	Vraća iterator koji daje *n*-torku kao izlaz koji sadrži dane u tjednu od 0-6 s danom u mjesecu.
itermonthdays3 (*godina, mjesec*)	To je nadograđena verzija `itermonth2` i vraća *n*-torku koja sadrži (godinu, mjesec, dan u mjesecu). Uzorak izlaza - (2020, 7, 29).
itermonthdays4 (*godina, mjesec*)	Vraća iterator koji sadrži *n*-torku u zadanom obliku - (godina, mjesec, dan u mjesecu, dan u tjednu). Na primjer - (2020, 7, 21, 1).
monthdays2calendar (*godina, mjesec*)	Vraća listu koja sadrži *n*-torke u obliku (dan u mjesecu, dan u tjednu) za dani mjesec u godini.
monthdayscalendar (*godina, mjesec*)	Vraća listu koja sadrži dane u mjesecu i neke dane prethodnog mjeseca i sljedećeg mjeseca koji upotpunjuju tjedan.
yeardays2calendar (*godina*)	Rezultat je u obliku liste koja sadrži *n*-torku - (dan u mjesecu, dan u tjednu) za kompletno navedenu godinu.

MODUL datetime

Modul **datetime** radi s datumom i vremenom:

```
>>> import datetime; dir (datetime)
['MAXYEAR', 'MINYEAR', '__all__', ...,
'date', 'datetime', 'datetime_CAPI',
'sys', 'time', 'timedelta', 'timezone',
'tzinfo']
```

Uobičajene klase u ovom modulu su:

- date
- time
- datetime
- timedelta

Dobit ćemo ih sa:

```
from datetime import
  ( datetime | date | time | timedelta )
```

U nastavku ih opisujemo tim redom.

datetime

```
>>> from datetime import datetime
>>> dir (datetime)
['__add__', ...,
'astimezone', 'combine', 'ctime',
'date', 'day', ..., 'weekday', 'year']
```

Modul `datetime` ima veliki broj metoda koje smo izostavili u ovom pregledu. Izdvojit ćemo samo nekoliko i prikazati kroz primjere.

```
>>> Datum = datetime.today(); Datum
datetime.datetime(2021, 5, 12, 21, 28,
41, 519490)
>>> Datum = datetime.now(); Datum
datetime.datetime(2021, 5, 12, 21, 29,
0, 563139)
>>> print (Datum)
2021-05-12 21:29:00.563139
```

Pozvali smo `today()` i `new()` metode definirane istovjetno u `datetime` klasi. Izravni prikaz i prikaz naredbom za ispis se razlikuju. Klasa `datetime` je definirana sljedećom sintaksom:

```
datetime ( year, month, day [, hour
[, minute [, second [, microsecond]]]])
```

gdje su `year`, ..., `microsecond` imena atributa.

```
>>> Y = datetime (2021, 5, 12); Y;
datetime.datetime(2021, 5, 12, 0, 0)
>>> print(Y)      2021-05-12 00:00:00
>>> print (Y.day, Y.month, Y.year)
12 5 2021
```

strftime()

Funkcija klase `datetime` koja ima jedan parametar, string formata, prikazan kao rezultat poziva. Na primjer, Y se može prikazati sa:

```
>>> Datum.strftime(
    "%d %m %Y %H %M %S %f" )
'12 05 2021 21 29 00 563139'
```

U sljedećoj su tablici dani parametri ove funkcije.

P	Opis	Primjer
%a	Dan u tjednu, kratka verzija	Sun
%A	Dan u tjednu, puna verzija	Sunday
%w	Dan u tjednu kao broj 0-6, 0 je nedjelja	0
%d	Dan u mjesecu, 01-31	12
%b	Ime mjeseca, kratka verzija	May
%B	Ime mjeseca, puna verzija	May
%m	Mjesec kao broj 01-12	05
%y	Godina, kratka verzija, bez stoljeća	21
%Y	Godina, puna verzija	2021
%H	Sat 00-23	22
%I	Sat 00-12	10
%p	AM/PM	PM
%M	Minuta 00-59	29
%S	Sekunda 00-59	00
%f	Mikrosekunda 000000-999999	563139
%j	Day number of year 001-366	129
%U	Broj tjedna u godini, nedjelja je prvi dan tjedna, 00-53	19
%W	Broj vikenda u godini, ponedjeljak je prvi dan vikenda, 00-53	18
%c	Lokalna verzija datuma i vremena	Sun May 9 22:21:08 2021
%x	Lokalna verzija datuma	05/09/21
%X	Lokalna verzija vremena	22:21:08
%%	% znak	%
%G	ISO 8601 godina	2021
%u	ISO 8601 dan u vikendu, 1-7	7
%V	ISO 8601 broj vikenda, 01-53	18

date

Možemo uvesti `date` klasu from `datetime` modula:

```
>>> from datetime import date
>>> dir (date)
['__add__', ...', 'ctime', 'day', ...,
'today', 'toordinal', 'weekday', 'year']
```

Mogu se izraditi instance datumskih objekata iz klase datuma. Objekt datuma predstavlja `date (godinu, mjesec, dan)`.

```
>>> from datetime import date
>>> d = date (2020, 3, 1); d
datetime.date(2020, 3, 1)
>>> print (d)                2020-03-01
```

`date()` u gornjem primjeru konstruktor je klase `date`. Konstruktor uzima tri argumenta: godinu, mjesec i dan. Varijabla `d` je `date` objekt.

Objekt `date` koji sadrži trenutni datum možemo stvoriti pomoću metode klase nazvane `today()`. Evo kako:

```
>>> from datetime import date
>>> Danas = date.today()
>>> print ("Tekući datum =", Danas)
Tekući datum = 2021-04-02
```

fromtimestamp()

Objekte date možemo stvoriti i iz timestamp (vremenske oznake). Unix vremenska oznaka je broj sekundi između određenog datuma i 1. siječnja 1970. Vremensku oznaku možete pretvoriti u datum pomoću metode fromtimestamp().

```
>>> from datetime import date
>>> T = date.fromtimestamp (1326255555)
>>> print ("Date =", T)
Date = 2012-01-11
```

Iz objekta datuma možemo lako dobiti godinu, mjesec, dan, dan u tjednu.

```
>>> from datetime import date
>>> # Danas
>>> Danas = date.today()
>>> print ("Godina:", Danas.year);
Godina: 2021
>>> print ("mjesec:", Danas.month)
mjesec: 4
>>> print ("dan:", Danas.day)
dan: 2
>>> x = date.today();  x
datetime.date(2021, 5, 11)
>>> print (x)                    2021-5-11
>>> d1 = date (2018, 1, 13)
>>> d2 = date (2018, 12, 31)
>>> d = d2-d1
>>> d.days                              352
>>> Δ = timedelta (days = 1000)
>>> d1 +Δ    datetime.date(2020, 10, 9)
>>> x = d1 +Δ; x.year, x.month, x.day
(2020, 10, 9)
>>> x = date(1993, 12, 14)
>>> print (x)                  1993-12-14
```

Datume možemo instancirati u rasponu od 1. siječnja prve godine do 31. prosinca 9999. To se može potražiti iz atributa min i max:

```
>>> from datetime import date
>>> print (date.min, date.max)
0001-01-01 9999-12-31
```

time

Ova je podklasa organizirana slično kao date.

```
>>> from datetime import time
>>> dir (time)
['__class__', ..., 'dst', 'fold',
'fromisoformat', 'hour', 'isoformat',
'max', 'microsecond', 'min', 'minute',
'replace', 'resolution', 'second',
'strftime', 'tzinfo', 'tzname',
'utcoffset']
>>> t = time (15, 15, 29); t; print (t)
datetime.time(15, 15, 29)
15:15:29
>>> print (time.min, time.max)
00:00:00 23:59:59.999999
>>> t = time (20, 4, 29)
>>> t.hour, t.minute, t.second
(20, 4, 29)
```

Svaka se instanca objekta klase time može promijeniti funkcijom replace:

```
>>> t = t.replace(hour=11, minute=59)
>>> t.hour, t.minute, t.second
(11, 59, 29)
```

Objekt time instanciran od time klase predstavlja lokalno vrijeme:

```
>>> from datetime import time
>>> # time (hour=0, minute=0, second=0)
>>> a = time (); print ("a =", a)
a = 00:00:00
>>> # time(hour, minute and second)
>>> b = time (11, 34, 56)
>>> print ("b =", b)        b = 11:34:56
>>> #   time (...,        microsecond)
>>> d = time(11, 34, 56, 234566)
>>> print("d =", d) d = 11:34:56.234566
```

Postoji neovisna klasa **time** sa svojim funkcijama i metodama:

```
>>> import time
>>> dir (time)
['_STRUCT_TM_ITEMS', '__doc__', ...,
'sleep', 'strftime', 'strptime', ...,
'time', 'time_ns', 'timezone',
'tzname']
```

Od velikog broja funkcija klase time izdvojimo **sleep()** koja zaustavlja izvršenje na zadani broj sekundi (tipa float). Primjer:

```
>>> from datetime import datetime
>>> import time
>>> datetime.now()
>>> time.sleep(5); datetime.now()
```

```
datetime.datetime(2021, 6, 6, 10, 55,
42, 820659)
datetime.datetime(2021, 6, 6, 10, 55,
47, 887596)
```

timedelta

Funkcija `timedelta()` koristi se za izračunavanje razlika u datumima, a može se koristiti i za manipulacije datumima u Pythonu. To je jedan od najlakših načina izvođenja manipulacija datumom. Pravilo pisanja je:

```
datetime.timedelta (days=0, seconds=0,
       microseconds=0, milliseconds=0,
       minutes=0, hours=0, weeks=0)
```

```
# Razlika_vremena.py
from datetime import timedelta
t1 = timedelta(weeks = 2, days = 5,
       hours = 5, seconds = 3)
t2 = timedelta(days = 3, hours = 22,
       minutes = 4, seconds = 54)
t3 = t1 - t2;  print ("t3 =", t3)
t3 = 15 days, 6:55:09
```

JSON

JSON je struktura podataka, tekst, napisana u JavaScript objektnoj notaciji. Python ima standardni modul **json** koji može raditi s podacima danim u JSON-u.

json

```
>>> import json; dir (json)
['JSONDecodeError', 'JSONDecoder',
'JSONEncoder', '__all__', ..., codecs',
'decoder', 'detect_encoding', 'dump',
'dumps', 'encoder', 'load', 'loads',
'scanner']
```

JSON imena i vrijednosti

Pravilo pisanja JSON-a slično je pisanju rječnika u Pythonu. JSON podaci zapisuju se kao parovi imena / vrijednosti. Par imena / vrijednosti sastoji se od imena polja (u dvostrukim navodnicima), nakon čega slijedi dvotočka, a zatim vrijednost:

```
JSON : { ime : vrijednost
         {, ime : vrijednost } }
ime  : string
```

Vrijednosti moraju biti sljedećeg tipa:

- *string*
- *broj*
- JSON *objekt*
- *polje* (*n-torka* ili *lista*)
- *logička vrijednost* (true ili false)
- null

Stringovi moraju biti napisani s dvostukim znakom navoda.

```
{ "ime": "Jojo" }
```

Konverzija iz JSON-a u Python

String u JSON-u konvertira se u rječnik (mapu) Pythona koristeći metodu **json.loads()**.

```
>>> import json
>>> # JSON
>>> x = '{ "ime" : "Igor", "god" : 30,
           "grad" : "Zagreb" }'
>>> y = json.loads(x); print (y)
{'ime': 'Igor', 'god': 30, 'grad':
'Zagreb'}
>>>
```

Konverzija iz Pythona u JSON

Ako imamo objekt u Pythonu može se konvertirati u JSON-ov string koristeći metodu **json.dumps()**.

```
>>> import json
>>> x = { "ime"  : "Jojo",
          "god"  : 11,
          "brat" : "Jacques" }
>>> # konvert u JSON:
>>> y = json.dumps (x)
>>> print (y)
{"ime":  "Jojo",  "god":  11,  "brat":
"Jacques"}
>>>
```

Rad s internetom

Python ima module za pristup internetu i obavljanje određenih radnji. To su `urllib` i `webbrowser`.

urllib

Modul `urllib` je Pythonov modul za rukovanje URL-om (Uniform Resource Locators - Jednoobrazni lokatori resursa). `urllib` sadrži nekoliko modula (kaže se da je to „paket“) za rad s URL-ovima, kao što su:

- `urllib.request` za otvaranje i čitanje.
- `urllib.parse` za raščlanjivanje URL-ova
- `urllib.error` za navedene iznimke
- `urllib.robotparser` za raščlanjivanje datoteka `robot.txt`

Za naše daljne učenje Pythona bit će dovoljno da opišemo `urllib.request`.

urllib.request

```
>>> import urllib.request
>>> dir (urllib.request)
['AbstractBasicAuthHandler', ...,
'urlopen', 'urlparse', 'urlretrieve',
'urlsplit', 'urlunparse', 'warnings']
```

Od 112 metoda ovog modula jedino ćemo koristiti `urlopen()`. Prikazali smo ga u programima za dohvat tečajne liste.

webbrowser

Modul **`webbrowser`** pruža sučelje na visokoj razini koje omogućuje prikaz dokumenata temeljenih na web-korisnicima. U većini slučajeva jednostavno pozivanje funkcije `open()` iz ovog modula otvorit će URL pomoću zadanog preglednika. Mora se uvesti modul i koristiti funkciju `open()`.

```
>>> # Radio_Ri.py
>>> import webbrowser
>>> try: x = webbrowser.open (
      "https://radio.hrt.hr/stream/11/")
except Exception as e : print (str (e))
```

Izvršenjem bit će pozvana stranica Radio Rijeke i, prateći akcije na stranici, možete se uključiti uživo u program Radio Rijeke. Potom možete nastaviti raditi bilo što, kontrola događaja ostaje u pozadini, u Windowsima. Ili, izvršenjem sljedećih naredbi otvorit će se stranica Net-informations.com:

```
>>> import webbrowser
>>> webbrowser.open(
    'http://net-informations.com')
```

GOVORIMO PYTHONSKI

VLASTITI MODULI

Već smo pisali vlastite module od samog početka knjige. Da bismo ih koristili u programima, mogli smo ih testirati pišući

```
if __name__ == "__main__" :
   blok
```

koji se neće izvršavati pozivom modula. Dio *blok* (v. drugo poglavlje) sadržavat će naredbe za testiranje.

RJEŠAVANJE PROBLEMA S DATUMIMA

Odsad ne postoji nijedan razlog da ne bismo koristili standardne module u rješavanju mnogih problema u radu s datumima. Na primjer, ako bismo željeli znati koji je bio dan u tjednu zadanog datuma, ako je to datum našeg rođenja, i koliko je dana proteklo od tog dana, rješenje će biti jednostavno rabeći modul `calendar` i *n*-torku `Dani`:

Dan_u_tjednu.py

```
# Ispis dana u tjednu zadanog datuma
from Moj_modul import Input
from calendar  import *
Dani = ('ponedjeljak', 'utorak',
        'srijeda', 'četvrtak',
        'petak', 'subota', 'nedjelja')
D, M, G = Input( 'Upiši datum u obliku'
              ' dan, mjesec, godina: ')
i       = weekday (G, M, D)
print ( Dani[i] )
```

```
>>>
Upiši datum u obliku dan, mjesec, godina:
1, 3, 1952
subota
Upiši datum u obliku dan, mjesec, godina:
30, 5, 2021
srijeda
```

Poslije uvođenja modula calendar i njegove funkcije `weekday()` pokazat ćemo kako se jednostavno mogu "izračunati" datumi posljednje nedjelje u trećem mjesecu kada se pomiču kazaljke za jedan sat unaprijed i posljednje nedjelje u desetom mjesecu kad se vraćaju za jedan sat (samo je pitanje hoće li nam to trebati ubuduće s obzirom na to da se najavljuje prestanak takvih radnji!).

Posljednja_nedjelja.py

```
# Posljednja nedjelja u trećem i
# desetom mjesecu
from calendar import *
def Nedjelja (G) :
  print (str (G) +'. ', end = ' ')
  for m in [3, 10] :
      for d in range (31, 31-7, -1) :
         if weekday (G, m, d) == 6 :
            print ("%d. %d. " % (d, m),
                   end = ' ')
            break
G = 2021
print ('POSLJEDNJE NEDJELJE \n')
for g in range (G, G+2) :
    Nedjelja (g); print ()
```

```
>>>
POSLJEDNJE NEDJELJE

2021.  28. 3.  31. 10.
2022.  27. 3.  30. 10.
```

RAZLIKA DVAJU DATUMA

Razliku između dva vremena možemo izračunati kako je pokazano u sljedećem programu.

```
# Razlika_vremena.py
from datetime import datetime, date
t1 = date (2021, 6, 5)
t2 = date (1952, 3, 1)
t3 = t1 -t2; print ('t3 =', t3)
t4 = datetime (year = 2021, month = 6,
               day = 5, hour = 20,
               minute = 31, second = 0)
t5 = datetime (year = 2021, month = 1,
               day = 1, hour = 0,
               minute = 0, second = 0)
t6 = t4 - t5; print ('t6 =', t6)
```

```
>>>
t3 = 25298 days, 0:00:00
t6 = 155 days, 20:31:00
```

ZAVRŠETAK DOGAĐAJA

Često imamo potrebu izračunati kraj nekog događaja. Na primjer, dokad će trajati bon na mobitelu, kada će biti kraj godišnjeg odmora, itd. Sljedeći program će vam pomoći u tome.

Dodaj_dane.py

```
from datetime import datetime,
timedelta
def Ispis (T, d, m, g) :
  print (T, str(d)+'.'+str(m)+'.'
           +str(g)+'.')
S = datetime.now()
Ispis ("Početak", S.day, S.month,
                  S.year)
D = eval (input ("Koliko dana "))
B = S + timedelta (days = D)
Ispis ("Kraj    ", B.day, B.month,
                  B.year)
```

UZORAK U REGULARNIM IZRAZIMA

Uzorak ("pattern") se u Pythonu piše prema pravilu:

```
uzorak :  r'regularni_izraz' |
          r'' + nizovni_izraz
nizovni_izraz : 'regularni_izraz' |
          nizovna_varijabla |
          nizovni_izraz
 + nizovni_izraz
```

gdje je *nizovna_varijabla* varijabla Pythona tipa `str` koja sadrži niz znakova koji predstavlja regularni izraz. Evo nekoliko primjera pravilno napisanih uzoraka:

```
r'(P|p)ython'
C = '[a-z0-9.]+'
email_com = "(?i)" +C +"@" +C +".com$"
r'' +email_com
```

KONTRAKCIJE U ENGLESKOM JEZIKU

Poznate su kontrakcije u engleskom jeziku. Pogledajmo nekoliko primjera:

I'm → I am, It's → It is, doesn't → does not

Evo programa koji to radi. Značenje `'\g<1>` je prijepis prve grupe prepoznate u uzorku (ono što se nalazi između prvog para zagrada).

De_kontrakcija.py

```
def De_kontrakcija (s) :
  patterns = [
    (r"won't",     "will not"   ),
    (r"can't",     "cannot"     ),
    (r'(\w+)n\'t', '\g<1> not'  ),
    (r"\'re",      " are"       ),
    (r'(\w+)\'d',  '\g<1> would'),
    (r"\'ll",      " will"      ),
    (r"\'t",       " not"       ),
    (r'(\w+)n\'t', '\g<1> not'  ),
    (r'(\w+)\'ve', '\g<1> have' ),
    (r"\'m",       " am"        ),
    (r"(I|i)t's",  "\g<1>t is"  ) ]

  for (pattern, repl) in patterns:
    s, count = re.subn (pattern,
                        repl, s)
    if count > 0: break
  return s
```

```
>>>
>>> De_kontrakcija ("It's")      'It is'
>>> De_kontrakcija (
    "I'll call you back" )
'I will call you back'
>>> Yes_it_is = """
Please don't wear red tonight
This is what I said tonight
For red is the color that will make me
blue
In spite of you, it's true
Yes it is, it's true
Yes it is
(The Beatles) """
>>> Y = Yes_it_is. split ('\n')
>>> for y in Y :
    print (De_kontrakcija (y))
Please do not wear red tonight
This is what I said tonight
For red is the color that will make me
blue
In spite of you, it is true
Yes it is, it is true
Yes it is
(The Beatles)
>>>
```

Priloženi program ne rješava englesko ime u genitivu, na primjer, `John's`. Također može biti dvoznačnost u ulaznom nizu, na primjer `"she's"` je `"she is"` ili `"she has"`!

REKURZIJE

Poslije uvođenja modula za mjerenje vremena možemo se vratiti rekurzivnim funkcijama i detaljnije analizirati njihovu uporabu.

Fibonaccijev niz i rekurzija?

U trećem smo poglavlju pokazali kako se problem izračunavanja n-tog člana Fibonaccijevog niza može riješiti rekurzivnom funkcijom. Tada je to bila LAMBDA funkcija `fib`:

```
>>> fib = lambda n : (
        'nije definirano' if n < 0  else
        n                 if n <= 1 else
        fib(n-2) + fib(n-1)
```

Sada možemo napisati i program uz dodatak vremena (broj sekundi) izračunavanja.

```
from datetime import *
t = '\t'
def fib (n) :
  if n < 3 : return 1
  else     : return fib(n-2) +fib(n-1)

k1, k2 = 30, 45

for n in range (k1, k2+1) :
  t1 = datetime.now(); An = fib (n)
  t2 = datetime.now(); T  = t2 -t1
  dT = T.seconds +T.microseconds/10**6
  print (n, t, An, t, dT)
```

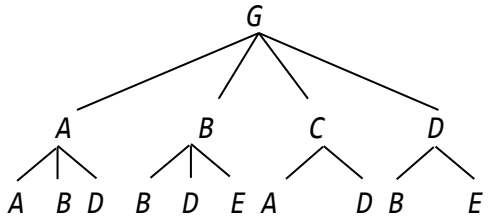
Ako testiramo danu funkciju za neki `n`, na primjer `n=10` ili `n=15`, bit ćemo zadovoljni vremenom izvršavanja. Ali, ako pokušamo izračunati član Fibonaccijevog niza za `n=45`, morat ćemo čekati nekoliko minuta. U slijedećoj je tablici dano vrijeme izvršavanja programu u sekundama i stotinkama sekunde (na vašem će se računalu dobiti drukčiji rezultati) pri izračunavanju članova a_{30} do a_{45}:

i	a_i	sec
30	832040	0.124671
31	1346269	0.187627
32	2178309	0.301325
...		
37	24157817	3.27594
38	39088169	5.323393
39	63245986	8.632276
...		
43	433494437	61.38437
44	701408733	99.187692
45	1134903170	160.685934

Ako s T_i označimo vrijeme trajanja izračunavanja člana a_i, zaključujemo da je:

$$T_i \approx T_{i-2} + T_{i-1}$$

Dakle, vremena izračunavanja odnose se jedno prema drugom kao i članovi Fibonaccijevog niza! No, prije nego što pokušamo dati odgovor zašto je to tako i da bismo rekurziju shvatili što potpunije, korisno je prikazati sve pozive potprograma stablom, koje tada nazivamo *stablo pozivanja potprograma*. Ako je, na primjer glavni program *G*, a potprogrami koje poziva su *A*, *B*, *C*, *D* i *E*, pozivi mogu biti prikazani stablom:



Iz ovakvog prikaza zaključujemo da su potprogrami *A* i *B* rekurzivni, potprogram *D* rekurzivan je implicitno jer poziva *B* koji mu može uzvratiti poziv, potprogram *C* nije rekurzivan ali poziva dvije rekurzivne procedure, *A* i *D*, i potprogram *E* nije rekurzivan. Ako je potprogram rekurzivan, onda je njegovo (pod)stablo *stablo rekurzije*.

To znači da ćemo analizirajući neki program u kojem se takav potprogram poziva iz glavnog programa svakim njegovim ponovnim pozivom unutar sebe imati ekspanziju stabla novim podstablom iz aktivnog čvora. Na primjer, rekurzivna funkcija `fakt` ima stablo rekurzije

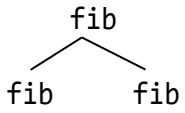
```
fakt
  |
fakt
```

Ako je pozvana s parametrom `n`, `n>0`, stablo rekurzije je:

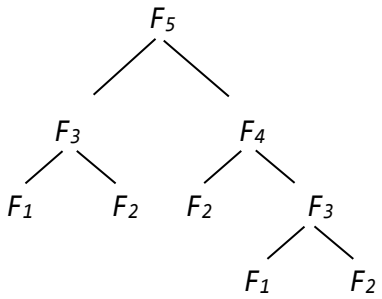
```
fakt(n)
  |
fakt(n-1)
  |
  ...
  |
fakt(0)
```

Kao što smo i pokazali u uvodnom primjeru, isto zaključujemo i iz stabla rekurzije funkcije `fakt`, izračunavanje faktorijela za dani `n` svodi se na niz pozivanja funkcije sve dok argument ne postane `0`. Tada se dobivena vrijednost vraća u prethodno pozvane funkcije sve do prvog poziva (korijena stabla). Zato se vrijeme izračunavanja faktorijela broja `n` rekurzivnom ili nerekurzivnom funkcijom ne razlikuju puno.

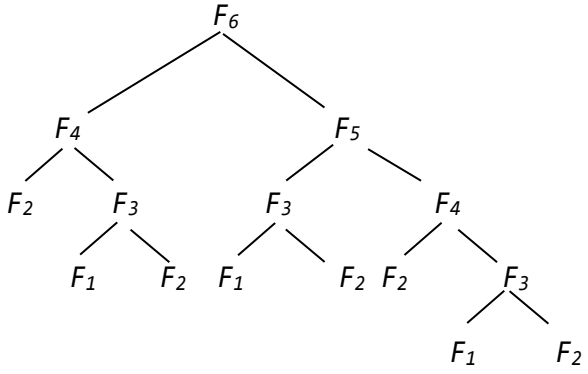
Sada se možemo vratiti našem primjeru rekurzivne funkcije `fib` za izračunavanje `i`-tog člana Fibonaccijevog niza. Najprije prikažimo stablo rekurzije:



Ako prikažemo stablo rekurzije za izračunavanje `i`-tog člana, označimo ga s F_i, na primjer za `i=5`, imamo:



ili, ukupno 9 poziva, a za `i=6`, imamo:



tj. ukupno 15 poziva funkcije `fib`.

Poslije ovih primjera nije teško zaključiti da je broj poziva za izračunavanje i-tog člana Fibonaccijevog niza jednak:

$$F_i = \begin{cases} 1 & \text{za } i=1,2 \\ F_{i-2} + F_{i-1} + 1 & \text{za } i>2 \end{cases}$$

odnosno, za `i>2` vrijedi:

$$F_i = a_i + i - 1$$

gdje je a_i `i`-ti član Fibonaccijevog niza. Sada nam je jasno da je za izračunavanje *30*-tog člana niza trebalo "samo" `832040+30` poziva funkcije `fib` što je na našem računalu trajalo `0.12` sec, a za izračunavanje

45-člana 1134903170+45 poziva što je trajalo 160.69 sec! Napišimo program koji će nam pomoći da procijenimo vrijeme iračunavanja 46-tog do 100-tog člana Fibonaccijevog niza ako bismo nastavili s rekurzivnom funkcijom.

Vrijeme_fib.py

```
a = 701408733;  t1 = 99.187692  # 44 clan
b = 1134903170; t2 = 160.685934 # 45 clan
for n in range (46, 101) :
  a, b   = b, a+b; t1, t2 = t2, t1+t2
  print ("%3d %-21d %10.2f" % (n, b,
    round (t2/(3600.0 *24 *365.25), 2)))
```

```
>>>
 70 190392490709135              0.85
...
 80 23416728348467685          105.03
...
 90 2880067194370816120      12917.27
 91 4660046610375530309      20900.58
 ...
 99 218922995834555169026   981882.16
100 354224848179261915075  1588718.72
```

Dakle, rekurzivno izračunavanje 100-tog člana Fibonaccijevog niza trajalo bi "samo" 1588718.72 (jedan milijun, 588 tisuća ...) godina!!!

Pamćenje međurezultata rekurzije

Trajanje rekurzivnog postupka izračunavanja članova Fibonaccijevog niza moglo bi se znatno skratiti ako bismo mogli pamtiti međurezultate pojedinih članova i time izbjeći ponovno računanje. To je moguće postići uporabom mape kao što smo prikazali u sljedećem programu:

Rek_Fib_1.py

```
Fib = {}
def fib (n):
  Fib[n] = (
     n      if n in [0, 1] else
     Fib[n] if n in Fib    else
     fib(n-1) + fib(n-2) )
  return Fib[n]
N = eval (input (
    'Fibonaccijev broj, n = ? '))
print (fib (N))
```

```
>>>
Fibonaccijev broj, n = ? 100
354224848179261915075
>>>
Fibonaccijev broj, n = ? 1000
43466557686937456435688527675040625802564660517371780402481729089536555417949051890403879840079255169295922593080322634775209689623239873322471161642996440906533187938298969649928516003704476137795166849228875
```

Ali, 1025-ti član nije moguće izračunati jer je dosegnuta maksimalna dubina rekurzivnih poziva:

```
>>>
Fibonaccijev broj, n = ? 1025
...
RuntimeError: maximum recursion depth exceeded in cmp
```

Međutim, postoji jedno interesantno rješenje, što je posljedica ljepote Pythona! Za brojeve veće od 1024 računat ćemo članove Fibonaccijevog niza parcijalno, svakih 1000, koristeći maksimalnu dubinu rekurzije! Evo rješenja u kojem koristimo istu funkciju fib, a dodali smo parcijalne pozive:

Rek_Fib_2.py

```
Fib = {}
def fib (n):
  Fib[n] = (
     n      if n in [0, 1] else
     Fib[n] if n in Fib    else
     fib(n-1) + fib(n-2) )
  return Fib[n]
N = eval (input (
    'Fibonaccijev broj, n = ? '))
if N > 1024 :
  for n in range (1024, N+1, 1000) :
    fib (n)
print (fib (N))
```

```
>>>
Fibonaccijev broj, n = ? 100000
25974069347221724166615503402127...
9895374653428746875
```

Rezultat je na ukupno 246 linija punog ekrana! Nismo mjerili vremena izračunavanja jer su praktički trenutna, i za izračunavanje 100-tog, 200-stotog kao i za 100000-tog člana.

```
>>>
```

P R O G R A M I

Rimski brojevi

Evo programa koji prepoznaje rimske brojeve. Uzorak (regularni izraz) za prepoznavanje rimskih brojeva opisali smo na dva načina (lista `Rim`). Rezultat, grupiranje po tisućama, stoticama, deseticama i jedinicama je jednak.

Rimski_brojevi.py

```
import re
def displaymatch ( match ) :
  if match is None: return None
  return ('<Match: %r, groups=%r>' %
        (match.group(), match.groups()))
Rim = ['(^M?M?M?)(CM|CD|D?C?C?C?)'
       '(XC|XL|L?X?X?X?)'
       '(IX|IV|V?I?I?I?)$',
       '(^M{0,3})(CM|CD|D?C{0,3})'
       '(XC|XL|L?X{0,3})'
       '(IX|IV|V?I{0,3})$' ]

R = input ('Rimski broj? '). upper()
for x in Rim:
  Re = re.compile (r''+x)
  print ( displaymatch (Re.match(R)))
```

```
>>>
Rimski broj? Mdcxxii
<Match: 'MDCXXII',
groups=('M', 'DC', 'XX', 'II')>
<Match: 'MDCXXII',
groups=('M', 'DC', 'XX', 'II')>)>

Rimski broj? Mi
<Match: 'MI', groups=('M', '', '', 'I')>
<Match: 'MI', groups=('M', '', '', 'I')>

Rimski broj? Mili
None
None
```

Registarske oznake u BiH

Regularne izraze često koristimo u sintaksnoj analizi. Ovdje ćemo to pokazati definirajući uzorak koji će prepoznati je li ulazni niz registarska oznaka vozila u Bosni i Hercegovini.

Ako je `s` veliko slovo `A`, `E`, `J`, `K`, `M`, `O` ili `T`, na Wikipediji se može naći da su tri vrste registarskih oznaka u BiH:

a) Stare registarske oznake, od `000-s-000` do `999-s-999`, na primjer `234-J-333`
b) Nove registarske oznake, od `s00-s-000` do `s99-s-999`, na primjer `A77-K-007`
c) Taksi vozila od `TA-000000` do `TA-999999`, na primjer `TA-222543`

BiH.py

```
# Registarske oznake u BiH
"""
a) 000-s-000 do 999-s-999
b) s00-s-000 do s99-s-999
c) TA-000000 do TA-999999
"""
# ^
import re
s  = "[AEJKMOT]"; c  = "-"; ili = "|"
B0 = "\d{3}$"; B1 = "^" +"\d{3}"
B2 = "^" +s +"\d{2}"; B = c+s+c+B0
T  = "TA-"; Taxi = T +"\d{6}$"
R    = (r"" +"(" +B2+B +ili+ B1+B
        +ili+ Taxi +")" )
print (R)
while 'unos' :
  print ()
  txt = input (
    'Unesi registarsku oznaku ')
  if not txt : break
  x = re.search (R, txt)
  if x : print ('OK')
  else : print (
    'Ne postoji registracija ' + txt)
```

```
>>>
(^[AEJKMOT]\d{2}-[AEJKMOT]-
\d{3}$|^\d{3}-[AEJKMOT]-\d{3}$|TA-
\d{6}$)

Unesi registarsku oznaku A11-T-534
OK

Unesi registarsku oznaku 123-O-987
OK
Unesi registarsku oznaku TA-123456
OK

Unesi registarsku oznaku M44-A-1234
Ne postoji registracija M44-A-1234

Unesi registarsku oznaku <Enter>
>>>
```

KALENDAR

Kalendar generiran za jedan mjesec ili godinu dobiven funkcijom `month()` ili `calendar()` je tekst (string). Prije njegova ispisa smo s funkcijom `replace()` njegov sadržaj, skraćenice imena mjeseci i puna imena mjeseci, preveli na hrvatski jezik.

Kalendar.py

```
from calendar import *
E = ('January ', 'February', ' March',
     '  April',  '  May  ',  ' June',
     ' July ',   'August ',
     'September', ' October',
     'November', 'December')
H = ('Siječanj', 'Veljača ', 'Ožujak',
     'Travanj', 'Svibanj', 'Lipanj',
     'Srpanj',   'Kolovoz' ,
     '  Rujan  ', 'Listopad',
     'Studeni ', 'Prosinac')
while 'godina' :
  yy = input ("Godina:           ")
  if not yy : break
  yy = eval (yy)
  if type (yy) != int : continue
  while 'mjesec' :
     mm = eval (input (
                "Mjesec, 0 za sve: "))
     if ( type (mm) == int and
          0 <= mm <= 12 ) : break
  print ()
  if mm :
     T = month (yy, mm)
     T = T. replace (
               'Mo Tu We Th Fr Sa Su',
               'Po Ut Sr Če Pe Su Ne')
  else :
     T = calendar (yy, w = 1,
                       c = 2, m = 2)
     T = T. replace (
               'Mo Tu We Th Fr Sa Su',
               'Po Ut Sr Če Pe Su Ne')
     for i in range (12) :
         T = T. replace (E[i], H[i])
  print ( T )
```

```
>>>
Godina:           1900
Mjesec, 0 za sve: 2

   February 1900
Po Ut Sr Če Pe Su Ne
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28
```

```
Godina:           2021
Mjesec, 0 za sve: 0

                  2021

      Siječanj              Veljača
Po Ut Sr Če Pe Su Ne  Po Ut Sr Če Pe Su Ne
             1  2  3   1  2  3  4  5  6  7
 4  5  6  7  8  9 10   8  9 10 11 12 13 14
11 12 13 14 15 16 17  15 16 17 18 19 20 21
18 19 20 21 22 23 24  22 23 24 25 26 27 28
25 26 27 28 29 30 31
...
      Studeni               Prosinac
Po Ut Sr Če Pe Su Ne  Po Ut Sr Če Pe Su Ne
 1  2  3  4  5  6  7         1  2  3  4  5
 8  9 10 11 12 13 14   6  7  8  9 10 11 12
15 16 17 18 19 20 21  13 14 15 16 17 18 19
22 23 24 25 26 27 28  20 21 22 23 24 25 26
29 30                 27 28 29 30 31
```

"CRNI PETAK"

Za one koji su pomalo praznovjerni evo primjera kako se primjenom funkcije `weekday()` modula `calendar` mogu dobiti datumi "crnih" petaka za ovu i još osam godina.

Crni_petak.py

```
# Mjeseci u kojima će biti "crni
# petak", od 2021. do 2029. godine
from calendar import *
G = 2021
print ('CRNI PETAK \n')
for g in range (G, G+10) :
  print ( str (g) +'. ', end = ' ')
  for m in range (1, 13) :
    if weekday (g, m, 13) == 4 :
       print (
       '  13.', str(m) +'.', end = ' ')
  print ()
```

```
>>>
CRNI PETAK

2021.    13. 8.
2022.    13. 5.
2023.    13. 1.   13. 10.
2024.    13. 9.   13. 12.
2025.    13. 6.
2026.    13. 2.   13. 3.   13. 11.
2027.    13. 8.
2028.    13. 10.
2029.    13. 4.   13. 7.
```

Dakle, treba izbjegavati 2., 3. i 11. mjesec 2026. godine!

Broj radnih dana i sati u godini (2)

Broj radnih sati u jednoj godini računa se kao zbroj radnih dana (dani bez subota i nedjelja) pomnoženih s 8.

Radni_dani.py

```
from datetime import datetime
while 'Unos' :
   G = eval (input ('Učitaj godinu '))
   if type (G) == int and G >= 2000 :
      break
Dan = ('pon', 'uto', 'sri', 'čet',
       'pet', 'sub', 'ned')
M   = [0, 31, 28 +(G % 4 == 0), 31, 30,
       31, 30, 31, 31, 30, 31, 30, 31]
RD = [0]
for i in range (1, 13) :
  # Radni sati po mjesecima. Subota i
  # nedjelja su 5-ti i 6-ti dan
  RD.append (
     sum ([1 for k in range (1, M[i]+1)
     if datetime(G, i, k).weekday()
        not in [5, 6]]))
del RD [0]
RS = [ d*8 for d in RD]
print ()
form = "%4d" *12
dana, sati = sum (RD), sum (RS)
d    = 'dan' +'a' *(dana % 10 != 1)
s    = 'sat' +'i' *(sati % 10 != 1)
print('Radnih dana i sati po mjesecima'
      ' u godini ' +str(G) +':\n')
print ('mjesec    ' +form %
       (tuple (range (1, 13))))
print ('rad. dana ' +form %
       (tuple (RD)))
print ('rad. sati ' +form %
       (tuple (RS))); print ()
print ('Ukupno:', dana, d, sati, s)
```

```
>>>
         Učitaj godinu 2020

Radnih dana i sati po mjesecima u godini
2020:

mjesec       1   2   3   4   5   6   7
8   9  10  11  12
rad. dana   23  20  22  22  21  22  23
21  22  22  21  23
rad. sati  184 160 176 176 168 176 184
168 176 176 168 184

Ukupno: 262 dana, 2096 sati
```

```
         Učitaj godinu 2021

Radnih dana i sati po mjesecima u godini
2021:
mjesec       1   2   3   4   5   6   7
   8   9  10  11  12
rad. dana   21  20  23  22  21  22  22
  22  22  21  22  23
rad. sati  168 160 184 176 168 176 176
 176 176 168 176 184

Ukupno: 261 dan, 2088 sati
```

Cijena Parkinga u zračnoj luci Zagreb (2)

U trećem smo poglavlju pokazali kako se uz pomoć uvjetnih izraza može izračunati cijena usluge parkiranja u zračnoj luci Zagreb. Morali smo znati koliko je dana, sati i minuta trajalo parkiranje. Također smo provjeravali valjanost tih podataka.

U ovoj inačici programa početne podatke dobivamo kao razliku vremena završetka i početka parkiranja. Ostali dio programa nismo mijenjali.

Parking_Zračna_luka_2.py

```
# Usluga parkiranja na parkiralištu
# Zračne luke Zagreb
from datetime import datetime
# datetime ( break, month, day, hour,
#            minute )

def Input (s) :
  while 'Ok' :
    print (s)
    D, M, Y, H, m = eval (input (
    'dan, mjesec, godina, sat, '
    'minuta '))
    try :
       t = datetime (
         year = Y, month = M, day = D,
         hour = H, minute = m)
       return t
    except : pass

T1   = Input ('Početak:')
T2   = Input ('Kraj:')
T    = T2 -T1
D    = T.days
S, M = divmod (T.seconds, 3600)
M    = M // 60

T = round (D*24 +S +M/100, 2)
d = D +((T % 1) > 0)
```

```
  C = 0  if T <=  0.1  else  \
     27  if T <=  1.0  else  \
     47  if T <=  2.0  else  \
     70  if T <=  3.0  else  \
     78  if T <=  6.0  else  \
    110  if T <= 12.0  else  \
    150  if T <= 24.0  else  \
    210 +( 0 if d <= 2 else
           72 *(d-2) if d <=5 else
           72 *3 +68 *(d-5) )
  print ( "Usluga parkiranja za %d dana,"
          " %d sati i %d min... %d kn"
          % (D, S, M, C) )
```

```
>>>
Početak:
dan, mjesec, godina, sat, minuta 7, 6,
2021, 10, 10
Kraj:
dan, mjesec, godina, sat, minuta 21, 6,
2021, 9, 30
Usluga parkiranja za 13 dana, 23 sati i
20 min... 1038 kn
```

MJENJAČNICA (3)

Nastavljamo s modifikacijom naše mjenjačnice. U ovoj inačici koristimo internetski pristup tečajnoj listi HNB na adresi:

https://www.hnb.hr/tecajn/htecajn.htm

Preostali je dio jednak onom u inačici (2).

Mjenjačnica_3.py

```
from Moj_modul import *
class RECORD :
  def __init__ ( s, x, y ):
    for i in range (len (x)) :
      exec ('s.' +x[i] +' = '
            +'str(y[i])')

# UČITAJ TEČAJNU LISTU U TL
Dat = 'TL.txt'
Y = input (
    'Učitavam novu tečajnu listu (d/n)? ')
if Y[0].upper() == 'D' :
  from urllib.request import *
  try :
    TL = urlopen(
"https://www.hnb.hr/tecajn/htecajn.htm")
    tl = TL.read().decode("utf-8")
  except :
    print( 'isključen internet!' ); quit()
  dat = open (Dat, 'w')
  tl = tl.replace ('\r', '')
  dat. write(tl)
  dat. close()
try    : dat = open (Dat, 'r')
except :
  print('NE POSTOJI DATOTEKA', Dat)
  quit ()
datum = dat.readline()[11:19]
tl  = dat. read(). replace (',', '.')
TL0 =  ['000HRK001       1.000000       '
      + '1.000000       1.000000']
TL0 += tl.split ('\n')

# 'RAZBIJ' TEČAJNU LISTU
TL = []
for x in TL0 :
  x = x[3:6] +' ' +x[:3] +' ' +x[6:]
  TL.append (x.split())

# ISPIŠI TEČAJNU LISTU
print (NL + 'Tečajna lista na dan: ',
      datum[:2] +'.' +datum[2:4]
      +'.' +datum[4:] +'.')
print ()
print ('RB Val Šif Par  Kupovni'
       '  Srednji  Prodajni')
print ('----------------------'
       '--------------------')
RB = 0;  f = "%2s %s %s %s " +3*"%9s"
for x in TL :
  if x :
    t = tuple ([str(RB)] +x)
    print (f % t); RB += 1
print ()

# MJENJAČNICA
while True :
  try :
    s = input('Redni broj ulazne valute, '
     'iznos i redni broj izlazne valute ')
    if not s : break
    atr = ('val', 'šif', 'par', 'kup',
           'sre', 'pro')
    i, X, j = eval (s)
    if (0 <= i < len(TL) and
        0 <= j < len(TL) ) :
      # UL/IZ - ulazna/izlazna lista
      UL = RECORD (atr, TL[i])
      IZ = RECORD (atr, TL[j])
      print ( X, UL.val, '= ',
        round (X *1/int (UL.par)
          *eval (UL.kup +'/' +IZ.pro), 2),
           IZ.val )
    else : print (
           'greška, redni broj valute?' )
  except :
    print ('POGREŠKA PRI UNOSU PODATAKA!')
    continue
```

```
>>>
Učitavam novu tečajnu listu (d/n)? d
```

```
Tečajna lista na dan:  05.06.2021.

RB Val Šif Par  Kupovni  Srednji  Prodajni
----------------------------------------
 0 HRK 000 001  1.000000 1.000000 1.000000
 ...
12 BAM 977 001  3.824756 3.836265 3.847774
13 EUR 978 001  7.480573 7.503082 7.525591
14 PLN 985 001  1.674218 1.679256 1.684294

Redni broj ulazne valute, iznos i redni
broj izlazne valute 13, 1000, 0
1000 EUR =  7480.57 HRK
Redni broj ulazne valute, iznos i redni
broj izlazne valute <Enter>
```

API TEČAJNA LISTA

Tečajnu listu HNB možemo dohvatiti s njihove stranice

https://www.hnb.hr/web/api

Tamo su dane i opće napomene:

- Tečaj je iskazan u kunama (HRK).
- Podatke je moguće dohvatiti u `json` (zadana vrijednost) ili `xml` formatu.
- Datumi su formatirani prema Hr standardu (`dd.MM.yyyy`).
- Valute se upotrebljavaju prema standardu ISO 4217.

Servis za navedene parametre vraća podatke o tečaju. Ako je datum napisan u obliku:

datum : GGGG-MM-DD

gdje su: `GGGG` godina, `MM` mjesec i `DD` dan, mogu se dohvatiti:

- trenutačni tečaj za sve valute:

  http://api.hnb.hr/tecajn/v1

- tečaj na *datum* za sve valute:

  http://api.hnb.hr/tecajn/v1?datum=***datum***

- dohvat podataka za odabranu valutu (EUR):

  http://api.hnb.hr/tecajn/v1?valuta=EUR

- dohvat podataka za više odabranih valuta (EUR i USD):

  http://api.hnb.hr/tecajn/v1?valuta=EUR&valuta=USD

- dohvat podataka za razdoblje:

  http://api.hnb.hr/tecajn/v1?datum-od=***datum*** &datum-do=***datum***

API_tečajna_lista.py

```
import json
from   urllib.request import *
from   time           import gmtime

G, M, D = gmtime()[:3]
dat     = ("?datum=%04d-%02d-%02d"
           % (G, M, D))
TecL    = urlopen (
    'http://api.hnb.hr/tecajn/v1' +dat)
# konverzija u listu mapa:
TecL    = json.load (TecL)
print ( 'Datum primjene',
        TecL[0]['Datum primjene'])
TL0  = ['000HRK001       1.000000'
  '        1.000000       1.000000']
TL = []
for x in TL0 :
  x = x[3:6] +' ' +x[:3] +' ' +x[6:]
  TL.append (x.split())

# ISPIŠI TEČAJNU LISTU
NL = '\n'
print ()
print ('RB Val Šif Par   Kupovni'
       '   Srednji  Prodajni')
print ('------------------------'
       '--------------------')
RB = 0;  f = "%2s %s%4s %-3s" +3*"%10s"
for x in TecL :
  TL.append ([ str (x["Šifra valute"]),
     str (x["Valuta"]),
     "%03d" % x["Jedinica"],
     (x["Kupovni za devize"]).
       replace (',', '.'),
     (x["Srednji za devize"]).
       replace (',', '.'),
     (x["Prodajni za devize"]).
       replace (',', '.') ] )
for x in TL :
  t = tuple ([str(RB)] +x)
  print (f % t); RB += 1
print ()

# MJENJAČNICA
# ... kao u Mjenjačnica_3.py
```

```
>>>
...
>>> # Primjer rezultata (JSON):
>>> print (TecL[12])
```

```
{'Broj tečajnice': '10', 'Datum
primjene': '18.01.2021', 'Država':
'EMU', 'Šifra valute': '978', 'Valuta':
'EUR', 'Jedinica': 1, 'Kupovni za
devize': '7,549974', 'Srednji za
devize': '7,572692', 'Prodajni za
devize': '7,595410'}
```

Kao drugi primjer programa dohvatimo srednji tečaj za devize švicarskog franka (CHF) u periodu 01.01.2007. do `01.04.2021`. godine na `01.mm`, gdje je `mm` mjesec, `01`, `04`, `07` i `10` tekuće godine.

API_TL_CHF.py

```
# Tečaj CHF u zadanom periodu
"""
Dohvat podataka za odabranu valutu:
GET http://api.hnb.hr/tecajn/
v1?valuta=EUR

Dohvat podataka za razdoblje:
GET http://api.hnb.hr/tecajn/
v1?datum-od=2007-03-02&datum-do=2014-
04-02
"""
import json
from   urllib.request import *

TL = []; D = []; M = ['']
for m in range (1, 13) :
  M. append ("%02d-01" % m)
for G in range (2007, 2022) :
  for m in M[1:13:3] :
    D. append (str (G) +'-' +m)

for d in D :
  dat = "?datum-od=" +d +"&datum-do="+d
  TecL = urlopen (
       'http://api.hnb.hr/tecajn/v1'
       +dat )
  TecL = json.load (TecL) # >lista mapa
  for x in TecL :
    if x["Valuta"] == "CHF" :
      TL.append (
         [d, x["Srednji za devize"].
             replace (',', '.')] )
for t in TL : print (t[0], '\t', t[1])
```

```
>>>
2007-01-01      4.571248
...
2009-01-01      4.916034
...
2011-01-01      5.904603
...
2012-01-01      6.194817
...
2013-01-01      6.250803
...
2015-01-01      6.369216
...
2016-01-01      7.051989
...
2018-01-01      6.431816
...
2019-01-01      6.577957
...
2020-01-01      6.864804
...
2021-01-01      6.971280
2021-04-01      6.840799
```

OXFORDSKI RJEČNIK

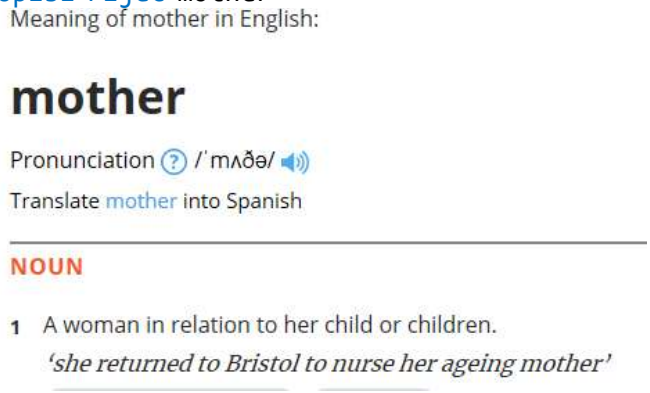
Završavamo s jednostavnim programom koji omogućuje vezu s Oxfordskim rječnikom. Za zadanu riječ engleskog jezika bit će prikazano njezino značenje. Tako ćemo, na primjer, naučiti da imenice `mother` i `beach` mogu biti i glagoli, a glagol `have` i imenica! Također ćemo vidjeti da riječ `string` može biti imenica sa šest značenja, ili glagol s također šest značenja.

WEB.py

```
import webbrowser
OX = (
"http://www.oxforddictionaries.com/"
"definition/english/")
while True :
  w = input ('Upiši riječ ')
  if not w : break
  webbrowser.open (OX +w)
```

Ovdje je OX adresa (URL) stranice Oxfordskog rječnika.

```
>>>
Upiši riječ mother
```


Meaning of mother in English:

mother

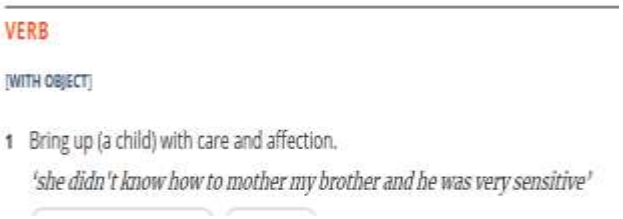
Pronunciation /ˈmʌðə/

Translate mother into Spanish

NOUN

1 A woman in relation to her child or children.

'she returned to Bristol to nurse her ageing mother'



VERB

[WITH OBJECT]

1 Bring up (a child) with care and affection.

'she didn't know how to mother my brother and he was very sensitive'

Ako želimo ulaznu riječ (ili rečenicu) w prevesti na hrvatski jezik, rabit ćemo adresu HR

```
HR =
"https://translate.google.hr/?hl=hr&tab=wT#
en/hr/"
```

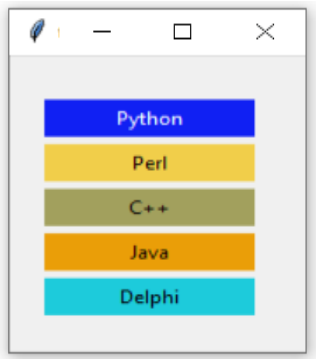
i poziv

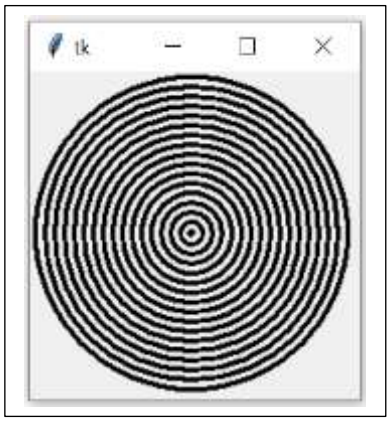
```
webbrowser.open (HR +w)
```

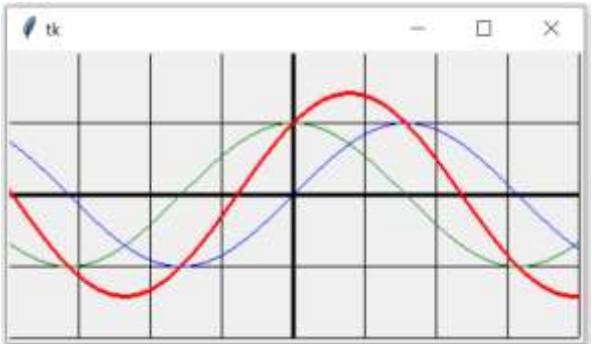
13.

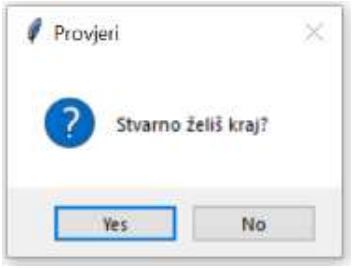
GRAFIČKO KORISNIČKO SUČELJE (G U I)

Puna primjena objektno orijentiranog programiranja dolazi do izražaja uporabom grafičkog korisničkog sučelja (GUI). Postoji nekoliko desetaka GUI modula namijenjenih Pythonu. Mi ćemo u ovoj knjizi uvodno opisati ***tkinter****, tradicionalni (standardni) GUI modul Pythona, koji možda nije najbolji, ali je relativno jednostavan i još uvijek prilično rasprostranjen. Kažemo „uvodno" jer bi popuniji opis Tkintera zahtijevao jednu malo veću knjigu.*









Uvod

Svi programi koje smo koristili u prvih dvanaest poglavlja komunicirali su s korisnikom preko teksta. Na primjer, ako bismo za zadani polumjer kruga, r, željeli izračunati opseg kružnice i površinu kruga, napisali bismo program:

```
# Izračunava opseg i površinu kruga
pi = 3.14
r  = eval (input (
          'Zadaj polumjer, r = '))
print( 'O = %0.4f, P = %0.4f'
       % (2 *r *pi, r**2 *pi) )

Zadaj polumjer, r = 12.5
O = 78.5000, P = 490.6250
```

Ako bismo željeli izračunati opseg i površinu za nekoliko polumjera, bez ograničenja broja pozivanja, morali bismo preurediti unos polumjera r u:

```
while True :
  r  = input ('Zadaj polumjer, r = ')
  if not r : break
  r = float (r); print ('O = %0.4f, '
    'P = %0.4f' % (2 *r *pi, r**2 *pi))
```

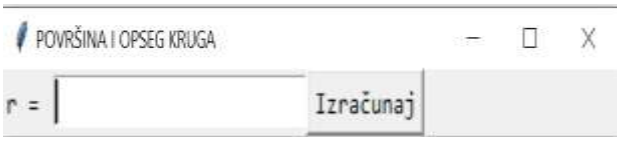
Ali, postoje i složeniji načini za predstavljanje i prosljeđivanje informacija. Grafičko korisničko sučelje (*engl.* Graphical User Interface – **GUI**) omogućuje da korisnik vizualno komunicira s računalom. Svi najpopularniji operacijski sustavi osobnih računala koriste GUI, što interakciju s korisnikom čini jednostavnijom i dosljednijom. Python ima veliki broj GUI modula,

https://wiki.python.org/moin/GuiProgramming

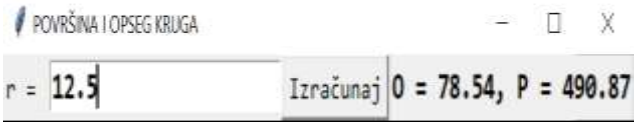
Mi ćemo u ovoj knjizi opisati **tkinter**, standardni GUI modul Pythona. Tkinter je dio ***Tcl***-a (*Tool Command Languagea*), jezika koji omogućava izradu, na primjer, web i desktop aplikacija. Uz njega se često veže i objektno orijentirano grafičko sučelje pod imenom ***Tk***, koje je standardni GUI ne samo za Tcl, nego i za mnoge druge dinamičke jezike (C, C++, Perl, PHP, Ruby itd.), ustrojenih na različitim operacijskim sustavima (od Linuxa i MAC-a do Windowsa). Na primjer, u Tkinteru bismo mogli napisati program za izračunavanje opsega i površine kruga:

```
from tkinter import *
font1 = 'Consolas', 12, 'normal'
font2 = 'Consolas', 14, 'bold'

root1  = Tk ()
root1  . title('POVRŠINA I OPSEG KRUGA')
root1  . option_add ('*font', font1)
Poruka = Label (root1, text = 'r = ')
Poruka . pack (side = LEFT)
r      = StringVar()
Input  = Entry(root1, textvariable = r,
               font = font2)
Input  . pack(side = LEFT)
Input  . focus_set()
def OP () :
  from math import pi
  global r
  R = eval (r.get())
  O = 2 *R *pi; P = R**2 *pi
  Rezultat['text'] = (
    "O = %0.2f, P = %0.2f" % (O, P) )
Button (root1, text = 'Izračunaj',
        command = OP).pack(side = LEFT)
Rezultat = Label (root1, text = ' '*15,
                  font = font2)
Rezultat.pack (side = LEFT)
root1.mainloop()
```

Izvršavanjem programa bit će prikazano:



Unosimo vrijednost polumjera i pritišćemo gumb na kojem piše "Izračunaj". Vrijednosti opsega i površine bit će prikazani iza gumba:



Izvršavanje programa nije prekinuto, pa ako bismo trebali izračunati opseg i površinu kružnice s nekim drugim polumjerom, treba umjesto 12.5 upisati novu vrijednost i pritisnuti gumb za izračunavanje.

Python nudi razne mogućnosti za razvoj grafičkih korisničkih sučelja (GUI). Najvažnije značajke su:

- **tkinter** - Pythonovo sučelje za Tk GUI alatni paket isporučen s Pythonom.
- **wxPython** - Pythonovo sučelje otvorenog koda za GUI alate za wxWidgets.
- **PyQt5** - Ovo je ujedno i Pythonovo sučelje za popularnu višeplatformsku Qt GUI biblioteku.

Dostupna su mnoga druga sučelja koja možete pronaći na mreži. Od predložena tri programa za GUI programiranje PyQt5 pruža najveće mogućnosti. Mi smo se

ipak odlučili da uvodno opišeno `tkinter` modul koji se standardno ispuručuje s Pythonom. Python u kombinaciji s tkinterom pruža brz i jednostavan način za stvaranje GUI aplikacija. Tkinter pruža moćno objektno orijentirano sučelje Tk GUI alata. Stvaranje GUI aplikacije pomoću tkintera jednostavan je zadatak. Sve što trebate učiniti je izvršiti sljedeće korake:

- Uvesti *Tkinter,* tkinter.
- Kreirati glavni prozor GUI aplikacije.
- Dodati jedan ili više komponenti (widgets) GUI aplikacija.
- Upisati glavnu petlju za događaje.

Na primjer:

```
import tkinter
top = tkinter.Tk()
# Kod i komponente #
top . mainloop()
```

Temeljni pojmovi

Prije nego što prijeđemo na opis Tkintera uvedimo nekoliko temeljnih pojmova.

Window

Prozor, koji ima različito značenje u različitim kontekstima, ali općenito je to pravokutni prostor, bilo gdje na zaslonu, kontroliran aplikacijom.

Control

Kontrola, objekt grafičkog sučelja koji se koristi za upravljanje aplikacijom. Kontrola ima svojstva i najčešće generira događaje. Kontrole odgovaraju objektima na razini aplikacije, a događaji su povezani s metodama odgovarajućeg objekta tako da kada se dogodi neki događaj, objekt izvršava jednu od svojih metoda. Neke kontrole pridružene komponenti, kao na primjer sat, nisu vidljive.

Widget

Generički pojam za bilo koji blok ili komponentu koja čini program u grafičkom korisničkom sučelju. To su vidljive kontrole kojima se manipulira od strane korisnika ili programera. Na primjer, to su gumbi (*buttons*), radio-gumbi (*radiobuttons*), tekstualna polja (*text fields*), okviri (*frames*), tekstualni natpisi (*text labels*) itd.

Frame

Okvir. U Tkinteru je to komponenta `Frame()` i predstavlja osnovnu jedinicu za organiziranje složenog izgleda. Okvir je pravokutni prostor koji sadrži druge komponente.

Label

Natpis. Sadrži neki natpis (*label*) ili sliku (ikonicu). Ne generira nikakav događaj ali može biti modificirana događajem iz nekih drugih komponenti.

Button

Gumb. Komponenta s tekstom i/ili slikom (ikonicom) na koju se može kliknuti pri čemu se izvršava neka procedura.

Text Entry

Komponenta koja može prikazati i/ili primati tekst. To može biti unos pojedinačnog retka na obrascu ili multilinijskom unosu, poput prozora za uređivanje teksta. Komponente teksta često mogu sadržavati i druge komponente, poput slika.

Menu

Komponenta koja predstavlja kontrolu izbornika. Izbornik sadrži stavke i/ili podizbornike. Izbornici pružaju sve mehanizme za navigaciju hijerarhijom Menu izbornika. Stavke izbornika, ako su odabrane, emitiraju događaje koji se mogu obraditi.

Canvas

"Platno" za crtanje. Sadrži grafičke oblike i slike. Canvas objekti normalno sadrže metode koje dopuštaju crtanje geometrijskih oblika, dijagrama, itd.

Messagebox

Mali dijaloški okvir koji općenito prikazuje vrlo jednostavne upite ili zahtjeve za jednostavnim vrstama korisničkog unosa.

Geometry

Svaki prozor i dodatak imaju geometriju ili skup koordinata koji ukazuju na njegovo mjesto i veličinu. Različiti alati različito predstavljaju ove informacije. Tkinter koristi format (širina, visina). Podaci o lokaciji ako su potrebni prikazuju se kao: (x-koordinata, y-koordinata) i relativni su prema sadržaju komponente.

Dialog
Posebna vrsta prozora koji je u vlasništvu nadređene aplikacije, ali se može samostalno premještati po zaslonu. Dijalozi mogu biti modalni, što znači da morate zatvoriti dijaloški okvir prije nego što aplikacija reagira na bilo koje druge radnje ako dijalog radi paralelno s glavnim prozorom aplikacije.

Layout
Kontrole se postavljaju unutar okvira prema određenom skupu pravila ili smjernica. Ova pravila čine raspored (*layout*). Izgled se može specificirati na više načina, bilo pomoću koordinata na zaslonu specificiranih u pikselima, koristeći relativni položaj prema ostalim komponentama (lijevo, gore itd.) ili pomoću rešetke ili rasporeda tablice. Koordinatni sustav lako je razumjeti, ali je njime teško upravljati kada se, na primjer, promijeni veličina prozora. Savjetujemo vam da koristite prozore koji se ne mogu mijenjati ako radite s rasporedima temeljenim na koordinatama. Još bolje, koristite nekoordinirane izglede i prepustite alatu da upravlja stvarima umjesto vas.

Parent-child
Roditelj-dijete. Kada je bilo koja komponenta (*widget*) kreirana, kreirana je i relacija roditelj-dijete (*parent-child*). Na primjer, ako se tekstualni natpis (*text label*) nalazi unutar okvira, okvir (frame) je roditelj tekstualnom natpisu. To je, u biti, nasljeđivanje svojstava u notaciji objektnog programiranja.

GUI aplikacije sastoje se od hijerarhije komponenti/ kontrola. Okvir najviše razine koji sadrži prozor aplikacije sadrži podokvire koji opet sadrže još više okvira ili kontrola. Te se kontrole mogu vizualizirati kao strukturu stabla, pri čemu svaka kontrola ima jednog roditelja i određeni broj djece. U stvari je normalno da komponente izričito pohranjuju ovu strukturu kako bi programer, ili češće samo okruženje GUI, često mogao izvršiti neku zajedničku radnju nad kontrolom i svom njenom podređenom jedinicom. Na primjer, zatvaranje gornje komponente rezultira zatvaranjem svih podređenih komponenti. Komponenta koji sadrži druge komponente naziva se roditelj.

Focus
Kada se prozor fokusira, on postaje aktivni prozor u kojem će svi pritisci tipki i klikovi miša ići na taj prozor i njegove podređene komponente. Na primjer, program za obradu teksta može imati dijaloški okvir za pretraživanja. Korisnik može prebaciti fokus između glavnog prozora i dijaloškog okvira klikom miša na bilo koji prozor koji će primiti ulaz.

Top-level window
Korijenski („vršni“) prozor koji egzistira neovisno na zaslonu. Može biti omeđen standardnim okvirom (frame) i kontrolama za uređenje radnog prostora i može se dopustiti ili ne njegovo pomicanje po cijelom radnom prostoru i promjena dimenzija.

Komponente

Elemente GUI-a kreiramo tako da instanciramo objekte klasa iz modula `tkinter`. Te ćemo klase nazvati "komponente" (eng. *widget*). Općenita sintaksa je:

```
element_GUI :
  komponenta ( parent
               {, opcija = vrijednost } )
```

parent predstavlja nadređeni prozor, u koji treba postaviti komponentu, a opcije su njezini atributi. Tkinter nudi razne komponente, poput gumba, naljepnica i tekstualnih okvira koji se koriste u GUI aplikaciji. Trenutno u Tkinteru postoji 18 vrsta dodataka. U sljedećoj su tablici dane sve klase (komponente) modula `tkinter`.

Komponenta	*Opis*
`Button`	Gumb. Izvršava određenu akciju kad ga korisnik aktivira.
`Canvas`	Platno. Polje za crtanje linija, kružnica, umetanje slika, teksta itd.
`Checkbutton`	Gumb za potvrđivanje. Omogućuje izbor (ili ne) opcije.
`Dialog`	Dijalog.
`Entry`	Ulazni string. Prihvaća i prikazuje jedan red teksta. Podržava izbor fontova.
`Frame`	Okvir. Sadrži druge elemente grafičkog korisničkog sučelja
`Label`	Natpis (oznaka). Prikazuje nepromjenljiv natpis ili ikonicu.
`Listbox`	Lista. Lista izabranih imena.
`Menu`	Opcije povezane s gumbom `Menubutton` ili prozorom najviše razine.
`Menubutton`	Gumb koji otvara izbornik opcija / podizbornika koji se mogu odabrati.
`Message`	Poruka. Polje za prikaz teksta.
`OptionMenu`	Sastavljeno: padajući popis za odabir.
`Radiobutton`	Radio gumb. Omogućuje izbor jedne od ponuđene grupe opcija.
`Scale`	Klizač, komponenta s skalabilnim položajima Windowsa kojim upravlja upravitelj prozora.
`Scrollbar`	Traka za pomicanje ostalih komponenti (npr. Okvira popisa, platna, teksta).

ScrolledText	Text kojem je pridružen Scrollbar
Text	Tekst. Prihvaća i prikazuje više redova teksta. Podržava izbor fontova.
Toplevel	Prozori najviše razine kojima upravlja upravitelj prozora.

Ovome treba dodati još dva objekta, BitmapImage i PhotoImage. BitmapImage smješta bitmap slike unutar nekih klasa, a PhotoImage slike u boji.

OPCIJE Tkintera

U sljedećoj tablici dan je pregled opcija tkintera.

Opcija	*Opis*
active-background	Boja pozadine kada je gumb ispod kursora.
activebor-derwidth	Širina granice nacrtane oko izbora kada je ispod miša. Zadana vrijednost je 1 piksel.
active-foreground	Boja prednjeg plana, kada je gumb ispod kursora.
anchor	Sidro, pozicija teksta u gumbu. Na primjer, ako je anchor = tk.NE, tekst je postavljen u gornjem desnom kutu gumba.
bd, borderwidth	Širina okvira oko vanjske strane gumba. Zadana postavka je dva piksela.
bg, background	Normalna boja pozadine.
bitmap	Za prikaz monokromatske slike na gumbu, postavlja tu opciju u bitmap.
borderwidth	Veličina granice oko indikatora. Zadana postavka je 2 piksela.
closeenough	Realni broj, sloj koji određuje koliko miš mora biti blizu stavke koju treba razmotriti.
confine	Ako je vrijednost True (zadano), platno se ne može pomicati izvan regije za pomicanje (vidi ispod).
command	Funkcija ili metoda koja se poziva kad se klikne na gumb.
compound	Ovu opciju koristite za prikaz teksta i grafike, koja može biti bitmapa ili slika na gumbu. Dopuštene vrijednosti opisuju položaj grafike u odnosu na tekst, a može biti bilo koji od tk.BOTTOM, tk.TOP, tk.LEFT, tk.RIGHT ili tk.CENTER. Na primjer, compound = tk.LEFT pozicionirao bi grafiku lijevo od teksta.
cursor	Odabire kursor koji će se prikazati kada miš pređe preko gumba.
default	tk.NORMAL je zadana vrijednost; upotrijebite tk.DISABLED ako će gumb u početku biti onemogućen (sivo, ne reagira na klikove miša).
digits	Način na koji vaš program čita trenutnu vrijednost prikazanu u komponenti razmjere vrši se kroz kontrolnu varijablu. Kontrolna varijabla skale može biti IntVar, DoubleVar ili StringVar. Ako je riječ o znakovnoj varijabli, opcija digits kontrolira koliko će se znamenki koristiti kada se vrijednost numeričke ljestvice pretvori u string.
direction	Postavite direction=LEFT za prikaz izbornika s lijeve strane gumba; koristite direction=RIGHT za prikaz izbornika s desne strane gumba; ili upotrijebite direction = 'above' da biste postavili izbornik iznad gumba.
disabled-background	Boja pozadine koja će se prikazati kada je komponenta u stanju tk.DISABLED. Za vrijednosti opcija pogledajte gore bg.
disabled-foreground	Boja prednjeg plana koja se koristi kad je gumb onemogućen (disabled).
element-borderwidth	Širina obruba oko vrhova strelica i klizača. Zadana vrijednost je -1, što znači koristiti vrijednost opcije borderwidth.
export-selection	Ako prema zadanim postavkama odaberete tekst unutar Entry() komponente, on se automatski izvozi u međuspremnik. Da biste to izbjegli, upotrijebite exportselection=0.
fg, foreground	Boja teksta.
font	Font teksta koji će se koristiti za oznaku gumba.
format	Formatiranje niza. Nema zadane vrijednosti.
from_	Realna ili cijelobrojna vrijednost koja definira jedan kraj raspona skale.
handlepad	Zadana vrijednost je 8.
handlesize	Zadana vrijednost je 8.
height	Visina gumba u retcima teksta (za tekstualne gumbe) ili pikselima (za slike).
highlight-background	Boja naglaska fokusa kada komponenta nema fokus.
high-lightcolor	Boja naglaska fokusa kada komponenta ima fokus.
highlight-thickness	Debljina žarišta fokusa.
image	Slika koja će se prikazati na gumbu (umjesto teksta).
indicatoron	Gumb checkbutton() obično je prikazan kao indikator okvira koji pokazuje je li gumb postavljen ili nije. To se ponašanje može postići postavljanjem indicatoron=1. Međutim, ako postavite indicatoron=0, indikator nestaje, a cjelokupna komponenta postaje pritisni gumb koji izgleda podignut kad se očisti, i utone kad se postavi. Možda treba povećati borderwidth vrijednost da bi se lakše vidjelo stanje takve kontrole.
insert-background	Boja kursora koji prikazuje točku u tekstu gdje će se umetnuti novi unos s tipkovnice. Inicijalno je crne boje.
insert-borderwidth	Inicijalno je kursor za umetanje jednostavni pravokutnik. Može se dobiti s tk.RAISED postavljanjem insertborderwidth na dimenziju trodimenzionalne granice. Ako to učinite, provjerite je li opcija insertwidth barem dvostruko veća od te vrijednosti.
insert-offtime	Prema zadanim postavkama kursor za umetanje trepće. Možete postaviti vrijeme umetanja na vrijednost u milisekundama da odredite koliko vremena provodi kursor za umetanje.

	Zadana vrijednost je 300. Ako je `insertofftime = 0`, kursor neće treptati.
insertontime	Slično `insertofftime`, i ova opcija određuje koliko vremena kursor provodi po treptaju. Inicijalno je 600 (milisekundi).
insertwidth	Postavljanje kursora na dimeziju. Inicijalno je širok 2 piksela.
jump	Ova opcija kontrolira što se događa kada korisnik povuče klizač. Uobičajeno (`jump = 0`), svako malo povlačenje klizača uzrokuje pozivanje povratnog poziva naredbe. Ako je `jump = 1`, povratnog poziva neće biti dok korisnik ne otpusti tipku miša.
justify	Prikazivanje više redaka teksta: tk.LEFT za poravnavanje svakog retka lijevo; tk. CENTER da ih centrira; ili tk.RIGHT na desno.
label	Oznaku možete prikazati u komponenti skale postavljanjem ove opcije na tekst labele. Oznaka se pojavljuje u gornjem lijevom kutu ako je skala vodoravna ili u gornjem desnom kutu ako je okomita. Zadana postavka nije labela.
labelAnchor	Specificira gdje će biti smještena labela.
length	Duljina komponente `scale()`. To je `x` dimenzija ako je skala vodoravno ili `y` dimenzija ako je okomita. Zadana vrijednost je `100` piksela.
menu	Pridruživanje `menubutton()` skupu izbora, skupu objekta `menu` koji sadrži te izbore. Taj objekt mora biti stvoren predajom pridruženog izbornika (menubuttona) konstruktoru kao njegov prvi argument.
offrelief	Prema zadanim postavkama, `Checkbutton()` koristi stil `tk.RAISED` kada je gumb isključen (obrisan); koristite ovu opciju da odredite drugačiji stil reljefa koji će se prikazivati kad je gumb isključen.
offvalue	Uobičajeno, pridružena kontrolna varijabla komponenti `Checkbutton()` postavit će se na `0` kada se poništi (isključi). Možete unijeti alternativnu vrijednost za isključeno stanje postavljanjem vrijednosti `offvalue` na tu vrijednost.
onvalue	Uobičajeno, pridružena kontrolna varijabla gumba `Checkbutton` postavit će se na `1` kad je postavljena (uključeno). Možete unijeti zamjensku vrijednost za uključeno stanje postavljanjem vrijednosti `onvalue` na tu vrijednost.
overrelief	Reljefni stil koji se koristi dok je miš na gumbu; zadano je `tk.RAISED`.
padx	Vanjska *x* podloga koja se dodaje izvan lijeve i desne strane komponente.
pady	Vanjska *y* podloga koja se dodaje izvan gornje i donje strane komponente.
postcommand	Ovu opciju možete postaviti na proceduru koja će biti pozvana svaki put kad netko otvori ovaj izbornik.
readonlybackground	Boja pozadine koja će se prikazati kada je opcija komponente "readonly ".
relief	Određuje vrstu reljefa za gumb. Zadana vrijednost je `tk.RAISED`.
repeatdelay	Pogledati `repeatinterval`, ispod.
repeatinterval	Gumb se obično aktivira samo jednom kada korisnik pusti tipku miša. Ako želite da se gumb aktivira u redovitim intervalima sve dok je tipka miša pritisnuta, postavite ovu opciju na broj milisekundi koja će se koristiti između ponavljanja i ponovite odgodu na broj milisekundi koje treba pričekati prije početka ponavljanja. Na primjer, ako navedete "repeatdelay = 500, repeatinterval = 100", gumb će se aktivirati nakon pola sekunde, a nakon toga svake desetine sekunde, sve dok korisnik ne otpusti tipku miša. Ako korisnik barem nekoliko milisekundi ne pritisne tipku miša, tipka će se normalno aktivirati.
resolution	Inicijalno, `Scale()` se može mijenjati samo u cijelim jedinicama. Postavite ovu opciju na neku drugu vrijednost da biste promijenili najmanji prirast vrijednosti skale. Na primjer, ako je `from _ = -1,0` i `to = 1,0`, a postavite `resolution=0.5`, skala će imati 5 mogućih vrijednosti: `-1,0`, `-0,5`, `0,0`, `+0,5` i `+1,0`.
sashcursor	Nema inicijalne vrijednosti.
sashrelief	Inicijalna vrijednost je `RAISED`.
sashwidth	Inicijalna vrijednost je `2`.
scrollregion	n-torka (*w*, *n*, *e*, *s*) koja definira na kojoj velikoj površini se može pomicati platno (`Canvas`), gdje je *w* lijeva strana, *n* vrh, *e* desna strana, a *s* dno.
selectbackground	Boja pozadine za prikaz odabranih stavki..
selectborderwidth	Širina obruba koji će se koristiti oko odabranih jedinki.
selectcolor	Boja Checkbutton kad je postavljena. Inicijalno je `selectcolor='red'`.
selectimage	Ako ovu opciju postavite na sliku, ona će se pojaviti na potvrdnom gumbu gdje je postavljena.
selectforeground	Boja u prvom planu za prikaz odabranih stavki.
show	Obično se u komponenti `Entry()` pojavljuju znakovi koje upisuju korisnici. Da biste unijeli unos "password" u kojem svaki znak odgovara zvijezdicom, postavite `show='+'`.
showhandle	Nema inicijalnu vrijednost.
showvalue	Trenutna vrijednost skale obično se u tekstualnom obliku prikazuje klizačem (iznad njega za vodoravnu skalu, lijevo za okomitu). Postavite ovu opciju na `0` da biste potisnuli tu oznaku.
sliderlength	Klizač je obično 30 piksela duž duljine skale. Tu duljinu možete promijeniti postavljanjem opcije duljine klizača na željenu duljinu.
spacing1	Određuje koliko se dodatnog okomitog prostora stavlja iznad svakog retka teksta. Ako se linija premota, taj se prostor dodaje samo prije prvog retka koji zauzima na zaslonu. Zadana vrijednost je `0`.
spacing2	Određuje koliko dodatnog okomitog prostora treba dodati između prikazanih redaka teksta

	kada se logička linija premota. Zadana vrijednost je 0.
spacing3	Određuje koliko se dodatnog okomitog prostora dodaje ispod svakog retka teksta. Ako se linija premota, taj se prostor dodaje tek nakon zadnjeg retka koji zauzima na zaslonu. Zadana vrijednost je 0.
state	Postavite ovu opciju na tk.DISABLED da zasivite gumb i onemogućite ga. Ima vrijednost tk.ACTIVE kada je miš iznad nje. Zadana vrijednost je tk.NORMALNO.
tabs	Kontrola kako su tabulatori postavljeni u tekstu.
takefocus	Obično fokus tipkovnice se odnosi na gumbe, a razmak se ponaša jednako kao klik mišem, "pritiskajući" gumb. Opciju fokusa možete postaviti na nulu kako biste spriječili da fokus posjeti gumb.
tearoff	Izbornik se obično može “otkinuti”, prvo mjesto (položaj 0) na popisu izbora zauzima element otkidanja, a dodatni izbori dodaju se počevši od položaja 1. Ako postavite `tearoff = 0`, izbornik neće imati značajku otkidanja, a odabir će biti dodan počevši od položaja 0.
text	Tekst prikazan na gumbu. Koristite interne redove za prikaz više redaka teksta.
textvariable	Instanca klase StringVar () koja je povezana s tekstom na gumbu. Ako se vrijednost promijeni, bit će prikazana na gumbu.
tickinterval	Za prikaz pojedinačnih vrijednostilj skale, postavite ovu opciju na broj, a vrijednosti će se prikazivati na višekratnicima te vrijednosti. Na primjer, ako je `from_ = 0,0, to = 1,0` i `tickinterval=0.25`, oznake će se prikazivati duž skale na vrijednosti `0,0`, `0,25`, `0,50`, `0,75` i `1,00`. Te se oznake pojavljuju ispod skale ako su vodoravne, s lijeve strane ako su okomite. Zadana vrijednost je 0, što suzbija prikaz vrijednosti.
title	Uobičajeno će naslov prozora `Menu()` biti jednak tekstu `Menubutton`-a ili kaskade koja vodi do `Menu`-a. Ako želite promijeniti naslov tog prozora, postavite opciju `title` na taj niz.
to	Realna ili cijela vrijednost koja definira jedan kraj raspona skale; drugi kraj definiran je opcijom `from_`. Za vertikalne skale vrijednost `to` definira dno skale; za vodoravne, desni kraj.
troughcolor	Boja korita.
underline	Zadana vrijednost je -1, što znači da nijedan znak teksta na gumbu neće biti podvučen. Ako nije negativan, odgovarajući tekstni znak bit će podvučen. Na primjer, podcrtavanje = 1 podcrtalo bi drugi znak teksta gumba.
validate	Ovu opciju možete koristiti za postavljanje komponente tako da njezin sadržaj provjerava funkcija provjere valjanosti u određeno vrijeme.
validatecommand	Povratni poziv koji potvrđuje tekst komponente.
value	Kad korisnik uključi radiobutton, njegova se kontrolna varijabla postavlja na trenutnu opciju vrijednosti. Ako je kontrolna varijabla `IntVar`, daje se svakom radio gumbu u grupi različitu opciju cjelobrojnih vrijednosti. Ako je kontrolna varijabla `StringVar`, daje se svakom radio gumbu različitu opciju vrijednosti stringa.
values	*n*-torka koja sadrži važeće vrijednosti za tu komponentu. Poništava `from/ to/ increment`.
variable	Kontrolna varijabla koja prati trenutno stanje komponente `checkbutton`. Obično je ova varijabla `IntVar`, a `0` znači poništeno, a `1` znači postavljeno. Pogledajte opcije `offvalue` i `onvalue`.
vcmd	Isto kao i `validatecommand`.
width	Širina gumba slovima (ako se prikazuje tekst) ili pikselima (ako se prikazuje slika).
wrap	Kontrolira prikaz preširokih linija. Postavite `wrap = WORD` i prekinut će redak nakon posljednje riječi koja će stati. Sa zadanim ponašanjem, `wrap = CHAR`, svaki predug redak bit će prekinut na bilo kojem znaku.
wraplength	Ako je ova vrijednost postavljena na pozitivan broj, to će biti reci teksta koji će stati unutar te duljine..
xscrollincrement	Uobičajeno se platna mogu vodoravno pomicati u bilo koji položaj. To se ponašanje može postići postavljanjem `xscrollincrement` na nulu. Ako ovu opciju postavite na neku pozitivnu dimenziju, platno se može postaviti samo na višekratnike te udaljenosti, a vrijednost će se koristiti za pomicanje pomoću jedinica za pomicanje, na primjer kada korisnik klikne strelice na krajevima trake za pomicanje.
xscrollcommand	Ako se platno može pomicati, postavite ovu opciju na `.set ()` metodu vodoravne trake za pomicanje.
yscrollincrement	Djeluje poput `xscrollincrement`, ali upravlja vertikalnim kretanjem.
yscrollcommand	Ako se platno može pomicati, ova bi opcija trebala biti metoda `.set()` vertikalnog klizača.

U sljedećoj tablici dane su komponente i sa „**+**“ označeno koje opcije sadrže:

	Button	Canvas	Checkbutton	Entry	Frame	Label	LabelFrame	Listbox
activebackground	+		+			+		
activeforeground	+		+			+		
anchor	+		+			+		

	Button	Canvas	Checkbutton	Entry	Frame	Label	LabelFrame	Listbox
bd ili borderwidth	+	+	+	+	+	+	+	+
bg ili background	+	+	+	+	+	+	+	+
bitmap			+			+		
closeenough		+						
confine		+						
command	+		+					
compound			+			+		
cursor	+	+	+	+	+	+	+	+
default	+							
disabledbackground				+				
disabledforeground	+		+	+		+		
exportselection				+				
fg ili foreground	+		+	+		+		
font	+		+	+		+	+	+
height	+	+	+		+	+	+	+
highlightbackground	+	+	+	+	+	+	+	
highlightcolor	+	+	+	+	+	+	+	+
highlightthickness	+	+	+	+	+	+	+	+
image	+		+			+		
indicatoron			+					
insertbackground				+				
insertborderwidth				+				
insertofftime				+				
insertontime				+				
insertwidth				+				
justify	+		+	+		+		
labelAnchor							+	
offrelief			+					
offvalue			+					
onvalue			+					
overrelief	+		+					
padx	+		+		+	+		
pady	+		+		+	+		
readonlybackground				+				
relief	+	+	+	+	+	+	+	+
repeatdelay	+							
repeatinterval	+							
scrollregion		+						
selectbackground		+		+				
selectborderwidth		+		+				

	Button	Canvas	Checkbutton	Entry	Frame	Label	LabelFrame	Listbox
selectcolor			+					
selectimage			+					
selectforeground		+		+				+
show				+				
state	+		+	+		+		
takefocus	+	+	+	+	+	+		
text	+		+			+	+	
textvariable	+		+	+		+		
underline	+		+			+		
validate				+				
validatecommand				+				
variable			+					
width	+	+	+	+	+	+	+	
wraplength	+		+			+		
xscrollincrement		+						
xscrollcommand		+		+				
yscrollincrement		+						
yscrollcommand		+						

	Menubutton	Menu	Message	PanedWindo	Radiobutto	Scale	Scrollbar	Spinbox	Text
activebackground	+	+			+		+		
activeforeground	+	+			+				
activeborderwidth		+							
anchor	+		+		+				
bd ili borderwidth	+	+	+	+		+	+	+	+
bg ili background	+	+	+	+	+	+	+	+	+
bitmap	+		+		+				
borderwidth				+	+				
command					+	+	+	+	
compound					+				
cursor	+	+	+	+	+	+	+	+	+
digits						+			
direction	+								
disabledbackground								+	
disabledforeground	+	+			+			+	
elementborderwidth							+		

	Menubutton	Menu	Message	PanedWindo	Radiobutto	Scale	Scrollbar	Spinbox	Text
exportselection									+
fg ili foreground	+	+	+		+	+		+	+
font		+	+		+	+		+	+
format								+	
from_						+		+	
handlepad				+					
handlesize				+					
height	+		+	+	+				+
highlightbackground					+	+	+		+
highlightcolor	+				+	+	+		+
highlightthickness					+		+		+
image	+	+	+		+				
indicatoron					+				
insertbackground									+
insertborderwidth									+
insertofftime									+
insertontime									+
insertwidth									+
jump							+		
justify	+		+		+			+	
label						+			
labelAnchor									
length						+			
menu	+								
offrelief					+				
offvalue									
orient				+		+	+		
overrelief					+				
padx	+		+		+				+
pady	+		+		+				+
postcommand		+							
relief	+	+	+	+	+	+		+	+
repeatdelay						+		+	
repeatinterval							+	+	
resolution						+			
sashcursor				+					
sashrelief				+					

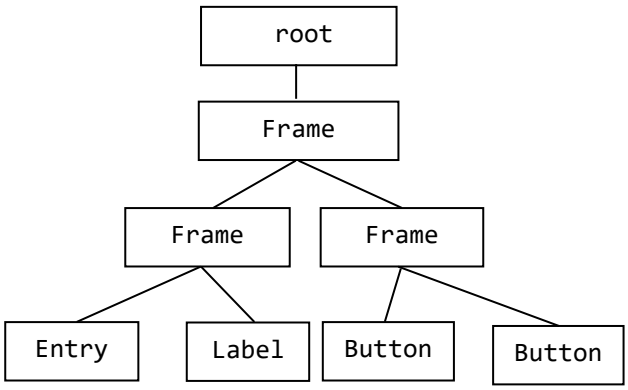
	Menubutton	Menu	Message	PanedWindo	Radiobutto	Scale	Scrollbar	Spinbox	Text
sashwidth				+					
selectbackground									+
selectborderwidth									+
selectcolor		+			+				
selectimage					+				
showhandle				+					
showvalue						+			
sliderlength						+			
spacing1									+
spacing2									+
spacing3									+
state	+				+	+		+	
tabs									+
takefocus					+	+	+		
tearoff		+							
text	+		+		+				
textvariable	+		+		+			+	
title		+							
to						+		+	
troughcolor						+	+		
underline	+		+		+				
validate								+	
validatecommand								+	
value					+				
values								+	
variable					+	+			
vcmd								+	
width	+		+	+	+	+	+	+	+
wrap								+	+
wraplength	+		+		+				
xscrollcommand									+
yscrollcommand									+

Struktura tkintera

U nastavku dajemo osnovnu strukturu dijelova programa napisanog u tkinteru.

KORIJENSKI PROZOR

Osnova GUI programa jest njegov korijenski prozor (*root window*), koji sadrži sve druge elemente (ili "komponte") grafičkog sučelja. Ako GUI zamislimo kao stablo, korijenski prozor bi bio njegov korijen. Stablo može imati grane koje se šire na sve strane, ali je svaki dio stabla neposredno ili posredno vezan s njegovim korijenom (što slijedi iz definicije stabla).



Najčešće ćemo modul **tkinter** uvesti izravno u globalni opseg programa.

```
from tkinter import *
```

IZRADA KORIJENSKOG PROZORA

Da bi se napravio korijenski prozor, instanciramo objekt klase Tk iz modula tkinter:

```
# kreiranje korijenskog prozora
root = Tk()
```

UPRAVITELJI RASPOREDA

Najprije opišimo upravitelje rasporeda, izgleda ili geometrije, kako ih se ponekad naziva. Kasnije ćemo opisati izgradnju komponenti Tkintera u GUI aplikacijama. Ovdje dajemo opis uređenja (postavljanja) objekata komponenti u prozore. To su:

- pack (paket)
- grid (rešetka ili mreža), i
- place (mjesto)

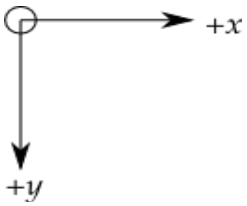
Tri upravitelja izgleda nikada se ne smiju miješati u istom glavnom prozoru! Različite su im namjene. Oni:

- raspoređuju komponente na zaslonu
- registriraju komponente unutar osnovnog sustava prozora
- upravljanje prikazom komponenti na zaslonu

Raspored komponenti na zaslonu uključuje određivanje veličine i njihovog položaja. Komponente mogu pružiti informacije o veličini i poravnanju upraviteljima geometrije, ali upravitelji geometrije uvijek imaju "zadnju riječ" o pozicioniranju i veličini.

Koordinatni sustav

Ishodište koordinatnog sustava nalazi se u gornjem lijevom kutu, s koordinatom *x* koja se povećava udesno, a u koordinatom *y* koja se povećava prema dnu:



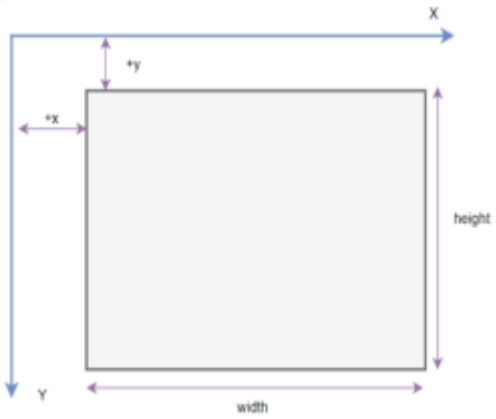
Osnovna jedinica je piksel, a gornji lijevi piksel ima koordinate (0,0). Koordinate koje navedete kao cijele brojeve uvijek se izražavaju u pikselima, ali bilo koja koordinata može biti navedena kao dimenzionirana veličina.

. geometry()

No, prije nego što prijeđemo na opis triju rasporeda komponenti, dajemo opis metode geometry(), koja je temeljna za određivanje veličine, položaja i neke druge atribute izgleda zaslona koji ćemo stvoriti. Pravilo pisanja je:

```
w.geometry ("widthxheight±x±y")
```

gdje su ***width*** i ***height*** pozitivni cijeli brojevi koji predstavljaju širinu i visinu zaslona, ne uključujući njegovo zaglavlje, a ***x*** i ***y*** cijeli brojevi koji, ako su pozitivni, predstavljaju udaljenost rubova zaslona od lijevog ishodišta ekrana, a ako su negativni, od desnog gornjeg vrha. Pišu se unutar stringa s "x" (puta) između, bez razmaka.



Na primjer, u sljedećoj smo skripti fiksirali veličinu prozora na 300 x 150:

```
from tkinter import *; gui = Tk (
      className = 'Prozor 300 x 150')
gui . geometry ('300x150')
gui . mainloop()
```

. pack()

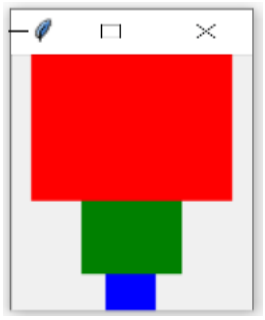
Paket (pack) je najjednostavniji za korištenje. Umjesto da moramo precizno deklarirati gdje bi se komponenta trebala pojaviti na zaslonu, njezine položaje možemo deklarirati međusobno. Naredba pack brine o detaljima. Bez obzira na jednostavnost paketa, upravitelji ovog rasporeda ograničeni su u svojim mogućnostima u usporedbi s preostala dva. Preporučujemo ga za jednostavne aplikacije. Na primjer, za postavljanja brojnih komponenti jedne pored druge ili jedne iznad druge. Pravilo pisanja ovoga rasporeda je:

```
pack          : pack ( [ pack_opcije ] )
```

```
# pack.py
from tkinter import *
root = Tk()
w1   = Label (root, text = "crveno",
                    bg   = "red2")
w1 . pack()
w2 = Label (root, text = "zeleno",
            bg = "green2"). pack()
w3 = Label(root, text = "plavo",
                 bg = "blue2"). pack()
mainloop()
```

Evo još jednog primjera s komponentom Frame():

```
# Frames.py
from tkinter import *
win  = Tk(); Boja = ['red', 'green',
                     'blue']
for i in range (3) : a = 100/2**i; \
  Frame (win, width=a,
         height=a, bg = Boja[i]).pack()
mainloop()
```



Upravitelj pack() poznaje četiri mogućnosti popunjavanja: unutarnje, vanjsko i dodavanje u smjeru x i y. Značenje je sljedeće:

expand

Kada se postavi na True, komponenta se proširuje kako bi se ispunio bilo koji prostor koji se inače ne koristi u njezinu roditelju.

fill

Utvrđuje hoće li komponenta ispuniti bilo koji dodatni prostor koji joj je dodijelio pack ili zadržava vlastite minimalne dimenzije: NONE (zadano), X (ispuniti samo vodoravno), Y (ispuniti samo vertikalno) ili BOTH (ispuniti vodoravno i okomito). Na primjer, promijenili smo definiciju pakiranja w2 u prethodnoj skripti:

```
w1. pack()
w2. pack(fill = X)
w3. pack()
```

padx

Vanjska *x* podloga koja se dodaje izvan lijeve i desne strane komponente.

```
w1 . pack (fill = X, padx = 10)
w2 . pack (fill = X); w3 . pack()
```

pady

Vanjska *y* podloga koja se dodaje izvan gornje i donje strane komponente.

```
w1. pack (fill = X,
          padx = 10)
w2. pack (fill = X)

w3. pack (pady = 10)
```

ipadx

Unutarnja *x* podloga koja se dodaje unutar lijeve i desne strane komponente.

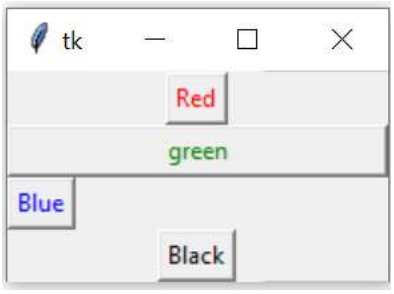
ipady

Unutarnja *y* podloga koja se dodaje unutar gornje i donje strane komponente.

side

Utvrđuje na kojoj će se strani nadređene komponente pakirati: `TOP` (zadano), `BOTTOM`, `LEFT` ili `RIGHT`.

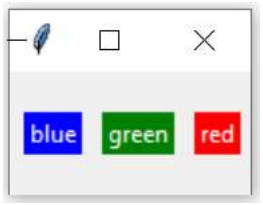
```
# pack0.py
from tkinter import *
root   = Tk()
frame  = Frame (root) . pack ()
bframe = Frame (root)
bframe . pack (side = BOTTOM)

w1 = Button  (frame, text= "Red",
                     fg  = "red")
w1 . pack( expand = True)

w2 = Button (frame, text= "green",
                    fg  = "green")
w2 . pack( fill = BOTH )

w3 = Button (frame, text = "Blue",
                    fg   = "blue")
w3 . pack( side = LEFT )

w4 = Button (bframe, text= "Black",
                     fg  = "black")
w4 . pack( side = BOTTOM)

root. mainloop()
```



Želimo smjestiti tri labele jednu uz drugu i malo skratiti tekst:

```
# pack5.py
from tkinter import *
root = Tk()
C = ['red', 'green', 'blue']
for i in range (3) :
  Label (root, text= C[i],
    bg = C[i], fg = "white").pack (
    padx = 5, pady = 20, side = LEFT)
mainloop()
```

Ako umjesto LEFT napišemo RIGHT, dobit ćemo boje u obrnutom redu:



. grid()

Ovaj upravitelj svaki prozor ili okvir tretira kao tablicu - mrežu redova i stupaca. Da bismo prikazali komponentu *w* na zaslonu, moramo pozvati metodu `grid()` prema sintaksi:

```
w.grid ( [ opcija = vrijednost
         {, opcija = vrijednost }] )
```

Opcije su:

column columnspan in_ ipadx ipady padx pady row rowspan sticky

Opcije **ipadx, ipady, padx i pady** iste su kao i za raspored `pack()`, a preostale su:

column
Broj stupca na kojem se komponenta želi umrežiti, računajući od nule. Zadana vrijednost je nula.

columnspan
Raspon stupaca. Uobičajeno komponenta zauzima samo jednu ćeliju u mreži. Međutim, možete zauzeti više ćelija u retku i spojiti ih u jednu veću ćeliju, postavljanjem broja raspona stupca na broj ćelija. Na primjer,

```
w.grid( row=0, column=2, columnspan=3)
```

stavit će komponentu *w* u ćeliju koja obuhvaća stupce 2, 3 i 4 retka 0.

in_
Da bi komponenta w bila podređena komponenti w2, piše se in_=w2. Novi roditelj w2 mora biti potomak nadređene komponente koji se koristila kad je stvoren w.

row
Broj retka u koji želite umetnuti komponentu, računajući od 0. Inicijalno je sljedeći nenastanjeni redak s većim brojem.

rowspan
Raspon redova. Uobičajeno komponenta zauzima samo jednu ćeliju u mreži. Možete zgrabiti više susjednih ćelija stupca, međutim, postavljanjem opcije raspona redova na broj ćelija za hvatanje. Ova se opcija može koristiti u kombinaciji s opcijom raspona stupaca da biste zgrabili blok ćelija. Na primjer,

```
w.grid (row=3, column=2, rowspan=4,
        columnspan=5)
```

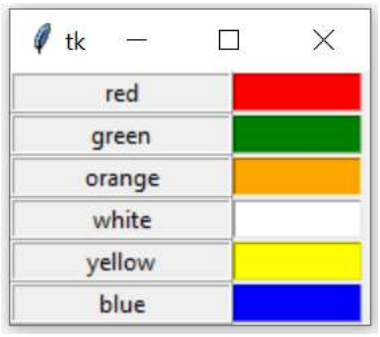
postavit će kompoenentu *w* u područje nastalo spajanjem 20 ćelija, s brojevima redaka 3-6 i brojevima stupaca 2-6.

sticky
Ova opcija određuje kako rasporediti svaki dodatni prostor unutar ćelije koju komponenta ne zauzima u svojoj prirodnoj veličini. Inicijalno je komponenta centrirana u ćeliji. Inače, sticky može biti spajanje nizova nula ili N, E, S, W, NE, NW, SE i SW, stranice kompasa koje označavaju stranice i uglove ćelije na koje se komponenta nalazi.

Upravitelj geometrije mreže postavlja komponentu u dvodimenzionalnu tablicu koja se sastoji od određenog broja redaka i stupaca. Položaj komponente definiran je brojem reda i brojem stupca. Komponenta s istim brojem stupca i različitim brojevima redaka bit će jedni iznad drugih ili ispod njih. Sukladno tome, komponente s istim brojem retka, ali različitim brojevima stupaca, nalazit će se na istoj "liniji" i bit će jedan pored drugog, tj. lijevo ili desno.

Veličina mreže ne mora biti definirana, jer upravitelj automatski određuje najbolje dimenzije za upotrijebljene widgete. Primjer:

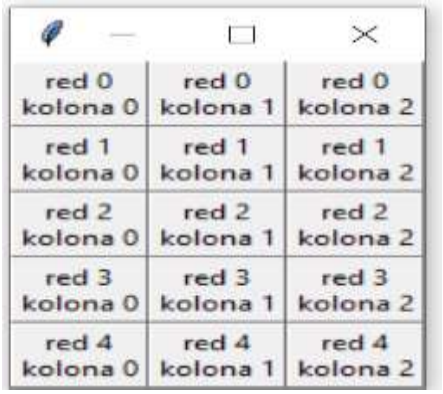
```
# grid.py
from tkinter import *
colours = ['red','green','orange',
           'white','yellow','blue']
r = 0
for c in colours:
  Label(text=c, relief = RIDGE,
     width=15).grid(row=r, column=0)
  Entry(bg=c, relief=SUNKEN,
     width=10).grid(row=r, column=1)
  r += 1
mainloop()
```



Sljedeća skripta koda pomoći će vam u stvaranju mreže 5 × 3 okvira s upakiranim komponentama Label:

```
# Mreža.py
from tkinter import *
win = Tk()
for i in range(5):
  for j in range(3):
    frame = Frame (
      master = win,
      relief = RAISED,
      borderwidth = 1 )
    frame.grid(row=i, column=j)
```

```
    label = Label (master=frame,
      text = f"red {i}\nkolona {j}")
    label.pack()
win.mainloop()
```



. place()

Upravitelj geometrije `place()` omogućuje izričito postavljanje položaja i veličine prozora, bilo u apsolutnom smislu, bilo u odnosu na drugi prozor. Može se primijeniti na sve standardne komponente. Pravilo pisanja je uobičajeno:

w.place (*place_opcije*)

Lista mogućih opcija je:

anchor

Točno mjesto ("sidro") komponente na koje se odnose druge opcije: može biti `N`, `E`, `S`, `W`, `NE`, `NW`, `SE` ili `SW`, smjernice kompasa koje označavaju kutove i stranice komponente; zadani je `NW` (gornji lijevi kut komponente).

bordermode

`INSIDE` (zadano) da naznači da se druge opcije odnose na roditeljevu unutrašnjost (zanemarujući roditeljevu granicu); `OUTSIDE` inače.

height, width

Visina i širina komponente u pikselima.

relheight, relwidth

Visina i širina (komponente) kao realne vrijednosti između `0.0` i `1.0`, kao djelić visine i širine nadređene komponente.

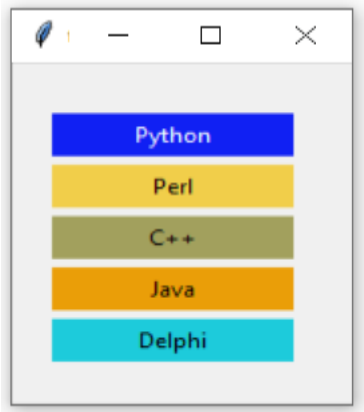
relx, rely

Horizontalni i vertikalni ofset (pomak) kao realni broj između `0.0` i `1.0`, frakcija visine i širine nadređene komponente.

x, y

Horizontalni i vertikalni pomak u pikselima. Pogledajmo primjer:

```
# place.py
from tkinter import *
from random  import randrange as rnd
root = Tk()
# widthxheight+x_offset+y_offset:
root.geometry("170x200+30+30")
L=['Python','Perl','C++','Java','Delphi']
for i in range(5):
    RGB     = [rnd(256)
               for x in range(3)]
    s       = (0.299*RGB[0]
              +0.587*RGB[1] +0.114*RGB[2])
    RGB_hex = ("%02x%02x%02x"
                % tuple(RGB) )
    bg_col   = '#' +RGB_hex
    l        = Label(root, text = L[i],
      fg = 'White' if s<120 else 'Black',
      bg = bg_col)
    l.place(x = 20, y = 30 + i*30,
            width=120, height=25)
root.mainloop()
```



U ovom se primjeru poigravamo s bojama, tj. svakoj labeli dodijeljujemo drugu boju koju slučajno kreiramo metodom `randrange()` slučajnog modula. Izračunavamo svjetlinu (vrijednost sive) svake boje. Ako je svjetlina manja od `120`, boju naprijed (`fg`) naljepnice postavljamo na bijelu, inače na crnu, kako bi se tekst lakše čitao. Još jedan primjer:

```
# Place2.py
from tkinter import *
master = Tk()
master . geometry ("200x200")
b1     = Button (master,
           text = "Pritisni me!")
b1     . place (relx = 1, x =-2,
                y = 2, anchor = NE)
```

```
l = Label (master,
           text = "sjevero zapad")
l  . place (anchor = NW)
b2 = Button (master, text = "CENTAR")
b2 . place  (relx = 0.5, rely = 0.5,
             anchor = CENTER)
mainloop()
```

DIMENZIJE

Različite duljine, širine i druge dimenzije dodataka mogu se opisati u mnogo različitih jedinica.

- Ako dimenziju postavite na cijeli broj, pretpostavlja se da je u pikselima.
- Možete odrediti jedinice postavljanjem dimenzije na niz koji sadrži broj iza kojeg slijedi:

c	centimetri
i	inči
m	milimetri
p	redovi printera (oko 1/72″)

TKINTEROVE VARIJABLE

Neke se komponente (unos teksta, radio gumbi i tako dalje) mogu izravno povezati s varijablama programa pomoću posebnih opcija: varijabli, tekstualnih varijabli, uključenih ili isključenih vrijednosti. Ova veza djeluje u oba smjera: ako se varijabla promijeni iz bilo kojeg razloga, komponenta s kojom je povezana ažurirat će se tako da odražava novu vrijednost. Te se kontrolne varijable koriste kao redovite Python varijable za zadržavanje određenih vrijednosti. Nije moguće predati redovnu Python varijablu widgetu putem varijable ili opcije varijabilnog teksta. Jedine vrste varijabli za koje ovo djeluje su varijable koje su podklase iz klase `Variable`, definirane u Tkinterovom modulu. Deklarirane su kao:

```
x = StringVar()  # string;inic. je ""
x = IntVar()     # int; inic. je 0
x = DoubleVar()  # float;inic. je 0.0
x = BooleanVar() # boolean; 0 False,
                   1 za True
```

Čitanje trenutne vrijednosti takve varijable je metodom `get()`, a promjena njezine vrijednosti je metodom `set()`. Stvaranje varijable je na uobičajeni način. Na primjer, *`label_text = tk.StringVar()`*

STANDARDNI ATRIBUTI

Prije nego što pogledamo komponente, pogledajmo kako su navedeni neki od njezinih uobičajenih atributa - poput veličina, boja i fontova.

- Svaka komponenta ima skup opcija koje utječu na njezin izgled i ponašanje - atribute poput fontova, boja, veličina, tekstualnih naljepnica i slično.
- Mogu se odrediti opcije prilikom pozivanja konstruktora komponente koristeći argumente ključnih riječi, kao što je `text='PANIC!'` ili `high=20`.
- Nakon što je stvorena komponenta, kasnije se može promijeniti u bilo koju opciju pomoću metode `.config()` komponente. Možete pristupiti trenutnoj postavci bilo koje opcije pomoću metode `.cget()`.

BOJA

Postoje sljedeći načini specificiranja boje:

- Niz koji određuje udio crvene, zelene i plave boje u heksadecimalnim znamenkama:

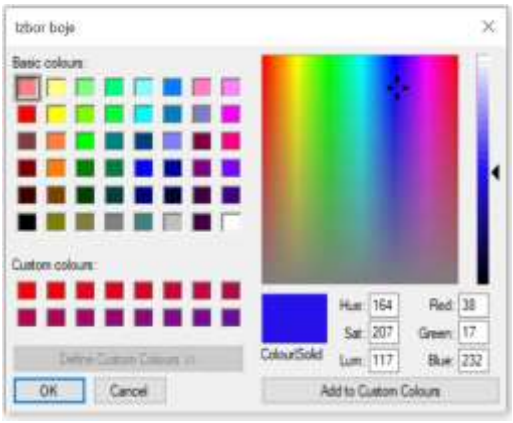
`#rgb`	Četiri bita po boji
`#rrggbb`	Osam bitova po boji
`#rrrgggbbb`	Dvanaest bitova po boji

Na primjer, `'#fff'` je bijela, `'#000000'` je crna, `'#000fff000'` je zelena, a `'#00ffff'` je cijan (zelena plus plava).

- Može se koristiti bilo koji lokalno definirani standardni naziv boje. Boje `'white'`, `'black'`, `'red'`, `'green'`, `'blue'`, `'cyan'`, `'yellow'`, i `'magenta'` uvijek će biti dostupne.

Tablicu boja možemo dobiti sa skriptom:

```
# Boje_Tablica.py
from tkinter.colorchooser import (
     askcolor )
Boja = askcolor (color = "blue",
            title = "Izbor boje")
S = Boja [1];  R, G, B = Boja [0]
print (int(R), int(G), int(B), S)
```



Na primjer, izabrana je boja: `37 22 226 #2516e2`

FONT

Ovisno o vašoj platformi, mogu postojati do tri načina za određivanje stila tipa. Navodimo dva najčešća.

- Kao *n*-torka čiji je prvi element obitelj fontova, nakon čega slijedi veličina (u točkama ako je pozitivna, u pikselima ako je negativna), po želji slijedi niz koji sadrži jedan ili više modifikatora stila podebljano, kurziv, podcrtano i precrtano. Primjeri: `('Helvetica', '16')` za normalnu Helveticu visine 16; `('Times', '24', 'bold italic')` za Times visine 24 točke, podebljano, kurziv, a za Times od 20 piksela, podebljano, pisat ćemo `('Times', -20, 'bold')`.
- Možete stvoriti „objekt fonta“ uvozom modula `import tkFont`, i upotrebom njegovog konstruktora klase Font: `font = tkFont.Font(opcija, ...)`, gdje opcije uključuju:

family	font `family` ime, kao string.
size	visinu fonta kao cijeli broj. Ako je visina *n* piksela, koristiti *-n*.
weight	`'bold'` ili `'normal'`
slant	`'italic'` za ukošen, `'roman'` za neukošen
underline	1 za podcrtano, 0 za normalno
overstrike	1 za precrtano, 0 za normalno.

Na primjer,

```
Helv36 = tkFont.Font(
         family = 'Helvetica',
         size   = 36, weight = 'bold')
```

Da bi se dobio popis svih porodica fontova dostupnih na vašoj platformi, pozovite ovu funkciju:

```
tkFont.families()
```

Povratna vrijednost je popis nizova. Prije pozivanja ove funkcije morate stvoriti svoj korijenski prozor.

Sljedeće su metode definirane na svim objektima fontova:

. actual (*option*=None)

Ako ne proslijedite nikakve argumente, vratit ćete rječnik stvarnih atributa fonta, koji se mogu razlikovati od onih koje ste zatražili. Da biste vratili vrijednost atributa, dodajte njegovo ime kao argument.

. cget (*opcija*)

Vraća vrijednost dane opcije.

. tagure (*opcija, ...*)

Ovom metodom se može promijeniti jednu ili više opcija na fontu. Na primjer, ako imate objekt `Font` pod nazivom `F`, ako ga pozovete

```
F.configure(family = 'times',size = 16)
```

taj će se font promijeniti u 16pt Times, a promijenit će se i sve komponente koje koriste taj font.

. copy()

Vraća kopiju Font objekta.

. measure (*tekst*)

Proslijedite ovoj metodi tekst i vratit će broj piksela širine koji će zauzeti u fontu. Upozorenje: neki ukošeni znakovi mogu se protezati izvan tog područja.

. metrics ([*opcija*])

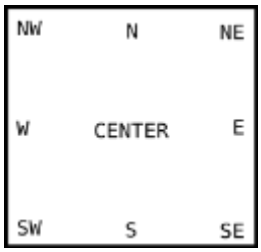
Ako je ova metoda pozvana bez argumenta, vratit će rječnik svih mjernih podataka fonta. Vrijednost samo jednog mjernog podatka može se dohvatiti tako da se njegovo ime navede kao argument. Mjerni podaci uključuju:

ascent	Broj piksela visine između osnovne linije i vrha najvišeg uspona.
descent	Broj piksela visine između osnovne linije i dna najnižeg uspona.
fixed	`0` za font promjenjive širine i `1` za font s jednostrukim razmakom.
linespace	Ukupno piksela visine. Ovo je vodeći tip temeljnog skupa u danom fontu.

SIDRA

Postoje brojne konstante sidra pomoću kojih se može kontrolirati gdje su stavke postavljene u odnosu na

njihov kontekst. Na primjer, sidra mogu odrediti gdje se komponenta nalazi unutar okvira kada je okvir veći od komponente. Te su konstante dane kao točke kompasa, gdje je sjever gore, a zapad lijevo. Konstante sidra prikazane su na ovom dijagramu:



Na primjer, ako se izradi mala komponenta unutar velikog okvira i upotrijebi se opciju `anchor = tk.SE`, komponenta će se postaviti u donji desni kut okvira. Ako se umjesto toga upotrijebi `anchor = tk.N`, komponenta će biti centrirana uz gornji rub.

Sidra se također koriste za definiranje mjesta na kojem se tekst nalazi u odnosu na referentnu točku. Na primjer, ako se `tk.CENTER` koristiti kao sidro teksta, tekst će biti centriran vodoravno i okomito oko referentne točke. Sidro `tk.NW` postavit će tekst tako da se referentna točka podudara sa sjeverozapadnim (gornjim lijevim) kutom okvira koji sadrži tekst. Sidrište `tk.W` usredotočit će tekst uspravno oko referentne točke, pri čemu će lijevi rub okvira za tekst prolaziti kroz tu točku i tako dalje.

RELJEFNI STILOVI

Reljefni stil komponente odnosi se na određene simulirane trodimenzionalne efekte oko njegove vanjske strane. Evo snimke zaslona niza gumba koji prikazuju sve moguće stilove reljefa: Širina ovih granica ovisi o opciji širine obruba komponente. Gornja grafika pokazuje kako izgledaju s obrubom od 5 piksela; zadana širina obruba je 2.

BITMAPE

Za `bitmap` opcije u komponentama, ove bitmape su zajamčeno dostupne:

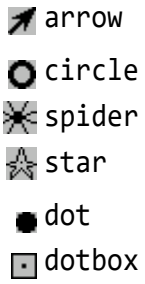
Ovdje su prikazane `Button` komponente koje nose standardne bitmape. S lijeva na desno su:

```
'error'  'gray75'   'gray50' 'gray25'
'gray12' 'hourglass' 'info' 'questhead'
'question' 'warning'.
```

Može se koristiti vlastite bitmape. Svaka datoteka u `.xbm` (X bitna mapa) formatu će raditi. Umjesto standardnog imena bitmape, treba upotrijebititi niz `'@'` nakon kojeg slijedi naziv puta `.xbm` datoteke.

KURSORI

Na raspolaganju je popriličan broj različitih pokazivača miša. Njihova imena i grafike prikazani su tablici:



Točna grafika može se razlikovati ovisno o vašem operativnom sustavu.

PROGRAMIRANJE VOĐENO DOGAĐAJIMA

GUI su programi najčešće *vođeni događajima* (engl. *event driven*), što znači da se odazivaju na akcije korisnika bez obzira na redosljed kojim se one odvijaju. Takav pristup uvodi malo drugačiji način razmišljanja o pisanju softvera.

Pišući program vođen događajima povezujemo (pridružujemo) događaje (ono što se može dogoditi objektima programa) s procedurama za obradu događaja, tj. kodom koji se izvršava kad se događaj desi.

Definiranjem svih objekata, događaja i procedura za njihovu obradu određujemo kako će naš program raditi. Potom se pokreće izvršavanje programa tako što počinje *petlja za događaje* u kojoj program čeka da nastanu događaji koje smo opisali. Kad god nastane jedan od tih događaja, program ga obrađuje onako kako smo zadali. Pravilo pisanja je:

```
widget.bind (događaj, procedura)
```

Ako se u *widgetu* desi događaj koji se podudara s opisom događaja, dani se obrađivač poziva s procedurom koja opisuje događaj. Primjer:

```
# Klikovi.py
from tkinter import *
root = Tk()
def callback(event):
    print ("clicked at",
           event.x, event.y)
frame = Frame (root,
        width = 100, height = 100)
frame . bind ("<Button-1>", callback)
frame . pack ()
root  . mainloop()
```

Formati događaja

Tkinter koristi takozvane sekvence događaja kako bi korisniku omogućio da definira koje događaje, i specifične i općenite, želi povezati s komponentama. To je prvi argument "događaj" metode vezanja. Slijed događaja dan je u obliku niza, koristeći sljedeću sintaksu:

```
<modifikator-tip-detalj>
```

Polje tipa važan je dio specifikatora događaja, dok polja "modifikator" i "detalj" nisu obavezna i u mnogim su slučajevima izostavljena. Koriste se za pružanje dodatnih informacija za odabrani "tip". "Tip" događaja opisuje vrstu događaja koji se veže, npr. radnje poput klikova mišem, pritiskanja tipki ili komponente da bi se dobio fokus unosa.

<Button>

Pritisnuta je tipka miša s pokazivačem miša preko komponente. Dio detalja određuje koji gumb, npr. lijeva tipka miša definirana je događajem **<Button-1>**, srednja tipka **<Button-2>**, a krajnja desna tipka miša **<Button-3>**. **<Button-4>** definira događaj pomicanja prema gore na miševima s potporom kotača i **<Button-5>** pomicanje prema dolje.

Ako se pritisne tipka miša preko komponente i drži pritisnuta, Tkinter će automatski „zgrabiti" pokazivač miša. Daljnji događaji miša poput pokreta i otpuštanja poslat će se trenutnoj komponenti, čak i ako je miš premješten izvan nje. Trenutačni položaj pokazivača miša, u odnosu na komponentu, naveden je u **x** i **y** članovima događaja prosljeđenih povratnom pozivu. Može se koristiti **ButtonPress** umjesto **Button** ili ga čak potpuno izostaviti: **,,** i <1> su svi sinonimi.

<Motion>

Miš se pomiče pritisnutim gumbom miša. Da bi se odredila lijeva, srednja ili desna tipka miša, upotrijebiti **<B1-Motion>**, **<B2-Motion>** ili **<B3-Motion>**.

Trenutačna pozicija pokazivača miša navedena je u **x** i **y** članovima objekta događaja u proslijeđenom povratnom pozivu, tj. **event.x**, **event.y**

<ButtonRelease>

Događaj, ako je gumb otpušten. Da biste odredili lijevu, srednju ili desnu tipku miša, upotrijebite

<ButtonRelease-1>, <ButtonRelease-2> ili <ButtonRelease-3>.

Trenutačna pozicija pokazivača miša navedena je u **x** i **y** članovima objekta događaja proslijeđenih povratnom pozivu.

<Double-Button>

Slično događaju <button>, ali se gumb klikne dvaput umjesto jedanput. Da bi se odredila lijeva, srednja ili desna tipka miša, upotrijebiti

<Double-Button-1>, <Double-Button-2>, ili <Double-Button-3>.

Kao prefikse može se koristiti Double ili Triple. Imati na umu da će se, ako se vežete na jedan klik (<Button-1>) i dvostruki klik (<Double-Button-2>), pozvati obje *veza.objekt* će biti proslijeđena.

<Enter>

Pokazivač miša ušao je u komponentu. Pažnja: To ne znači da je korisnik pritisnuo tipku Enter! U tu svrhu koristi se **<Return>**.

<Leave>

Pokazivač miša je lijevo od komponente.

<FocusIn>

Fokus tipkovnice premješten je na ovu komponentu ili na podklasu te komponente.

<FocusOut>

Fokus tipkovnice premješten je s ove na drugu komponentu.

<Return>

Korisnik je pritisnuo tipku Enter. Može se vezati za gotovo sve tipke na tipkovnici: posebne tipke su Cancel (tipka Break), BackSpace, Tab, Return (tipka Enter), Shift_L (bilo koja tipka Shift),

Control_L (bilo koja tipka Control), Alt_L (bilo koja tipka Alt), Pause, Caps_Lock, Escape, Prior (Page Up), Next (Page Down), End, Home, Left, Up, Right, Down, Print, Insert, Delete, F1, F2, F3, F4, F5, F6, F7, F8, F9, F10, F11, F12, Num_Lock i Scroll_Lock.

<Key>
Korisnik je pritisnuo bilo koju tipku. Tipka je navedena u članu char objekta događaja koji se prosljeđuje povratnom pozivu (ovo je prazan niz za posebne tipke).

a
Korisnik je pritisnuo tipku "a". Većina znakova koji se mogu ispisati mogu se koristiti takvi kakvi jesu. Izuzetak su razmak i minus. Treba imati na umu da je 1 veza na tipkovnici, dok je <1> obvezujući gumb.

<Shift-Up>
Korisnik je pritisnuo strelicu prema gore, držeći pritisnutu tipku Shift. Mogu se koristiti prefiksi poput Alt, Shift i Control.

<Configure>
Veličina komponente se promijenila. Nova veličina navedena je u atributima širine i visine objekta događaja koji se prosljeđuju povratnom pozivu. Na nekim platformama to može značiti da se lokacija promijenila.

Atributi događaja

Definirani su atributi događaja opisani u nastavku.

widget
Komponenta koja generira ovaj događaj. Ovo je valjana instanca komponente tkintera, ne ime. Ovaj atribut je postavljen za sve događaje.

x, y
Tekuća pozicija miša, u pikselima.

x_root, y_root
Tekuća pozicija miša relativna na najviši lijevi kut zaslona, u pikselima.

char
Znakovni kôd (samo za događaje tipkovnice), kao string.

keysym
Simbol tipkovnice (samo za događaje tipkovnice).

keycode
Kôd tipkovnice (samo za događaje tipkovnice).

num
Broj gumba (samo događaji gumba miša).

width, height
Nova veličina komponente, u pikselima (samo konfiguriranje događaja).

type
Tip događaja.

Mijenjanje korijenskog prozora

Korijenski prozor može se modificirati pozivanjem njegovih metoda. Na primjer:

```
# Pokretanje_petlje.py
from tkinter import *
root = Tk()
# modificiranje prozora
root.title ("Jednostavni GUI")
root.geometry ("300x100")
```

Metoda title() zadaje naslov korijenskom prozoru, a metoda geometry() zadaje njegove dimenzije u pikselima. Argument je string koji sadrži širinu ("300") i visinu ("200") prozora razdvojene znakom "x".

Pokretanje petlje za događaje

Na kraju se mora pokrenuti petlja za događaje u korijenskom prozoru, što čini metoda mainloop() korijenskog prozora:

```
# pokreće izvršavanje petlje za
# događaje u prozoru
root.mainloop()
```

Rezultat toga je da će prozor biti otvoren i čekat će na događaje koje će obraditi. S obzirom na to da nismo definirali nijedan događaj, prozor neće ništa „raditi“. Ali, to je kompletan prozor nad kojim su definirane opcije za podešavanje širine i visine, minimaliziranje ili zatvaranje.

Frame

Frame (okvir) je pravokutno područje na zaslonu. Uglavnom se koristi kao glavna geometrija za druge komponente, ili za pružanje dodataka između ostalih komponenti.

Label

Komponenta **Label** (natpis, oznaka) prikazuje jedan ili više redaka teksta u istom stilu, ili bitmap ili slike. Ne postoje posebne metode za widgete naljepnica, osim uobičajenih.

LabelFrame

Komponenta **LabelFrame**, slično komponenti Frame, je prostorni spremnik - pravokutno područje koje može sadržavati druge komponente. Međutim, za razliku od komponente Frame, ova komponenta omogućuje prikazivanje natpisa kao dijela obruba oko područja.

Evo primjera komponente LabelFrame koja sadrži dva gumba. Imajte na umu da oznaka "Važne kontrole" prekida obrub. Ova komponenta ilustrira zadani reljef GROOVE i zadano sidro naljepnice „nw“, koje oznaku postavlja na lijevu stranu vrha okvira.

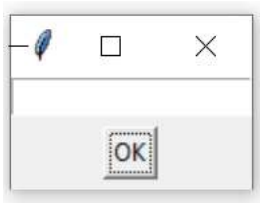
ŽARIŠTE

Metode focus_set() i focus_get() su univerzalne metode komponenti. Mogu se primijeniti i na Tk () metodi.

. focus_set()

Ova se metoda koristi za postavljanje fokusa na željenu komponentu onda i samo ako je glavni prozor fokusiran. Primjer:

```
# Focus_set.py
from tkinter import *
korijen = Tk()
e1 = Entry (korijen)
e1 . pack (expand = 1, fill = BOTH)
e2 = Button(korijen, text ="OK")
e2 . focus_set ()   # fokus!
e2 . pack (pady = 5)
mainloop()
```



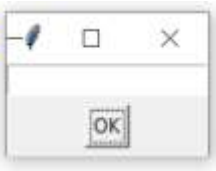
. focus_get()

Ova metoda vraća ime komponente koja je trenutno u fokusu. Može se koristiti za bilo koju komponentu.

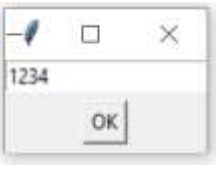
U sljedećem smo programu objedinili obje metode.

```
# Focus.py
from tkinter import *
korijen = Tk()

def focus(event): # klik na Button-1
  widget = korijen.focus_get()
  print (widget, "ima focus")

e1 = Entry (korijen)
e1 . pack (expand = 1, fill = BOTH)
e2 = Button(korijen, text = "OK")
e2 . focus_set () # fokus!
e2 . pack (pady = 5)
korijen.bind_all ("<Button-1>",
                  lambda e: focus(e))
mainloop()
```

```
.!button ima focus
```




```
.!button ima focus
.!entry ima focus
```



Pokretanjem programa gumb je bio označen kao fokusiran. Prvim klikom bilo gdje na zaslonu izvan komponente Entry bilo je ispisano button ima focus. Klikom u polje komponente Entry fokus je prešao na tu komponentu što je potvrđeno ispisom entry ima focus.

GOVORIMO PYTHONSKI

U ovom ćemo dijelu prikazati neke komponente Tkintera „u akciji“. Za razliku od wxPythona i PyQt5, nedostatak je Tkintera što nema programa za dizajniranje, već moramo „metodom pokušaja“ predvidjeti položaj pojedinih komponenti.

KADA KORISTITI FRAME

`Frame` se koristi za grupiranje ostalih komponenti u složene izglede (rasporede). Također se koristi za popunjavanje i kao osnovna klasa pri implementaciji složenih dodataka. `Frame` se može rabiti i kao dekoracija. Na primjer,

```
# Frame.py
from tkinter import *
master = Tk()
Label (text = "prvi").pack()
separator = Frame (height = 2, bd = 1,
                   relief = SUNKEN)
separator.pack(fill = X, padx = 5,
               pady = 5)
Label (text = "drugi").pack()
mainloop()
```

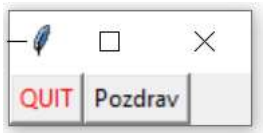
`Frame` se može koristiti kao držač mjesta za video prekrivače i druge vanjske procese. Da bi se na ovaj način koristio okvir, treba postaviti boju pozadine na prazan niz (to sprječava ažuriranja i ostavlja kartu boja na miru), pakirati je kao obično i upotrijebiti metodu `window_id` da biste dobili „ručku“ prozora koja odgovara okviru.

```
frame = Frame(width     = 768,
              height    = 576,
              bg        = "",
              colormap  = "new")
frame.pack()
```

Button()

`Button()` je standardni Tkinterov dodatak koji se koristi za razne vrste gumba. Gumb je komponenta koja je dizajnirana za interakciju korisnika, tj. ako se pritisne gumb klikom miša, mogla bi se pokrenuti neka radnja. Oni također mogu sadržavati tekst i slike poput naljepnica (labela). Iako naljepnice mogu prikazivati tekst u raznim fontovima, gumb može prikazivati tekst samo jednim fontom. Tekst gumba može obuhvaćati više redaka. Funkcija ili metoda Pythona mogu se povezati s gumbom. Ova funkcija ili metoda izvršit će se ako se tipka pritisne na neki način. Sljedeća skripta definira dva gumba: jedan za napuštanje aplikacije i drugi za radnju, tj. Ispis teksta `"Tkinter je lagan!"` na ekranu.

```
# Button.py
from tkinter import *
def write_slogan () :
    print ("Tkinter je lagan!")
root   = Tk()
button = Button (root, text = "QUIT",
                        fg = "red",
                   command = quit)
button.pack (side = LEFT)
slogan = Button (root,
            text    = "Pozdrav",
            command = write_slogan)
slogan.pack(side = LEFT)
root.mainloop()
```



BOJA

Postoje aplikacije u kojima korisnik treba imati mogućnost odabira boje. Tkinter nudi skočni izbornik za odabir boje. U tu svrhu moramo uvesti modul `tkinter.colorchooser` i koristiti metodu `askColor`:

```
tkinter.colorchooser.askcolor (
        color, option = value, ...)
```

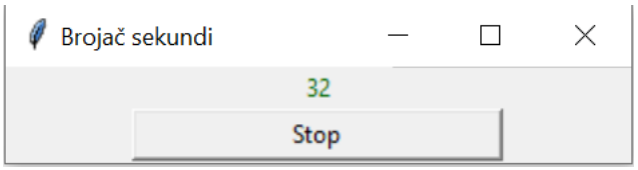
Ako kliknemo gumb "Izaberi boju" na skočnom prozoru, povratna vrijednost askcolor() skup je s dva elementa, oba predstavljaju odabranu boju, npr. ((106, 150, 98), '#6a9662'). Povratak prvog elementa je skup (R, G, B), s RGB prikazom u decimalnim vrijednostima (od 0 do 255). Povratak drugog elementa je heksadecimalni prikaz odabrane boje. Možemo birati izabrane boje u kvadratima lijevo ili neku boju „između", desno. Inicijalna boja je "grey".

```
# Boje.py
from tkinter import *
from tkinter.colorchooser import (
             askcolor )
def callback ():
  Boja = askcolor (color = "#6A9662",
           title = "Izbor boje")
  print (Boja)
root = Tk()
Button (root,
  text    = 'Izaberi boju',
  fg      = "darkgreen",
  command = callback
  ).pack(side=LEFT, padx=10)
Button (text = 'Kraj',
  command = root.quit,
  fg      = "red"). pack(side=LEFT,
  padx    = 10)
mainloop()
```

Label()

Sljedeća skripta prikazuje primjer, gdje se oznaka dinamički uvećava za 1 dok se ne pritisne gumb za zaustavljanje:

```
# Label0.py
from tkinter import *
n = 0
def n_label (label) :
  def Brojač () :
    global n
    n += 1
    label.config (text = str(n))
    label.after  (1000, Brojač)
  Brojač ()
root = Tk()
root . title ("Brojač sekundi")
label = Label (root,
              fg = ("dark green"))
label.pack();  n_label(label)
button = Button (root, text = 'Stop',
    width = 25, command = root.destroy)
button.pack()
root.mainloop()
```



Evo još jednog primjera dinamičke oznake, digitalnog sata:

```
# digitalni_sat.py
from tkinter import Label
from   time  import strftime as Time
dig_sat           = Label()
dig_sat[ 'text' ] = '00:00:00'
dig_sat[ 'font' ] = 'Consolas 40 bold'
dig_sat[ 'fg'   ] = 'blue'
dig_sat           . pack()
def tic(): dig_sat[ 'text' ] = Time(
           "%H:%M:%S" )
def tac(): tic(); dig_sat.after (1000,
           tac)
tac()
```

Font

Neke komponente Tk, poput naljepnice, teksta i platna, omogućuju nam da odredimo fontove koji se koriste za prikaz teksta. To se može postići postavljanjem atributa font, obično putem opcije konfiguracije font. Atribut fg može se koristiti za tekst u drugoj boji, a atribut bg za promjenu boje pozadine. Slijedi primjer:

```
# Fontovi.py
from tkinter import *
root = Tk()
Label (root,
  text = "Crveno, font Times",
  fg   = "red",
  font = "Times").pack()
Label (root,
  text = "Zeleno, font Helvetica",
```

```
    fg   = "light green",
    bg   = "dark green",
    font = "Helvetica 16 bold italic"
           ).pack()
Label (root,
    text = "Plavo, font Verdana bold",
    fg   = "blue",
    bg   = "yellow",
    font = "Verdana 10 bold").pack()
Label (root,
    text = "Žuto, font Consolas bold",
    fg   = "yellow",
    bg   = "black",
    font = "Consolas 20 bold").pack()

root.mainloop()
```

Optionmenu()

OptionMenu je u osnovi padajući ili skočni izbornik koji prikazuje grupu objekata na kliku ili tipkovnici i omogućuje korisniku da odabere jednu po jednu opciju. Evo jdnostavnog primjera:

```
# Optionmenu.py
from tkinter import *

root = Tk();  s = StringVar()
s . set ('a') # inicijalni izbor
om          = OptionMenu (root, s, 'a',
              'b', 'c', 'd')
om['font'] = 'Consolas', 14, 'bold'
om.pack()
root.mainloop()
```

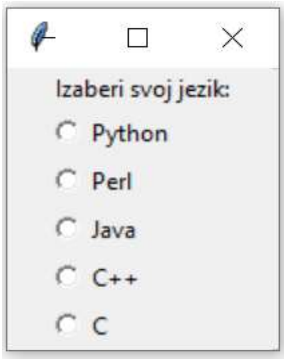
Radiobutton()

Radiobutton, koji se ponekad naziva i opcijski gumb, omogućava korisniku da odabere (točno) jednu od unaprijed definiranih skupova opcija. Radio gumbi mogu sadržavati tekst ili slike. Gumb može prikazati tekst samo jednim fontom. Python funkcija ili metoda mogu se povezati s radio gumbom.

Ova funkcija ili metoda pozvat će se ako pritisnete ovaj radio gumb. Radio gumbi su nazvani prema fizičkim tipkama koje se koriste na starim radio uređajima za odabir valnih područja ili unaprijed postavljenih radio stanica. Pritiskom na gumb vrijednost ove varijable mijenja se u unaprijed određenu vrijednost.

Općenito, postoji više od dva radio gumba. Bilo bi nezgrapno kad bismo morali definirati i zapisati svaki gumb. Rješenje je prikazano u sljedećem primjeru. Imamo popis "jezika", koji sadrži tekstove gumba i odgovarajuće vrijednosti. Pomoću *FOR petlje* možemo stvoriti sve radio gumbe. Popis *n*-torki 'jezika' sadrži i jezike i vrijednosti koje će se dodijeliti varijabli 'V' ako se klikne na odgovarajući jezik.

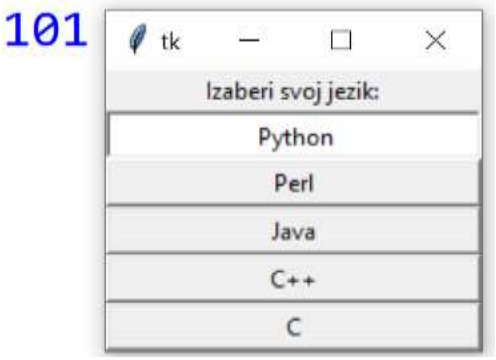
```
# Radiobutton.py
from tkinter import *

root = Tk()
V = IntVar ()
V . set (1)  # inicijalni izbor
Jezici = [
  ("Python", 101), ("Perl", 102),
  ("Java",   103), ("C++",  104),
  ("C",      105) ]

def Prikazi (): print (V.get())
Label ( root,
  text    = "Izaberi svoj jezik:",
  justify = LEFT,
  padx    = 20). pack()

for jezik, val in Jezici:
  Radiobutton (root,
    text     = jezik,
    padx     = 20,
    variable = V,
    command  = Prikazi,
    value    = val). pack (anchor = W)

root.mainloop()
```



Umjesto radio gumba s kružnim rupama koje sadrže prazan prostor, možemo imati radio gumbe s cjelovitim tekstom u okviru. To možemo učiniti postavljanjem opcije indikatora (treba stajati "indikator on") na `0`, što znači da neće biti zasebnog indikatora radio gumba. Zadana vrijednost je `1`. Zamjenjujemo definiciju `Radiobutton` gumba u prethodnom primjeru sa:

```
Radiobutton (root,
  text        = jezik,
  indicatoron = 0,
  width       = 20,
  padx        = 20,
  variable    = V,
  command     = Prikazi,
  value       = val). pack(anchor=W)
```

Sada je, izborom „Python“, prikazano:

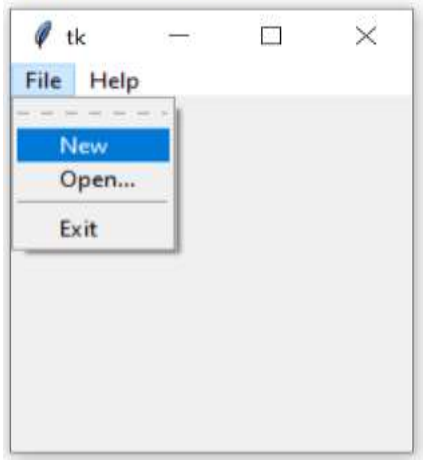


Menu()

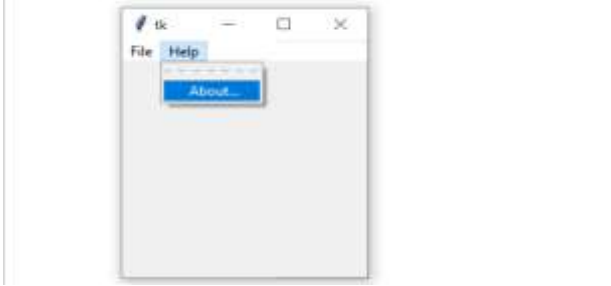
Evo primjera padajućih izbornika programa Tkinter, tj. liste na vrhu prozora koji se pojavljuju (ili povlače prema dolje) ako se klikne stavku poput, na primjer "File" ili "Help".

```
# Menu.py
from tkinter import *
from tkinter.filedialog import (
     askopenfilename )
def NewFile()   : print ("New File!")
def OpenFile() :
  name = askopenfilename(); print(name)
def About()    :
  print ("Ovo je primjer jednostavnog"
         " menu-a")
root  = Tk();  menu = Menu(root)
root  . config(menu = menu)
Fmenu = Menu (menu)
menu  . add_cascade (label = "File",
                     menu = Fmenu)
Fmenu . add_command (label   = "New",
                     command = NewFile)
Fmenu . add_command (label = "Open...",
                     command = OpenFile)
Fmenu . add_separator()
Fmenu . add_command (label = "Exit",
                     command = root.quit)
Hmenu = Menu(menu)
menu  . add_cascade (label = "Help",
                     menu  = Hmenu)
Hmenu . add_command (
                     label = "About...",
                    command  = About)
mainloop()
```

New File!



Ovo je primjer jednostavnog menu-a



Text()

Tekstualnu komponentu izrađujemo metodom `Text()`. Visinu smo postavili na 2, tj. dva retka, a širinu na 30, tj. 30 znakova. Možemo primijeniti metodu `insert()` na objekt `T`, koji je metoda `Text()` vratila, da bismo uključili tekst. Dodamo dva retka teksta.

```
# Text1.py
from tkinter import *
root = Tk()
T = Text (root, height = 2,
                width  = 30)
T . pack(); T . insert (END,
           "Ovo je Text() komponenta\n"
           "u dva reda\n" )
mainloop()
```

Dijelovi teksta mogu biti u posebnom fontu i boji. Da bi se to postiglo, prvo se tekst mora „tegirati“ metodom `tag_add`:

tag_add (*ime_tega*, *od*, *do*)

gje je *ime_tega* string, a *od* i *do* stringovi koji sadrže početni red i poziciju znaka i krajnji red i poziciju znaka. Redovi se broje od `1` nadalje, a pozicije od `0`. Kraj pozicije mora biti veći za `1` (kao i kod svih isječaka u Pythonu). Odvojeni su točkom.

Metodom `config_tag` definiraju se opcije dijela teksta koji je selektiran imenom tega:

`tag_config ( ime_tega, opcija, ...)`

Evo programa u kojem je to pokazano. Jedini je problem što pozicije unutar teksta moramo brojati „na prste“. Moramo li?

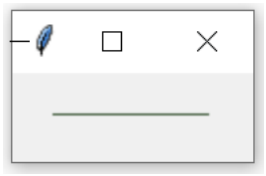
```
# Text_pos.py
from tkinter import *

f = ('Consolas', 15, 'normal')
F = ('Consolas', 20, 'bold')

root = Tk()
text = Text(root, width=30, height=2)
text.insert(INSERT, "Python...")
text.insert(END, "Bye Bye...")
text.pack()

text.tag_add ("python", "1.0",  "1.6")
text.tag_add ("bye1",   "1.9",  "1.12")
text.tag_add ("bye2",   "1.13", "1.16")
text.tag_config ("python", font = F,
                 background = "yellow",
                 foreground = "blue")
text.tag_config ("bye1", font = f,
                 background = "white",
                 foreground = "red")
text.tag_config ("bye2",
                 foreground = "blue")
root.mainloop()
```

Canvas()

`Canvas()` pruža grafičke sadržaje: linije, krugove, grafikone i crteže, slike, editore pa čak i druge komponente te implementira razne vrste prilagođenih komponenti.

U našem prvom primjeru pokazujemo kako povući crtu. Metoda `create_line (koordinate, opcije)` koristi se za crtanje ravne crte. Koordinate "koordinate" dane su kao četiri cjelobrojna broja: `x1`, `y1`, `x2`, `y2`. To znači da linija ide od točke `(x1, y1)` do točke `(x2, y2)`.

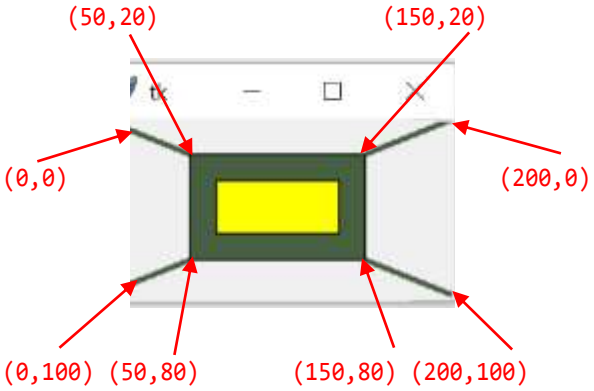
```
# Canvas1.py
from tkinter import *
korijen = Tk()
C_w, C_h = 80, 40
w = Canvas (
  korijen, width = C_w, height = C_h)
w.pack()
y = int(C_h / 2)
w.create_line (0, y, C_w, y,
               fill = "#476042")
mainloop ()
```



Nakon ovih koordinata slijedi popis dodatnih parametara odvojenih zarezom, koji može biti prazan. Na primjer, boju crte postavili smo u posebnu zelenu boju našeg web mjesta: `fill="#476042"`. Prvi smo primjer zadržali namjerno vrlo jednostavnim. Izrađujemo platno i u njega crtamo ravnu vodoravnu crtu. Ova linija okomito presijeca platno na dva područja. Postavljanje na cjelobrojnu vrijednost u dijelu `y=int`

(visina_platna/2) suvišno je, jer `create_line` može raditi i s realnim vrijednostima koje se automatski pretvaraju u cjelobrojne vrijednosti. U nastavku se može vidjeti kôd naše prve jednostavne skripte:

```
# Canvas2.py
from tkinter import *
glavni = Tk()
C_w = 200; C_h = 100
w = Canvas (glavni,
            width = C_w, height = C_h)
w.pack();  B = "#476042"

def Crtaj_P (w, x1,y1, x2,y2, b = B) :
  w.create_rectangle (x1, y1, x2, y2,
                      fill = b)
def Crtaj_L (w, x1,y1, x2,y2,
    b = B, d = 3) :
    w.create_line (x1, y1, x2, y2,
                   fill = b, width = d)
Crtaj_P (w, 50,20, 150,80)
Crtaj_P (w, 65,35, 135,65, "yellow")
Crtaj_L (w, 0,0,    50,20)
Crtaj_L (w, 0,C_h,  50,80)
Crtaj_L (w, 150,20, C_w,0)
Crtaj_L (w, 150,80, C_w,C_h)

mainloop()
```



Pravokutnik

Za crtanje pravokutnika imamo metodu

`create_rectangle` (*koordinate, opcije*)

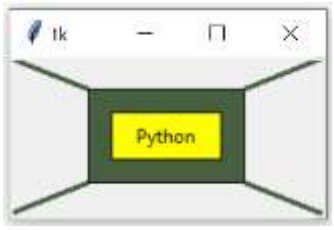
Koordinate se opet definiraju s dvije točke, ali ovaj put prva je gornja lijeva točka i donja desna točka pravokutnika. Prozor, koji vidite gore, stvoren je sljedećim Python tkinter kodom. Na slici smo označili koordinate za bolje razumijevanje primjene

`create_lines` i `create_rectangle`

u našem prethodnom primjeru.

Metoda `create_text()` može se primijeniti za pisanje teksta na platnu. Prva dva parametra su *x* i *y* položaji tekstualnog objekta. Prema zadanim postavkama, tekst je centriran na ovom položaju. To možete nadjačati sidrenom opcijom. Na primjer, ako bi koordinata trebala biti gornji lijevi kut, postavite sidro na SZ. Pomoću teksta parametra ključne riječi možemo definirati stvarni tekst koji će se prikazivati na platnu (dodajemo na kraj prethodnog primjera):

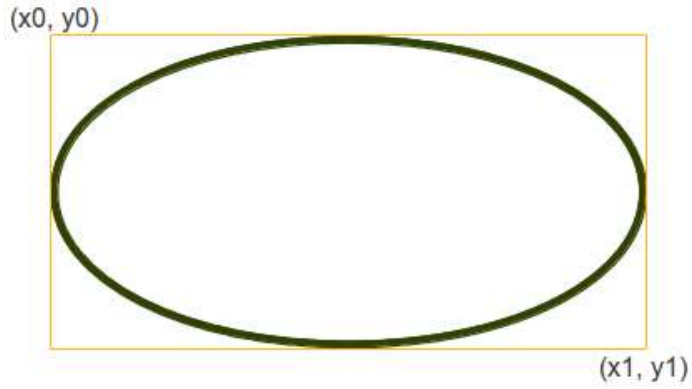
```
w.create_text (C_w / 2, C_h / 2,
               text = "Python")
```



Ovalni objekti

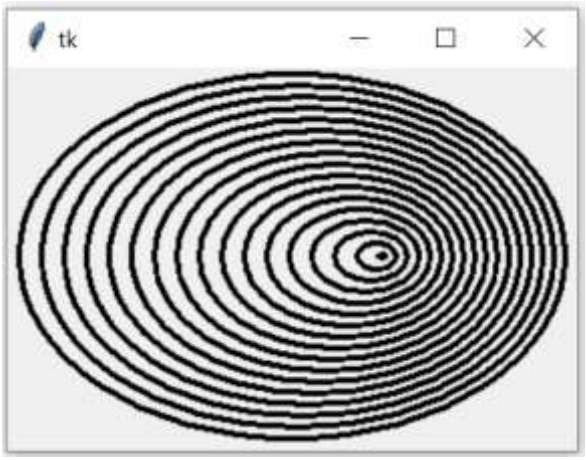
Riječ ovalni potječe od latinskog “ovum” što znači "jaje". To je lik koji nalikuje obliku jajeta. Oval je građen od dva para lukova, s dva različita polumjera. Krug je poseban slučaj ovala. Nalikuje elipsi, ali nije elipsa. Pojam "ovalni" nije dobro definiran. Mnogo različitih krivulja naziva se ovalima, ali svima im je zajedničko:

- Razlikovne su, jednostavne (ne presijecaju se same), konveksne, zatvorene, ravne krivulje
- Po obliku su vrlo slične elipsama
- Postoji barem jedna os simetrije




```
# Canvas3.py
from tkinter import *
from random  import randint
C_w, C_h = 300, 200
glavni = Tk()
w = Canvas (glavni,
    width = C_w, height = C_h)
w.pack()
def circle (canvas,x,y, r):
```

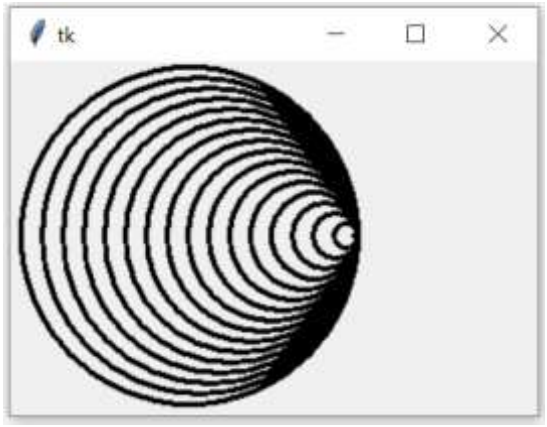
```
    canvas.create_oval (x-r*2,
      y-r,x+r,y+r, width = 3)
for r in range (1, 101, 6) :
  circle (w, 200, 100, r)

mainloop()
```




```
# Canvas4.py
from tkinter import *
from random  import randint

C_w, C_h = 200, 200
glavni = Tk()
w = Canvas (glavni,
    width = C_w, height = C_h)
w.pack()

def circle (canvas,x,y, r):
   canvas.create_oval (x-r,y-r,x+r,y+r,
      width = 3)

for r in range (1, 101, 6) :
  circle (w, 100, 100, r)

mainloop()
```

Može se definirati mala funkcija crtanja krugova koristeći metodu `create_oval()`:

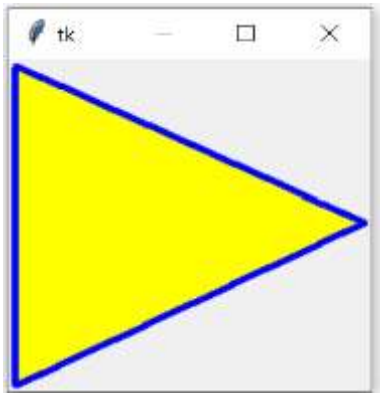
```
def circle (canvas,x,y, r):
   canvas.create_oval (x-r,
      y-r,x+r,y+r, width = 3)
```



Poligon

Ako se želi nacrtati poligon, mora se navesti najmanje tri koordinatne točke: `create_polygon (x0,y0,x1, y1,x2,y2,...)`. U sljedećem primjeru crtamo trokut ovom metodom:

```
# Canvas5.py
from tkinter import *
C_w, C_h = 200, 200
Obod     = "blue"; d = 4
glavni   = Tk()
w = Canvas (glavni, width = C_w,
            height = C_h)
w.pack()
Točke = [d,d, C_w,C_h/2, d, C_h]
w.create_polygon (Točke,
  outline = Obod,
  fill    = 'yellow', width = d)

mainloop()
```



Evo još jednoga primjera:

```
# Canvas6.py
from tkinter import *
C_w, C_h = 200, 200
Obod     = "black";  d = 4
glavni   = Tk()
w = Canvas (glavni,
    width  = C_w, height = C_h)
w.pack()

Točke = [100,140, 110,110, 140,100,
         110, 90, 100, 60, 90,  90,
         60, 100, 90, 110]

w.create_polygon (Točke,
  outline = Obod,
  fill    = 'red', width=3)

mainloop()
```

Točka

Nažalost, ne postoji način da se na platnu naslika samo jedna točka. Ali ovaj problem možemo prevladati pomoću malog ovalnog oblika:

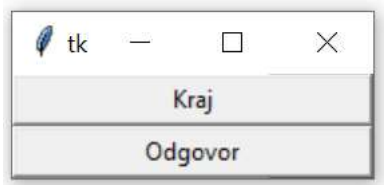
```
# Točka.py
from tkinter import *
glavni = Tk()
w = Canvas (glavni,
  width  = 400, height = 200)
w.pack()
def Točka (w, x, y, d, b) : # točka
  w.create_polygon (x,y, x,y,
    x,y, x,y, outline = b,
    fill = b, width   = d)
for x in range (10, 40, 10) :
  Točka (w, x*10, 100, x, 'blue')
mainloop ()
```

Dijalozi

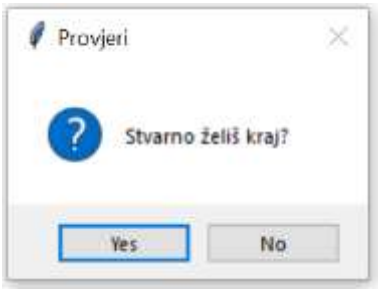
Tkinter pruža skup dijaloga koji se mogu koristiti za prikaz okvira za poruke, upozorenja ili pogreške ili komponenti za odabir datoteka i boja. Postoje i jednostavni dijalozi u kojima se traži da korisnik unese niz, cijele ili realne brojeve. Pogledajmo tipičnu GUI sesiju s dijalozima i okvirima za poruke:

```
# Dijalozi.py
from tkinter import *
from tkinter import messagebox as mb

def Odgovor () :
    mb.showerror("Odgovor",
       "Oprosti, nema odgovora!")

def Uzvrati ():
    if mb.askyesno ('Provjeri',
          ' Stvarno želiš kraj?'):
       mb.showwarning ('Da',
          'Nije još implementiran')
    else :
       mb.showinfo('Ne',
          'Kraj je odgođen')

Button(text = 'Kraj',
     command = Uzvrati).pack(fill = X)
Button(text='Odgovor',
     command = Odgovor).pack(fill = X)

mainloop()
```

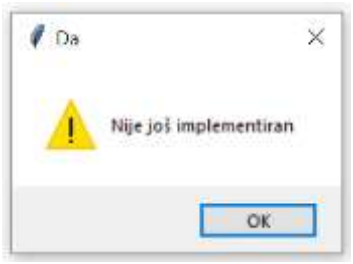
Gumb za pokretanje dijaloga je "Kraj" u sljedećem prozoru:



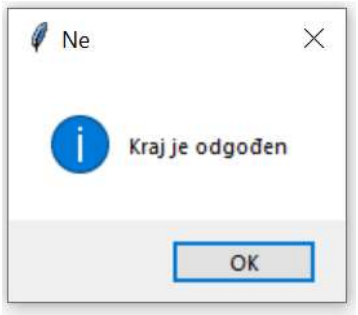
Ako je pritisnuto „Kraj", otvoren je prozor:



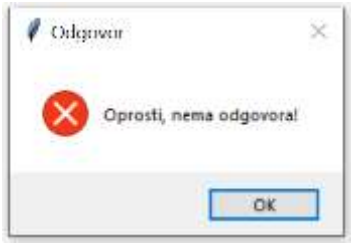
(Primijetiti da je „Da“ i „Ne“ ignorirano). Ako smo pritisnuli „Yes“, bilo bi ispisano:



Na „No“ bi bilo prikazano:



Da smo u prvom dijalogu pritisnuli „Odgovor“, bila bi prikazana poruka:



Na kraju dajemo program koji omogućuje „crtanje“ uz pomoć miša:

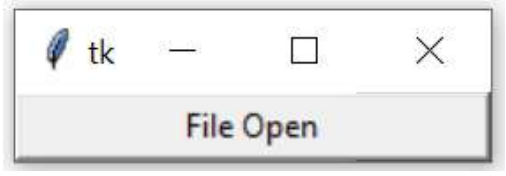
```
# Crtanje.py
from tkinter import *
C_w, C_h = 500, 300

def Crtaj ( event ):
  boja = "blue"
  x1, y1 = (event.x -1), (event.y -1)
  x2, y2 = (event.x +1), (event.y +1)
  w.create_oval( x1, y1, x2, y2,
    outline = boja, width = 10 )
root = Tk()
root.title ("Crtanje koristeći Ovals")
w = Canvas (root,
      width = C_w, height = C_h)
w.pack (expand = YES, fill = BOTH)
w.bind ( "<B1-Motion>", paint )
Poruka = Label ( root,
  text = "Pritisni i pomakni miša" )
Poruka.pack ( side = BOTTOM )
mainloop()
```

filedialog ()

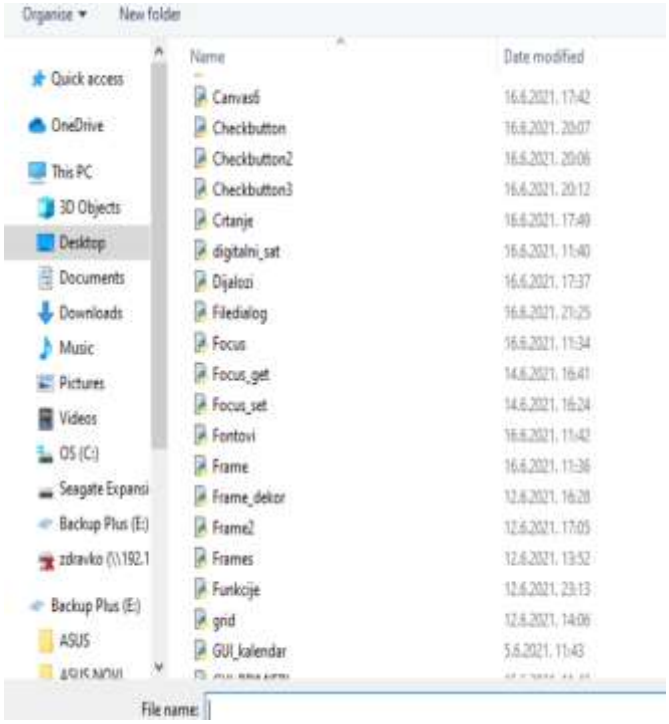
Rijetko da postoji ozbiljna aplikacija koja ne treba način čitanja iz datoteke ili pisanja u datoteku. Nadalje, takva aplikacija možda će morati odabrati imenik. Tkinter u ove svrhe nudi modul `filedialog`. Primjer:

```
# Filedialog.py
from tkinter import *
from tkinter import filedialog as fd
def callback ():
    Ime = fd.askopenfilename()
    print (Ime)
errmsg = 'Error!'
Button (text = 'File Open',
    command = callback).pack(fill = X)
mainloop()
```

Gornji kôd stvara prozor s jednim gumbom s tekstom "File Open".



Ako se pritisne gumb, pojavit će se sljedeći prozor, u tekućem folderu:



Nastavljamo na uobičajeni način kao u Windowsima.

Checkboxes()

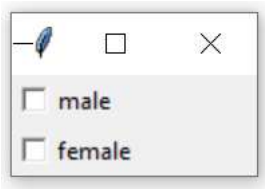
`Checkboxes()`, također poznati kao potvrdni okviri ili okviri za potvrdu, dodaci su koji korisniku omogućuju višestruki odabir iz niza različitih opcija. To se razlikuje od radio gumba, gdje korisnik može napraviti samo jedan izbor.

Obično su potvrdni okviri na zaslonu prikazani kao četvrtasti okviri koji mogu sadržavati razmake (za **`False`**, tj. nije označeno) ili oznaku ili `X` (za **`True`**, tj. označeno).

Natpis koji opisuje značenje potvrdnog okvira obično se prikazuje uz potvrdni okvir. Stanje potvrdnog okvira mijenja se klikom miša na okvir. Alternativno se to može učiniti klikom na naslov ili korištenjem prečaca na tipkovnici, na primjer razmaknice. Potvrdni okvir ima dva stanja: uključeno ili isključeno.

`Checkbutton` može sadržavati tekst, ali samo jednim fontom ili slikama, a gumb može biti povezan s funkcijom ili metodom. Kad se pritisne tipka, Tkinter poziva povezanu funkciju ili metodu. Tekst gumba može obuhvaćati više redaka. Sljedeći primjer predstavlja dva potvrdna okvira "male" i "female". Svaki potvrdni okvir treba različito ime varijable (`IntVar()`).

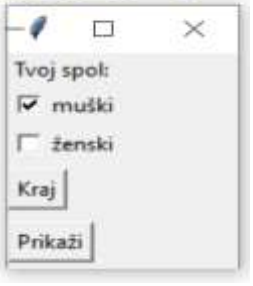
```
# Checkbutton.py
from tkinter import *
root   = Tk()
V1, V2 = IntVar(), IntVar()
Checkbutton(root, text = "male",
     variable = V1).grid(row=0, sticky=W)
Checkbutton(root, text = "female",
     variable = V2).grid(row=1, sticky=W)
mainloop()
```



Ovaj primjer možemo malo poboljšati. Prvo mu dodamo oznaku. Zatim dodajemo dva gumba, jedan za napuštanje aplikacije, a drugi za prikaz vrijednosti `V1` i `V2`.

```
# Checkbutton2.py
from tkinter import *
root = Tk()
def Status ():
   print ("muški: %d, ženski: %d"
          % (V1.get(), V2.get()))
Label (root,
  text = "Tvoj spol:").grid(row = 0,
                            sticky   = W)
V1 = IntVar()
Checkbutton(root, text = "muški",
   variable=V1).grid(row = 1, sticky = W)
V2 = IntVar()
Checkbutton(root, text = "ženski",
   variable=V2).grid(row = 2, sticky = W)
Button(root, text = 'Kraj',
  command = root.quit).grid(row = 3,
                            sticky = W,
                            pady   = 4)
Button(root, text = 'Prikaži',
  command = Status).grid(row=4,
                         sticky=W, pady=4)
mainloop()
```

```
muški: 1, ženski: 0
```

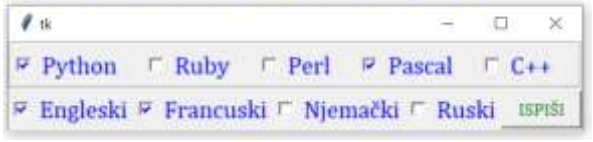


Napišimo aplikaciju koja prikazuje popis programirnih jezika, npr. ['Python','Ruby','Perl', 'Pascal','C++'] i popis prirodnih jezika, npr. ['Engleski', 'Francuski','Njemački'] kao potvrdne okvire. Tako je moguće odabrati programirne jezike i prirodne jezike. Nadalje, imamo gumb " PRIKAŽI " za provjeru stanja varijabli potvrdnog okvira.

```
# Checkbutton3.py
from tkinter import *

class Checkbar (Frame) :
  def __init__(_, parent = None,
               picks = [], side = LEFT,
               anchor = W) :
    Frame.__init__ (_, parent)
    _.vars = []
    for pick in picks :
      Var = IntVar()
      chk = Checkbutton(_,
            font = 'Cambria 16',
            fg = 'blue', text = pick,
            variable = Var)
      chk.pack(side = side,
          anchor = anchor,
          expand = YES)
      _.vars.append(Var)

  def state ( _ ):
    return map((lambda var: var
                .get()), _.vars)

if __name__ == '__main__' :
    root = Tk()
    lng = Checkbar(root, ['Python',
          'Ruby','Perl','Pascal','C++'])
    tgl = Checkbar(root, ['Engleski',
          'Francuski', 'Njemački',
          'Ruski'])
    lng.pack(side=TOP,  fill=X)
    tgl.pack(side=LEFT)
    lng.config(relief=GROOVE, bd=2)

    def allstates () :
       print( list(lng.state()),
              list(tgl.state()) )

    Button(root, text='  ISPIŠI  ',
      font = 'Cambria 12', fg = 'green',
      command= allstates).pack(side=RIGHT)
    root.mainloop()
```

Izborom jezika za programiranje, prirodnih jezika i pritiskom na gumb ISPIŠI bilo bi prikazano:

```
[1, 0, 0, 1, 0] [1, 1, 0, 0]
```



POGREŠKE BEZ DOJAVE

Pri pisanju GUI programa može se dogoditi pogreška koju naš dobri Python ne dojavljuje! Jednostavno se ne dobije očekivani rezultat, a ne zna se razlog. Tada slijedi testiranje i traženje mjesta koje je „zaštekalo“.

P R O G R A M I

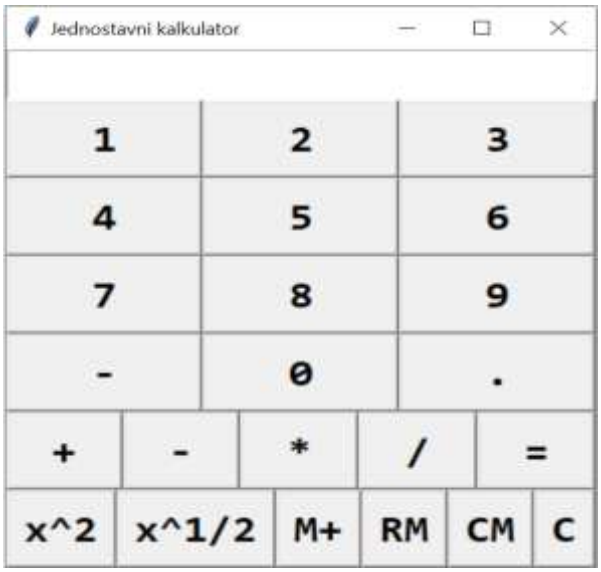
KALKULATOR

Sljedeći program je jednostavni kalkulator koji podržava četiri binarne operacije realnih izraza, kvadriranje, drugi korijen, pamćenje i pozivanje međurezultata izračunavanja.

Kalkulator.py

```
from tkinter import *
def frame (root, side):
  w = Frame (root)
  w.pack    (side   = side,
             expand = YES,
             fill   = BOTH)
  return w

def button (root, side, text,
            command = None) :
  w = Button (root, text = text,
                 command = command)
  w.pack     (side   = side,
              expand = YES,
              fill   = BOTH)
  return w

class Kalkulator (Frame):
  def __init__ (_):
    Frame.__init__ (_)
    _.option_add ('*Font',
                  'Consolas 20 bold')
    _.pack (expand = YES, fill   = BOTH)
```

```
_.master.title    ('Kalkulator')
_.master.iconname ("calc1")
_.M = 0;   _.B = False
display = StringVar()
Entry (_, relief = SUNKEN,
  textvariable = display).pack (
     side = TOP, expand = YES,
     fill = BOTH)

def x2  ()   : display.set(
        str (eval (
        display.get() +'**2')))
def Sqrtx () : display.set(
        str (eval (
        display.get() +'**0.5')))
def Mplus () : _.M += eval (
                  display.get())
def RMem  () : display.set (
                display.get() +
                str (_.M))
def CMem  () : _.M = 0

for key in ("123", "456", "789",
            "-0.") :
  keyF = frame (_, TOP)
  for char in key:
    button (keyF, LEFT, char,
    lambda w = display, c = char:
           w.set (w.get() + c))
opsF = frame(_, TOP)
for char in "+-*/=":
  if char == '=':
    btn = button (opsF, LEFT,
                  char)
    btn.bind (
      '<ButtonRelease-1>',
      lambda e, s = _,
      w = display :
          s.calc (w, char), '+')
  else:
    btn = button (opsF, LEFT,
      char, lambda w = display,
      s = ' %s ' % char :
      w.set (w.get()+s))
X2    = frame (_, LEFT)
button (X2, LEFT,'x^2',command = x2)
sqrtX = frame (_, LEFT)
button (sqrtX, LEFT, 'x^1/2',
        command = Sqrtx)
M     = frame (_, LEFT)
button (M, LEFT, 'M+',
           command = Mplus)
RM    = frame (_, LEFT)
button (RM, LEFT, 'RM',
        command = RMem)
    CM    = frame (_, LEFT)
    button (CM, LEFT, 'CM',
            command = CMem)
    if _.B :
       display.set('');  _.B = False
    clearF = frame(_, LEFT)
    button (clearF, LEFT, 'C',
      lambda w=display: w.set(''))
  def calc (_, display, c):
    try :
      display.set (str (eval (
              display.get()) ) )
      if c == '=' : _.B = True
    except :
      display.set ("ERROR")
Kalkulator().mainloop()
```



CRTANJE FUNKCIJA

Evo programa koji crta funkcije

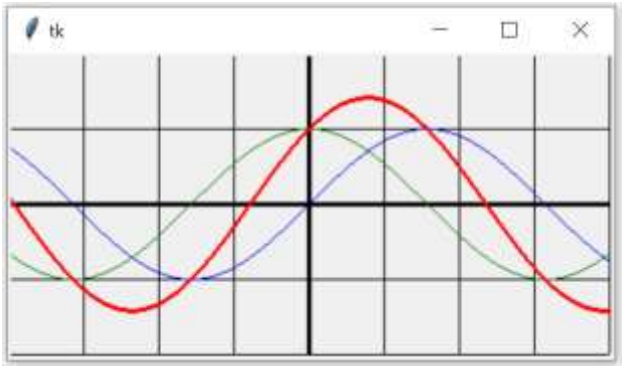
```
y = sin(X), z = cos(X)
```

i njihov zbroj na intervalu `[-4, 4]`. Širina platna je `800`, a visina `200` pikesla. Crteži se dobivaju nizom točaka dobivenih pozivanjem procedure T.

Funkcije.py

```
from tkinter import *
from math    import *
glavni = Tk()
d = 50;  C_w, C_h = 8*d, 4*d
w = Canvas (glavni, width  = C_w,
                    height = C_h)
w.pack()

# mreža koodinata
for i in range (0, 5) :
```

```
    # horizontalno
    P = 1; y = i*d
    if i == 2 : P = 3
    w.create_line (0, y, C_w, y,
                   fill = "black", width = P)
  for i in range (0, 9) : # vertikalno
    P = 1; x = i*d
    if i == 4 : P = 3
    w.create_line (x, 0, x, C_h,
                   fill = "black", width = P)
  def T (w, x, y, d, b) : # točka
    w.create_polygon (x,y, x,y, x,y, x,y,
      outline = b, fill = b, width = d)

  for x in range (-C_w//2, C_w//2+1) :
    X = x/d; y = sin(X) *d; z = cos(X) *d
    T (w, C_w/2 +x, C_h/2 -y, 2, 'blue')
    T (w, C_w/2 +x, C_h/2 -z, 2, 'green')
    T (w, C_w/2 +x, C_h/2 -(y+z), 3,
                                  'red')
  mainloop ()
```



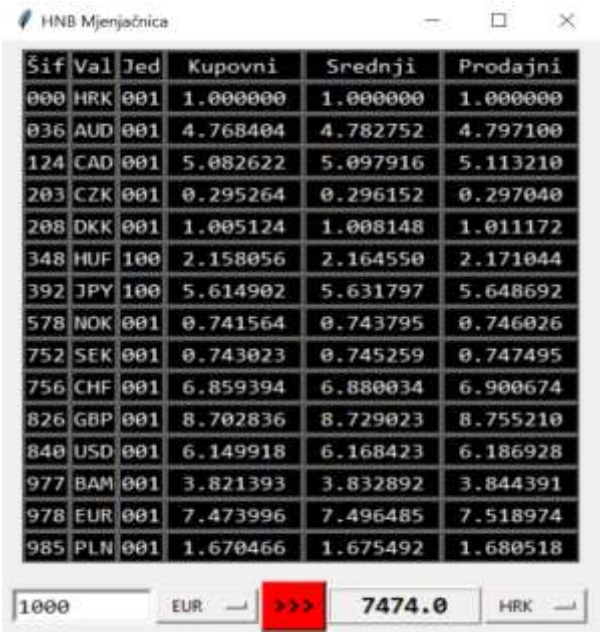
MJENJAČNICA (4)

Još jedna inačica mjenjačnice, sada u grafičkom okruženju.

Mjenjačnica_GUI.py

```
from urllib.request import urlretrieve
from tkinter import *
def konvertiraj ():
  try :
    x = float (X.get());  u = Ul.get()
    if u in options :
       u = options.index(u)
    i = Iz.get()
    if i in options :
      i = options.index(i)
    y = round((x / float(s[u][2])) *
      (float(s[u][3])/
       float(s[i][5])), 2)
    Y.config (text = y)
  except : pass

# dohvaćanje podataka
URL = ('https://www.hnb.hr/'
       'tecajn/htecajn.htm')
urlretrieve (URL, "TL.txt")

# obrada podataka
D = open('TL.txt'); D = D.read()
D = D.replace('\r', '')
D = D.replace(',', '.')
D = D.split('\n')
D[0] = ('000HRK001       1.000000'
     '       1.000000       1.000000')
s = [['Šif', 'Val', 'Jed', 'Kupovni',
      'Srednji', 'Prodajni']]
for line in D:
  line = (line[:3] +' ' +line[3:6]
            +' ' +line[6:])
  line . split()
  s.append(line.split())
options = [valuta[1] for valuta in s]
options[0] = ''
root = Tk()
root . title ('HNB Mjenjačnica')
Frame1 = Frame (bd = 7)
Frame1 . pack()
Frame2 = Frame (bd = 7)
Frame2 . pack()
f = ('Consolas', 12, 'normal')
F = ('Consolas', 14, 'bold')

def Tečajna_lista (s, i, od, do, w) :
  for j in range(od, do):
    b = Label(Frame1, text=s[i][j],
          relief=GROOVE,
          font = f, width = w,
          fg = 'white',
          bg = 'black')
    b.grid(row = i, column = j)

for i in range( len(s) ):
  Tečajna_lista (s, i, 0, 3, 3)
  Tečajna_lista (s, i, 3, 6, 10)
X = Entry(Frame2,
      bd = 3, width = 10, font = f)
X . pack(side = LEFT)
Ul = StringVar(Frame2)
Ul.set('EUR') # početna vrijednost
Ul_valuta = OptionMenu (Frame2, Ul,
                        *options)
Ul_valuta.config (width = 5)
Ul_valuta.pack   (side = LEFT)
```

```
Button(Frame2, text = '>>>',
  font = F, bg = 'red',
  command = konvertiraj).pack (
                          side = LEFT)
Y  = Label(Frame2, bd = 3, width = 10,
       font = F, relief = RIDGE)
Y  . pack(side = LEFT)
Iz = StringVar(Frame2)
Iz . set('HRK') # početna vrijednost
Iz_valuta = OptionMenu (Frame2,
              Iz, *options)
Iz_valuta.config (width = 5)
Iz_valuta.pack   (side = LEFT)

X.focus_set()
mainloop()
```



LEKSIKON

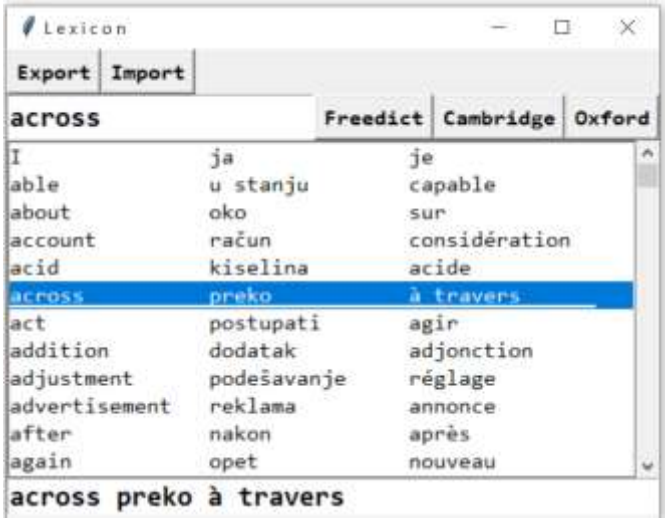
Dajemo program koji pokazuje kako možete napraviti mini leksikon za 850 engleskih riječi s prijevodom na hrvatski i francuski, s vezom na kembridžski i oksfordski leksikon. Riječi su upisane u tekstualnu datoteku i odvojene su tabulatorom.

Leksikon_850.py

```
# 850 enleskih riječi i prijevod na
# hrvatski i francuski

from tkinter import *
from tkinter.filedialog import Open
from pickle  import *

global Lx

def IMPORT () :
  global Lx
  Add = open ('Lex.TxT', 'r',
              encoding = 'utf8')
  ADD = []
  for line in Add :
    ADD.append (line[:-1].split('\t'))
  Lx = {}
  for W in ADD :
    if W != [''] :
       Lx[W[0]] = (W[1], W[2])
  Add.close ()
  f = open ('Lex.dat', 'wb')
  dump (Lx, f)
  f  . close()

def Open_Lex () :
  global Lx
  try :
    f  = open ('Lex.dat', 'rb')
    Lx = load (f); f. close()
  except :
    IMPORT ()

Open_Lex() # Lx <- radni leksikon

def EXPORT () :
  global Lx
  T = '\t'
  L = list (Lx.keys());  L.sort()
  Lex = open ('Lexicon.TxT', 'w',
              encoding = 'utf8')
  for w in Lx :
    W = Lx[w]; print (
        w, T, W[0], T, W[1], file = Lex)
  Lex.close()

class LEXICON (Frame):
  global v, Wrd, Lx
  def __init__ (_, master):
    global Wrd
    Frame.__init__ (_, master)
    _.grid (row = 0,  column = 10,
            columnspan=3)
    _. Ch_Lx  = False
    _. Fn     = 'Consolas', 12, 'normal'
    _. Fb     = 'Consolas', 12, 'bold'
    _. Fbh    = 'Consolas', 15, 'bold'
    _. parent = master
    _. create_widgets ()
    _. Lexicon ()

  def create_widgets (_):
    global aa, v, Wrd

    def CopyToInput0 (event) :
      global ind, v
      ind = _.LexBox.curselection()
```

```
    # index u listbox (0, ), ... (n, )
    _.input.delete (0, END)
    tx = _.LexBox.get (ANCHOR)
    _.wordLex.delete (0, END)
    tx = tx.split()
    _.wordLex.insert (0, tx[0])
    # _.v.set (tx[0])
    tx = ' '.join (tx)
    _.input.insert (0, tx)

def Search ( event ) :
    # nađi riječ (prefiks)
    global L
    c = event.char.lower()
    try :
      if ord(c) >= 32 or ord(c) == 8 :
         x = _.wordLex.get().lower()
         X = x +c if ord(c) > 32\
             else x[:-1]
         k = len (X)
         _.LexBox.delete(0, END)

         for x in L:
           if X in x[:k].lower():
              _.LexBox.insert (END, x)
    except : pass

  def key0 (event) :
    c = event.char
    if c == '\r' : CopyToInput0 (event)

  oFrame    = Frame (_)
  _.Export  = Button (oFrame,
    text    = 'Export',
    font    = _.Fb,
    command = EXPORT).grid (
    column  = 0, row = 0, sticky = W)

  _.Import  = Button (oFrame,
    text    = 'Import', font = _.Fb,
    command = _.Import). grid (
    column  = 1, row = 0, sticky = W)

  oFrame.pack  (side = TOP, fill=X)

  # IZABRANA RIJEČ
  wFrame = Frame (_)
  _.W       = Button (wFrame,
    text    = "Oxford", font = _.Fb,
    command = _.Oxford).pack (
    side    = RIGHT)
  _.W       = Button (wFrame,
    text    = "Cambridge", font = _.Fb,
    command = _.Cambridge).pack (
    side    = RIGHT)
  _.W       = Button (wFrame,
    text    = "Freedict", font = _.Fb,
    command = _.FreeDict).pack (
    side    = RIGHT)
  # print (Wrd)
  _.v            = StringVar()
  try    : _.v.set (Wrd)
  except : pass
  _.wordLex      = Entry (wFrame,
    textvariable = _.v, width=13,
    font = _.Fbh)
  _.wordLex.pack   (side = RIGHT,
    fill = BOTH, expand = 1)
  wFrame.pack      (fill = X)
  _. wordLex.focus_set ()
  _. wordLex.bind  ("<Key>",
     Search)

  # LexBox  lista leksikona
  iFrame      = Frame (_)
  iScrollbar  = Scrollbar (iFrame,
                orient = VERTICAL)
  _.I         = StringVar()
  _.I.set       ('')
  _.LexBox    = Listbox (iFrame,
    height    = 12, width = 50,
    font           = _.Fn,
    listvariable   = _.I,
    yscrollcommand = iScrollbar.set,
    selectmode     = EXTENDED)
  iScrollbar.configure (
    command = _.LexBox.yview)
  _.LexBox.pack (side = LEFT,
                 fill = BOTH,
               expand = 1)
  _.LexBox.bind ("<Double-Button-1>",
                 CopyToInput0)
  _.LexBox.bind ("<Key>", key0)
  iScrollbar.pack(side = RIGHT, fill=Y)
  iFrame.pack (fill = BOTH, expand = 1)

# IZABRANA REČENICA
  Input   = Frame (_)
  _.input = Entry (Input, font = _.Fbh)
  _.input.pack (side   = RIGHT,
                fill   = BOTH,
                expand = 1)
  Input.pack   (fill = BOTH)
  try :
    if Wrd in Lx :
      tx = (Wrd +' ' +Lx[Wrd][0]
            +'\t' +Lx[Wrd][1]
            +'\t' +Lx[Wrd][2])
```

```
        _.wordLex.insert (0, tx[0])
        _.input.insert   (0, tx)
    except : pass

    lex_tr      = Frame (_)
    lex_tr.pack (anchor = W, fill = Y,
                 expand = 0)
    _.aFrame = Frame(_)
    _.aFrame.pack (fill = BOTH,
                   expand = 0) # RF
    _.RF = False

def Lexicon (_):
    global L
    Words = list (Lx.keys())
    Words.sort()

    L = [];  i = 0
    for x in Words :
       L.append ("%-15s %-15s %-15s"
                 % (x, Lx[x][0],
                    Lx[x][1]))
    _.LexBox.delete (0, END)
    for i in L : _.LexBox.insert(END,i)

def Import ( _ ):
  IMPORT (); _. Lexicon ()

def FreeDict ( _ ) :
  global Url
  Url = (
  "http://www.thefreedictionary.com/")
  _.Lex ()

def Cambridge (_)  :
  global Url
  Url = ("http://dictionary.cambridge."
         "org/dictionary/english/")
  _.Lex ()

def Oxford (_)     :
  global Url
  Url = (
  "http://www.oxforddictionaries.com/"
                 "definition/english/")
  _.Lex ()

def Lex    (_) :
  import webbrowser
  global Url
  W    = _.wordLex.get()
  URL  = (
     "http://dictionary.cambridge.org/"
     "search/english/") +W
  webbrowser.open (Url +W)

if __name__ == "__main__" :
    root2 = Tk()
    root2.title ("L e x i c o n")
    app2 = LEXICON (root2)
    root2.mainloop()
```



Oxford

Riječ koju pretražujete u danim rječnicima (leksikonima) ne mora biti dio našeg leksikona.

GUI_KALENDAR (2)

Evo programa koji generira GUI kalendar za mjesec zadane godine.

GUI_Kalendar.py

```
from tkinter  import *
from calendar import *
class Kalendar (Frame):
  def __init__ (_, master):
    Frame.__init__(_, master)
    _.grid()
```

```
    year  = int (input ('Godina? '))
    month = int (input ('Mjesec? '))
    _.create_widgets(year, month)
  def create_widgets(_, year, month):
    days = ['pon', 'uto', 'sri', 'čet',
            'pet', 'sub', 'ned']
    for i in range (7):
      label = Label (_, fg   = 'blue',
                        text = days[i])
      label.grid (row = 0, column = i)

    weekday, numDays = monthrange (
                       year, month )
    week = 1
    for i in range (1, numDays +1):
      dan  = "%2d" % i
      boja = ('black' if weekday < 6
                      else 'red')
      button = Button(_,
             font = 'Consolas 10 bold',
             fg = boja, text = dan)
      button.grid (row    = week,
                   column = weekday)
      weekday = (weekday +1) % 7
      week   += 1 * (weekday == 0)
root=Tk()
obj = Kalendar (root)
root.mainloop()
```

Nazivi svih boja

Na kraju dajemo nazive svih boja koje se mogu koristiti kao stringovi u aplikacijama. Primjeri:

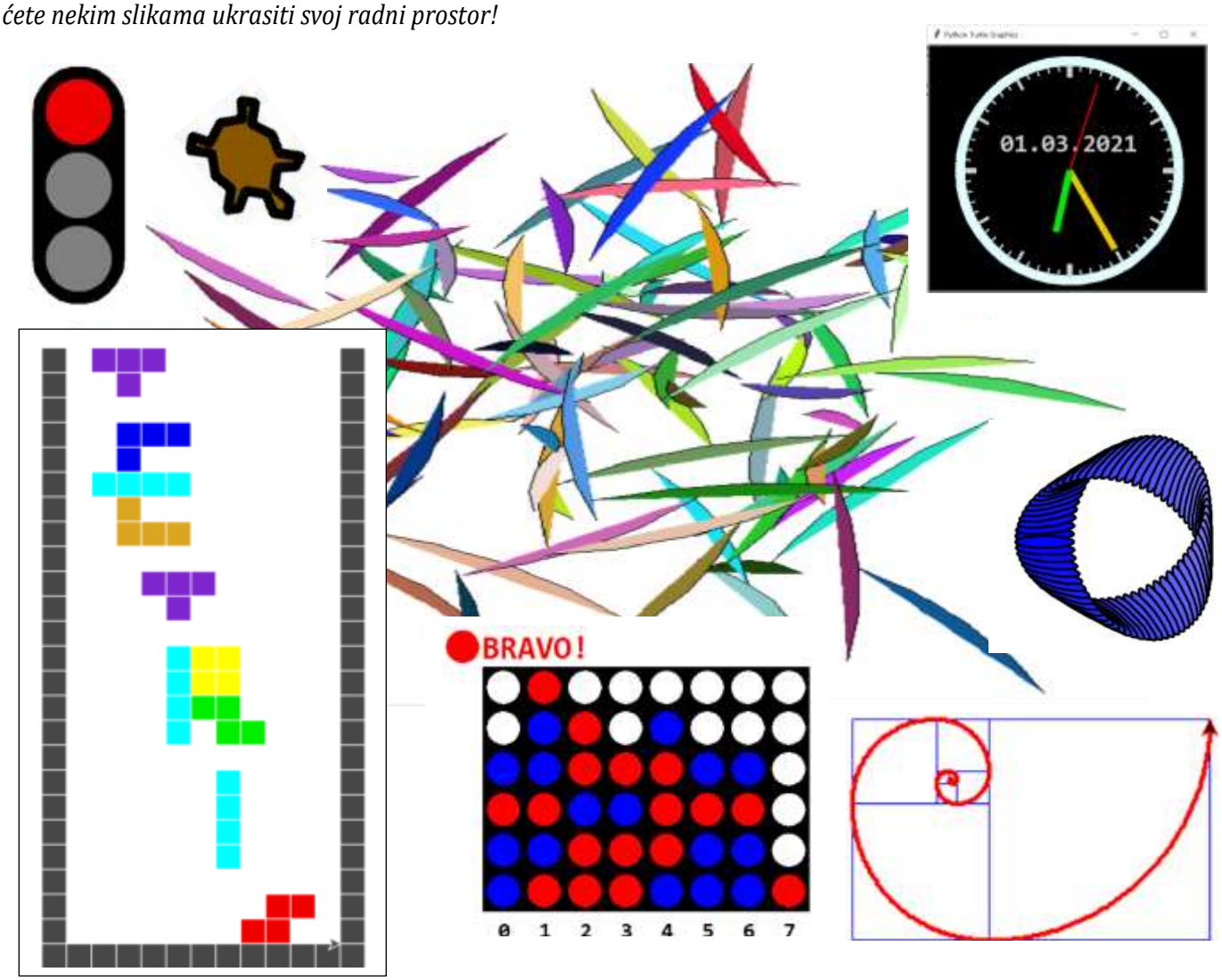
```
'deep sky blue' 'green3' 'green yellow'
```

14.

KORNJAČINA GRAFIKA

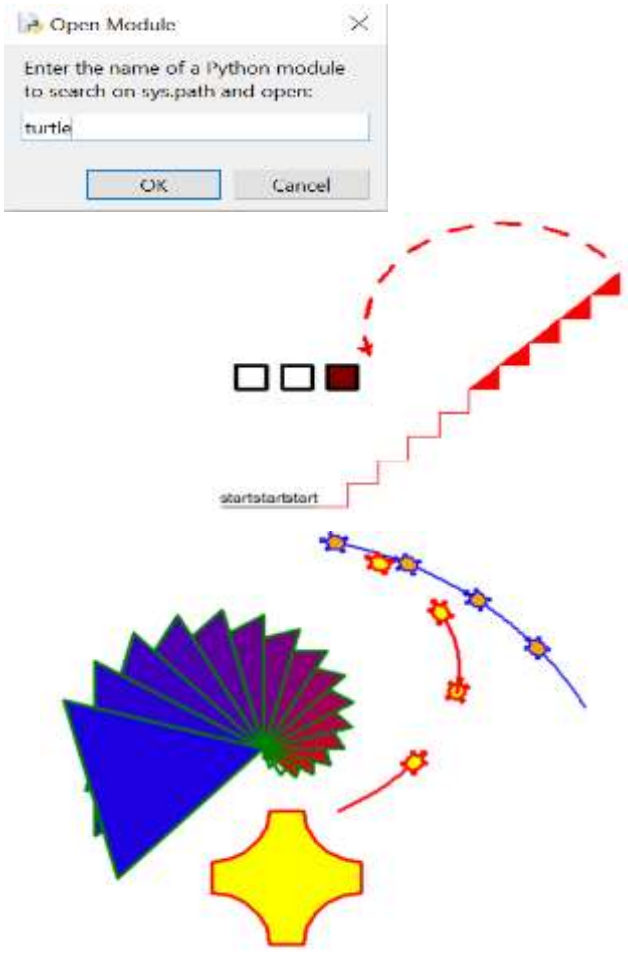
Kornjačina grafika (turtle graphics) izvorno je razvijena 1967. godine kao dio programirnog jezika Logo (autori su Wally Feurzeig, Seymour Papert i Cynthia Solomon), s namjerom upoznavanja djece s programiranjem na popularan način. Osim jezika Logo bila je još i ona kao dio programirnog jezika Turbo Pascal 3.0.

Pythonov modul `turtle` *sadrži jezik kornjačine grafike čije mogućnosti znatno nadilaze navedene prijašnje implementacije. Pruža grafičke primitive kornjača, kako na objektno orijentirani, tako i na proceduralni način, te na strukture i tipove podataka Pythona, na čijim temeljima je postavljena implementacija jezika kornjačine grafike. Kao i dosad, a u ovom poglavlju posebno, pretpostavka je da se sve naredbe i skripte paraleno s njihovim uvođenjem iskušaju i izvrše. Na kraju ovog poglavlja moći ćete generirati slike, satove i igrice kao što je ovdje prikazano. Možda ćete nekim slikama ukrasiti svoj radni prostor!*



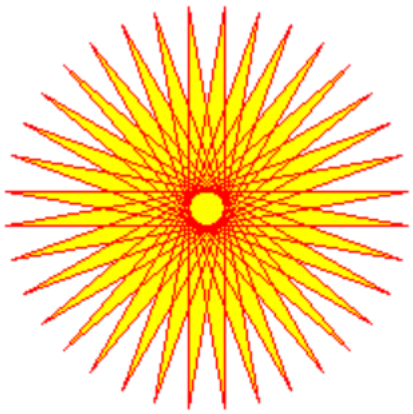
Uvod

Pythonova kornjačina grafika sadržana je u standardnom modulu `turtle`. Sa svojim procedurama i funkcijama daleko nadilazi Logo i Turbo Pascal. Pogledajmo najprije demostraciju nekih mogućnosti otvaranjem i izvršavanjem modula `turtle` (`Alt+M` iz glavnog menu-a editora):



Kornjača može crtati složene oblike pomoću programa koji ponavljaju jednostavne poteze. Evo dva primjera jednostavnih programa koji to potvrđuju.

```
# suncokret.py
from turtle import *
color ('red', 'yellow')
begin_fill()
while 1 :
  forward (200); left (170)
  if abs(pos()) < 1 : break
end_fill(); ht()
```



```
# spiralna_zavojnica.py
from turtle import *
boje = ['red', 'purple', 'blue',
        'green', 'orange', 'yellow']
for x in range (120):
  pencolor (boje[x%6])
  width    (x/100 + 1)
  fd(x); lt(59)
ht()
```

„Voilà!“, rekli bi Francuzi. Vjerujemo da su ovi uvodni primjeri dovoljna motivacija da uđete u svijet kornjačine grafike Pythona. Ono što dajemo u ovom poglavlju je samo uvod u jezik kornjače jer bi potpuni opis mogao biti jedna posebna knjiga.

MODUL `turtle`

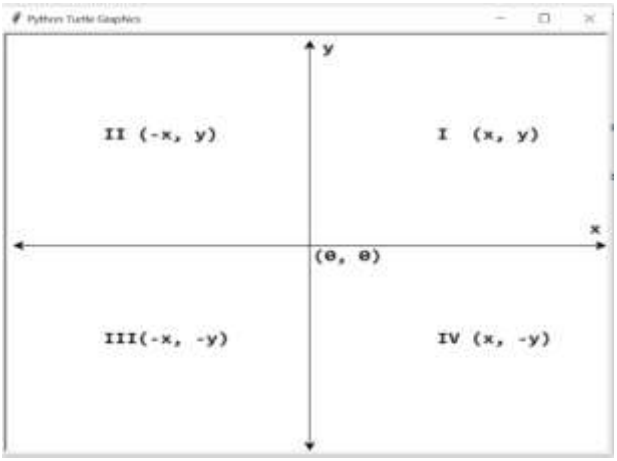
Modul **`turtle`** sadrži veliki broj klasa, funkcija i procedura:

```
>>> import turtle
>>> dir (turtle)
['Canvas', 'Pen', 'RawPen', RawTurtle',
..., 'xcor', 'ycor']
```

S obzirom na to da kornjačina grafika sa svojim metodama (procedurama i funkcijama) predstavlja poseban jezik unutar Pythona, možemo reći da su to njezine naredbe.

ZASLON

Crta "kornjača" koja, krećući se naprijed, nazad ili pod nekim kutem, ostavlja svoje "tragove" na posebnom zaslonu (prozoru) za crtanje. Zaslon se može zamisliti kao koordinatni sustav s četiri kvadranta, s ishodištem u sredini, kao što je prikazano u sljedećem crtežu.




```
>>> from turtle import *
>>> home () # otvoren zaslon
```

Izvršenjem prve komande jezika kornjačine grafike otvoren je zaslon. Podijelimo radni prostor Windowsa na dva dijela, po pola ekrana. Lijevo je prostor za interaktivno izvršavanje naredbi Pythona, a desno zaslon kornjačine grafike.

Poslije izvršenja komande **home()** postavljena je strelica („kornjača") usmjerena udesno u ishodište zamišljenog koordinatnog sustava.

Sljedeće dvije komande vraćaju širinu i visinu kornjačinog zaslona na vašem računalu.

window_width ()

Vraća širinu kornjačinog prozora u pikselima.

```
>>> window_width() # inicijalno      640
```

Poslije klika na maksimalni prozor:

```
>>> window_width()                   1280
```

window_height ()

Vraća visinu kornjačinog prozora u pikselima.

```
>>> window_height () # inicijalno    540
```

Poslije klika na maksimalni prozor:

```
>>> window_height ()                 657
```

mode (*mode* = None)

Postavlja môd kornjače na zadani, ili vraća tekući môd ako je pozvan bez argumenta.

mode : "standard" | "logo" | "world"

Značenje modova dano je u sljedećoj tablici.

	môd	*značenje*
➤	"standard"	Kompatibilan je sa starom kornjačom. Kornjača je orijentirana prema „istoku". Pozitivni kut je suprotan od kretanja kazaljke na satu. Inicijalno se kornjača nalazi u ovom modu.
▲	"logo"	Kompatibilan je većim dijelom s grafikom kornjače jezika Logo. Inicijalno je kornjača orijentirana prema „sjeveru". Pozitivni kut je u smjeru kretanja kazaljke na satu.
➤	"world"	Koristi korisnički definirane "svjetske koordinate". U ovom načinu rada kutovi izgledaju izobličeno ako omjer razmjera x/y nije jednak 1.

Svakom promjenom moda resetira se zaslon.

```
>>> from turtle import *
>>> home(); mode()              'standard'
>>> mode ('logo'); mode()          'logo'
```

Kretanje kornjače

Zamislimo robotsku kornjaču koja se inicijalno, nakon uvoza modula `turtle`, nalazi u ishodištu zaslona i čeka se na izvršavanje komandi.

POMICANJE I CRTANJE

U nastavku dajemo opis svih komandi za kretanje kornjače, pomicanje i crtanje. Neke komande imaju dva ili tri imena. Svejedno je koje se koristi.

position, pos ()

Vraća tekuću poziciju `(x,y)` *kornjače.*

```
>>> home(); pos()              (0.00,0.00)
```

xcor ()

Vraća tekuću poziciju koordinate x *kornjače.*

```
>>> xcor() # inicijalno                0.00
```

ycor ()

Vraća tekuću poziciju koordinate y *kornjače.*

```
>>> ycor() # inicijalno                0.00
```

forward, fd (*p*)

Pomiče pero naprijed za p *piksela, ako je* p>0*, inače unazad* p *piksela ako je* p<0*.*

```
>>> pos ()                          (0.00,0.00)
>>> forward (100); pos ()  (100.00,0.00)
>>> fd (-50); pos ()          (50.00,0.00)
```

right, rt (*k*)

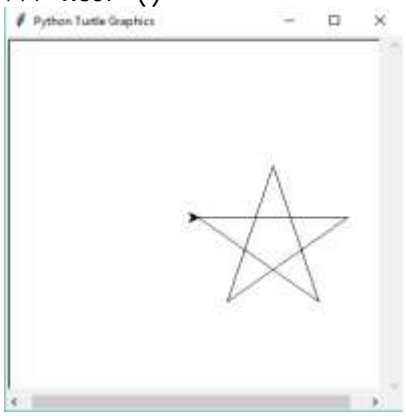
Zaokreće pero udesno (u smjeru okretanja kazaljke na satu) za k *stupnjeva (radijana), ako je* k>0 *ili za* k *stupnjeva suprotno smjeru kazaljke na satu ako je* k<0*.*

```
>>> forward (150); right (144)
```

Pazite, vrh strelice je prema dolje!

„Kornjača“ je, krećući se naprijed, ostavila trag duljine 150 piksela i potom se zaokrenula za 144 stupnja udesno (u smjeru kazaljke na satu). Ako to ponovimo još četiri puta:

```
>>> for i in range(4): fd(150); rt(144)
>>> xcor ()                            0.00
```



na kraju je dobiven crtež petokrake. „Kornjača“ se vratila u ishodište i okrenuta je udesno pod kutem 0 stupnjeva. Da je bilo izvršeno:

```
>>> for _ in range (3):
    right (120); fd(120)
```

bio bi nacrtan jednakostranični trokut stranice jednake 120 piksela.

left, lt (*k*)

Zaokreće pero ulijevo (suprotno smjeru okretanja kazaljke na satu) za k *stupnjeva (radijana) ili za* k *stupnjeva u smjeru kazaljke na satu ako je* k<0*.*

```
>>> for _ in range (8):
    left(45); fd(2) # oktagon
```

Komanda lt(-k) ekvivalentna je rt(k), a rt(-k) komandi lt(k).

backward, back, bk (*p*)

Pomiče pero nazad za p *piksela, ako je* p>0*, inače naprijed* p *piksela ako je* p<0*.*

Komanda bk(-p) ekvivalentna je fd(p), a fd(-p) komandi bk(p).

```
>>> pos ()                          (0.00,0.00)
>>> back (100); pos ()      (-100,0.00)
>>> bk (-100); pos ()         (0.00,0.00)
```

pensize, width (*width* = None)

Postavlja debljinu crte na zadanu širinu width*. Poziv bez argumenta vraća trenutnu debljinu. Inicijalno je jednaka* 1*.*

```
>>> pensize () # inicijalno             1
>>> fd (50); lt (90);
>>> width (10); pensize ()             10
>>> fd (50)
```

heading ()

Vraća tekući kut pod kojim je kornjača usmjerena (vrijednost je ovisna o modu grafike kornjače).

```
>>> home ();   heading ()              0.0
>>> rt   (75); heading ()            285.0
>>> mode ('logo'); heading ()          0.0
>>> rt (75); heading ()               75.0
```

degrees (*fullcircle = 360.0*)

Postavka jedinice mjerenja kuta, tj. postavka broja "stupnjeva" za puni krug. Zadana (inicijalna) vrijednost je 360 stupnjeva.

fullcircle – broj veći od 0

Svakom promjenom vrijednosti „stupnjeva“ punoga kruga mijenja se i vrijednost tekućeg kuta usmjerenja kornjače:

```
>>> mode    ('standard'); lt (90)
>>> heading ()                    90.0
>>> degrees (400); heading ()    100.0
>>> degrees (100); heading ()     25.0
>>> degrees (1);   heading ()     0.25
>>> degrees (2 *3.1415926) # 2*pi
>>> heading () # pi/2          1.5707963
```

radians ()

Postavka jedinice mjerenja kuta u radijane. Postavka je ekvivalentna postavci:

```
degrees ( 2*math.pi )
```

```
>>> lt(90);    heading()           90.0
>>> radians(); heading()  1.57079632679
```

penup, pu, up ()

Diže pero. Ne crta pri premještanju pera (crtanje „isključeno“).

pendown, pd, down ()

Spušta pero. Crta pri premještanju pera (crtanje „uključeno“).

home ()

Postavlja (pomiče) kornjaču u ishodište, (0,0) i na početnu orijentaciju (što ovisi o načinu rada, mode()). Ako tekuća lokacija nije bila (0,0) i ako je pero spušteno, povlači (crta) liniju do ishodišta.

speed (*speed = None*)

Postavlja brzinu crtanja kornjače na cijelu vrijednost u rasponu 0..10. Ako nije naveden nijedan argument, vraća trenutnu brzinu. Ako je unos broj veći od 10 ili manji od 0.5, brzina se postavlja na 0.

Brzina se može zadati imenom sadržanim u stringu, sa sljedećim značenjem:

- "fastest" : 0
- "fast" : 10
- "normal" : 6
- "slow" : 3
- "slowest" : 1

Brzine od 1 do 10 nameću sve bržu animaciju crtanja linija i okretanja kornjača. Pozor: *speed* = 0 znači da se ne odvija animacija. fd() i bk() tjeraju kornjaču da „skače“, a jednako tako lt() i rt() trenutno okreću kornjaču.

```
>>> speed ()   # inicijalno          3
>>> speed ("fast"); speed ()        10
>>> speed (5); speed ()              5
```

setheading, seth (*to_angle*)

Postavlja orijentaciju kornjače na dani kut.

to_angle – cijeli ili realni broj

Evo nekoliko karakterističnih kuteva i njihovih orijentacija, ovisno o modu:

"standard"	"logo"
0 - istok	0 - sjever
90 - sjever	90 - istok
180 - zapad	180 - jug
270 - jug	270 - zapad

```
>>> mode ()                      'standard'
>>> heading ()                          0.0
>>> setheading (90); heading ()        90.0
>>> seth      (-90); heading()        270.0
```

setx (x)

Postavlja prvu kornjačinu koordinatu na x, druga koordinata ostaje nepromijenjena.

```
>>> position()              (0.00,0.00)
>>> setx(100); pos()      (100.00,0.00)
```

sety (y)

Postavlja drugu kornjačinu koordinatu na y, prva koordinata ostaje nepromijenjena.

```
>>> position()              (100.00,0.00)
>>> sety(100); pos ()   (100.00,100.00)
```

bye ()

Prekid rada u kornjačinoj grafici.

VIDLJIVOST KORNJAČE

hideturtle, ht ()

Kornjača postaje nevidljiva. To ćemo činiti ako radimo neki složeni crtež, jer skrivanje kornjače ubrzava crtanje.

showturtle, st ()

Čini kornjaču vidljivom.

isvisible ()

Logička funkcija koja vraća `True`*, ako je kornjača vidljiva,* `False` *ako nije.*

Kontrola zaslona

Evo još nekoliko komandi za kontrolu zaslona.

screensize (canvwidth = None, canvheight = None, bg = None)

Ako nisu dani argumenti, vraća tekuće parametre platna (zaslona) (širinu i visinu). Inače mijenja veličinu platna na kojem kornjače crtaju. Nemojte mijenjati prozor za crtanje. Da biste promatrali skrivene dijelove platna, upotrijebite trake za pomicanje. Ovom metodom mogu se učiniti vidljivim oni dijelovi crteža koji su prije bili izvan platna.

canvwidth – pozitivni cijeli broj, nova širina platna u pikselima
canvheight – pozitivni cijeli broj, nova visina platna u pikselima
bg – string ili *n*-torka boje, nova boja pozadine

```
>>> screensize ()                (400, 300)
>>> screensize (2000, 1500)
>>> screensize ()              (2000, 1500)
```

setworldcoordinates (llx, lly, urx, ury)

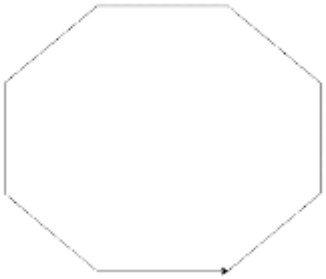
Postavlja korisnički definirani koordinatni sustav i po potrebi prebacuje se u način rada "world". Ovo izvodi `screen.reset()`*. Ako je način rada „world“ već aktivan, svi se crteži precrtavaju prema novim koordinatama.*

llx, lly - x i y koordinate donjeg lijevog kuta zaslona
urx, ury - x i y koordinate gornjeg desnog kuta zaslona

U korisnički definiranim koordinatnim sustavima kutovi mogu izgledati iskrivljeno.

```
>>> reset()
>>> setworldcoordinates(-50,-7.5,50,7.5)
>>> for _ in range (8):
    left(45); fd(2) # oktagon
```

```
>>> setworldcoordinates (-5,-5,5,5)
```



reset, resetscreen ()

Briše sve tragove kretanja kornjače i vraća zaslon na početno stanje.

clear, clearscreen ()

Briše sve crteže i sve kornjače sa zaslona i vraća prazni zaslona početno stanje: bijela pozadina, bez pozadinske slike, bez vezanja događaja i praćenja.

PARAMETRI I MJERE

colormode (cmode = None)

Postavlja môd kornjače na zadani, ili vraća tekući môd.

cmode : 1 | 255

```
>>> colormode() # inicijalno          1.0
>>> colormode(100); colormode()       1.0
>>> colormode(255); colormode()     255.0
>>> colormode(100); colormode()     255.0
```

Ako argument nije `1` niti `255` ostaje posljednje definirani *cmode*, bez dojave pogreške.

Vrijednost *cmode* određuje raspon *R*, *G* i *B* komponenti boje (*Red*, *Green* i *Blue*) koji moraju biti u rasponu od `0` do *cmode*.

distance (x, y = None)

Vraća udaljenost kornjače do koordinate (x, y) zadanog vektora ili dane druge kornjače, u jedinicama koraka kornjače.

x – broj, par (n-torka) ili instanca kornjače
y – broj, ako je *x* broj, inače None.

```
>>> home ()
>>> distance (100, 100)          141.421356
>>> A = (40, 30)
>>> distance ( A )                    50.0
>>> distance ( (30, 40) )             50.0
>>> Jo = Turtle ()
>>> Jo. fd(55)
>>> distance (Jo)                     55.0
>>> Jo. lt(90); Jo. fd(55)
>>> distance (Jo)         77.78174593052023
>>> # udaljenost je:
>>> (2 *55**2) **0.5      77.78174593052023
```

towards (x, y = None)

Vraća kut između crte položaja kornjače u položaj određen (x, y), vektorom ili drugom kornjačom. Kut je ovisan o startnom načinu rada (modu).

x – broj, par (n-torka) ili instanca kornjače
y – broj, ako je *x* broj, inače None.

```
>>> mode ()                       'standard'
>>> towards(0,0)                         0.0
>>> goto (50, 50); towards(0,0)        225.0
>>> mode ('logo'); towards(0,0)         90.0
>>> goto (50, 50); towards(0,0)        225.0
>>> isvisible()                         True
>>> ht() if isvisible() else ''
>>> isvisible()                        False
```

KONTROLA BOJE I ISPUNE

Tri su metode za kontrolu boje i ispune: `color()`, `pencolor()` i `fillcolor()`.

color ()

Vraća trenutnu boju pera i trenutnu boju ispune kao par nizova specifikacije boja.

```
>>> from turtle import *
>>> home(); color() # inicijalno
('black', 'black')
```

color (color_par)

Vraća ili postavlja boju pera za crtanje i boju ispune. Dopušteno je nekoliko formata ulaznih argumenata.

color_par : [*color* [, *color*]]
color : *naziv_boje* | (*r*, *g*, *b*) | "*#rrggbb*"

naziv_boje

Naziv boje je string koji sadrži engleski naziv boja. Na primjer: `'red'`, `'blue'`, `'green2'` itd. Nazive svih boja dali smo na kraju prethodnog poglavlja.

color (color)

Ako je naveden jedan argument *color*, boja pera i ispune bit će jednake.

```
>>> color ('green')
>>> color()       ('green', 'green')
```

color (color, color)

Prvi argument će biti boja pera, a drugi boja ispune.

```
>>> color ('green', 'blue')
>>> color()       ('green', 'blue')
```

(r, g, b)

Ime boje sadrži 479 unaprijed definiranih imena boja (v. prethodno poglavlje). Ako želimo definirati neku svoju, „miješanu“, boju, tada ćemo koristiti trojku

(*r*, *g*, *b*)

gdje su:

r – koeficijent udjela crvene boje („red“)
g – koeficijent udjela zelene boje („green“)
b – koeficijent udjela plave boje („blue“)

Vrijednost ovih koeficijenata je:

`0.0` do `1.0` za *cmode* = 1
`0` do `255` za *cmode* = 255

Ako je oblik poligon, obris i unutrašnjost tog poligona iscrtani su novo postavljenim bojama.

```
>>> color (0.5, 0.5, 0.5)
>>> color()
((0.50196, 0.50196, 0.50196),
 (0.50196, 0.50196, 0.50196))
>>> color ('red', (1, 0.5, 0))
>>> color()  ('red', (1.0, 0.50196, 0.0))
>>> colormode (255)
>>> color()  ('red', (255.0, 128.0, 0.0))
>>> 128/255            0.5019607843137255
```

"#rrggbb"

String `"#rrggbb"` je drugi način definiranja (*r*,*g*,*b*) boje u kojem su vrijednosti *r*, *g* i *b* prikazani kao

znakovni niz od tri heksadecimalna broja *rr*, *gg* i *bb* u intervalu od `0x00` do `0xff`:

```
r = float (0xrr)  g = float (0xgg)
b = float (0xbb)
```

```
>>> color ("#285078","#a0c8f0"); color()
((0.15686, 0.31373, 0.47059),
 (0.62745, 0.78431, 0.94118))
>>> colormode (255);  color()
((40.0, 80.0, 120.0), (160.0, 200.0,
240.0))
```

pencolor (*color*)

Postavlja ili vraća boju pera za crtanja.

```
>>> home()
>>> pencolor() # inicijalno       'black'
```

Promjenom boje pera mijenja se i prvi parametar metode `color()`:

```
>>> color ()           ('black', 'black')
>>> pencolor('blue'); pencolor()   'blue'
>>> color ()            ('blue', 'black')
>>> pencolor ("#FF00ff")
>>> pencolor ()           (1.0, 0.0, 1.0)
>>> color ()
((1.0, 0.0, 1.0), (1.0, 1.0, 1.0))
```

I obrnuto, promjenom prvog parametra metode `color()`, mijenja se i boja pera:

```
>>> color('red'); color()  ('red', 'red')
>>> pencolor ()                    'red'
```

fillcolor (*color*)

Postavlja ili vraća boju ispune.

```
>>> home()
>>> fillcolor () # inicijalno     'black'
```

Promjenom boje ispune mijenja se i drugi parametar metode `color()`:

```
>>> color ()           ('black', 'black')
>>> fillcolor ("yellow")
>>> color ()          ('black', 'yellow')
>>> fillcolor ("#00ffff")
>>> fillcolor ()          (0.0, 1.0, 1.0)
>>> color ()   ('black', (0.0, 1.0, 1.0))
```

I obrnuto, promjenom drugog parametra metode `color()`, mijenja se i boja ispune:

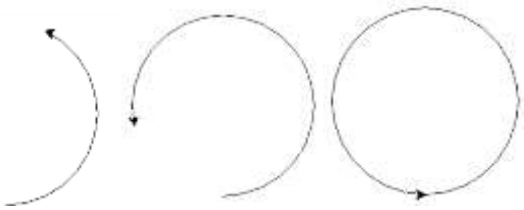
```
>>> color ('red', 'green')
>>> color ()              ('red', 'red')
>>> fillcolor ()                  'green'
```

circle (*radius, extend* = None, *steps* = None)

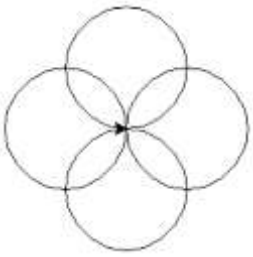
Crta kružnicu s danim radijusom. Zaokreće pero ulijevo (suprotno smjeru okretanja kazaljke na satu).

Središte su jedinice radijusa lijevo od kornjače; opseg - kut - određuje koji je dio kruga nacrtan. Ako opseg nije naveden, crta se cijeli krug. Ako opseg nije puni krug, jedna krajnja točka luka je trenutni položaj pera. Crta se luk u smjeru suprotnom od kazaljke na satu ako je polumjer pozitivan, inače u smjeru kazaljke na satu. Konačno se smjer kornjače mijenja u odnosu na opseg. Kako se krug aproksimira upisanim pravilnim poligonom, koraci određuju broj koraka koji će se koristiti. Ako se ne zada, izračunat će se automatski. Može se koristiti za crtanje pravilnih poligona.

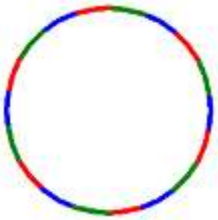
```
>>> circle (100)
```



```
# kružnice.py
from turtle import *; home()
#1 četiri kružnice
for i in range(4) : circle(50); lt(90)
```



```
input () # pauza
#2 "šarena" kružnica
reset(); pensize(5); delay (20)
B = ['red', 'blue', 'green']
for i in range (18) :
  pencolor (B[i %3])
  circle (100, 20); ht ()
```



```
input ()
```

```
#3 kružnica s promjenom boje
reset ()
pensize(8); delay (0); ht ()
for i in range(360) :
  x = i/360; color (1.0, 1.0 -x, x)
  circle (100, 1)
```

```
input ()
#4 simbol olimpijade
reset ()
pensize(8)
r = 50; d = 12; delay (10); ht()
C = ['blue', 'black', 'red',
     'yellow', 'green']
for i in range (5) :
  if i == 3 : up(); goto (r+d/2,-r); pd()
  color (C[i]); circle (r); up()
  fd (2*r+d); pd()
```

Funkciju `delay()` smo kasnije opisali.

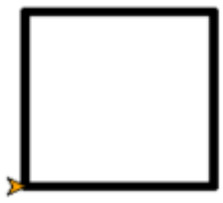
begin_fill ()

Pamti početnu točku poligona koji će biti ispunjen.

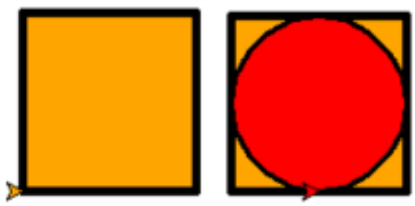
end_fill ()

Zatvara poligon i popunjava ga s tekućom bojom ispune.

```
>>> fillcolor ('orange'); pensize(5)
>>> color ()        ('black', 'orange')
>>> begin_fill ()
>>> for i in range(4) : fd (100); lt (90)
```



```
>>> end_fill ()
```



```
>>> fillcolor ('red'); pensize (3)
>>> up (); goto (50, 0); pd ()
>>> begin_fill (); circle (50)
>>> end_fill()
```

Evo još jednog primjera popune kruga crvenom bojom, potom njegovih dijelova u drugim bojama:

```
>>> reset(); color ('black', 'red')
>>> begin_fill (); circle (80)
>>> end_fill()
```

```
>>> fillcolor ('green')
>>> begin_fill (); circle (80, 90)
>>> end_fill()
>>>
>>> fillcolor ('blue')
>>> begin_fill (); circle (80, 90)
>>> end_fill()
>>>
>>> fillcolor ('yellow')
>>> begin_fill (); circle (80, 180)
>>> end_fill()
```

Popunjava se lik između luka i tetive koja spaja njegove početne i konačne točke na kružnici.

filling ()

Logička funkcija koja vraća `True` ako je startan početak ispune, inače `False`.

```
>>> begin_fill ()
>>> d = 5 if filling () else 3
>>> pensize (d)
```

dot (*size = None, *color*)

Crta krug („točku“) s danim promjerom `size`. *i bojom* `color`. *Ako veličina nije navedena, koristi se maksimum:*

size = max (pensize() +4, 2 *pensize())

```
>>> from turtle import *
>>> home ()
>>> dot (); fd (50); dot (20, "blue")
>>> fd(50); position()      (100.00,0.00)
```

Početna točka ima promjer `max(5,4)`, a druga `20`.

stamp ()

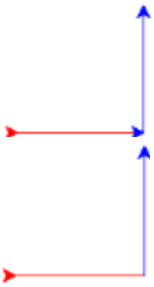
Ostavlja otisak (žig) lika kornjače na njezinoj trenutnoj lokaciji i vraća `stamp_id`, *identifikator koji se može koristiti za njezino brisanje pozivom* `clearstamp (stamp_id)`.

```
>>> color ('red'); stamp ()  11
>>> fd (100)
```

clearstamp (*stamp_id*)

Ukida otisak s identifikatorom stamp_id.

```
>>> color ('blue')
>>> _1 = stamp ()
>>> lt (90); fd (100)
>>>
>>> clearstamp (_1)
>>> pos() (100.00,100.00)
>>>
```



clearstamps (*n = None*)

Briše sve ili prvih / zadnjih ***n*** *otisaka kornjače. Ako je* ***n*** *=* None, *brišu se svi otisci, ako je* ***n>0*** *briše se prvih* ***n*** *žigova, inače ako je* ***n<0*** *briše se posljednjih* ***n*** *žigova.*

```
>>> for i in range (8) :
    fd (10 +i*5); stamp ()
5
...
12
>>>
```

```
>>> clearstamps(2); clearstamps(-2)
```

```
>>> clearstamps()
```

undo ()

Ukida posljednju akciju kornjače. Broj dostupnih radnji poništavanja određuje se veličinom undo bufera.

```
>>> for i in range ( 5) : fd(50); lt(72)
>>> for i in range (10) : undo()
```

goto, setpos, setposition (*x, y = None*)

Pomiče kornjaču na apsolutnu lokaciju s koordinatama x i y. Ako je pero spušteno, povlači crtu. Ne mijenja orijentaciju kornjače.

Ako je y = None, *x* mora biti par koordinata ili `Vec2D()` (vektor, v. `Vec2D()`).

```
>>> pos ()                    (0.00,0.00)
>>> A = Vec2D (10, 20) # vektor
>>> pos ()                    (0.00,0.00)
>>> A                        (10.00,20.00)
>>> goto (A)
>>> pos()                    (10.00,20.00)
```

Kontrola pera

STANJE CRTANJA

pen (*pen = None, **pendict*)

Vraća ili postavlja atribute pera u „pen-rječnik“.

pen - rječnik s nekim ili svim dolje navedenim ključevima.

pendict - jedna ili više ključnih riječi - argumenata s dolje navedenim ključevima kao ključnim riječima

```
'shown'         : True | False
'pendown'       : True | False
'pencolor'      : color
'fillcolor'     : color
'pensize'       : n > 0
'speed'         : n iz intervala 0..10
'resizemode'    : 'auto' | 'user' |
                  'noresize'
'stretchfactor': (n > 0, n > 0)
'outline'       : n > 0
'tilt'          : n (nagib)
```

Inicijalni sadržaj pen-rječnika je:

```
>>> pen ()
{'shown': True, 'pendown': True,
 'pencolor': 'black',
 'fillcolor': 'black',  'pensize': 1,
 'speed': 3, 'resizemode': 'noresize',
 'stretchfactor': (1.0, 1.0),
 'shearfactor': 0.0, 'outline': 1,
 'tilt': 0.0}
```

S obzirom na to da pen() ima strukturu rječnika (mape), možemo generirati sortiranu listu parova atributa i njihovih vrijednosti:

```
>>> sorted (pen().items())
[('fillcolor', 'black'), ('outline',
1), ('pencolor', 'black'), ('pendown',
True), ('pensize', 1), ('resizemode',
'noresize'), ('shearfactor', 0.0),
('shown', True), ('speed', 3),
('stretchfactor', (1.0, 1.0)), ('tilt',
0.0)]
```

Ovaj se rječnik može koristiti kao argument za sljedeći poziv pen(), za vraćanje prethodnog stanja olovke. Štoviše, jedan ili više od ovih atributa mogu se navesti kao argumenti za ključne riječi. To se može koristiti za postavljanje nekoliko atributa olovke u jednom iskazu.

```
>>> pen ()['shown']                 True
>>> ht(); pen ()['shown']           False
```

Atributi pera ne mogu se mijenjati izravno,

```
>>> pen()['shown'] = True
>>> pen()['shown']                  False
```

već izvršenjem pojedinih komandi ili navodeći ih kao argumente pozivom metode pen(). Na primjer:

```
>>> pen(shown = True, )
>>> pen()['shown']                  True
>>> pu (); pen()['pendown']         False
>>> pen (pendown  = True,
         pencolor = 'red')
>>> pen ()['pendown']               True
>>> pen ()['pencolor']              'red'
```

isdown ()

Logička funkcija koja vraća True*, ako je pero spušteno,* False *ako nije.*

```
>>> isdown ()                       True
>>> up (); isdown ()                False
```

Izgled kornjače

A gdje je kornjača? Zar je nismo vidjeli (čak dvije) u demo programu? Pitanje je na mjestu, jer riječ je o „kornjačinoj grafici“, a nje nema!

Sada ćemo konačno pokazati kako se pero može promijeniti u lik kornjače i/ili neki drugi standardni ili vlastito dizajnirani lik. I dalje ćemo u tekstu koristiti riječ „kornjača“ umjesto pero, posebno kad se radi o komandama za kretanje (crtanje). Parametri kornjačine grafike (ili varijable) dani su u nastavku.

shape (*name* = None)

Postavljanje oblika kornjače u oblik s danim imenom ili, ako ime nije dano, vraća ime trenutnog oblika.

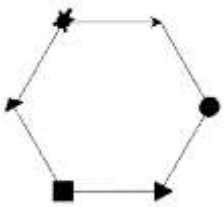
name : "classic"| "circle"| "triangle"|
"square" | "arrow" | "turtle" |
"blank" | *ime_složenog_lika*

Oblik s imenom mora postojati u rječniku oblika kornjače. U početku postoje sljedeći (standardni) oblici čije su slike prikazane u nastavku.

"classic" je inicijalni lik.

```
# shapes.py
from turtle import *

SHAPES = ("circle", "triangle",
          "square", "arrow", "turtle" )
print (shape()); a = 100; fi = 60
fd(a); stamp()
for S in SHAPES :
  shape(S); print (S); rt(fi); fd(a)
  stamp()
```

```
classic
circle
triangle
square
arrow
turtle
```



SLOŽENI OBLIK KORNJAČE

Da bi se koristio složeni oblik, koji se sastoji od nekoliko poligona različite boje, koristi se `Shape()` metoda definirana na sljedeći način:

```
Shape ( type_, data )
type_ : "polygon"| "image"| "compound"
```

data je struktura podataka koja modelira složeni oblik. Mora biti usklađena s *type_*, kao što je prikazano u sljedećoj tablici.

*type_*	*data*
"polygon"	*n*-torka parova koordinata, duljine veće od 2.
"image"	slika (u ovom se obliku koristi samo interno!)
"compound"	None (Oblik će biti konstruiran koristeći **addcomponent()** metodu)

Prvo treba stvoriti prazan oblik (objekt) tipa "compound". Ovom objektu dodaje se onoliko komponenata koliko se želi, koristeći metodu **addcomponent()**, prema sljedećoj sintaksi:

```
addcomponent (poly, fill, outline=None)
```

gdje su:

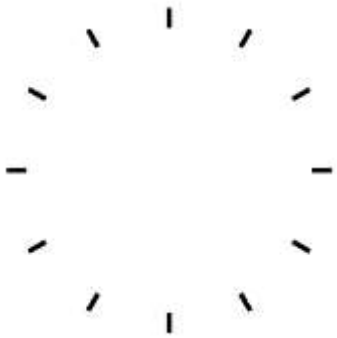
poly - poligon (*n*-torka parova brojeva)
fill - boja popune poligona
outline - boja ruba poligona (ako je dana)

```
>>> from turtle import *
>>> s = Shape ( "compound" )
>>> p = ((0,0),(10,-5),(0,10),(-10,-5))
>>> s.addcomponent (p, "red", "blue")
>>> p = ((0,0),(10,-5),(-10,-5))
>>> s.addcomponent (p, "blue", "red")
```

Kreirani oblik `s` dodaje se listi oblika i poziva:

```
>>> register_shape ( "moj_shape", s )
>>> shape          ( "moj_shape" )
```

```
# crte.py
from turtle import *
Crta = Shape ( "compound" )
p1   = ((2,0),(2, 20), (-2,20),
        (-2, 0), (2,0))
Crta . addcomponent (p1, "black",
                         "black")
register_shape ( "crta", Crta )

pu(); shape ( "crta" ); lt(90)
for i in range (12) :
  fd (150); pd(); stamp(); pu();
goto(0,0); rt(30); ht()
```



resizemode (*rmode* = None)

Postavlja `resizemode` *na jednu od vrijednosti: "auto", "user", "noresize". Ako* `rmode` *nije dan, vraća trenutni* `resizemode`. *Različiti resizemodovi imaju sljedeće učinke:*

"auto" - prilagođava izgled kornjače koji odgovara vrijednosti `pensize()`. Ako je kornjača poligon, taj se poligon crta jednakom debljinom crte.

"user" - prilagođava izgled kornjače prema vrijednostima `stretchfactor` i `outlinewidth` (outline), koje su postavljene sa `shapesize()`.

"noresize" - ne događa se prilagođavanje izgleda kornjače.

```
>>> resizemode()              'noresize'
>>> resizemode('auto')
>>> resizemode()                 'auto'
```

shapesize, turtlesize (stretch_wid=None, stretch_len=None, outline=None)

Vraća ili postavlja x/y atribute kornjače i/ili debljinu obodne linije.

`stretch_wid` – faktor širine kornjače
`stretch_len` – faktor duljine kornjače
`outline` – debljina vanjske linije

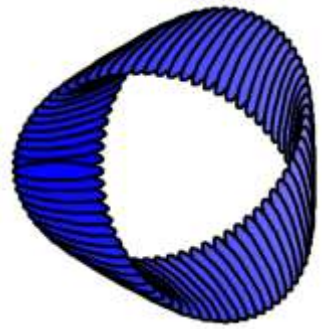
Postavite `resizemode` na `"user"`. Tada će se kornjača prikazati rastegnutom u skladu s njezinim faktorima rastezanja: `stretch_wid` je faktor okomit na njezinu orijentaciju, `stretch_len` je faktor u smjeru svoje orijentacije.

```
>>> from turtle import *
>>> home       ()
>>> shape      ("turtle")
>>> shapesize  ()          (1.0, 1.0, 1)
>>> color      ('black', 'orange4')
>>> shapesize  (5, 5, 10)
>>> turtlesize ()         (5.0, 5.0, 10)
>>> pen () ['outline']                10
```

tilt (*kut*)

Zaokreće lik kornjače za kut od trenutnog kuta nagiba, ali ne mijenja smjer kornjače (smjer kretanja).

```
# shape_tilt.py
from turtle import *

reset(); shape ("circle")
pu(); goto(-150, 0); fillcolor (0, 0, 1)
shapesize(5, 1, 4); bk (24); lt (90)
stamp()
for i in range (1, 73):
  x = i/72 if i < 36 else (73 -i)/72
  fillcolor (x, x, 1)
  fd (12); rt (5); tilt (7.5); stamp()
```



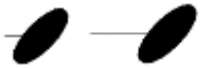
settiltangle (*kut*)

Okreće kornjaču u smjeru usmjerenom kutom, bez obzira na trenutni kut nagiba. Ne mijenja smjer kornjače (smjer kretanja).

```
>>> reset (); shape ("circle")
>>> shapesize (5, 2)
```

```
>>> settiltangle (45)
>>> fd (50)
```

```
>>> settiltangle(-45)
>>> fd (50)
```



tiltangle (*angle=None*)

Postavlja ili vraća trenutni kut nagiba. Ako je dan kut, okreće kornjaču u smjeru usmjerenom kutom, bez obzira na trenutni kut nagiba. Ne mijenja smjer kornjače (smjer kretanja). Ako kut nije dan, vraća trenutni kut nagiba, tj. kut između orijentacije oblika kornjače i smjera kornjače (njezin smjer kretanja).

```
>>> reset(); shape ("turtle"); tilt (45)
>>> tiltangle ()   45.0
```

Posebne metode

begin_poly ()

Početak pamćenja vrhova mnogokuta. Trenutni položaj kornjače prvi je vrh poligona.

end_poly ()

Prestanak pamćenja vrhova mnogokuta. Trenutni položaj kornjače zadnji je vrh poligona. To će biti povezano s prvim vrhom.

get_poly ()

Vraća posljednje snimljeni poligon.

```
>>> home(); begin_poly()
>>> fd(100); lt(20); fd(30)
>>> lt(60); fd(50)
>>> end_poly()
>>> get_poly ()
((0.00,0.00), (100.00,0.00),
(128.19,10.26), (136.87,59.50))
>>> p = get_poly ()
>>> register_shape ("mojPoly", p)
```

```
>>> shape ("mojPoly")
>>> reset()
```

Vec2D (*x*, *y*)

Dvodimenzionalna vektorska klasa, koja se koristi kao pomoćna klasa za implementaciju grafike kornjače. Može biti korisna i za programe kornjačine grafike. Izvedeno iz n-torke, tako da je vektor n-torka – uređeni par!

U prehodnim je poglavljima pokazano kako se kompleksni brojevi i *n*-torke (parovi) mogu promatrati i kao vektori. Ali, objekti klase Vec2D pružaju više mogućnosti, jer u svojoj definiciji sadrži binarne i unarne operacije nad vektorima. Ako su *a* i *b* vektori i *k* broj, evo pregleda operacija:

`a + b`	zbroj
`a - b`	razlika
`a * b`	unutarnji umnožak
`k * a`	množenje sa skalarom
`abs(a)`	apsolutna vrijednost od *a*
`a.rotate(k)`	rotacija za kut *k*

Iz matematike je poznato da je unutarnji (skalarni) proizvod dvaju vektora jednak umnošku njihovih apsolutnih vrijednosti i kosinusa kuta među njima.

```
>>> from turtle import *
>>> a = Vec2D (2, 2); b = a.rotate(60)
>>> a; b
(2.00,2.00)
(-0.73,2.73)
>>> abs(b)              2.8284271247461903
>>> abs(a)              2.8284271247461903
>>> a * b               4.000000000000002
>>> abs(a) * abs(b) * 0.5 # cos(pi/3)
4.000000000000001
```

UNOS PODATAKA

Jezik kornjačine grafike ima dvije metode za unos podataka: `textinput()` i `numinput()`.

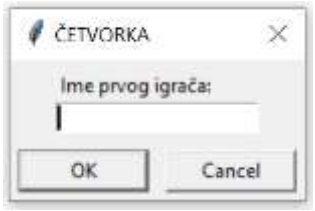
textinput (*title, prompt*)

Otvara dijaloški prozor i vraća uneseni znakovni niz. Oba parametra su stringovi. Ako je dijalog otkazan, vraća None.

- *title* Naslov dijaloškog prozora.
- *prompt* Opis podatka koji treba unijeti.

Primjer:

```
>>> X = textinput ("ČETVORKA",
                   "Ime prvog igrača:")
```



Unosom teksta i pritiskom na gumb „OK“ prozor se zatvara i vraća uneseni tekst. Potom:

```
>>> Y = textinput ("ČETVORKA",
                   "Ime drugog igrača:")
>>> X, Y              ('Dario', 'Mario')
```

numinput

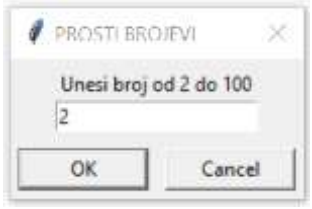
(*title, prompt,* default = None, minval = None, maxval = None)

Otvara dijaloški prozor za unos broja. `title` *je naslov dijaloškog prozora,* `prompt` *je tekst koji uglavnom opisuje koje numeričke informacije treba unijeti. Sljedeća tri parametra su opcionalna, tipa broj sa značenjem:*

`default` - *ulazna vrijednost,*
`minval` - *minimalna ulazna vrijednost,*
`maxval` - *maksimalna ulazna vrijednost.*

Ulazna vrijednost mora biti u rasponu `minval`... `maxval`, ako su navedeni. Ako nije, ispisuje se poruka i dijalog ostaje otvoren za korekciju. Vraća uneseni broj. Ako je dijalog otkazan, vraća None.

```
>>> from turtle import *
>>> Do = numinput("PROSTI BROJEVI",
    "Unesi broj od 2 do 100",2, 2,100)
100.0
```



ISPIS TEKSTA

Za ispis teksta koristi se funkcija **`write()`**.

write ()

Tekst se ispisuje od tekuće pozicije kornjače. Pozicija ostaje nepromijenjena poslije ispisa.

```
write (arg, move = False, align = 'left',
       font = ('Arial', 8, 'normal'))
```

arg - tekst koji se ispisuje
move - True ili False
align - 'left' | 'center' | 'right'
font - *n*-torka (*ime, visina, tip*)

```
# Write.py
from turtle import *
home(); ht(); color ('blue'); H = 10;
pu(); goto (-250, 0)
for x in range (5) :
  write ('Python',
         font = ('Cambria', H, 'bold'))
  pu(); H += 5
  fd (4*H); pd()
```

Python Python Python Python Python

Kontrola animacije

Sljedeće tri metode omogućuju animaciju:

- delay()
- tracer()
- update()

delay (*delay* = None)

Odgoda, zadržavanje izvršavanja komandi za zadanu vrijednost milisekundi. To je aproksimativni interval između dva uzastopna ažuriranja crtanja. Brzina animacije je veća ako je usporenje manje. Poziv bez argumenta vraća trenutačnu vrijednost usporenja. Inicijalno je jednaka 10 milisekundi.

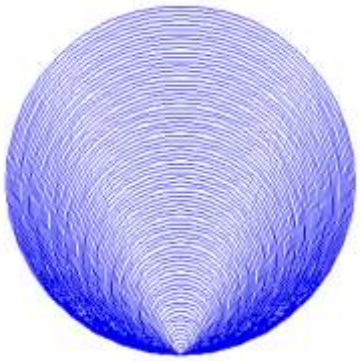
```
>>> home(); delay() # inicijalno     10
>>> delay (100);     delay()        100
>>> delay (10.999);  delay()         10
>>> delay(0);        delay()          0
>>> delay (-10);     delay()        -10
```

Vrijednost odgode je cijeli broj veći ili jednak 0. Realni argumet pretvara se u int(). Dopušteno je pisanje i negativnog argumenta. Bez obzira što neće biti dojavljena pogreška i s delay() će biti prikazana negativna vrijednost, stvarna odgoda jednaka je 0.

tracer
(n = None, delay = None)

Uključuje / isključuje animaciju kornjače i postavlja odgodu za ažuriranje crteža.

Ako je dano n, stvarno se izvodi samo svako n-to redovito ažuriranje zaslona. (Može se koristiti za ubrzavanje crtanja složenih grafika.) Drugi argument postavlja vrijednost odgode.



```
# test_tracer.py
from turtle   import *
from datetime import datetime
home(); ht(); pu(); sety (-50); pd()
R    = list( range( 1, 152, 2 ) )
boja = ("red2", "blue2", "green2")
def Test (s, b = "black") :
  # s - naredba; b - boja
  print (s);             color (b)
  t1 = datetime.now();  exec  (s)
  t2 = datetime.now();  T = t2 -t1
  dT = round (T.seconds
              +T.microseconds/10**6, 4)
  print (dT)
t = {
0 : "for r in R : circle( r )",
1 : """for r in R :
    tracer( r ); circle( r )""",
2 : """tracer( 300 )
for r in R : circle( r )""" }
for i in range ( len(t) ) :
    Test(t[i], boja[i])
```

```
>>>
for r in R : circle( r )
168.4187
for r in R :
    tracer( r ); circle( r )
1.4093
tracer( 300 )
for r in R : circle( r )
0.1625
```

update ()

Ažuriranje zaslona. Koristi se kada je `tracer()` isključen. Pogledajte i `speed()` metodu.

Rad s događajima

Grafika kornjače ima metode za rad s događajima. Opisat ćemo samo one koje se najčešće koriste.

mainloop, done ()

Pokreće petlju događaja - pozivanje Tkinterove funkcije glavne petlje. To mora biti zadnja naredba u programu kornjačine grafike. Ne smije se koristiti ako se skripta izvodi iz IDLE-a (Nema potprocesa), za interaktivnu upotrebu kornjačinih grafika.

onclick, onscreenclick (*fun*, btn = 1, add = None)

Povezivanje funkcije fun klikom miša na kornjaču ili na zaslon.

fun Funkcija s dva argumenta kojoj će biti dodijeljene koordinate kliknute točke na platnu.
btn broj gumba miša. Zadana vrijednost je 1, lijevi gumb. 3 je desni gumb.
add True ili False. Ako je True, dodat će se nova veza, inače će zamijeniti raniju vezu (binding)

Povezuje *fun* sa klikom na miša na ovom okviru. Ako je *fun* jednaka None, postojeća veza je maknuta. Slijedi implementacija gornjih metoda:

```
# onclick.py
from turtle import *
 def f (x, y) : rt (90); fd (100)
speed (1); fd (100)
onclick (f) # kliknuti na kornjaču
```

```
# onclick2.py
from turtle import *
def fx (x, y):
  up(); goto  (x, y); pd(); dot(6)
  write ("(" +str(x) +"," +str(y) +")")
def X (x, y) : undo(); undo()
w = Screen (); ht()
w . onclick (fx);  w . onclick (X, 3)
```

Kontrola zaslona

Rekli smo da se o Pythonovoj grafici kornjače može napisati posebna knjiga i da je ovo poglavlju samo uvod. Ipak, ne možemo preskočiti jedan veliki dio metoda koje se odnose na kontrolu zaslona danih u ovom podpoglavlju.

bgcolor (*color*)

Postavlja ili vraća pozadinske boje zaslona.

```
>>> bgcolor ()                    'orange'
>>> bgcolor ("#800080")
>>> bgcolor ()                (128, 0, 128)
```

bgpic (*picname* = None)

Postavlja pozadinsku sliku ili vraća ime trenutne pozadinske slike.

picname – string. ime GIF datoteke, 'nopic' ili None

Ako je ime datoteke naziv datoteke, postavit će se odgovarajuća slika kao pozadina. Ako je naziv "nopic", bit će izbrisana pozadinska slika, ako postoji.

```
# BGpic.py
from turtle import *
home(); ht();          print (bgpic ())
bgpic ("cvijeće.gif"); print (bgpic ())
```

```
nopic
cvijeće.gif
```

Brisanje pozadinske slike:

```
>>> bgpic ("nopic"); bgpic ()      'nopic'
```

clear, clearscreen ()

Briše sve crteže i sve kornjače sa zaslona. Vraća potom prazan zaslon na početno stanje: bijela pozadina, bez pozadinske slike, bez vezanja događaja i praćenja. Ova je metoda dostupna kao globalna funkcija samo pod nazivom **clearscreen**. *Globalna funkcija* **clear** *još je jedna izvedena iz Turtle metode* clear.

reset, resetscreen ()

v. str. 290..

listen (xdummy=None, ydummy=None)

Usredotočuje (fokusira) se na zaslon (kako bi se mogli izvršiti ključni događaji).

onkey (*fun, key*)

Povezuje fun s ključem. Ako je fun jednak None*, ne vežu se događaji. Napomena: da bi se mogli registrirati ključni događaji, mora biti fokus na zaslon (pogledajte metodu* listen ()*.)*

fun – fukcija bez argumenata ili None
key – string: na primjer "a" ili simbol, npr. "space"

```
>>> def f (): fd(50); lt(60)
>>> onkey (f, "Up")
>>> listen ()
```

ontimer (*fun, t=0*)

Inicira se sat koji poziva funkciju fun (bez argumenata) poslije t milisekundi.

```
>>> running = True
>>> def f () :
    if running:
       fd(50); lt(60); ontimer (f, 250)
>>> f() # tjera kornjaču da kruži okolo
>>> running = False
```

getcanvas ()

Vraća platno tekućeg zaslona. Korisno za one koji znaju što učiniti s Tkinterovim platnom.

```
>>> getcanvas()
<turtle.ScrolledCanvas object
.!scrolledcanvas>
```

getshapes ()

Vraća listu imena tekuće dostupnih oblika.

```
>>> getshapes()
['arrow', 'blank', 'circle', 'classic',
'square', 'triangle', 'turtle']
```

register_shape, addshape (*name, shape = None*)

Postoje tri različita načina za poziv ove fukcije:

name ime gif-datoteke
shape None: instalira odgovarajuću sliku

```
>>> register_shape ("turtle.gif")
```

Oblici slike se ne okreću prilikom okretanja kornjače, pa ne prikazuju naslov kornjače! *name* je proizvoljan niz, a oblik je skup parova koordinata instaliranih od odgovarajućeg poligona.

```
>>> register_shape ("triangle",
    ((5,-3), (0,5), (-5,-3)))
```

Samo se registrirani oblici mogu koristiti izvršenjem naredbe shape(*shapename*).

title (*titlestring*)

Postavlja titlestring u prozor zaglavlja kornjače.

```
>>> title ("Welcome to the turtle
zoo!")
```

setup (width = _CFG ["width"], height = _CFG ["height"], startx = _CFG ["leftright"], starty = _CFG ["topbottom"])

Postavlja veličinu i poziciju glavnog prozora. Inicijalna vrijednost argumenata pohranjena je u konfiguracijskom rječniku i može biti promijenjena preko turtle.cfg datoteke.

width – ako je int, širina je u pikselima, ako je float, širina je frakcija od zaslona; inicijalna je vrijednost 50% zaslona
height – ako je int, visina je u pikselima, ako je float, visina je frakcija od zaslona; inicijalna je vrijednost 75% zaslona
startx – ako je pozitivno, početna pozicija je u pikselima od lijeve strane zaslona, ako je negativna od desne strane zaslona, ako je None, horizontalni centar prozora
starty – ako je pozitivno, početna pozicija je u pikselima od gornje strane zaslona, ako je negativna od donje strane zaslona, ako je None, vertikalni centar prozora

```
>>> setup (200, 200, 0, 0)
>>> # postavlja prozor na 200x200
>>> # piksela u gornji lijevi zaslon
>>> setup (0.75, 0.5, None, None)
>>> # postavlja prozor 75% duljine i 50%
>>> # visine prozora u sredini
```

KLASA TK

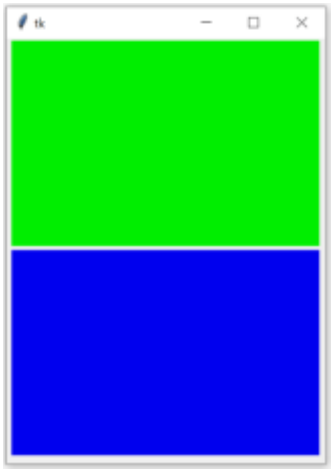
Kornjačina grafika sadrži kompletan modul tkinter. nazvan TK. Napravimo pokus:

```
>>> from turtle import TK
>>> TK
<module 'tkinter' from ...>
>>> dir (TK)
['ACTIVE', 'ALL', ..., 'wantobjects']
>>> import tkinter
```

```
>>> dir (tkinter)
['ACTIVE', 'ALL', ..., 'wantobjects']
>>> len (dir(TK))                    165
>>> len (dir(tkinter))               165
>>> dir (TK) == dir (tkinter)        True
```

Ta činjenica u mnogome povećava mogućnosti uporabe kornjačine grafike. Na primjer, otvorimo dva platna:

```
# Canvas0.py
from turtle import TK
root = TK.Tk()
cv1  = TK.Canvas (root, width=300,
          height = 200, bg = 'green2')
cv1.pack()
cv2 = TK.Canvas (root, width=300,
         height = 200, bg = 'blue2')
cv2.pack()
```



JAVNE KLASE

RawTurtle, RawPen (*canvas*)

Kreiraju kornjaču. Kornjača ima sve metode opisane kao u klasi Turtle.

canvas – TK.Canvas

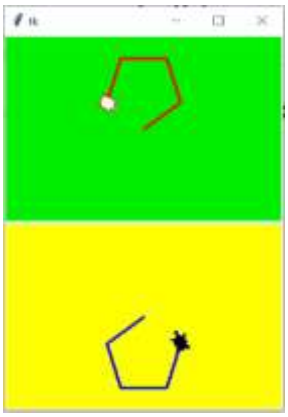
TurtleScreen (*cv*)

Pruža opisane metode orijentirane na zaslon poput setbg() *itd.*

cv – TK.Canvas

Sljedeći program je primjer koji objedinjuje prethodne tri klase:

```
# Canvas.py
# Demonstrira uporabu dva platna
from turtle import (TurtleScreen as TS,
     RawTurtle as RT, TK)
root = TK.Tk()
cv1  = TK.Canvas (root, width = 300,
          height = 200, bg = "#ddffff")
cv1.pack()
cv2 = TK.Canvas (root, width = 300,
         height = 200, bg = "#ffeeee")
cv2.pack()
s1 = TS (cv1); s1.bgcolor("green2")
s2 = TS (cv2); s2.bgcolor("yellow")
p = RT (s1); q = RT(s2)
p.color("red", "white");  p.width(3)
q.color("blue", "black"); q.width(3)

for t in p, q:
    t.shape("turtle"); t.lt(36)
q.lt(180)

# crtanje na dva platna:
for i in range(5) :
    for t in p, q : t.fd(50); t.lt(72)

# izmještanje kornjača:
for t in p, q :
    t.lt(54); t.pu(); t.bk(50)

TK.mainloop() # može biti izostavljeno
```



Screen

Podklasa od TurtleScreen, *s četiri dodane metode:* bye(), exitonclick(), setup() *i* title().

Turtle

Podklasa od RawTurtle, *ima isto sučelje, ali crta na zadanom objektu* Screen *stvorenom automatski..*

GOVORIMO PYTHONSKI

Već smo više puta rekli da je Python jezik za istraživače. Sada, kad smo uveli kornjačinu grafiku, vidici su se znatno proširili, navode nas na kreativnost i otkrivanje vlastitih dizajnerskih sposobnosti. Ponekad će se dogoditi, kao što se dogodilo i autoru ove knjige, da ćemo nenamjernom „pogreškom“ dobiti neočekivanu sliku.

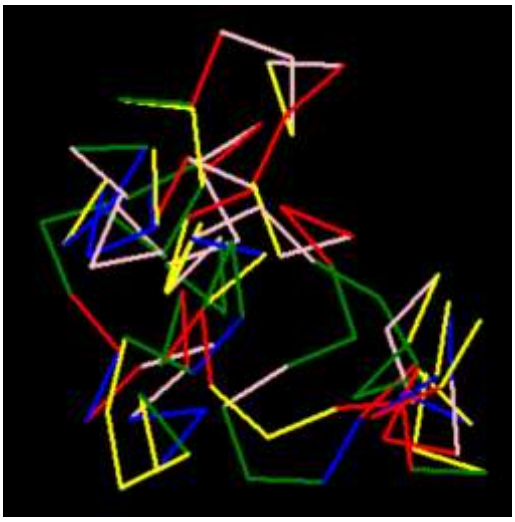
U ovom smo poglavlju opisali kornjačinu grafiku, možemo reći jedan posebno interesantan jezik, koji u kombinaciji sa strukturama podataka i naredbama Pythona omogućuje pristup programiranju u rješavanju problema iz matematike, fizike, kemije i mnogih drugih disciplina.

CRTEŽI

Evo nekoliko primjera koji pokazuju kako se s jednostavnim programima mogu dobiti interesantni crteži.

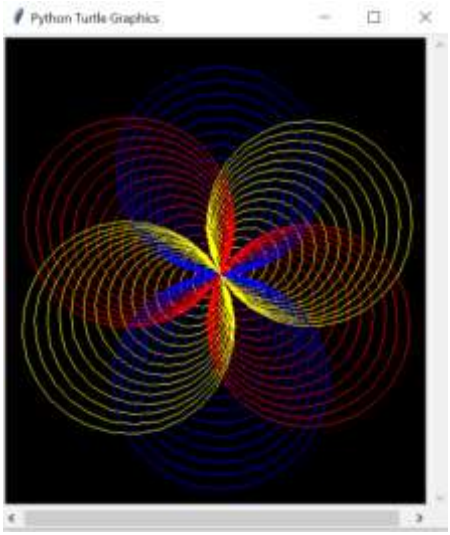
škrabotina.py

```
from turtle import *
from random import randint, choice
bgcolor ("black"); pensize (3)
def crta (n, d):
  tracer (0)
  for x in range (n):
    r = rt (randint (0, 360))
    l = lt (randint (0, 360))
    color (choice(["blue", "red",
           "green", "yellow", "pink"]))
    choice ([r, l]);  fd (d)
crta (100, 50)
```



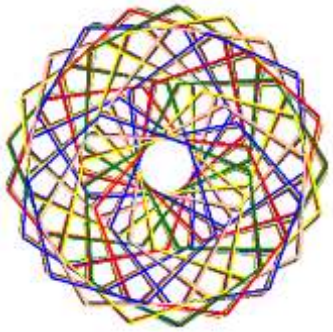
slika.py

```
from turtle import *
bgcolor("black")
boje = ["red", "yellow", "blue"]
tracer(0, 0)
for x in range(100):
  circle (x)
  color  (boje [x%3])
  lt     (60)
```



Crtež.py

```
from turtle import *
boja = ('blue', 'red', 'green',
        'yellow', 'pink')
speed(0); Screen(); width (3)
up(); rt(45); fd(90); rt(135); pd()
for x in range (121) :
  pencolor (boja[x % 5])
  for _ in range (6) :
    fd(120); rt(61)
  rt(11.1111)
ht ()
exitonclick()
```



YIN I YANG SIMBOL

Yin-Yang filozofija kaže da je svemir sastavljen od konkurentskih i komplementarnih sila mraka i svjetlosti, sunca i mjeseca, muškarca i žene. Filozofija je stara najmanje 3.500 godina, o njoj se govori u tekstu iz IX. stoljeća prije Krista, poznato kao I Ching ili Knjiga promjena, i utječe na filozofije taoizma i konfucijanizma. Simbol yin-yang povezan je s drevnom metodom koja se koristila za praćenje kretanja sunca, mjeseca i zvijezda. Više na:

https://www.thoughtco.com/yin-and-yang-629214

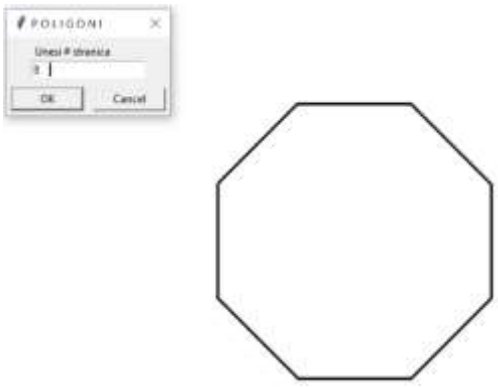
Yin_yang.py

```
from turtle import *

r = 100; r2 = r/2; r3 = r/3
c = circle; width (3)

# crta kružnicu, polumjera r
# "pola" je crno
begin_fill(); c (r, 180); lt(180)
c (-r2, 180); c (r2, 180)
end_fill()
lt (180); c (-r, 180); pu (); ht ()

# crta manje krugove (kao točke)
goto (0, 3*r2); dot (r3)
color ('white')
goto (0, r2); dot (r3)
```

GEOMETRIJSKI LIKOVI

Crta_poligon.py

```
from turtle import *
def Poligon (n):
  st(); px = 1000/n; kut = 360/n
  up(); goto( 0, 200); pensize(3); pd()
  for x in range( n ): fd(px); rt(kut)
  ht()
setup (750, 750, 0, 0)
ht(); Crtaj = True
while Crtaj :
  N = int( textinput( "P O L I G O N I",
                      "Unesi # stranica"))
  if N == 0 : Crtaj = False
  else      : clear(); Poligon( N )
```

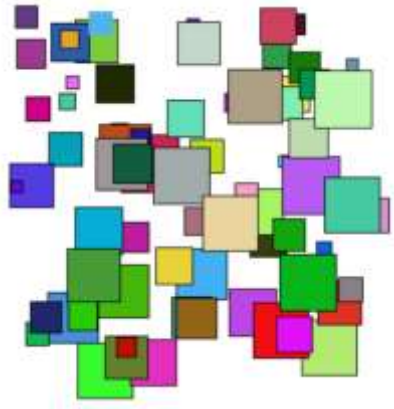


likovi.py

```
from turtle import *
from random import *
s = int (input (
    '0 - krugovi; ' +
    'n - poligon (od n stranica) '))
n = int (input ('Koliko uzoraka? '))
title ("likovi.py")
setup (600, 600, 0, 0)

def poligon (n, d):
  # n - broj stranica; d - duljina
  for x in range (n):
    fd (d)
    rt (360/n) # rt(360/d) crta
               "mahune"!
ht(); tracer(0); a = 150
for x in range (n):
  xpos = randint(-a, a)
  ypos = randint(-a, a)
  up (); goto (xpos, ypos); pd ()

  # boja
  RGB = random(), random(), random()
  fillcolor (RGB)

  # lik
  begin_fill()
  if s == 0 : circle (randint (10, 40))
  else : poligon (max (3, s),
                  randint (10, 50))
  end_fill()

>>>
```

```
0 - krugovi; n - poligon (od n stranica) 0
Koliko uzoraka? 50
```

```
0 - krugovi; n - poligon (s n stranica) 4
Koliko uzoraka? 75
```



Moj modul

Treba dodati u `Moj_modul.py` funkciju `Pauza`.

+ Moj_modul.py

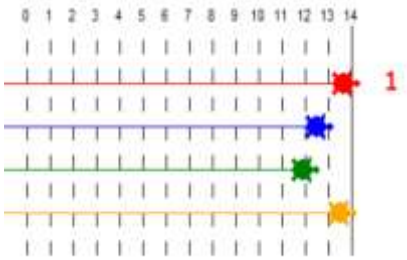
```
from datetime import *
def Pauza (sec) :
  t0 = datetime.now()
  while 'pauza' :
    t  = datetime.now()
    T  = t -t0
    dT = (T.seconds +T.microseconds/
                      10**6)
    if dT > sec : break
  return
```

Animacija

Prisustvujemo utrci četiriju kornjača, svaka u svojoj boji. Ujedno je to primjer četiri objekta klase `Turtle()`.

Utrka.py

```
from turtle import *
from random import sample, randint as rnd
speed(0); penup(); goto( -140, 140 )
begin_fill() # staza
for k in range(15) :
  write( k, align ='center' ); rt(90)
  if k < 14 :
    for i in range(8) :
       pu(); fd(10); pd(); fd(10)
  else                 : pd(); fd (160)
  pu(); bk(160); lt(90); fd(20)
end_fill()
def igrač ( boja) :
  X = Turtle (); X . color (boja)
  X . shape ('turtle')
  return X
def START ( I, y, s, x = -160 ) :
  I.pu(); I.goto(x, y); I.pd()
  for i in range (abs(360//s)) : I.rt(s)
# igrači i startne pozicije
_1 = igrač ('red');    START (_1,100, 36)
_2 = igrač ('blue');   START (_2, 70, -5)
_3 = igrač ('green');  START (_3, 40,  6)
_4 = igrač ('orange'); START (_4, 10, 12)
# utrka
d = [0] *4 # pređena udaljenost
R = []     # rang lista
while len (R) < 4 :
  x = sample (range (1, 6), 4)
  if _1 not in R : _1.fd( x[0] )
  if _2 not in R : _2.fd( x[1] )
  if _3 not in R : _3.fd( x[2] )
  if _4 not in R : _4.fd( x[3] )
  for i in range (4) :
    d[i] += x[i]; Y = eval ('_'+str(i+1))
    if d[i] > 283 and Y not in R :
      R.append (Y); Y.fd (3); Y.stamp()
      Y.ht(); Y.pu()
      Y.goto(170, Y.ycor()-10); pd()
      Y.write (str(len(R)),
           font = ('Consolas',15,'bold'))
```



DIGITALNI SAT (2)

U prethodnom smo poglavlju pokazali kako se može napisati program u Tkinteru koji bi simulirao digitalni sat. Ovdje dajemo njegovu realizaciju u kornjačinoj grafici.

Digitalni_sat.py

```
from datetime import datetime
from turtle   import *

def Vrijeme () :
    now = datetime.now()
    T = now.strftime("%H:%M:%S")
    T = T.split(":")
    return (int(T[0]),int(T[1]),int(T[2]))

def Prikazi (h, m, s) :
   write (str(h).zfill(2) +":"
         +str(m).zfill(2) +":"
         +str(s).zfill(2),
     font = ("Consolas", 30, "bold"))

setup (220, 80, 0, 0)
pu(); goto(-90, -20); pd()
bgcolor ("blue2"); color ('white'); ht()
h, m, s = Vrijeme(); Prikazi (h, m, s)
s0 = 60
while True :
  h, m, s = Vrijeme()
  if s != s0 : undo(); Prikazi (h, m, s)
  s0 = s
```

ISPIS I ZASLON

Sljedeći smo program napisali da bismo prikazali koordinatni sustav zaslona kornjačine grafike na početku ovoga poglavlja. Prilažemo ga kao primjer uporabe naredbe `write()`.

Zaslon.py

```
from turtle import *
home(); delay(0)
fnt = ("Consolas", 16, "bold")
X, Y = 310, 260

def Označi (a, b, c, d, e, f) :
  goto (xcor() +a, ycor() +b); pd()
  write (c, font = fnt); pu()
  goto (d, e); pd();
  write (f, font = fnt); pu()

for i in range (4) :
  if i in [0, 2] : fd (X)
  else           : fd (Y)
  stamp(); pu();
  if   i == 0 : Označi (-15, 10, 'x',
       X/2 -20, Y/2, 'I  (x, y)')
  elif i == 1 : Označi (15, -20, 'y',
       -X/2-60, Y/2, 'II (-x, y)')
  elif i == 2 : Označi (0, 0, '',
       -X/2-60, -Y/2, 'III(-x, -y)')
  else        : Označi (0, 0, '',
       X/2 -20, -Y/2, 'IV (x, -y)')
  goto(0,0); pd(); lt(90)

pu(); Označi (5, -25, '(0, 0)', 0, 0, '')
ht()
```

P R O G R A M I

Dajemo nekoliko „starih“ programa, uz dodatak grafike, i nekoliko novih programa u kojima su rabljene gotovo sve metode modula `turtle`. Nastojali smo maksimalno primijeniti i sve ono što smo naučili u prethodnim poglavljima.

POVRŠINA I OPSEG TROKUTA (3)

Prvo ćemo izdvojiti klase `točka` i `trokut` iz prethodne inačice programa za izračunavanje površine i opsega trokuta i izvršiti ih (kompilirati).

trokut.py

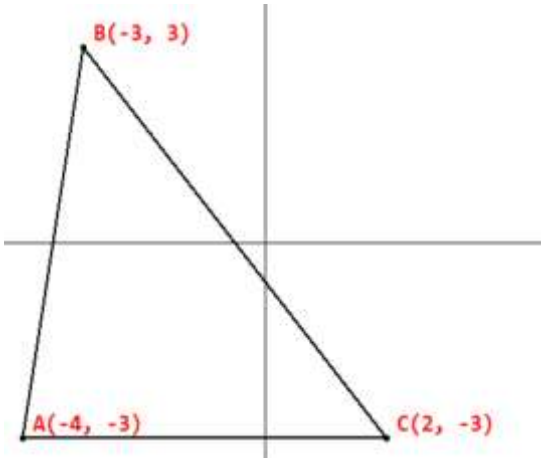
```
# KLASE (točka i trokut)
class točka:
  def __init__ (_, t, ozn = '') :
    _.x, _.y = t
    _.ozn = (ozn +'(' +str(_.x)
                 +', ' +str(_.y) +')')
  def d (_) :
    return sqrt (_.x**2 +_.y**2)
  def set_ozn (_, s = '') :
    _.ozn = ( s +'(' +str(_.x) +', '
              +str(_.y) +')' )
```

```
class trokut:
  def __init__ (o, A, B, C) :

    d = lambda X, Y : (
      round (((X.x -Y.x)**2 +
              (X.y -Y.y)**2) **0.5, 4))
    o.A, o.B, o.C = A, B, C
    o.a, o.b, o.c = (d(o.B, o.C),
         d(o.A, o.C), d(o.A, o.B))
    o.O = o.opseg(); o.P = o.površina()
   def opseg (o):
     return round (o.a +o.b +o.c, 4)
   def površina (o):
     s = o.opseg () /2
     return round ( (s*(s-o.a)
           *(s-o.b)*(s-o.c))**0.5, 4)
```

Slijedi glavni program koji koristi `trokut.py` kao svoj modul.

trokut_3.py

```
from Moj_modul import *
from turtle    import *
from trokut    import *

d = 50
def GOTO (x, y) :
  X = Vec2D (x, y) *d; goto (X)

def prikazi (t) :
  pd(); width (2); GOTO (t.x, t.y)
  dot(); up(); color ('red')
  write(' '+t.ozn, True, "left",
       ("Consolas", 14, "bold"))
  color ('black'); GOTO (t.x, t.y)

up(); GOTO (-6, 0); pd(); GOTO (6, 0)
up(); GOTO (0, -5); pd(); GOTO (0, 5);
pu()
ht()

A = Input ('Koordinate točke A ')
A = tocka (A, 'A')
up(); GOTO (A.x, A.y); prikazi(A)
B = Input ('Koordinate točke B ')
B = tocka (B, 'B'); prikazi(B)
C = Input ('Koordinate točke C ')
C = tocka (C, 'C'); prikazi(C)

pd(); GOTO (A.x, A.y)
T = trokut(A, B, C)
print( 'O =', T.O, 'P =', T.P )
```




```
>>>
Koordinate točke A -4, -3
Koordinate točke B -3, 3
Koordinate točke C 2, -3
O = 19.893 P = 18.0001
```

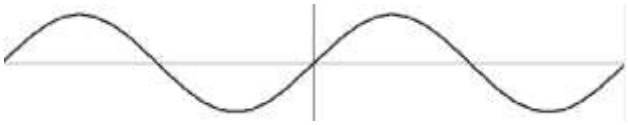
DULJINA GRAFA FUNKCIJE (2)

Računanje duljine grafa funkcije na zadanom intervalu dali smo u petom poglavlju. Ovdje dodajemo njezin graf i prikaz rezultata.

Funkcija.py

```
from math   import *
from turtle import *
_1 = 50
y  = lambda x : eval (fx)

while 1 :
  fx = input ('Upiši f(x) =  ')
  try :
    a, b = eval (input (
                 'interval a, b '))
    y(a), y(b)
    break
  except : print ('Pogreška!')

# Crtanje f(x)
home(); ht(); tracer(0)
begin_poly()
goto (-350, 0); goto(350,0); up();
home(); pd(); goto(0, 350)
goto(0,-350); home(); up()
goto (_1*a, _1*y(a)); pd()

x = a; d = abs(a -b)/100;  pensize (2)
while x <= b :
  goto (_1*x, _1*y(x)); x += d
goto (_1*b, _1*y(b))

end_poly();  update()
```

```
f  = lambda x : complex (x, y(x))

P = 1e-5;       'točnost (preciznost)'
n = 2; L0 = 0;
while True :
  d = abs (a-b)/n; A = f(a)
  L = 0; x = a+d
  while x < b+d/2 :
    if x > b -d : x = b
    B  = f(x); L += abs(A-B); A = B
    x += d
  if L != L0 and abs(L-L0) < P : break
  L0 = L; n *= 2

print ('d =', L)
>>>
Upiši f(x) =  sin(x)
interval a, b -2*pi, 2*pi
d = 15.280789276551097
```



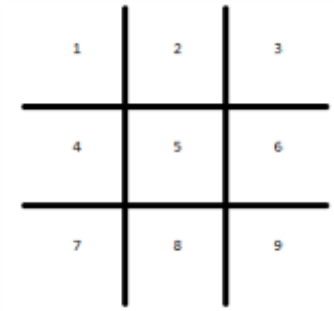
KRIŽIĆ – KRUŽIĆ (2)

Sada možemo dodati grafiku u ovu popularnu igricu. Prvo igra plavi. Pritišću se brojevi od 1 do 9.

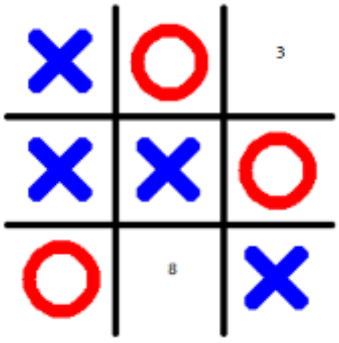
križić_kružić.py

```
from turtle import *
Križ = Shape ( "compound" )
p1   = ((-1,1),(24,26),
        (26, 24),(1, -1), (-1, 1))
p2   = ((24,-1),(-1,24),
        (1, 26),(26, 1), (24, -1))
Križ . addcomponent (p1, "blue", "blue")
Križ . addcomponent (p2, "blue", "blue")
register_shape ( "križić", Križ )

T = [1, 2]; k = 80; XY = ['']
for j in range (2, -1, -1) :
  for i in range (3) :
    XY.append ((i*k, j*k))

def Ploca () :
  speed(0); ht()
  # for _ in range (4) : fd (3*k); lt(90)
  pensize (5)
  for j in T :
    up(); goto (0, k*j); pd(); fd (3*k)

  lt (90)
  for i in T :
    up(); goto (k*i, 0); pd(); fd (3*k)
  rt (90); up()
  for i in range (1, 10):
    x, y = XY[i]
    goto  (x +k/2, y +k/2)
    write (str(i), True, "left",
           ("Consolas", 10, "normal"))

def Simbol (i, s) :
  pensize (10)
  x, y = XY[i]
  goto (x +k/2,y +k/2); color ("white")
  write(str(i), True, "left",
        ("Consolas", 10, "bold"))
  if s == A :
    shape ('križić')
    shapesize (1.5, 1.5, 8)
    goto (x +k/4, y +k-20)
  else      :
    shape ("circle")
    shapesize  (2.4, 2.4, 10)
    color ("red", "white")
    goto (x +k/2, y +k/2)
  stamp()

S = '123 456 789 147 258 369 159 357'
Ploca()
A = 'x'; B = 'o'; I = A; b = 0
Kraj = False; a = 0
def X (a) :
  global S, I, b, A, B, Kraj
  if Kraj or b == 9: return
  if len(a) == 1 and a in S :
    S = S.replace(a, I)
    Simbol (int(a), I)
    if 3*I in S :
      Kraj = True
      print ('BRAVO,', I); return
    b += 1
    if b == 9 : Kraj = True; return
    I = B if I == A else A

"""
def _1 () : X ('1')
onkey (_1, '1')
...
def _9 () : X ('9')
onkey (_9, '9')
"""
Proc = """
def _C () : X ("C")
onkey (_C, "C") """
```

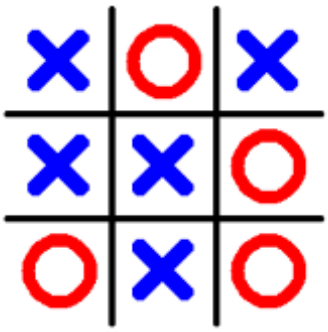
```
for C in '123456789' :
  if not Kraj :
     proc = Proc.replace ('C', C)
     exec (proc)
  else : break

listen   ()
mainloop ()
```



Pobjeda križić



ili neodlučno:

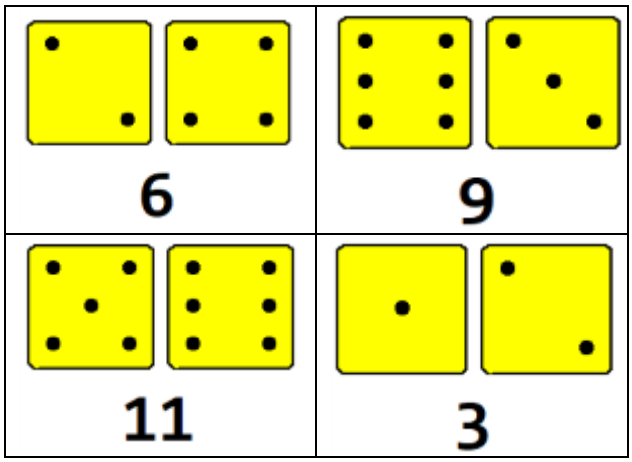


DVIJE KOCKE

„Bacaju“ se dvije kocke i prikazuje njihov zbroj. Mapa K sadrži po tri reda (*n*-torke) za svaki broj, gornji, srednji i donji. Na primjer, broj 5 je

```
5 : ('* *', ' * ', '* *')
```

Stringovi su duljine 3. Predstavljaju kolone. Znak `'*'` predstavlja točkicu.



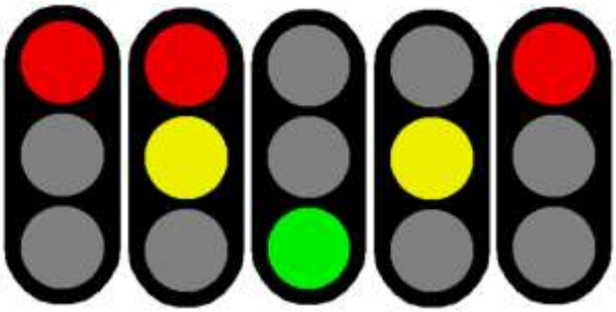
dvije_kocke.py

```
from turtle import *
from random import *

title ("DVIJE KOCKE")
setup (450, 400, 0, 0); ht ()
K = { 1 : ('   ', ' * ', '   '),
      2 : ('*  ', '   ', '  *'),
      3 : ('*  ', ' * ', '  *'),
      4 : ('* *', '   ', '* *'),
      5 : ('* *', ' * ', '* *'),
      6 : ('* *', '* *', '* *') }

def Kocka (x):
  n = randint (1, 6); tracer (0)
  begin_fill(); width(2); pu ();
  setpos (x, 0);  pd()
  for i in range (4):
    fd(90); lt(45); fd(5); lt(45)
  fillcolor ("yellow")
  end_fill()

  def Točkice (P) :
    tracer (0); begin_fill(); width(10)
    d = (1, 3, 5); h = (80, 50, 20)
    for i in range (3) :
      p = P[i]
      for k in range (3) :
        if p[k] == '*' :
          pu(); setpos (x +15*d[k],
                        h[i])
          pd(); dot(12); pu()
  Točkice (K[n])
  end_fill()
  return n
S0 = False
def Baci_kocke () :
  global S0
  exec ("undo()" *S0)
```

```
    S = str (Kocka (-105) +Kocka (5))
    x = -35 +15*(len(S) == 1)
    pu(); setpos (x, -70); color ("black")
    write (S, True, "left",
           ("Consolas", 40, "bold")); pu()
    S0 = True

onkey (Baci_kocke, "Return")
listen()
mainloop()
```

SEMAFOR

Vertikalni semafor ima tri boje: crvenu, žutu (ili narančastu) i zelenu. Crvena traje desetak sekundi, potom zajedno s njom žuta oko dvije sekunde, zelena dvadesetak sekundi, ponovo žuta oko dvije sekunde i, na kraju, crvena. Program dan u nastavku simulira takav rad.



Semafor.py

```
from turtle import *
from time   import sleep
boje = ["red2", "yellow2", "green2"]
poz  = [   200,       100,        0]

def Semafor (i, ON = None) :
  ht(); goto (0, poz[i]); st()
  Boja = "gray" if not ON else boje[i]
  color (Boja); stamp()

def Uključi () :
  tracer(1); pu()
  Semafor (0, True); sleep (15)
  Semafor (1, True); sleep ( 2);
  Semafor (0); Semafor(1)

  Semafor (2, True); sleep (20);
  Semafor (2)
  Semafor (1, True); sleep ( 3);
  Semafor (1);       Semafor (0, True);

def Okvir () :
  a = 100; d = 90; r = 60;
  fillcolor ('black')
  tracer(0); ht(); pu(); pensize (5)
  goto (-60, 0); rt(90)

  begin_fill()
  pd(); circle (r, 180)
  fd(2*a); circle (r, 180); fd (2*a)
  end_fill();

  pu(); tracer(0); shape ('circle')
  shapesize (4, 4); st()
  for s in range (3) : Semafor (s)

Okvir (); Uključi ()
```

ANALOGNI SAT

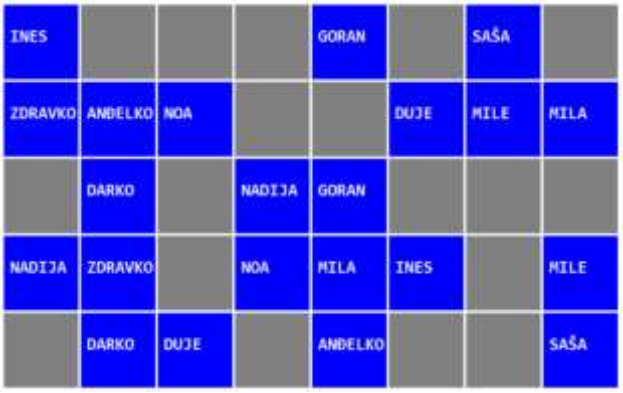
Napišimo program koji će simulirati analogni sat kao što je prikazano na slici. Primijetiti da je dobiveno efikasno rješenje ako se radi u `'logo'` modu.

Moj_sat.py

```
from turtle   import *
from datetime import datetime
from winsound import Beep

mode ('logo') # !!!
def Kazaljka (ime, r, d, k, b) :
  X = Turtle(); s = Shape ("compound")
  p = ((d,0),(d, r), (-d,r),
       (-d, 0), (d,0))
  s . addcomponent (p, b, b)
  register_shape (ime, s)
  X.shape (ime); X.degrees (k); X.ht()
  return X
def Vrijeme () :
  now = datetime.now()
  T = now.strftime("%H:%M:%S")
  T = T.split(":")
  return (int(T[0]) %12, int(T[1]),
          int(T[2]))
```

```
def Otkucavaj () :
  global s0
  S, m, s = Vrijeme ()
  if s != s0 :
     Beep (15500, 2);
     if s == 0 :
       _m.settiltangle( m )
       S += m/60; _S.settiltangle( S )
     _s.settiltangle( s )
     s0 = s

# Inicijalizacija
setup (450, 400, 100, 100); r = 150

Crta = Shape ("compound");
b    = "light grey"
p    = ((2,0),(2, 20), (-2,20),
        (-2, 0), (2,0))
Crta . addcomponent (p, b, b)
register_shape ( "crta", Crta )
bgcolor ("black"); color ("light cyan")
tracer(0); fillcolor ("black")
begin_fill()
pu(); shape ( "crta" ); ht(); lt(90)
for i in range (12) :
  if i % 3 == 0 : shapesize(2, 1)
  else          : shapesize(1, 1)
  goto(0,0); fd(r); pd(); stamp(); pu()
  if i == 0 :
    fd(26); lt(90); pd(); pensize (16);
    circle (r+26); pu(); rt(90)
  for j in range (4) :
    goto(0,0); lt(6); fd(r+10)
    shapesize(0.5, 0.5); pd(); stamp()
    pu()
  lt(6)
ht()
end_fill()
ht()

# Definiranje kazaljki
Sec, Min, Sat = 'sec', 'min', 'sat'
_s = Kazaljka (Sec, r-5,  1, 60, "red")
_m = Kazaljka (Min, r-5,  5, 60,
"gold2")
_S = Kazaljka (Sat, r-50, 5, 12,
"green2")

tracer (1)
now = datetime.now()
T = now.strftime ("%d.%m.%Y")

# Ispis datuma
color ("light grey")
fnt = ("Consolas", 30, "bold")
pu(); goto (-110, 20); pd()
write (T, font = fnt); pu()

# Početna postavka kazaljki
S0, m0, s0 = Vrijeme(); S0 += m0/60
tracer(0)
begin_fill()
_m.settiltangle( m0 ); _m.st()
_S.settiltangle( S0 ); _S.st()
_s.settiltangle( s0 ); _s.st()
end_fill(); tracer(1)

onkey   (bye, "Escape")
listen  ()

# Pokreni sat
while True : Otkucavaj ()
```

IGRA MEMORIJE



U matrici koja sadrži skrivene parove imena ili boja i brojeva na engleskom jeziku, u 5 redova i 8 stupaca, treba ih otvaranjem polja upariti.

igra_memorije.py

```
from turtle   import *
from random   import *
from winsound import *
from time     import sleep

global X, O

A  = (
  ('DARKO',  'IGOR',  'MARIN','JOZO',
   'VESNA',  'NADIJA','NOA',  'ZDRAVKO',
   'INES',   'ANA',   'MIRKO','GORAN',
   'TRISTAN','IVAN',  'MILE', 'SUZANA',
   'DUJE',   'SAŠA',  'MILA', 'ANĐELKO'),
```

```
  ('WHITE', 'BLACK', 'RED',   'YELLOW',
   'BLUE',  'PINK',  'GREEN', 'GREY',
   'BROWN', 'ONE',   'TWO',   'ORANGE',
   'THREE', 'FOUR',  'FIVE',  'SIX',
   'SEVEN', 'EIGHT', 'NINE',  'TEN ') )

Jj  = randint (0, 1)
Out = 'imena' if Jj == 0 else 'engleski'
T   = ( sample (A[Jj], 20)
       +sample (A[Jj], 20) )

title ("IGRA MEMORIJE")
setup (900, 600, 0, 0)
m = 5; n = 8; O = [];  X = []
a = 86; d = 90; x0, y0 = -350, 250
tracer (False); ht()

def Mreža ():
  global MR
  MR = [(x0 +j*d+2, y0 -i*d-2,
         x0 +j*d+a+2, y0 -i*d-a-2)
         for i in range (m)
         for j in range (n) ]

def Polje () :
  for i in range (m) :
    y = y0 -i*d
    for j in range (n) :
      x = x0 +j*d; up()
      setpos (x +2, y -2)
      pd(); color ("gray", "gray")
      begin_fill()
      for _ in range(4) : fd(a); rt(90)
      end_fill()
  update()
def Otv (b) :
  global X, O
  c = T [b];  i, j = divmod (b, n)
  x = x0 +j*d; y = y0 -i*d; up()
  setpos (x +2, y -2); pd()
  color ("blue", "blue")
  begin_fill()
  for _ in range(4) : fd(a); rt(90)
  end_fill(); update()
  up(); setpos (x +10, y -50)
  color ("white")
  write (c, True, "left",
         ("Consolas", 15, "bold"))
  if len (X) == 2 :
    if T [X[0]] == T [X[1]] :
      O += X; X = []
    else :
      sleep(2)
      color ("gray", "gray")
      for q in X :
        i, j = divmod (q, n);
        x = x0 +j*d; y = y0 -i*d; up()
        setpos (x +2, y -2); pd()
        begin_fill()
        for _ in range (4) :
          fd(a); rt(90)
        end_fill(); update()
      X = []

def nadji (x, y) :
  i = -1
  for (x1, y1, x2, y2) in MR :
    i += 1
    if ( x1 <= x <= x2 and
         y2 <= y <= y1     ) : return i
  else : return -1

def getPos(x, y):
  global X, O
  b = nadji (x, y)
  if b != -1 and b not in O+X:
     X.append(b); Otv (b)

Mreža();  Polje()
onscreenclick (getPos)
mainloop()
```

LOGO OLIMPIJSKIH IGARA

Logo olimpijskih igara sačinjen je od pet kružnica (karika) plave, crne, crvene, žute i zelene boje.

olimpijada0.py

```
from turtle import *
setup (1000, 700); pensize(8)
r = 50; d = 12; ht()
C = ['blue', 'black', 'red', 'yellow',
     'green']
for i in range (5) :
  if i == 3 :
     up(); goto (r +d/2, -r); pd()
  color (C[i]); circle(r)
  up(); fd (2*r+d); pd()
```

Ovaj je logo „aproksimacija“ pravog loga jer karike nisu ulančane (v. sliku). To čine žuta i zelena karika koje povezuju plavu, crnu i crvenu kariku. To smo, poslije podešavanja duljine lukova žute i zelene karike, postigli sljedećim programom.

simbol_olimpijade.py

```
from turtle import *
setup (1000, 700)
pensize(8); r = 50; d = 12; ht()
C = ['blue', 'black', 'red', 'yellow',
     'green']
for i in range (5) :
  if i < 3         :
    color(C[i]); circle(r); up()
    fd(2*r+d); pd()
  else             :
    if i == 3   :
       up(); goto(r+d/2, -r); pd()
    color(C[i])
    exec ( 2*("circle(r, 82); up();" +
              "circle(r, 16); pd();"))
    circle(r, 164); up()
    fd (2*r+d); pd()
```

FIBONACCIJEVA SPIRALA

Zlatni rez („božanski omjer“), (znak φ)

https://www.enciklopedija.hr/natuknica.aspx?ID=67302

odnos je dijelova a i b neke dužine kod kojega se cijela dužina $a+b$ odnosi prema većemu dijelu a kao što se veći dio a odnosi prema manjemu b:

$$\varphi = \frac{a+b}{a} = \frac{a}{b} = (\sqrt{5} + 1)/2 = 1.6180339887...$$

Često se za zlatni rez ne uzima njegova točna vrijednost nego približna, zbog praktičnih razloga. Kao prva aproksimacija uzima se omjer `8 : 5 = 1.6`. Omjer susjednih članova Fibonaccijeva niza, (osim prva dva) odgovara približnoj vrijednosti zlatnoga reza.

```
>>> F = [1, 1]
>>> for i in range(20) :
    F += [F[-2] +F[-1]]

>>> fi = []
>>> for i in range (3, len(F)-1) :
    fi += [round (F[i+1]/F[i], 8)]

>>> print (* fi)
1.66666667 1.6 1.625 1.61538462
1.61904762 1.61764706 1.61818182
1.61797753 1.61805556 1.61802575
1.61803714 1.61803279 1.61803445
1.61803381 1.61803406 1.61803396
1.618034 1.61803399
```

Na stranici

https://hr.wikipedia.org/wiki/Fibonaccijev_broj

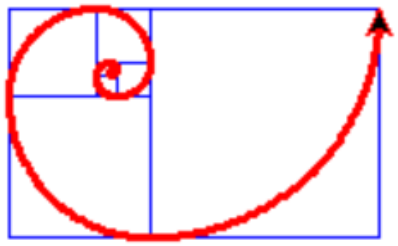
može se naći interesantnih primjera broja φ i njegove povezanosti sa Fibonaccijem i prirodom:

1. U pčelinjoj zajednici, košnici, uvijek je manji broj mužjaka pčela nego ženki pčela. Kada bismo podijelili broj ženki s brojem mužjaka pčela, uvijek bismo dobili broj φ.
2. Kada bismo izračunali odnos svakog spiralnog promjera kućice puža prema sljedećem, dobili bismo broj φ
3. Sjeme suncokreta raste u suprotnim spiralama. Međusobni odnos promjera rotacije je broj φ.
4. Izmjerimo li čovječju dužinu od vrha glave do poda, zatim to podijelimo s dužinom od pupka do poda, dobijamo broj φ.

Fibonaccijeva_spirala.py

```
from turtle import *
def fiboPlot(n):
  a = _a = 0; b = _b = 1
  pencolor ("blue")
  for _ in range(3) : fd(b *f); lt(90)
  fd(b * f)
  _a, _b = _b, _a +_b

  # crtanje ostatka kvadrata
  for i in range(1, n):
```

```
    bk(_a *f); rt(90)
    for _ in range(2) : fd(_b *f); lt(90)
    fd(_b * f)
    _b, _a = _a +_b, _b
  pu(); setposition(f, 0); seth(0); pd()
  pencolor("red")

  # Fibonaccijeva spirala
  lt(90)
  for i in range (n):
    p  = pi *b *f /2;  p /= 90
    for j in range (90): fd(p); lt(1)
    b, a = a+b, b

while 1 :
  n = eval( input(
      'Broj iteracija ( > 1): ')) # 12
  if type(n) == int and n > 1 : break

pi = 3.14159; f = 1; speed(100);
fiboPlot(n)
```



IGRA ČETVORKE (2)

Nekad, dok nije bilo mobitela ni osobnih računala, među djecom su bile popularne mnoge igre. Među njima je i „igra četvorke“. Prvu verziju bez grafike, dali smo u 10-om poglavlju. Algoritam ostaje nepromijenjen. Samo je dodana grafika utemeljena na događajima. Igrači pritišću brojeve od 0 do 7. Za to bismo trebali napisati 8 procedura:

```
def _0 () :
  global OK
  if OK : Igra (I, '0')
...
def _7 () :
  global OK
  if OK : OK = False; Igra (I, '7')
```

i 8 poziva ovisno o pritisnutom broju:

```
onkey (_0, "0")
...
onkey (_7, "7")
```

Umjesto toga možemo napisati generiranje koda:

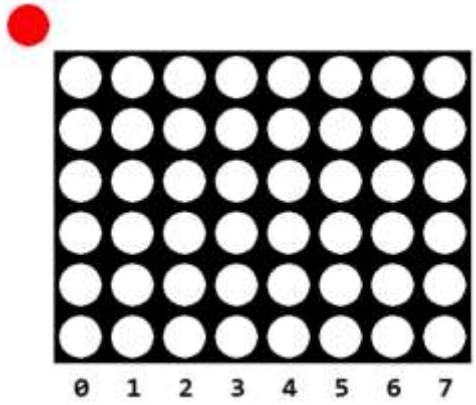
```
Proc = """
def _C () :
  global OK
  if OK : Igra (I, 'C')
onkey (_C, "C")
"""
for C in '01234567' :
  proc = Proc.replace ('C', C)
  exec (proc)
```

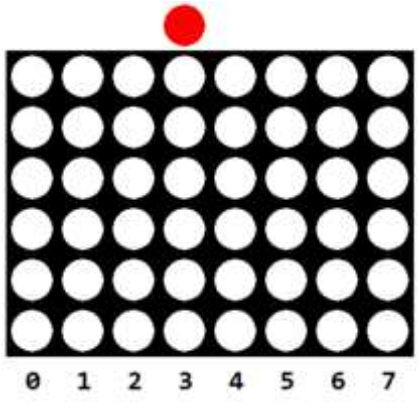
Voilà! Preostaje da priložimo cijeli kôd.

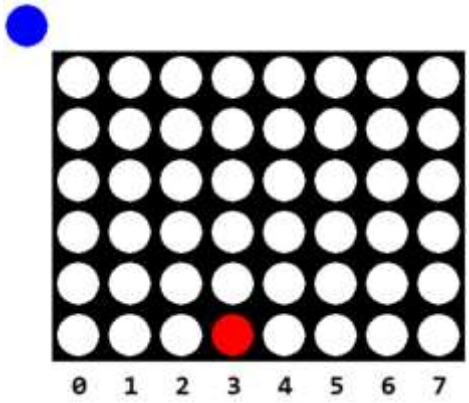
četvorka.py

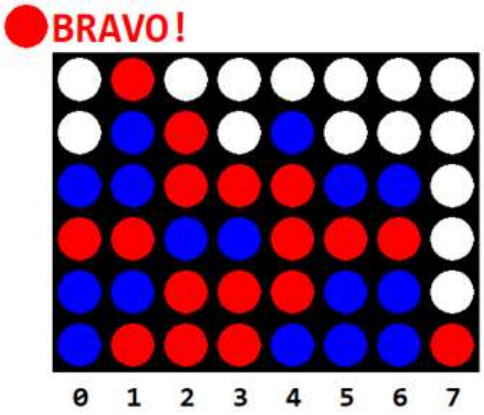
```
# IGRA ČETVORKE
from turtle import *
from time   import sleep
from random import randint
global ST

# parametri
# setup (500, 500, 0, 500)
d = 50
X0, Y0 = -4*d, -2*d
x0, y0 = X0 -d/2, 4*d +d/2
M = 6;  N = 8;  St = [0]*N
T = {}; Q = {}
for m in range (M) : T[m] = list (' '*N)
for n in range (N) : Q[n] = list (' '*M)
x, y = x0 +d , Y0 +d/2
Y = [y +d*i for i in range (M)]
X = [x +d*i for i in range (N)]
Boja = { 'P' : 'blue', 'C' : 'red' }
Pobjeda = Kraj = False
OK = True

def Izgradi_Niz (i, j) :
  global S

  def dom_M (m) : return m in range (M)
  def dom_N (n) : return n in range (N)
  S =  ''. join (T[i]) +' '
  S += ''. join (Q[j]) +' '

  for k in range (-3, 4) :
    if dom_M (i+k) and dom_N (j+k) :
      S += T[i+k][j+k]
  S += ' '
  for k in range (3, -4, -1) :
    if dom_M (i+k) and dom_N (j-k) :
      S += T[i+k][j-k]

def Igra (b, St = None) :
  global I, ST, S, Kraj, Pobjeda, OK
  OK = False
```

```
    if Kraj or Pobjeda : return
    boja = Boja[b]; pu()
    color (boja); tracer(1);
    #shape('circle'); shapesize (2,2)
    if not St : goto(x0, y0); ST = stamp()
    if St :
      St = int(St)
      if ' ' in Q[St] :
        clearstamp(ST); st(); delay(50)
        goto (x0 +(St+1)*d, y0)
        ST = stamp()
        sleep(0.25); clearstamp(ST)
        i = Q[St]. index (' ')
        up (); delay(50)
        goto (X[St], Y[i]);
        stamp(); Q[St][i] = I; T[i][St] = I
        Izgradi_Niz (i, St)
        if 4*I in S : # Pobjeda
          Kraj = Pobjeda = True
          tracer(0); goto(x0, y0); stamp()
          pu(); goto (x0 +d/2, y0 -d/2);
          write('BRAVO!', True, "left",
                ("Consolas", 30, "bold"))
          return
        if not Kraj :
           Kraj = not ' ' in T[M-1]
        if not Kraj :
           I = 'C' if I == 'P' else 'P'
        pu (); ht (); delay (0); Igra (I)
    OK = True

def Ploča () :
    global I, ST
    pu(); ht();  goto (4*d, Y0); pd()
    tracer(0); fillcolor ("black")
    begin_fill()
    for _ in range (2) :
      lt(90); fd (6*d)
      lt(90); fd (8*d)
    end_fill()
    color ('white'); pu()
    shape ('circle'); shapesize (2, 2)
    for i in range (N) :
      for j in range (M) :
        goto (X[i], Y[j]); stamp()
    up(); color ('black')
    x = X0 +20; y = Y0 -40
    for j in range (8):
      pu(); goto (x, y);
      write (str(j), True, "left",
             ("Consolas", 20, "bold"))
      pu()
      x += d
      I = 'P' if randint (0, 1) == 0 else 'C'
      Igra (I)
Proc = """
def _C () :
  global OK
  Igra (I, 'C') if OK else ''
onkey (_C, "C")
"""
for C in '01234567' :
  proc = Proc.replace('C', C); exec(proc)
Ploča ();  listen();  mainloop()
```

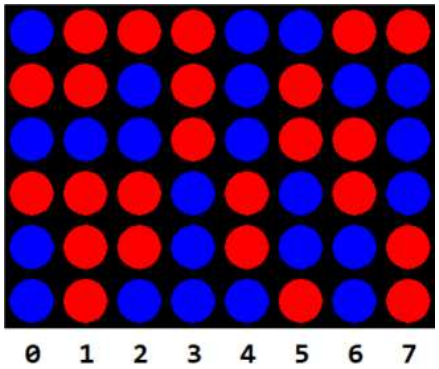








Ako su igrači iskusni, igra se može završiti bez pobjednika. Evo primjera:



MINOLOVAC (MINESWEEPER)

Minolovac (Minesweeper) poznata je igra koju je izmislio Curt Johnson devedesetih godina prošloga stoljeća, a Robert Donner ju je 1992. godine prebacio u Windowse. Otad se nalazi u svim inačicama Windowsa.

Ideja igre je da se pronađu sve skrivene mine postavljene u polju (matrici) dimenzija $m \times n$. Količina mina ili „gustoća“ je određeni postotak, na primjer 15%, od ukupnog broja lokacija. Pomoć u pronalaženju mina jest oznaka broja mina koje se nalaze uz svaku lokaciju. Prazne lokacije su one koje nemaju nijednu minu u svojoj okolini.

Igra počinje s otvorenim praznim poljem (poklon od autora programa) u sredini mreže, jer će tada biti otvorene sve prazne lokacije povezane s njom, te sve lokacije koje predstavljaju granicu s praznim lokacijama.

Inačica programa u Windowsima mjeri i ukupno vrijeme do otkrivanja svih polja. U našoj implementaciji programa nije nam cilj mjeriti vrijeme pronalaženja svih mina i možda uvoditi nervozu, već pokazati uporabu rekurzije u kreiranju algoritma.

Krenimo sada napisati program Minolovac.py. Najprije ćemo definirati konstante, deklarirati varija-ble, te napisati pomoćne procedure i funkcije. Polje `MP` pridružuje komponentama `x` i `y` na lokaciji `[i,j]` vrijednost `'0'`. Potom se “postavlja” `Mn` mina. Poslije toga se lokacijama uz mine povećava sadržaj za `1`.

Procedura `Otvori` poziva se klikom na lokaciju `[i,j]`. Ako nije izabrana lokacija koja sadrži minu, poziva se procedura `Prikazi(i,j)`. To je rekurzivna procedura koja će prikazati sadržaj lokacije `[i,j]`. Ako nije prazna (`MP[i,j].x!='0'`), bit će ispisan sadržaj lokacije, a ako je prazna, bit će otvorene sve lokacije u njezinom okruženju, matrici s indeksima `[i0,j0]`, s domenom indeksa:

```
[i-1..i+1, j-1..j+1]
```

Ako je lokacija `[i0,j0]` prazna, procedura `Prikazi` poziva sama sebe, `Prikazi(i0,j0)`. Rješenje istog problema bez uporabe rekurzije bilo bi prilično složeno.

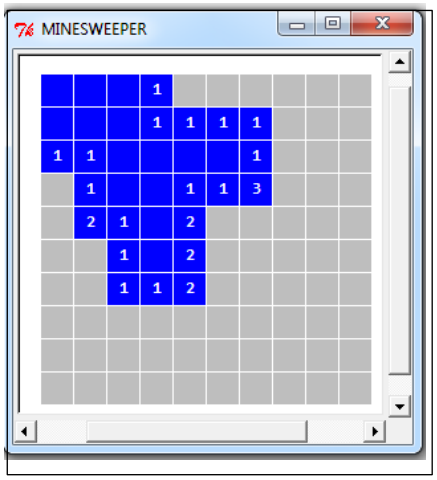
Ako se „nagazi“ na minu, poziva se procedura `Eksplozija`. koja koristi metodu `PlaySound()` iz standardnog `modula winsound`. Izvršit će se datoteka `explosion.wav` koju možete kopirati s priložene adrese. Evo programa i primjera za prikaz njegova rada.

Minesweeper.py

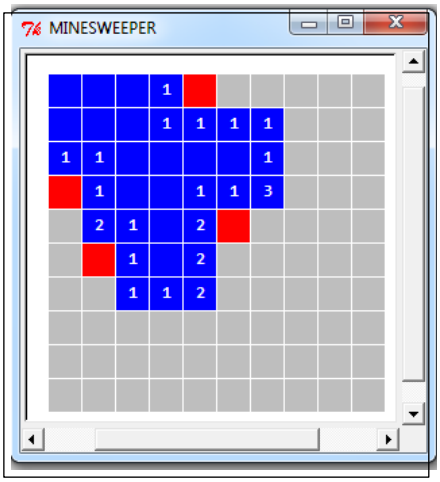
```
# Inicijalno će se otvoriti dio polja.
# S desnim gumbom miša označiti mjesta
# gdje ste sigurni da je mina.
from turtle import *; from random import *
from winsound import *; from time import *
M, N = eval (textinput ("",
    'Zadaj broj redova i stupaca (M, N)'))
Mn   = int (0.15 *M *N) # 15% mina
a = 23; d = 25;  x0, y0 = -d*N//2, d*M//2
tracer (False); ht()
```

```
def Mreza ():
  global MR
  MR = [(x0 +j*d+1, y0 -i*d-1,
     x0 +j*d+a+1, y0 -i*d-a-1)
     for i in range(M) for j in range(N) ]

def Polje (s = False) :
  for i in range (M) :
    y = y0 -i*d
    for j in range (N) :
      c = MP[i][j]; x = x0 +j*d; up()
      setpos (x +1, y -1)
      pd(); color ("gray", "gray")
      if s : color ("blue", "blue")
      begin_fill()
      for _ in range( 4) : fd(a); rt(90)
      end_fill()
      if s :
        if c == '0' : c = ' '
        if c != ' ' :
          b='white' if c != '*' else 'red'
          up(); x1, y1 = 11, 19; h = 10
          if c == '*': x1,y1 = 7,31; h=18
          setpos (x +x1, y -y1); color (b)
          write(c, True, "left",
                ("Consolas", h, "bold"))
  update()

def dom (i, j) : return (0 <= i < M
                         and 0 <= j < N )
def Prikazi (i, j) :
  c = MP[i][j]
  if  c == '0' :
    MP[i][j] = ' '; Otv (i, j)
    for i0 in range (i -1, i +2) :
      for j0 in range (j -1, j +2) :
        if dom (i0, j0) :
          c0 = MP[i0][j0]
          if c0 != '*' :
            if c0 == '0' : Prikazi(i0, j0)
            else         : Otv    (i0, j0)
  else : Otv (i, j)
def Eksplozija () :
  #https://www.freesoundeffects.com/free-
                                      sounds/
  PlaySound( "explosion.wav", True )
  sleep( 0.5)
  Polje( True )
  exitonclick()
  # KRAJ

def Otv (i, j) :
  c = MP[i][j]
  x = x0 +j*d; y = y0 -i*d; up()
  setpos (x +1, y -1); pd()
  color ("blue", "blue")
  begin_fill()
  for _ in range(4) : fd(a); rt(90)
  end_fill(); update()
  if c != ' ' :
    b = 'white' if c != '*' else 'red'
    up(); x1, y1 = 11, 19; h = 10
    if c == '*' : x1, y1 = 7, 31; h = 18
    setpos (x +x1, y -y1); color (b)
    write( c, True, "left", ("Consolas",
                      h, "bold") )
def Otvori (i, j) :
  if MP[i][j] == '*': Eksplozija () # STOP
  else       : Oznaci(i,j); Prikazi (i, j)

def Oznaci (i, j) :
  x = x0 +j*d; y = y0 -i*d; up()
  setpos (x +1, y -1); tracer (0)
  color ("red", "red")
  begin_fill()
  for _ in range(4) : fd(a); rt(90)
  end_fill(); update()

def nadji (x, y) :
  i = -1
  for (x1, y1, x2, y2) in MR :
    i += 1
    if ( x1 <= x <= x2 and
         y2 <= y <= y1 ) : return i
  else : return -1

def getPos( x, y ):
  b = nadji( x, y )
  if b != -1 :
    i, j = divmod( b, N); Otvori( i, j )

def getPos2( x, y ):
  b = nadji (x, y)
  if b != -1 :
    i, j = divmod( b, N); Oznaci( i, j )

def main () :
  global Početak
  if Početak :
     i = M // 2; j = MP[i].index('0')
     Otvori (i, j); Početak = False
  onscreenclick( getPos,  1)
  onscreenclick( getPos2, 3)
  mainloop()

Mreza(); MP = []
for i in range( M) : MP.append( ['0']*N)
Bm = 0
while Bm < Mn :
  i0 = randint(0,M-1); j0 = randint(0,N-1)
  if MP[i0][j0] == '0' :
     MP[i0][j0] = '*'; Bm += 1
```

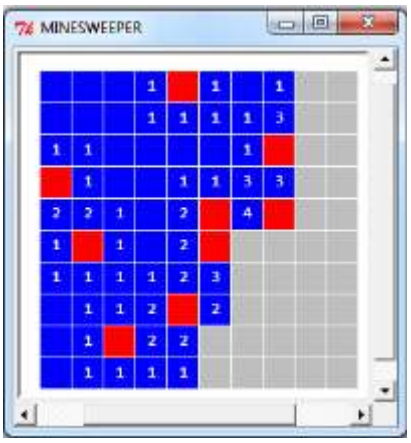
```
for i in range( M) :
  for j in range( N) :
    if MP[i][j] == '*' :
      # Povećanje sadržaja lokacija oko
      # mine za 1 '
      for i0 in range( max (i-1, 0),
                min (i+2, M)) :
        for j0 in range( max (j-1, 0),
                  min (j+2, N)) :
          x = MP[i0][j0]
          if x != '*' :
            x= nt(x)+1; MP[i0][j0] =str(x)
title ("MINESWEEPER");  Polje ()
Početak = True
main()
```

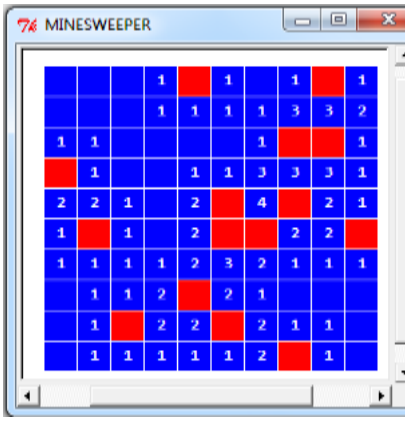


Početak igre. Veličina polja je `10x10`. Postavljeno je `15%` mina, ukupno `15`. Inicijalno je otvorena prva prazna lokacija u srednjem (petom) redu i prikazane su sve prazne lokacije povezane s njom, kao i sve lokacije koje predstavljaju granicu s praznim lokacijama. Zaključujemo da mina ne može biti na lokacijama `[1,6]`, `[1,7]`, `[5,1]`, ...(zašto?). Označimo klikom na desni gumb miša mjesta na kojima sigurno znamo da su mine.



Biramo `[1,6]`, `[1,7]`, `[3,7]`, `[5,1]`, `[6,1]`, `[7,1]`, ...



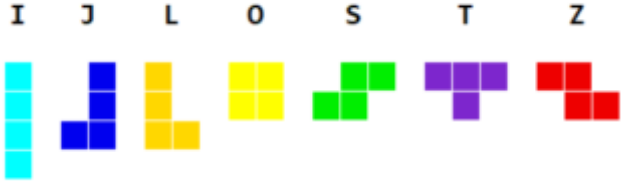


Dakle, nije bilo problema da se u svakom koraku otvori lokacija za koju smo sigurni da ne sadrži minu i tako smo otvaranjem posljednje lokacije, `[10,3]`, uspješno okončali igru i dobili prikaz kompletnog polja i lokacije mina.

TETRIS

Tetris (ruski: Тетрис) logička je videoigra, sadržana na gotovo svim igraćim platformama. Jedna je od najpoznatijih videoigara uopće. Tvorac igre je u lipnju 1984. bio ruski znanstvenik Aleksej Pažitnov. Radio je kao programer u računalnom centru Sovjetske akademije znanosti.

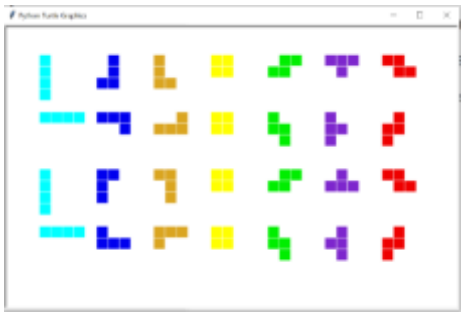
Ime "tetris" dolazi od grčkog prefiksa "tetra-" ("četiri"), jer su svi dijelovi sastavljeni od četiri segmenta, i tenisa, Pažitnovljevog omiljenog sporta. Ima ukupno 7 tetrada koje možemo označiti slovima **I**, **J**, **L**, **O**, **S**, **T** i **Z** jer svojim izgledom podsjećaju na njih.



Modul `TETRADE.py` sadrži sve tetrade u osnovnom obliku i tri rotacije ulijevo za `90` stupnjeva.

TETRADE.py

```
from turtle import *

C = ['cyan',  'blue2', 'goldenrod',
     'yellow', 'green2', 'purple3',
     'red2', 'orange', 'gray28', 'white']
T = 'IJLOSTZ'
TR = TR0 = 100;  ' trajanje stanke '
Rd       =  24;  ' broj redaka  '
St       =  11;  ' broj stupaca '
x0, y0   = -200, 250; ' ishodište '
d        = 20;   ' stanica kvadrata '
t        = [ list(' ' *St)
             for i in range (Rd) ]
Empty    = [ '*           *'] *Rd +\
           ['*' *(St+2)]
C0       = ['white'] +C[:-1]
rot      = 0
kvadrat = lambda y, x : ( (y,x), (y+d,x),
          (y+d, x+d), (y,x+d), (y,x) )

# TETRADE
E = { 'I' : (
      ['1', '1', '1', '1'],    # I0
      ['1111'            ],    #  1
      ['1', '1', '1', '1'],    #  2
      ['1111'            ] ),  #  3
      'J' : (
      [' 2' , ' 2', '22' ],    # J0
      ['222', '  2'      ],    #  1
      ['22' , '2 ', '2 ' ],    #  2
      ['2  ', '222'      ] ),  #  3
      'L' : (
      ['3 ',  '3 ', '33' ],    # L0
      ['  3', '333'      ],    #  1
      [ '33', ' 3', ' 3' ],    #  2
      ['333', '3  '      ] ),  #  3
      'O' : (
      ['44' , '44'       ],    # O0
      ['44' , '44'       ],    #  1
      ['44' , '44'       ],    #  2
      ['44' , '44'       ] ),  #  3

      'S' : (
      [' 55', '55 '      ],    # S0
      ['5 ' , '55', ' 5' ],    #  1
      [' 55', '55 '      ],    #  2
      [ '5 ', '55', ' 5' ] ),  #  3

      'T' : (
      ['666', ' 6 '      ],    # T0
      ['6 ',  '66', '6 ' ],    #  1
      [' 6 ', '666'      ],    #  2
      [' 6' , '66', ' 6' ] ),  #  3

      'Z' : (
      ['77 ', ' 77'      ],    # Z0
      [' 7' , '77', '7 ' ],    #  1
      ['77 ', ' 77'      ],    #  2
      [' 7' , '77', '7 ' ] ) } #  3

def Tetrade ( ) :
  for t in T :
    for k in range ( len (E[t]) ) :
      R = E[t][k];  Y = ()
      for i in range ( len(R) ) :
        a = R[i]
        for j in range ( len(a) ) :
          if a[j] != ' ' :
            Y += ( (i*d, j*d), )
      s = Shape ( "compound" )
      for (y, x) in Y :
        s.addcomponent ( kvadrat (y, x),
          C[T.index(t)], "white" )
      register_shape ( t +str(k), s )
if __name__ == "__main__" :
  Tetrade ()
  setup (800, 500, 0, 100)
  X = 0; Y = 15*d; up(); ht(); Δ = 16
  for rot in range (4) :
    Y -= 5*d;  i = 22*d
    for c in T :
      delay(0); setpos (X, Y)
      shape(c +str(rot)); st(); delay (Δ)
      i -= 5*d; bk (i); stamp (); ht()
```



Manje je poznato da je prva verzija Tetrisa na IBM PC platformi bila napisana u Turbo Pascalu 3.0, jeziku koji je sredinom osamdesetih godina prošloga stoljeća bio vodeći algoritamski jezik i imao je grafiku kornjače. Kompletna igra slična originalu dana je u [Dov2011].

Ovdje dajemo nepotpunu igru koja nema bodovanja, promjenu brzine spuštanja tetrada i prikaz sljedeće tetrade. Cilj je pokazali gotovo sve naredbe, tipove podataka i standardne strukture Pythona, sumirajući sve procedure kornjačine grafike u animaciji. Objekti („tetrade“) sastavljeni od četiri kvadratića padaju s vrha polja za igru širokog 11, a visokog 24 kvadratića. Igrač ih može pritiskom na odgovarajuću tipku:

↑ rotirati
←→ pomaknuti lijevo-desno
↓ spustiti

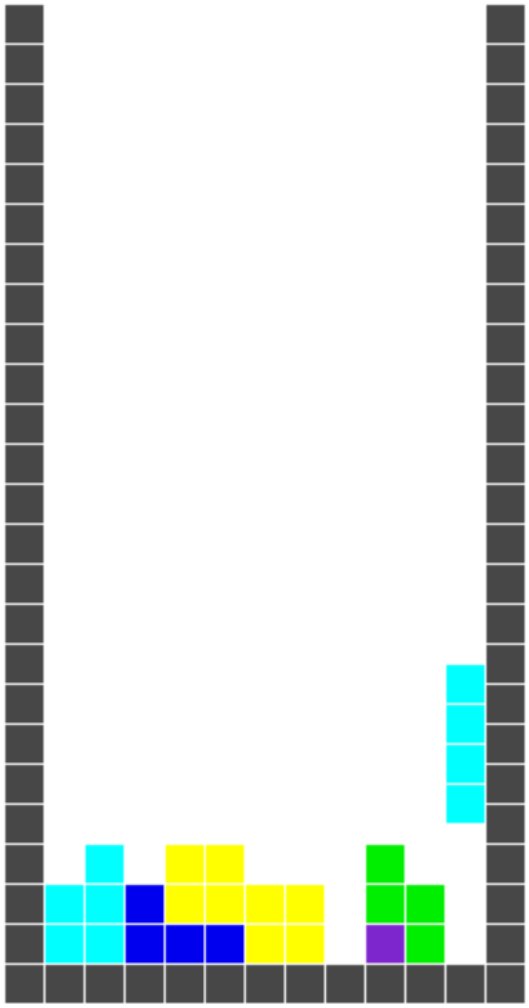
i slagati na način da između djelića ne ostaje prazan prostor. Kad se jedan red popuni djelićima, on nestaje, a kvadratići koji su bili iznad padaju u novostvoreni prazan prostor. Kada se djelići nagomilaju do vrha polja za igru, bez mogućnosti da se novi djelić pojavi, igra je gotova.

Postupak slučajnim odabirom određuje tetradu koji će se kao sljedeća pojaviti na vrhu polja. U nastavku je dan kompletan program.

TETRIS.py

```
from TETRADE import *; from random import *
Tetrade ()
global M, Start;  Start = False
Tetris = """
* 666       *
*  6        *
*           *
*  222      *
*  2        *
* 1111      *
*  3        *
*  333      *
*           *
*   666     *
*    6      *
*           *
*    144    *
*    144    *
*    155    *
*    1 55   *
*           *
*      1    *
*      1    *
*      1    *
*      1    *
*           *
*        77 *
*       77  *
************ """
Tetris = Tetris.split('\n')[1:-1]
def Prikaži () :
  global I, J, x0, y0
  st(); delay (TR)
  setpos (x0 +J*d, y0 -I*d)
def Lijevo () : global J;  Ok = Test ( 1 )
def Desno ()  : global J;  Ok = Test ( 2 )
def Spusti () : global TR; TR = TR0//8
def Umetni () :
  global I, J, B, L, D, Tet, M, TR, \
         Rd, x0, y0
  stamp(); ht(); i = I -1
  for r in B :
    i += 1; j = J-1
    for c in r :
      j += 1
      if (i < Rd and j < St
         and M[i][j] == ' ') : M[i][j] = c
  # Ukini
  k = 0; R = []
  for i in range (I, I+L) :
    if ' ' not in M[i] :
       M [i] = 'xxx'; k += 1
  if k > 0 : # Ukini k redova
    while 'xxx' in M :
      i = M. index ('xxx'); del M[i]
    # dodavanje k redova na vrh (početak)
    M = k * [list (' ' *St)] +M
    S = ['*' +''.join (M[i]) +'*'
      for i in range (Rd) ] +['*' *(St+2)]
    Pocetak (S)
def En (q, rot) :
  global B, L, D
  B = E[q][rot]; L, D = len(B), len(B[0])
def Rot () :
  global B, I, J, L, D, rot, Tet, q
  if Tet != 'O' :
    rot = (rot+1) % 4; delay(0)
    shape (Tet +str(rot)); En (q, rot)
    delay (TR)
def Test (s = 0) :
  global I, J, B, L, D, Tet, M,End, TR,TR0
  i = I-1; Ok = True
  if s == -1 : # početak
    TR = TR0; delay (500)
    for r in range (L) :
      i += 1; j = J-1
      for c in range (D) :
        j += 1; b = B[r][c]
        Ok = Ok and (i < Rd and j < St and
            (M[i][j] == ' ' or b == ' ' ))
    End = not Ok
  elif s == 0 : # dolje
    i += 1
    for r in range (L) :
      i += 1; j = J-1
      for c in range (D) :
        j += 1;  b = B[r][c]
        Ok = Ok and (i < Rd
           and j < St and (M[i][j] == ' '
            or b == ' ' ))
    if Ok : I += 1

  elif s == 1 and J > 0 : # lijevo
```

```
      j = J-1
      for i in range (I, I +L-1) :
        c  = B[i-I][0]
        Ok = Ok and (M[i][j]==' ' or c==' ')
      if Ok : J -= 1
      return Ok
    elif s == 2 and J < St-D: # desno
      j = J+D
      for i in range (I, I +L-1) :
        c  = B[i-I][-1]
        Ok = Ok and j < St and \
             (M[i][j] == ' ' or c == ' ' )
      if Ok : J += 1
      return Ok
    return Ok
  def Ekran (q) :
    global I, J, B, L, D, rot, Tet
    rot  = 0; ht(); Tet = q; delay (0)
    shape ( Tet +str(rot))
    I, J = 0, 4; setpos (x0 +J*d, y0); st()
    En (q, rot); Ok = Test (-1); delay (TR0)
    while Ok : Prikaži(); Ok = Test()
    Umetni()
  def Pocni () :
    global q, End, Start, M
    if Start : return
    Pocetak (Empty); Start = True
    M = [list(' ' *St) for i in range (Rd)]
    End   = False
    while not End :
      ht(); delay (0)
      [q] = sample(list (T),1); Ekran (q)
    Start = False

  def Pocetak (X) :
    global x0
    tracer(False); up(); x0 -=d;begin_fill()
    for i in range (Rd +1) :
      y = y0 -i*d
      for j in range (St +2) :
        x = x0 +j*d;up(); setpos (x,y); pd()
        k = 0 if X[i][j] == ' ' else \
            9 if X[i][j] == '*' else \
            int( X[i][j] )
        color (C0[0], C0[k]); begin_fill()
        for _ in range(4) : fd(d); rt(90)
        end_fill()
    update(); x0 += d; up (); tracer (True)
  Pocetak (Tetris)
  onkey (Pocni,  " ");    onkey (Rot, "Up")
  onkey (Lijevo, "Left"); onkey (Desno,
                                 "Right")
  onkey (Spusti, "Down"); onkey (bye,
                                 "Return")
  listen(); mainloop()
```



SLAGALICA (2)

Igru slagalice, sa skromnim grafičkim mogućnostima, opisali smo u desetom poglavlju. Sada smo u mogućnosti imati njezin grafički prikaz.

slagalica.py

```
from turtle   import *
from random   import *
from datetime import datetime
global S, Start, t0, Kraj, C, Q, T
Q = {
  0 : (1, 3),       1 : (0, 2, 4),
  2 : (1, 5),       3 : (0, 4, 6),
  4 : (1, 3, 5, 7), 5 : (2, 4, 8),
  6 : (3, 7),       7 : (4, 6, 8),
  8 : (5, 7) }

title("SLAGALICA")
setup(500, 500, 0, 0)
Start = Kraj = False

def Vrijeme () :
  global S, Start, t0, Kraj, Q
  def Ispis (t, c) :
```

```
    up(); setpos (-90, 120)
    color (c, "green2")
    write ("%3d" % t, True, "left",
           ("Consolas", 15, "bold"))
  s = s0 = 0
  Ispis (s, "black")
  while s <= T :
    t2 = datetime.now()
    t = t2 -t0;  s = t.seconds
    if s != s0 :
      Ispis (s0, "white")
      Ispis (s,  "black")
    s0 = s
  else :
    Start = False;  Kraj = True
    Ispis (s0, "white")
    up(); setpos (-85, 120)
    color ("red", "green2")
    write ("VRIJEME JE ISTEKLO!", True,
       "left", ("Consolas", 20, "bold"))

def Reset ():
  global t0, Start, S, Kraj, C, q
  if Kraj :
    up(); setpos (-85, 120)
    color ("white", "green2")
    write ("VRIJEME JE ISTEKLO!", True,
      "left", ("Consolas", 20, "bold"))
    Kraj = False
  S = sample(C,8) +[' ']; q = S.index(' ')
  Start = True;  Slagalica ()
  t0 = datetime.now()

def Slagalica ():
  global S, Start, Q;  a = 90; j = -1
  for y in (100, 0, -100)  :
    for x in (-100, 0, 100) :
      j += 1
      up();  goto (x+5, y-5);  pd()
      color ("white", zeleno)
      if S[j] == ' ' :
         color ("white", crno)
      begin_fill()
      for k in range (4) : fd (a); rt (90)
      up(); goto (x+40, y-60);
      if S[j] != ' ' :
        color ("white", zeleno)
        write(S[j], False, "left",
             ("Consolas", 30, "normal"))
      end_fill()
  update()
# Mreža
a = 90
M = [(x+5, y-5, x+a+5, y-a-5)
     for y in (100, 0, -100)
     for x in (-100, 0, 100) ]

ht (); tracer(0); zeleno = "green2"
crno   = "black"
def Ok () : return ''.join (S) == C +' '
global q, S
C = '12345678';  S = list (C) +[' ']
q = S.index(' ')

Slagalica ()

# Tipka "Reset"
up(); goto (-200, -105);  pd()
fillcolor ("red")
begin_fill()
for i in range (4) : fd (90); rt (90)
up(); goto (-190, -160)
write ("Reset", False, "left",
       ("Consolas", 20, "normal"))
end_fill()
update()

def getPos (x, y):
  global q, S, Start
  if Start :
    i = -1
    for (x1, y1, x2, y2) in M :
      i += 1
      if (x1 <= x <= x2 and
          y2 <= y <= y1) : b = S[i]; break
    else : b = ' '
    if b != ' ' and i in Q[q] :
      S[q] = b;  S[i] = ' '
      q = i; Slagalica()
  else :
    if (-200 <= x <= -110 and
        -195 <= y <= -105) :
      Start = True; Reset(); Vrijeme()
T = int (textinput ("",
    "Trajanje igre (sec)"))
onscreenclick (getPos)
mainloop()
```

Poslije unosa vremena trajanja igrice, treba pritisnuti na Reset poslije čega se gumbi biti slučajno raspoređeni i počinje odbrojavanje vremena.

Generiranje "lišća"

U programu likovi.py smo označili:

```
rt (360/n) # rt(360/d) crta "mahune"!

0 - krugovi; n - poligon (od n stranica)
6

Koliko uzoraka? 50
```

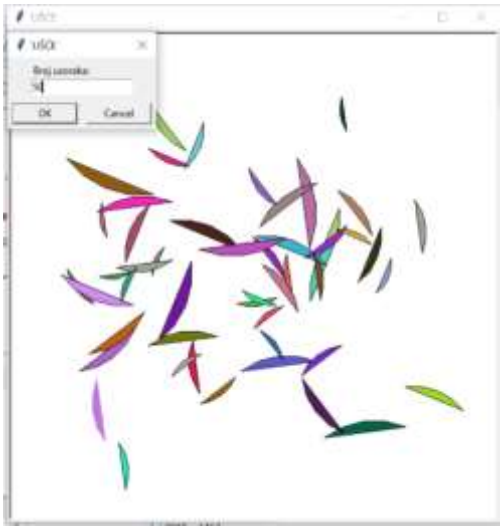
Pri testiranju tog programa bilo je napisano pogreškom `rt(360/d)` i pri odabiru poligona dobio se neočekivani rezultat:

Ponukani tom pogreškom napisali smo poseban program za generiranje lišća koji ovdje dajemo.

lišće.py

```
from turtle import *
from random import (random as rnd,
                    randint)

title ("'LIŠĆE'"); setup (600, 600, 0, 0)

def crtaj ( px ):
  x, y = xcor(), ycor(); rt(90)
  for _ in range (4) : fd(px); rt(15)
  goto (x, y)

ht(); tracer(0); Crtaj = True
while Crtaj :
  N = int(
      textinput("", "Broj uzoraka:") )
  if N == 0 : Crtaj = False
  else      :
    clear()
    for x in range(N) :
      xpos = randint( -200, 200 )
      ypos = randint( -200, 200 )
      d    = randint(   10, 30  )
      pu(); goto( xpos, ypos ); pd()
      fillcolor( rnd(), rnd(), rnd() )
      begin_fill(); crtaj(d); end_fill()
```

Za isti broj uzoraka generirano je „lišće“ i čeka se na ponovni unos, na primjer `100` itd.



15.

GOVORIMO PYTHONSKI

Evo nas na posljednjem poglavlju knjige. Sada već dobro "govorimo pythonski". Ali, to nije kraj! Python je, sigurni smo u to, kao nijedan drugi jezik za programiranje, otvoren za stalno otkrivanje novih mogućnosti.

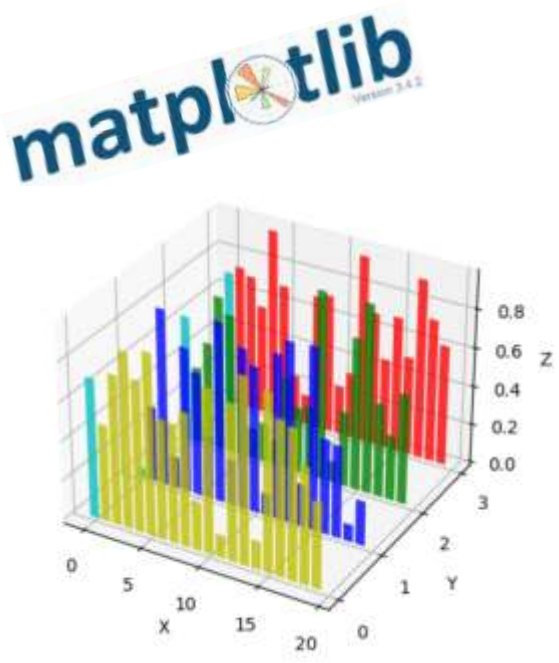
Pokazali smo da se uporabom Pythona, njegovih tipova i struktura podataka, velikog broja standardnih modula, te na internetu dostupnim modulima, mogu jednostavno i elegantno riješiti klasični algoritamski problemi, kao i problemi iz velikog broja disciplina, od matematike do obrade jezika.

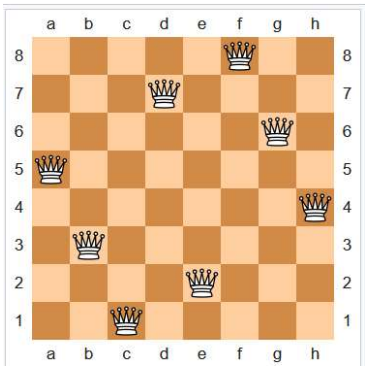
Ne zaboravimo Ctrl_F1 iz interaktivnog ili programskog moda i pozivanje on-line Pythonove dokumentacije! Osim toga, na stranici "The Python Standard Library" naći ćete sve o tipovima, strukturama podataka i standardnim modulima.

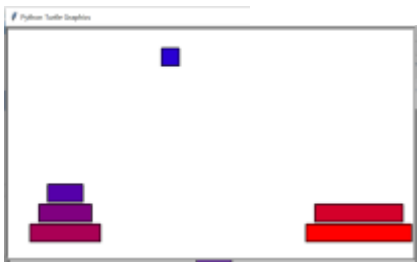
Ovdje smo pokazali kako se Python može iskoristiti za definiranje nekoliko nestandardnih struktura podataka, a posebno u teoriji grafova i formalnih jezika. S obzirom da se autor ove knjige bavi teorijom i praksom formalnih jezika i prevodilaca, dio ovog poglavlja posvećen je prikazu primjene Pythona u tim područjima, a kao motiv za vaše primjene.

Na kraju smo poglavlja dali kratki opis modula koje možete uključiti u rješavanju problema iz vaše struke. Vjerujemo da će to biti dovoljna motavacija da se krene u nove pobjede.

Živio Python!







Uvod

Ne zaboravite on-line Pythonovu dokumentaciju (F1). Tamo ćete uvijek naći odgovore na mnoga pitanja, tumačenje sintakse i semantike Pythona i primjere. Pogledat ćete pregled (popis) svih standardnih modula (Python Module Index) i možda ćete među njima naći modul koji će vam pomoći u rješavanju nekih problema.

S druge strane, na webu ćete naći puno besplatnih modula koji će još više obogatiti vaše aplikacije. Na primjer, moduli za GUI programiranje, grafiku, web programiranje, rad s bazama podataka, programiranje igrica itd.

UVOZ NESTANDARNIH MODULA I PAKETA

Uvoz nestandarnih modula i programskih paketa postiže se izvršenjem

pip : `pip install [-U]` *ime*

KOMPILIRANJE PROGRAMA

Iako je dosad pokretanje Pythonovih naredbi u interaktivnom modu ili iz programa (skripte) pomoću omiljenog uređivača teksta jednostavno, a bilo je i korisno za brže i bolje učenje, postoje neke situacije u kojima ćete radije sakriti sav kôd napisan u skripti (`.py`) unutar izvršne datoteke (`.exe`).

Možda trebate poslati skriptu nekome tko uopće ne kodira ili ćete možda trebati dogovoriti posao koji pokreće `.exe` u određeno vrijeme na vašem računalu. Neki od vas koji su programirali u nekim drugim jezicima, na primjer u Pascalu, Delphiju, Javi, C ili C++, smatrali su da Python nije dobar jer ga morate uvijek "vući" za sobom. Drugi su programirali u inačici `2.7` i znali su da za generiranje `.exe` datoteke iz Pythonove skripte postoji `py2exe` koji je to radio, ali ne radi za inačicu `3.x`.

Na internetu se mogu naći dva programa koji generiraju `.exe` datoteke: auto-py-to-exe i PyInstaller. Prvi je prikladniji za uporabu jer generira kôd i za Windows okruženje. Instalirajte ga sa:

```
pip install auto-py-to-exe
```

Potražite instalaciju na svom računalu. Trebala bi biti u folderu

C:\Python39\Lib\site-packages\auto_py_to_exe

gdje je Python39 glavni folder na kojem je instaliran Python. Prebacite je na željeni folder i napravite kraticu za poziv. Izrada izvršne datoteke je jednostavna. Evo kako smo, na primjer, dobili EXE datoteku programa `Mjenjačnica_GUI.py` iz poglavlja 13:

Poslije unosa izabrane datoteke birajte „One file" i „Windows Based".

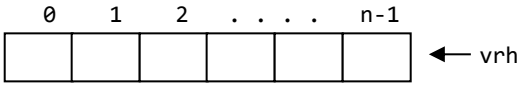
Nestandardne strukture podataka

Popularizacijom i sve većim mogućnostima jezika za programiranje šezdesetih i sedamdesetih godina prošlog stoljeća pojavio se veliki broj matematičara i znanstvenika računalnih znanosti koji su razvili nekoliko struktura podataka potrebnih za rješavanje mnogih problema. To su: red, stog, povezana lista, polje i stablo. Neke su postale dio tadašnjih jezika (npr. polje u Pascalu), a neke su do današnjih dana ostale kao dio realizacije algoritma u programskim jezicima.

Python sa svoje četiri standardne strukture podataka omogućuje, kao što ćemo prikazati, njihovu jednostavnu implementaciju.

STOG (stack)

Stog (engl. *stack*) je poseban slučaj liste u kojoj se dodavanje novog i brisanje postojećeg elementa liste obavlja na jednom njezinom kraju koji se naziva vrh. Često se stog naziva i potisna lista.



Na početku je stog prazan, pa je broj elemenata jednak 0. Dodavanjem novih elemenata na vrh stoga njihov broj se povećava. Operacija dodavanja elemenata u stog obično se naziva potiskivanje.

Brisanje elemenata stoga obavlja se također s vrha, tako da se uvijek briše element koji je na vrhu, odnosno koji je posljednji ušao u stog. Zato se u literaturi stog naziva **LIFO** (**L**ast **I**n - **F**irst **O**ut) lista. Elementi se, dakle, brišu obrnutim redom od onog kojim su ušli u stog.

Implementacija stoga u Pythonu je „prirodna“ preko liste. Za dodavanje nove vrijednosti u stog koristimo funkciju `append()`, a za izbacivanje elementa koristimo funkciju `pop()`. Ove funkcije rade učinkovito i brzo.

stog.py

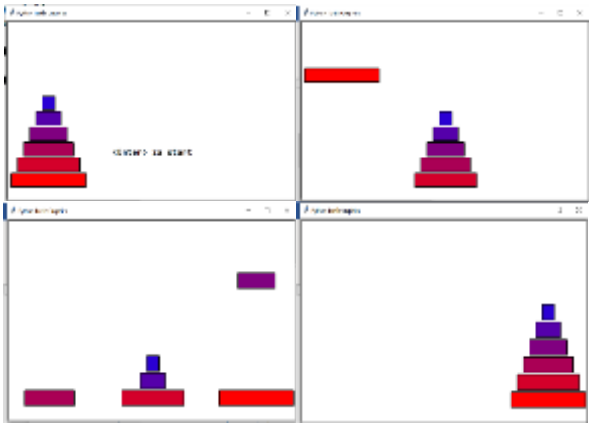
```
class STOG (list):
  _ = []

  def PUSH (_, x): _.append (x)

  def POP  (_)   :
      if len (_) : return _.pop ()
      print ('stog je prazan!')

>>> S = STOG ()
>>> for x in range(10, 51, 10): S.PUSH(x)
>>> S                [10, 20, 30, 40, 50]
>>> while S : S.POP()

50
40
30
20
10
```

Tornjevi Hanoa (2)

Vraćamo se tornjevima Hanoa (v. 10. poglavlje). Tornjevi su stogovi s diskovima kao elementima, pravokutnim oblicima (`"square"`) u kornjačinoj grafici, u različitim bojama i dimenzijama. Izvorna verzija programa je pronađena na internetu još prije desetak godina. Radila je u Pythonu 2.7. Preuredili smo je da radi u našem Pythonu.

Hanoi_2.py

```
from turtle import *
class Disk (Turtle):
  def __init__(_, n):
      Turtle.__init__(_,
      shape   = "square",
      visible = False)
      _.pu(); _.shapesize(1.5, n*1.5, 2)
      _.fillcolor(n/r, 0, 1-n/r); _.st()

class Stog (list):
  "Hanoev toranj je podklasa liste!"
  def __init__(_, x):
    """kreiranje praznog tornja.
       x je x-koordinata stupa"""
    _.x = x

  def push (_, d):
    d.setx(_.x);  d.sety(-150+34*len(_))
    _.append(d)

  def pop (_):
    d = list.pop(_);  d.sety(150)
    return d

def hanoi (n, from_, with_, to_):
  if n > 0:
      hanoi(n-1, from_, to_, with_)
      to_.push(from_.pop())
      hanoi(n-1, with_, from_, to_)

def play ():
  global Ok
  if Ok :
      Ok = False; clear()
      hanoi(N, t1, t2, t3)

global t1, t2, t3, N, r
setup (700, 500, 0, 0)
N = r = int( textinput("Hanoevi "
       "tornjevi", "Unesi broj diskova"))
ht(); pu(); goto(0, -225)
t1,t2,t3 = Stog(-250), Stog(0), Stog(250)

# postavi toranj od N diskova
for i in range(N,0,-1) : t1.push(Disk(i))

write ("<Enter> za start",
       align = "center",
       font  = ("Consolas", 16, "bold") )
Ok = True;  onkey (play, "Return")
listen ();  mainloop ()
```



RED (queue)

Red (engl. *queue*) je također posebni slučaj linearne liste. Za razliku od stoga, red ima dvije kontrolirane točke pristupa koje se nazivaju čelo i začelje. Novi elementi dodaju se u listu na začelju, a postojeći elementi brišu se s čela liste. Dakle, red radi na principu prvi ušao, prvi izašao. Zato se u literaturi naziva **FIFO** (**F**irst **I**n - **F**irst **O**ut) lista.

Implementacija u Pythonu je jednostavna. Ako je *Q* red, bit će to lista *Q* u kojoj će dodavanje biti uobičajeno s `Q.append(x)`, a izuzimanje s `Q.pop(0)`. Čelo ima indeks `0`, a začelje *n*-1 (ili -1), gdje je *n* broj elemenata.

POLJE (array)

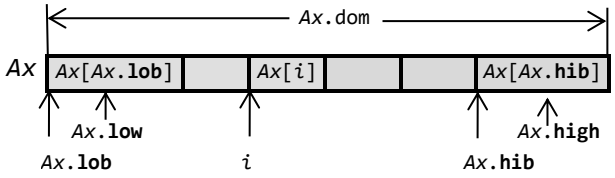
Pored primitivnih varijabli promatraju se i njihovi skupovi koji imaju isto ime, a različit „indeks“. To je struktura podataka koju nazivamo „polje“ (v. drugo poglavlje). Povijesno gledajući, u FORTRANu su, na primjer, takve varijable morale biti deklarirane s `DIMENSION` kojeg slijedi ime (ili imena) varijable i u zagradi cijeli broj veći ili jednak 1 koji je predstavljao „dimenziju“ varijable. U Pascalu su takve varijable bile deklarirane s `ARRAY` iza kojeg su u uglatoj zagradi bile navedene domene indeksa općenito višedimenzionalnog polja.

Dijkstra, [Dij1976], ne držeći se strogo rješenja u tadašnjim jezicima za programiranje, definira polje kao strukturu koja je opisana određenim atributima (svojstvima) i funkcijama (metodama), svojom domenom i semantikom. Sada se može reći da je takva struktura podataka imala obilježja klase (iako je tada Dijkstra bio protiv objektnog programiranja, v. 216 str.).

Polje može biti parametrizirano po bazičnom tipu (**`int`** ili **`bool`**). Prednost parametrizacije je u lakšem uvođenju novih tipova (pri proširenju jezika). Takvim pristupom u definiranju varijabli sa strukturom polja možemo smatrati da su strukture nad tipom **`int`** i **`bool`** jednake sve do razine parametara, tj. do elemenata strukture koji su iz domene mogućih vrijednosti bazičnog tipa.

Atributi polja

Neka je *Ax* varijabla sa strukturom polja, kao što je prikazano na slici:



U aktivnom dosegu polja *Ax* možemo izlučiti kao prvi atribut domenu i pišemo ***Ax*.dom**, a to je cijeli broj koji govori koliko elemenata sadrži polje *Ax* na određenom mjestu u programu. Indeksi polja *Ax* bit će iz neprekinutog intervala cijelih brojeva. Atributi ***Ax*.lob** i ***Ax*.hib** označivat će donju i gornju granicu domene polja *Ax*. Time je polje *Ax* definirano samo za indeks *i* koji zadovoljava uvjet:

$$Ax.\mathbf{lob} \le i \le Ax.\mathbf{hib} \wedge Ax.\mathbf{dom} \ge 0$$

Ova tri dosad uvedena atributa zadovoljavaju uvjet:

$$Ax.\mathbf{dom} = Ax.\mathbf{hib} - Ax.\mathbf{lob} + 1 \ge 0$$

Vrijednost polja *Ax* na mjestu *i* domene označivat ćemo s ***Ax*.ind(*i*)**. Za ***Ax*.dom** > 0 uvodimo dva atributa:

***Ax*.low** definiran kao ***Ax*.ind(*Ax*.lob)** i
***Ax*.high** definiran kao ***Ax*.ind(*Ax*.hib)**

Oni predstavljaju vrijednost polja *Ax* u najnižoj i najvišoj točki domene.

Inicijalizacija polja

Identifikator polja je rezervirana riječ **`array`** i pridružuje se varijabli naredbom za inicijalizaciju polja:

```
polje :
  array (tip , (broj {, konstanta } ))
  tip       : 'int'| 'bool'
  broj      : cijeli_broj
  konstanta : broj | False | True
```

Inicijalna domena polja određena je brojem navedenih konstanti. Ako nije navedena nijedna konstanta, inicijalna domena polja jednaka je nuli.

Pri pisanju konstanti mora biti zadovoljen kontekstni aspekt: tip konstante mora biti jednak bazičnom tipu polja. Definirajmo klasu `array`:

arr.py

```
class array :
  ER = 'Nije definirana operacija'
  def __init__ (_, tip, lob, x = []):
    _.tip = tip; _.lob = lob; _.val = x
    _.dom = len (x)
    _.hib = _.lob +_.dom -1
    _.low = None; _.high = None
    if _.dom > 0 :
      _.low = x[0]; _.high = x[-1]
  def in_dom (_, i) :
    return _.lob <= i <= _.hib
  def ind (_, i) :
    y  = 'NIJE DEFINIRANO'
    if _.in_dom (i) :
      i -= _.lob; y = _.val[i]
    return y
  def loext (_, x) : # Ax. loext (x)
    _.val = [x] +_.val;  _.dom += 1
    _.lob -= 1;           _.low = x
    if not _.high :      _.high = x
  def hiext (_, x) : # Ax.hiext (x)
    _.val.append (x); _.dom += 1
    _.hib += 1;       _.high = x
    if not _.low :    _.low  = x

  def lorem (_) : # Ax.lorem()
    if _.dom > 0 :
      _.val  = _.val[1:]
      _.dom -= 1; _.lob += 1
      if _.dom> 0 : _.low = _.ind(_.lob)
      else : _.low = None; _.high = None
    else : print (ER, 'LOREM')

  def hirem (_) : # Ax.hirem()
    if _.dom > 0 :
      _.val  = _.val[:-1]
      _.dom -= 1; _.hib -= 1
      if _.dom>0 : _.high = _.ind(_.hib)
      else : _.low  = None; _.high = None
    else : print (ER, 'HIREM')

  def lopop (_) : # x = Ax.lopop()
    if _.dom > 0 :
      x = _.low;  _.lorem(); return x
    else         : print (ER, 'LOPOP')

  def hipop (_) : # x = Ax.hipop()
    if _.dom > 0 :
      x = _.high; _.hirem(); return x
    else         : print (ER, 'HIPOP')
  def alt (_, i, x) :
    if _.in_dom (i) :
      if i == _.lob : _.low  = x
      if i == _.hib : _.high = x
      i -= _.lob; _.val[i] = x
    else : print ('Pogreška')
  def swap (_, i, j) : # Ax.swap (i, j)
    if _.in_dom (i) and _.in_dom (j) :
      i0 = i -_.lob;  j0 = j -_.lob;
      _.val[i0], _.val[j0] = \
                 _.val[j0], _.val[i0]
      _.low  = _.ind(_.lob)
      _.high = _.ind(_.hib)
    else :
      print ('Indeks polja van domene')
  def shift(_, k): _.lob += k; _.hib += k
  def atr ( _ ) :
    for x, y in _.__dict__.items() :
      print (x, '\t', y)
```

```
>>>
>>> Ax = array ('int', 1, [10, 20, 30])
>>> Ax. atr()
tip         int
lob         1
val         [10, 20, 30]
dom         3
hib         3
low         10
high        30
>>> A = array ('int', -7, [10, -12])
>>> A. atr()
tip         int
lob         -7
val         [10, -12]
dom         2
hib         -6
low         10
high        -12
>>> P = array ('bool', 1, [False,True])
>>> P . atr ()
tip         bool
lob         1
val         [False, True]
dom         2
hib         2
low         False
high        True
```

Referiranje na pojedine komponente polja piše se prema pravilu:

```
komponenta_polja :
        ime_polja ( .val| .ind ( i ) )
  i               : cjelobrojni_izraz
```

uz uvjet da je vrijednost cjelobrojnog izraza unutar domene indeksa polja.

```
>>> Ax.val                    [10, 20, 30]
>>> A.val                        [10, -12]
>>> A.ind(-6)                          -12
>>> A.ind(1)             'NIJE DEFINIRANO'
>>> P.val                     [False, True]
>>> P.ind (P.lob)                    False
>>> S.val                              []
```

Atributi **dom**, **lob** i **hib** su cjelobrojne vrijednosti i mogu se pojaviti kao operandi u cjelobrojnim izrazima. Atributi **low** i **high** i inicijalizirane komponente polja imaju tip jednak bazičnom tipu polja pa se, ovisno o svom bazičnom tipu, mogu pisati kao operandi u cjelobrojnim ili logičkim izrazima.

Operatori nad poljem

Nad strukturom polja, u njegovom aktivnom dosegu, postoji nekoliko operatora (ili funkcija). Ako je *Ax* varijabla sa strukturom polja i *k* cjelobrojni izraz, pravilo pisanja prvog operatora je sljedeće:

Ax. **shift** (*k*)

Operator **shift** će, ako je *k*>0, prouzročiti pomicanje granica domene navedenog polja *Ax* "udesno" za *k* mjesta; ako je *k*<0, za *k* mjesta "ulijevo". Ostali atributi i sadržaj polja se ne mijenjaju.

```
>>> X = array ('int', 1, [10,55,30,66,
        50,77,70,88,90]); X.atr()
tip         int
lob         1
val         [10, 55, 30, 66, 50,
             77, 70, 88, 90]
dom         9
hib         9
low         10
high        90
>>> X.shift (-1); X.atr()
tip         int
lob         0
val         [10, 55, 30, 66, 50,
             77, 70, 88, 90]
dom         9
hib         8
low         10
high        90
```

Sljedeća dva operatora proširuju definiranost polja s „gornjeg" ili „donjeg" kraja. Vrijednost funkcije u novoj točki dana je kao parametar *x*, a to je izraz koji mora biti iz bazičnog tipa polja. Pišu se prema pravilu:

Ax. **hiext** (*x*) i *Ax*. **loext** (*x*)

```
>>> Y2 = array ('int', 2, [4])
>>> Y2.loext (1); Y2.hiext (3**2)
>>> Y2.atr()
tip         int
lob         1
val         [1, 4, 9]
dom         3
hib         3
low         1
high        9
```

Sada uvodimo dva operatora, **hirem** i **lorem**, koji smanjuju domenu za jedan, odnosno ukidaju vrijednost na početku ili kraju polja. Definirane su pod uvjetom da je *Ax*.**dom**>0. Evo značenja tih operatora nad poljem Y2:

```
>>> Y2.lorem(); Y2.hirem(); Y2. atr()
tip         int
lob         2
val         [4]
dom         1
hib         2
low         1
high        9
```

Dalje se uvode još dva operatora

x = *Ax*.**hipop** i *x* = *Ax*.**lopop**

Prvi je semantički ekvivalentan s:

x = *Ax*.**high**; *Ax*.**hirem**

a drugi s:

x = *Ax*.**low**; *Ax*.**lorem**

Sljedeći operator preuređuje polje izmjenjujući međusobno dvije vrijednosti na mjestima *i* i *j* unutar domene. Označuje se sa **swap**. Na primjer, polje X inicijalizirano je naredbom:

```
>>> X = array ('int', 1, [10,55,30,66,50,
        77,70,88,90]); X.atr()
tip         int
lob         1
val         [10, 55, 30, 66, 50,
             77, 70, 88, 90]
dom         9
hib         9
low         10
high        90
```

Uz pretpostavku da je:

```
>>> i = X.lob
```

poslije izvršenja dijela programa:

```
>>> while i < (X.lob + X.hib)/2]
    X.swap (i, X.hib -i +1); i += 1
```

polje će X sadržavati vrijednosti u reverznom nizu, s promijenjenim atributima **low** i **high**:

```
>>> X. atr()
tip         int
lob         1
val         [90, 88, 70, 77, 50,
             66, 30, 55, 10]
dom         9
hib         9
low         90
high        10
```

Domena, donja i gornja granica domene ostali su nepromijenjeni.

Posljednji operand koji ćemo uvesti jest promjena vrijednosti varijable sa strukturom polja na mjestu *i* unutar njezine domene. Pisat ćemo:

Ax. **alt** (*i*, *x*)

pri čemu tip varijable (izraza) *x* mora biti jednak bazičnom tipu polja *Ax*. Primjer:

```
>>> X.alt(2, 22)
>>> X.val
[90, 22, 70, 77, 50, 66, 30, 55, 10]
```

FIBONNACI (6)

Primjena polja je u algoritmima s cijelim brojevima. Pokažimo to na poznatom problemu izračunavanja članova Fibonnacijevog niza. Sljedeći je program dobiven generiranjem Dijkstrinog algoritma iz jezika DDH, [Dov2013] i preuređen da bude čitljiviji. Koristi strukturu polja.

DDH_FIBONNACI.py

```
from arr import *
i = 1; a = 0; b = 1
Fib = array ('int', 1)
n = eval (input(
    'Ispis Fibonaccijevog niza do '
    'člana n, n>0 ? '))
if n>0 :
  while True :
    if i <= n :
      Fib.hiext (b)
      a, b = b, a+b
      i += 1
    else : break
elif n <= 0 : exit ()
print ('Prvih ', n, 'članova '
       'Fibonaccijevog niza:')
i = 1
while True :
  if i <= Fib.dom :
    print (Fib.ind (i), end = ' '); i +=1
  else : break
```

```
>>>
Ispis Fibonaccijevog niza do člana n, n>0
? 20
Prvih  20 članova Fibonaccijevog niza:
1 1 2 3 5 8 13 21 34 55 89 144 233 377
610 987 1597 2584 4181 6765
```

GRAFOVI

Graf je apstraktni matematički objekat. No, uobičajeno je da se njegov geometrijski prikaz - lik sastavljen od točaka i crta koje ih spajaju - naziva grafom. Grafovima (stablima) je moguće opisati mnoge strukture u računalskim znanostima, posebno u teoriji formalnih jezika i relacijskih baza podataka.

Neformalno, grafovi su likovi sastavljeni od točaka od kojih su neke (dvije po dvije) spojene krivuljama. Postoji nekoliko definicija grafa. Ovdje dajemo onu kojom se pojam grafa povezuje s pojmom binarne relacije.

Neka je $X=\{x_1, x_2, \ldots, x_n\}$ neprazan skup i ρ binarna relacija u X, $\rho \subseteq X \times X$. Uređeni se par $\Gamma=(X,\rho)$ naziva graf, elementi skupa X čvorovi, a elementi skupa ρ grane grafa. Na primjer, ako je

```
X1 = {1, 2, 3, 4}
ρ1 = {(1,1), (1,2), (2,3), (2,4),
      (3,4), (4,3)}
Γ1 = (X1, ρ1)
```

graf $\Gamma 1$ napisan je izravno u Pythonu!

Osim crtežom, graf može biti prikazan kvadratnom matricom čiji je red jednak broju čvorova. Takva se matrica naziva matrica susjedstva ili Booleova matrica (jer su joj elementi 0 ili 1). Element a_{ij} na presjeku i-tog retka i j-tog stupca u toj matrici, jednak je 1, ako je $(x_i, x_j) \in \rho$, odnosno 0 ako to nije ispunjeno. Na primjer, graf Γ_1 može biti prikazan sljedećom matricom susjedstva:

	1	2	3	4
1	1	1	0	0
2	0	0	1	1
3	0	0	0	1
4	0	0	1	0

Evo još nekoliko definicija vezani uz pojam grafa.

Niz čvorova $(x_0,\ldots, x_n)$, $n\geq 1$, jest put duljine n od čvora x_0 do čvora x_n ako postoji grana koja izlazi iz čvora x_{i-1} i ulazi u čvor x_i, za $1\leq i\leq n$. Ako između čvorova a i b grafa Γ postoji put duljine k, $k\geq 1$, tada je a prethodnik od b, odnosno, b je slijednik od a. Ako je $k=1$, a je izravni prethodnik od b. Tada je b izravni slijednik od a. Graf $\Gamma=(X,\rho)$ jest povezan ako postoji put od a do b za sve parove različitih čvorova.

Ulazni stupanj čvora x grafa $\Gamma=(X,\rho)$, $x\in X$, jest broj grana koje ulaze u x. Izlazni stupanj jest broj grana koje izlaze iz čvora x.

NAJKRAĆI PUTOVI U GRAFU

Zadan je graf s *n* čvorova i udaljenostima između njih. U programu je to matrica susjedstva, M. Potrebno je odrediti najkraće putove od svakog do nekog zadanog čvora. Opet dajemo Dijkstrin algoritam objavljen u [Dij1976] i generiran u jeziku DDH, [Dov2013]. I ovog puta malo uređenog.

DDH_PUTOVI.py

```
from arr import *;  N = q = 5
M = array ('int', 1, [
#         matrica susjedstva
#         1    2    3    4    5
          0,   1, 100,   4,   5,    # 1
          1,   0,   2, 100,   8,    # 2
        100,   2,   0,   3, 100,    # 3
          4, 100,   3,   0,   6,    # 4
          5,   8, 100,   6,   0, ]) # 5
Pre = array ('int', 1,[]) # prethodnik
Ud  = array ('int', 1,[]) # udaljenost
Slj = array ('int', 1,[]) # sljedbenik
j   = 1
while True :
  if j <= N :
    Ud.hiext (100000); Pre.hiext (0)
    j += 1
  else : break
Slj.hiext (q);  Ud.alt (q, 0);  j = 1
while True :
  if Slj.hib >= 1 :
    def BEGIN_1 () :
      global N, M, j, Ud, Slj, Pre
      LL = Slj.high;  KK = (LL-1)*N +j
      if (Ud.ind (j) -Ud.ind (LL)
          > M.ind (KK)) :
        Ud.alt (j, Ud.ind (LL) +M.ind
                (KK)); Slj.hiext (j)
        Pre.alt (j, LL); j = 1
      elif (Ud.ind (j) -Ud.ind (LL)
            <= M.ind (KK)) :
        if    j < N: j=j+1
        elif j == N: Slj.hirem(); j=1
    BEGIN_1 ()
  else : break
j = 1
while True :
  if j <= Pre.dom :
    print (
          '(', j, ') ->', Pre.ind (j))
    j += 1
  else : break
```

```
>>>
( 1 ) -> 5
( 2 ) -> 1
( 3 ) -> 2
( 4 ) -> 3
( 5 ) -> 0
```

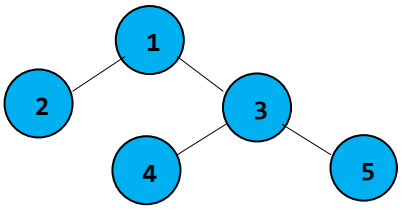
STABLO (tree)

Na kraju ovoga podpoglavlja dajemo još jednu definiciju strukture podataka koja se odnosi na posebnu skupinu grafova – stabla.

Stablo, označimo ga s τ, jest povezani graf $\Gamma=(X,\rho)$ s $n\geq 1$ čvorova i $m=n-1$ grana. Korijen stabla jest čvor x sa svojstvima:

- ulazni stupanj od x jednak je 0,
- svi ostali čvorovi stabla imaju ulazni stupanj jednak 1, i
- svaki je čvor iz X slijednik od x.

List stabla τ jest čvor z sa svojstvom da mu je izlazni stupanj jednak 0. Sljedeći je graf primjer stabla:



Korijen je označen s 1. Čvorovi 2 i 3 izravni su slijednici od 1, a čvor 3 izravni prethodnik čvorovima 4 i 5.

Ako je τ stablo, iz njegove definicije slijedi da ne sadrži petlje (veza čvora sa samim sobom) i da postoji

jedinstveni put od korijena do svakog njegova čvora.

Primjere stabla nalazimo u svakodnevnoj uporabi računala. Na primjer, folderi imaju takovu strukturu. Korijeni su particije diska: C, D, E..., a čvorovi su folderi koji se nalaze u njima.

Algoritmi

U prethodnim poglavljima dano je dosta jednostavnih i složenih algoritama. Evo još nekoliko. Prva grupa se odnosi na rješenje problema sortiranja, potom algoritam za izračunavanje n-te permutacije.

SORTIRANJE

Do pojave Pythona sortiranje i pretraživanje bili su jedni od glavnih algoritama. Iz „tradicionalnih" razloga ovdje dajemo nekoliko najpoznatijih postupaka radi usporedbe njihove efikasnosti (vremena izvršavanja) na primjeru liste N slučajnih cjelobrojnih vrijednosti. Opis postupaka dan je u [Dov2011]. Njihove realizacije u Pythonu istovremeno su dobar primjer programiranja uz korištenje velikog dijela naredbi, podataka i izraza. Ovdje je dodan i standardni algoritam sortiranja u Pythonu – Timsort.

Sortiranje.py

```
# Usporedba postupaka sortiranja
Met = { 0 : ('list.sort()', 'Y.sort()' ),
        1 : ('RAZMJENA',    'S_Raz()'  ),
        2 : ('U VALOVIMA',  'S_Val()'  ),
        3 : ('UMETANJE',    'S_Ume()'  ),
        4 : ('UMETANJE 2',  'S_Ume2()' ),
        5 : ('SHELL-SORT',  'Shell()'  ),
        6 : ('QUICK-SORT',  'Quick()'  ),
        7 : ('MERGE_S',     'Merge_S()') }
from random import *
def S_Raz () :
  for i in range (N-1) :
    M = min (Y[i+1:]); j = Y.index(M)
    if M < Y[i] :  Y[i], Y[j] = Y[j], Y[i]
def S_Val () :
  B = -1
  while 1 :
    Raz = False; B += 1
    for i in range (N -B) :
      if Y[i] > Y[i+1] :
         Y[i], Y[i+1] = Y[i+1], Y[i]
         Raz = True
    if not Raz : break

def S_Ume () : # sortiranje umetanjem
  L = 0
  while L <= N :
    i = L
    while i > 0 and Y[i] < Y[i-1] :
      Y[i], Y[i-1] = Y[i-1], Y[i];  i -= 1
    L += 1

def S_Ume2 () : # sortiranje umetanjem (2)
  for i in range (1, N+1) :
    T = Y[i]; p = 1; r = i -1;
    while p <= r :
      m = (p+r) //2
      if T < Y[m] : r = m-1
      else        : p = m+1
    for j in range (i-1, p-1, -1) :
        Y[j+1] = Y[j];  Y[p] = T

def Shell () :
    Ink = N
    while Ink > 1 :
      Ink = Ink //3 +1
      for i in range (N -Ink +1) :
        j = i +Ink
        while j <= N :
          k = j -Ink
          if Y [j] < Y[k] :
             Y[k], Y[j] = Y [j], Y[k]
          j += Ink

def Quick () : # Quick-sort
  B = 0;  n = len(Y)
  L_stog = [0];  D_stog = [n -1];  p = 0
  while 1 :
    Lijevo = L_stog[p];  Desno = D_stog[p]
    p -= 1
    while 2 :
      k = Lijevo; j = Desno
      T = Y[(Lijevo +Desno) //2]
      while 3 :
        while Y[k] < T : k += 1
        while Y[j] > T : j -= 1
        if k <= j :
           Y[k], Y[j] = Y[j], Y[k]
           k += 1;  j -= 1
        if k > j : break
      if k < Desno :
        p += 1
        if p < len (L_stog) :
          L_stog[p] = k; D_stog[p] = Desno
        else                :
          L_stog += [k]; D_stog += [Desno]
        Desno = j
      if Lijevo >= Desno : break
    B += 1
    if p == -1 : break
def Merge_S () :
  global Y
  def S_Raz (L) :
    n = len (L)
```

```
    for j in range (n-1) :
      m = j
      for k in range (j+1, n) :
        if L[k] < L[m] : m = k
      if m != j : L[m], L[j] = L[j], L[m]
    return L

  def merge (A, B) :
    global k
    i = j = 0; Y = []
    while i <= len(A)-1 and j <= len(B)-1:
      Ai = A[i];  Bj = B[j]
      if   Ai < Bj : Y.append(Ai); i += 1
      elif Ai > Bj : Y.append(Bj); j += 1
      else         :
         Y.append(Ai); i += 1
         Y.append(Bj); j += 1
    while i <= len(A)-1 :
       Y.append(A[i]); i += 1
    while j <= len(B)-1 :
       Y.append(B[j]); j += 1
    return Y

  M = len(Y) //2;    A = S_Raz (Y[:M])
  B = S_Raz (Y[M:]); Y = merge (A, B)

from datetime import *

N = 10000
X = [randint (1, N) for i in range (N+1)]
print ('SORTIRANJE '+ str(N)
      +' cjelobrojnih vrijednosti \n')
print ('Postupak         sec')
Top = []
for S in range(len(Met)) :
  Y  = list (X)
  t1 = datetime.now()

  exec (Met[S][1])

  t2 = datetime.now()
  T  = t2 -t1
  dT = T.seconds +T.microseconds/10**6.0
  Top.append ([round (dT,5), Met[S][0]])
Top.sort()
for x in Top : print (x[1], t, x[0])
print ('\n')
```

```
>>>
SORTIRANJE 10000 cjelobrojnih vrijednosti

Postupak         sec
list.sort()  0.00096
QUICK-SORT   0.01595
RAZMJENA     0.98593
MERGE_S      1.06606
UMETANJE 2   2.20339
UMETANJE     4.95872
U VALOVIMA   6.57101
SHELL-SORT  10.81688
```

Vidimo da je standardna funkcija `sort()` najbrža, mnogo brža od čuvenog Quick-Sorta! To je Timsort o čemu se može naći informacija u [4].

PERMUTACIJE

Problem permutacija niza podataka dugo je bio predmetom algoritma s rješenjima u mnogim jezicima. Među njima je posebno zapažen algoritam kojeg je Dijkstra opisao u osmom i trinaestom poglavlju svoje knjige [Dij1976]. To je knjiga koja sadrži desetak poznatih algoritamskih problema prikazanih u posebnom jeziku koji je Dijkstra razvio. Prije nego što prijeđemo na rješenje problema, dajemo nekoliko definicija. Neka je:

$$p = (p_0, p_1, \ldots, p_{n-1})$$

permutacija od n, $n>1$, različitih vrijednosti p_i, $0 \le i < n$, i neka je

$$q = (q_0, q_1, \ldots, q_{n-1})$$

permutacija različita od p, ali koju čini isti skup vrijednosti kao u p. Izjavu: „permutacija p prethodi permutaciji q u alfabetskom uređenju“ permutacije p i q ispunjavaju onda i samo onda ako za minimalnu vrijednost k za koju je $p_k \ne q_k$ imamo $p_k < q_k$. Na primjer, za n=3 i skup vrijednosti 2, 4 i 7, imamo:

$$\text{indeks}_3\ (2,4,7) = 0$$
$$\text{indeks}_3\ (2,7,4) = 1$$
$$\text{indeks}_3\ (4,2,7) = 2$$
$$\text{indeks}_3\ (4,7,2) = 3$$
$$\text{indeks}_3\ (7,2,4) = 4$$
$$\text{indeks}_3\ (7,4,2) = 5$$

Permutacija (4,2,7) prethodi permutaciji (4,7,2), jer za $p_2 \ne q_2$ vrijedi $p_2 < q_2$. indeks_n permutacije n različitih elemenata je indeks_n alfabetski prve s istom krajnjom vrijednošću uvećan za indeks_{n-1} permutacije preostalih n-1 krajnjih desnih vrijednosti. Na primjer:

$$\begin{aligned}\text{indeks}_3(4,7,2) &= \text{indeks}_3(4,2,7) \\ &\quad + \text{indeks}_2(7,2) \\ &= 2 + 1 = 3\end{aligned}$$

Početna permutacija ima indeks jednak 0, a posljednja $n!$-1, gdje je n broj članova. Kompletno se rješenje može naći u spomenutoj Dijkstrinoj knjizi [Dij1976].

U knjizi [Dov2013] generiran je, uz pomoć predprocesora DDH, prijevod tog algoritma u Python, kojeg ovdje prilažemo bez uljepšavanja koda:

DDN_NPERM.py

```
# PROGRAM Nperm
from arr import *
C = array ('int', 1, [0, 1, 2, 3, 4,
                      5, 6, 7, 8, 9] )
N=eval (input(
       'Permutacija (1 do 3628799)? '))
print ( C.val)
def BEGIN_1 () :
  global C,N
  S=0
  Kfac=1
  k=1
  i=0
  j=1
  while True :
    if k != C.hib :
      Kfac=Kfac*(k+1)
      k=k+1
    else : break
  while True :
    if S != N :
      while True :
        if N < S+Kfac :
          Kfac=Kfac/k
          k=k-1
        else : break
      i=C.hib-k
      j=i+1
      while True :
        if S+Kfac <= N :
          S=S+Kfac
          C.swap(i,j)
          j=j+1
        else : break
    else : break
  print ( N)
  print ( C.val)
BEGIN_1 ()
```

```
>>>
Permutacija (1 do 3628799)? 3628799
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
3628799
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
Permutacija (1 do 3628799)? 5555
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
5555
[0, 1, 3, 2, 8, 5, 6, 9, 7, 4]
```

Naravno, današnji Python ima modul koji sadrži funkciju za izračunavanje *n*-te permutacije. To je modul **itertools**, a funkcija je **permutations**:

```
permutations (niz, r = None)
```

Vraća uzastopne permutacije duljine r elemenata niza. Ako je r izostavljeno, duljina je jednaka `len(niz)`.

```
>>> from itertools import permutations
>>> perm = permutations (range(10), 10)
>>> p = list (perm); p[-1]
(9, 8, 7, 6, 5, 4, 3, 2, 1, 0)
>>> p[5555]
(0, 1, 3, 2, 8, 5, 6, 9, 7, 4)
```

KOMBINACIJE

Modul **itertools**, sadrži i funkciju **combinations**. Ta funkcija iz liste L vraća n-torke iz liste koje sadrže sve moguće kombinacije elemenata liste duljine n, n≤len(L). Kombinacije su generirane u leksikografskom uređenju (kako je lista definirana). Elementi se tretiraju kao jedinstveni na temelju njihova položaja, a ne vrijednosti. Dakle, ako su ulazni elementi jedinstveni, u svakoj kombinaciji neće biti ponovljenih vrijednosti.

 # Kombinacije.py

```
from itertools import combinations
L = eval (input ('Lista? '))
# Sve kombinacije duljine n
for n in range (1, len(L)+1) :
  Komb = combinations(L, n)
  for i in list (Komb) : print (i)
  print ()
```

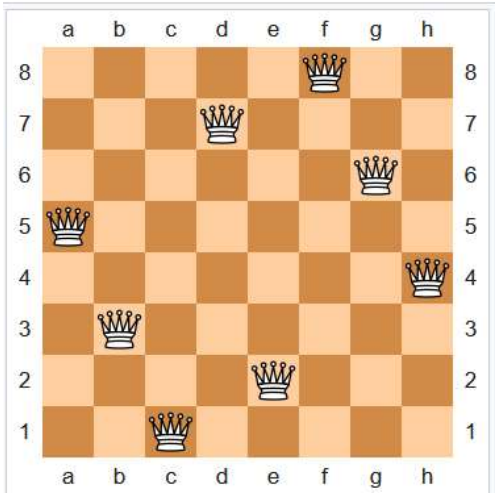
```
>>>
Lista? [1, 2, 3, 4]
(1,)
(2,)
(3,)
(4,)

(1, 2)
(1, 3)
(1, 4)
(2, 3)
(2, 4)
(3, 4)
(1, 2, 3)
(1, 2, 4)
(1, 3, 4)
(2, 3, 4)

(1, 2, 3, 4)
```

Ako želimo napraviti kombinaciju istog elementa s istim elementom, tada koristimo metodu:

```
combinations_with_replacement()
```

PROBLEM 8 KRALJICA

Problem osam kraljica odnosi se na postavljanje osam šahovskih kraljica na šahovnicu 8 × 8, tako da nigdje ni dvije kraljice ne prijete jedna drugoj. Prema tome, rješenje zahtijeva da dvije kraljice ne dijele isti red, stupac ili dijagonalu.



Bilo je nekoliko rješenja koje je predlagalo nekoliko znanstvenika računalnih znanosti početkom sedamdesetih godina prošlog stoljeća. Spomenimo opet Dijkstru koji je problem 8 kraljica iskoristio da bi ilustrirao snagu onoga što je nazvao strukturiranim programiranjem. Objavio je vrlo detaljan opis *backtracking* algoritma u rješavanju tog problema. Drugo rješenje je dao Niclaus Wirth, [Wir1976], u Pascalu.

Interesantno rješenje primjenom permutacija i kombinacija u Pythonu dao Jon Walsh u „Permutations and the N Queens Problem“, kojeg ćemo ovdje priložiti bez posebnih objašnjenja (dana su u spomenutom članku). Jedino ćemo napomenuti da je rješenje općenito i da vrijedi za ploču nxn, n>3.

Kraljice.py

```
from itertools import permutations, \
                      combinations
text = input ('Ploča n = ')
n    = int   (text)
x    = range (1, n+1)
def is_diagonal (point1, point2):
  x1 = point1[0];  y1 = point1[1]
  x2 = point2[0];  y2 = point2[1]
  gradient = (y2-y1)/(x2-x1)
  return abs (gradient) == 1
list_of_per = []
for per in permutations(range(1, n+1)):
  # print (per)
  y = per
  all_per = list (zip (x, y))
  list_of_per.append (all_per)
S = set()
for pos_sol in list_of_per:
  solutions = []
  for p1, p2 in combinations(
                  pos_sol, 2):
    solutions.append(
                is_diagonal(p1, p2))
  if True not in solutions:
    print (pos_sol)
    S |= { tuple (pos_sol) }
print (len(S))
```

```
>>>
Ploča n = 8
[(1, 1), (2, 5), (3, 8), (4, 6), (5, 3),
(6, 7), (7, 2), (8, 4)]
...
[(1, 8), (2, 4), (3, 1), (4, 3), (5, 6),
(6, 2), (7, 7), (8, 5)]
92
```

8 kraljica na šahovskoj ploči ima 92 rješenja! Jedno rješenje sadrži listu s 8 *n*-torki. Prvi broj u jednoj *n*-torki je red, a drugi stupac.

Formalni jezici

U ovom ćemo poglavlju, na primjeru izabranih tema teorije formalnih jezika: definiranja jezika, sintaksne analize i prevođenja jezika, pokazati da je programski jezik Python, sa svojom sintaksom i semantikom primitivnih i složenih naredbi te standardnim strukturama podataka (stringovima, *n*-torkama, listama i mapama), funkcijama i procedurama, pogodan za korištenje kao pseudojezik u opisu struktura i algoritama teorije formalnih jezika, [Dov2012a], [Dov2012b] i [Dov2013].

DEFINIRANJE JEZIKA

Skupovi stringova koji čine elemente nekog jezika nazivaju se *rečenice*, pa kažemo da je jezik skup rečenica. String konačne duljine koji se može promatrati kao jedinstvena cjelina naziva se *simbol* ili *riječ*. Skup svih simbola definiranih nad alfabetom *A*, označen s *V*, naziva se rječnik.

Postoji nekoliko metoda za specificiranje skupa koji čini jezik. Opisat ćemo ukratko dvije: generativni

sustav nazvan *gramatika* i *automat*. Svaka rečenica jezika može se izvesti koristeći pravila gramatike (nazvana „produkcije"). *Generator*i su automati koji mogu generirati rečenice jezika ali su češće u ulozi prepoznavača (u sintaksnoj analizi jezika), kako ćemo ih i ovdje prikazati.

Gramatike

Gramatika je četvorka, $G = (N,T,P,S)$, gdje su:

- N konačni skup *neterminalnih simbola*,
- T konačni skup *terminalnih simbola*, disjunktan skupu N,
- P konačan skup parova (α,β), gdje je:
  $\alpha = \alpha_1\gamma\alpha_2;\ \alpha_1,\ \alpha_2,\ \beta \in (N \cup T)^*,\ \gamma \in N$
  Element (α,β) iz P može se napisati kao $\alpha \rightarrow \beta$ i naziva se *produkcija*.
- S je poseban simbol iz N, $S \in N$, i naziva se *početni* (*startni*) *simbol*.

Ako u nekoj gramatici P sadrži sljedeće produkcije

$\alpha \rightarrow \beta_1 \ldots \alpha \rightarrow \beta_n$

skraćeno se piše

$\alpha \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n.$

Znak ' | ' se čita 'ili'. β_i su *alternative* za α.

Gramatika generira jezik rekurzivno. Ako je $G=(N,T,P,S)$ gramatika, njezina je *rečenična forma* definirana rekurzivno, kao što slijedi:

(1) S je rečenična forma (RF).

(2) Ako je $\alpha\delta\gamma$ RF, gdje je $\alpha,\gamma \in (N \cup T)^*$, i $\delta \rightarrow \beta$ je produkcija iz P, tada je $\alpha\beta\gamma$ također rečenična forma.

(3) Rečenična forma koja se ne može dalje izvoditi je *rečenica* jezika.

Ustroj gramatika u Pythonu

Gramatika $G=(N, T, P, S)$ može se izravno preslikati u četvorku u Pythonu u kojoj se elementi N i T mogu prikazati skupovima, element P strukturom mape (dict) i startni simbol S je znak iz liste N. Inicijalno ćemo gramatike definirati kao tekst pišući samo produkcije u obliku:

α -> β

Neterminali (nezavršni simboli) su velika slova, prazan string ε i oznaka produkcije je ->. Evo primjera produkcija gramatike jezika rimskih brojeva, aritmetičkih izraza, brojeva djeljivih s 3 i gramatike jezika s jednakim brojem znaka a, b, c i d, $\{a^n b^n c^n d^n, n>0\}$:

Gramatike.py

```
# Primjeri gramatika

Gs = ('Rim', 'Exp', 'n3', 'abcd')
# Rimski brojevi
Rim = """
R -> MA|CB|XD|I
M -> m|mm| mmm
A -> ε|CB| XD I
C -> c|cc|ccc|cd|d|dc|dcc|dccc|cm
B -> ε|XD|I
X -> x|xx|xxx|xl|l|lx|lxx|lxxx|xc
D -> ε|I
I -> i|ii|iii|iv|v|vi|vii|viii|ix """
# Izrazi
Exp = """
E -> T+E| T
T -> F*T| F
F -> a| b| c| (E) """
# Brojevi djeljivi s 3
n3 = """
A -> 1B| 2C| 3D| 4B| 5C| 6D| 7B| 8C| 9D
B -> 0B| 1C| 2D| 3B| 4C| 5D| 6B| 7C| 8D|
9B
C -> 0C| 1D| 2B| 3C| 4D| 5B| 6C| 7D| 8B|
9C
D -> ε| 0D| 1B| 2C| 3D| 4B| 5C| 6D| 7B|
8C| 9D """
# Jednak broj znakova a, b, c i d
abcd = """
S  -> aBCSd| abcd
Ba -> aB
Bb -> bb
Ca -> aC
Cb -> bC
Cc -> cc """
```

Procedura GRM() iz produkcija gramatike danih kao tekst (ili tekstualna datoteka) vraća definiciju gramatike G=(N,T,P,S) koja se može prikazati pozivom procedure za ispis, Write_GRM (G):

GRM.py

```
# Definiranje gramatika
from random    import *
from Gramatike import *
def GRM ( X, Name = 'G' ) :
  N = T = ''
  A = (X.replace (' ', '')).split('\n')
  Y = { 'name'  : Name,
        'start' : A[1][0],
        'α'     : []      }
  for a in A :
     if not a : continue
```

```
      [x, y] = a.split('->')
      y = tuple (y.split('|'))
      Y[x], Y['α'] = y, Y['α'] +[x]
      for C in y :
        for c in C : T += c *(not
                      c.isupper() and
                      not (c in T+'ε'))
      for c in x :
        N += c *(c.isupper()
             and not (c in N))
  N = list(N);  T = list(T)
  return list(N), list(T), Y, Y['start']

def Write_GRM ( G ) :
  N, T, P, Σ = G
  print ( P['name'], '= (N, T, P, Σ)' )
  print ( 'N = { ' +('%s, '*(len(N)-1)
       % tuple(N[:-1])) +N[-1] +' }' )
  print ( 'T = { ' +('%s, '*(len(T)-1)
        % tuple(T[:-1])) +T[-1] +' }' )
  print ( 'Σ =', P['start'] )
  print ( 'P :' )
  for x in P['α'] :
    print (x, '->', P[x][0], end =' ')
    for y in P[x][1:] :
      print ('|', y, end = ' ')
    print ()
  print ()
```

Na primjer, gramatika jezika rimskih brojeva je

```
>>> G=(N,T,P,Σ)=GRM(Rim); Write_GRM (G)
G = ( N, T, P, Σ )
N = { R, M, A, C, B, X, D, I }
T = { m, ε, c, d, x, l, i, v }
Σ = R

P :
R -> MA | CB | XD | I
M -> m | mm | mmm
A -> ε | CB | XD | I
C -> c | cc | ccc | cd | d | dc | dcc | dccc
| cm
B -> ε | XD | I
X -> x | xx | xxx | xl | l | lx | lxx | lxxx
| xc
D -> ε | I
I -> i | ii | iii | iv | v | vi | vii | viii
| ix
```

Procedura DER(P) generira rečenice jezika definiranog produkcijama P gramatike *G*. Ako je pozvana s DER(P,True) bit će prikazan niz rečeničnih formi (RF).

DER.py

```
from GRM import *

def DER ( P, DSP = False ) :
  RF = P['start'];  print ( RF, end=' ' )
  if not DSP : print ( '*=>', end=' ' )
  while True :
    for a in P['α'] :
      if len (RF) > 1000 :
        print ('RF ima više od 1000 '
               'znakova (rekurzija)')
        print ( RF );  return
      if a in RF :
        x = (''.join(sample (P[a], 1)))
        i = RF.find(a);  x *= (x != 'ε')
        RF = RF[:i] +x +RF[i+len(a):]
        if DSP : print ('=>', RF, end=' ')
        break
    else :
      print (RF if not DSP else '' )
      return RF
rf = input ('Prikaz izvođenja rečeničnih '
    'formi (d/n)? ').lower()[0] == 'd'
n  = int (input ('Koliko rečenica? '))
for g in Gs :
  G=(N,T,P,Σ) = GRM(eval(g)); Write_GRM(G)
  for _ in range (n) : DER ( P, rf )
  print ()
```

```
>>>
Prikaz izvođenja rečeničnih formi (d/n)? n
Koliko rečenica? 1

G = (N, T, P, Σ)
N = { R, M, A, C, B, X, D, I }
T = { m, c, d, x, l, i, v }
Σ = R
P :
R -> MA | CB | XD | I
M -> m | mm | mmm
A -> ε | CB | XDI
C -> c | cc | ccc | cd | d | dc | dcc |
dccc | cm
B -> ε | XD | I
X -> x | xx | xxx | xl | l | lx | lxx |
lxxx | xc
D -> ε | I
I -> i | ii | iii | iv | v | vi | vii |
viii | ix

R *=> mmm

G = (N, T, P, Σ)
N = { A, B, C, D }
T = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 }
Σ = A
P :
A -> 1B | 2C | 3D | 4B | 5C | 6D | 7B | 8C
| 9D
B -> 0B | 1C | 2D | 3B | 4C | 5D | 6B | 7C
| 8D | 9B
```

```
C -> 0C | 1D | 2B | 3C | 4D | 5B | 6C | 7D
| 8B | 9C
D -> ε | 0D | 1B | 2C | 3D | 4B | 5C | 6D
| 7B | 8C | 9D

A *=> 818683139422437939065612 43

G = (N, T, P, Σ)
N = { E, T, F }
T = { +, *, a, b, c, (, ) }
Σ = E
P :
E -> T+E | T
T -> F*T | F
F -> a | b | c | (E)

E *=> b

G = (N, T, P, Σ)
N = { S, B, C }
T = { a, d, b, c }
Σ = S
P :
S -> aBCSd | abcd
Ba -> aB
Bb -> bb
Ca -> aC
Cb -> bC
Cc -> cc

S *=> aabbccdd
```

SINTAKSNA ANALIZA

U praksi se često susrećemo s problemom da je poznata gramatika ili generator nekog jezika i zadan niz znakova, a postavlja se pitanje je li to rečenica jezika generiranog danom gramatikom ili generatorom. Takav se postupak naziva *sintaksna analiza*.

Ako je jezik definiran gramatikom, problem se svodi na nalaženje niza izvođenja (rečeničnih formi), počevši od *S*, koji bi rezultirao tim nizom (rečenicom). Takav se postupak sintaksne analize naziva *parsiranje*. Ustroj postupka parsiranja na računalu (program u izabranom jeziku za programiranje) naziva se *parser*.

Ako je jezik definiran automatom (generatorom), postavljamo pitanje: može li dani niz biti generiran danim generatorom. Tada je automat u ulozi prepoznavača jezika koji analizira ulazni niz i poslije konačno mnogo promjena svojih konfiguracija, krenuvši od početnog stanja, doseže konačno stanje ako je niz u jeziku i odgovara "da", ili se postupak prekida i odgovara "ne" ako ulazni niz nije u jeziku. Takav se postupak sintaksne analize naziva *prepoznavanje*, a automat koji to radi naziva se *prepoznavač*.

Automati najčešće imaju ulogu prepoznavača. Ovisno o jeziku koji se prepoznaje, odnosno o tipu gramatike koja generira takav jezik, postoje i vrste generatora. Opširnije u [Dov2012a] i [Dov2012b].

Pokažimo sintaksnu analizu beskonteksnih jezika primjenom stogovnog prepoznavača *P*. Najprije uvodimo definicije:

1) *Konfiguracija* stogovnog prepoznavača *P* jest (q, w, α) iz Q ×Σ*×Γ*, gdje su:

   q tekuće stanje
   w preostali dio ulaznog niza
   α niz znakova koji predstavlja sadržaj stoga; vrh je prvi znak niza,

2) *početna konfiguracija* je (q_0, w, Z_0),
3) *završna konfiguracija* je(q, ε, α), q∈F, α∈Γ*,
4) *pomak* stogovnog prepoznavača *P* jest relacija ⊢

$$(q, aw, Z\alpha) \vdash (q', w, \gamma\alpha)$$

ako δ(*q*, a, *Z*) sadrži (*q'*, γ) za q∈Q , a∈Σ∪{ε}, *w*∈Σ*, *Z*∈Γ. Kaže se da je ulazni niz *w* prihvatljiv s P ako

$$(q_0, w, Z_0) \vdash^* (q, \varepsilon, \alpha)$$

U Pythonu je struktura `dict` najpodesnija za implementaciju funkcije prijelaza jer predstavlja njezin preslik. Ako je *D* funkcija prijelaza prepoznavača, bilo kojeg tipa, općenito će njezini elementi imati strukturu:

```
D = { x0 : y0, x1 : y1, ..., xn : yn }
```

gdje je x_i n-torka domene, a y_i kodomena čija struktura ovisi o tipu automata. Na primjer, stogovni pretvarač jezika Exp generiranog gramatikom:

```
E -> E+E| E*E| (E)| a| b
```

napisan u Pythonu dan je u nastavku:

SP.py

```
# - STOGOVNI PREPOZNAVAČ
NL = '\n'
def Input_W (): # Učitavanje ulaznog niza
  return (input ('Upiši ulazni niz: ' )
                 ). replace (' ', '')

def Write_SP (Ime): # Ispis konfiguracije
  print (Ime)
  print (NL, 'SP = (Q,A,St, _1,D,s,F)',NL)
  print ('Q  =', Q, NL, ' A =', A, NL,
         'St =', St, NL, '_1 =', _1, NL,
         ' s =', s, NL, ' F =', F)
  print (NL, 'D:')
```

```
  S = list (D.keys());  S.sort()
  for d in S: print (' ', d, '=', D[d])
  print ()

def Write_C (y, C): print (y, C )

def SP (x):
  global Q, A, St, _1, D, s, F
  Ok = True; End = False
  q = s;  α = '$';  C = (q, x, α)
  Write_C ('', C)
  while len(x)>=0 and Ok and not End:
    X = ''; a = ''
    if len(x) > 0 : X = x[0]; x = x[1:]
    if len(α) > 0 : a = α[0]
    Ok = False; d = (q, X, a)
    if d in D :
      q, g = D[d]
      if g == '' and a != '': α = α[1:]
      if g != ''            : α = g +α[1:]
      Ok = True
    else : Ok = False
    if Ok:
      C = (q, x, α); Write_C ('   |--', C)
      if q in F and α == '' : End = True
      if End and x != ''    : Ok  = False

  return Ok and End

# Primjer
Exp = """
Q  = [0, 1]
A  = ['a', 'b', '+', '*', '(', ')']
St = ['$', '(']
_1 = '$';  s = 0;  F = [1]
D  = { (0, 'a', '$') : (1, '$'),
       (0, 'a', '(') : (1, '('),
       (0, 'b', '$') : (1, '$'),
       (0, 'b', '(') : (1, '('),
       (0, '(', '$') : (0, '($'),
       (0, '(', '(') : (0, '(('),
       (1, '+', '$') : (0, '$'),
       (1, '*', '$') : (0, '$'),
       (1, '+', '(') : (0, '('),
       (1, '*', '(') : (0, '('),
       (1, ')', '(') : (1, ''),
       (1,   '','$') : (1, '')   }
Ime = 'Exp'; DSP = (Q,A,St, _1,D,s,F) """

exec (Exp);  Write_SP (Ime)
w = Input_W(); print ()

while len(w) > 0:
  Ok = SP (w)
  if Ok: Write_C ('   |--', 'accept')
  else : Write_C ('   |--', 'error')
  print ()
  w = Input_W(); print ()
```

```
>>>
Upiši ulazni niz: a*(a+b)
 (0, 'a*(a+b)', '$')
   |-- (1, '*(a+b)', '$')
   |-- (0, '(a+b)', '$')
   |-- (0, 'a+b)', '($')
   |-- (1, '+b)', '($')
   |-- (0, 'b)', '($')
   |-- (1, ')', '($')
   |-- (1, '', '$')
   |-- (1, '', '')
   |-- accept
```

PREVOĐENJE

Ako je Σ ulazni alfabet, a Δ izlazni alfabet, prevođenje iz jezika $L_1 \subseteq \Sigma^*$ u jezik $L_2 \subseteq \Delta^*$ je relacija T iz $\Sigma^* \times \Delta^*$ tako da je L_1 domena, L_2 kodomena od T. Rečenica y, tako da je (x, y) u T naziva se izlaz za x.

Sintaksno-upravljano prevođenje

Jedan od formalizama za definiranje prevođenja jest shema sintaksno-upravljanog prevođenja. Intuitivno, shema sintaksno-upravljanog prevođenja jest gramatika u kojoj su elementi prevođenja pridruženi svakoj produkciji.

Kad god bi neka produkcija bila upotrijebljena u izvođenju ulazne rečenice, element prevođenja se koristi kao pomoć u izračunavanju dijela izlazne rečenice pridružene dijelu ulazne rečenice generirane tom produkcijom. Translacijska forma od T definira se na sljedeći način:

1) (S, S) je translacijska forma i prvi S pridružen je drugom S.
2) Ako je $(\alpha A\beta, \alpha' A\beta')$ translacijska forma i ako je $A \to \gamma, \gamma'$ pravilo u R, tada je $(\alpha\gamma\beta, \alpha'\gamma'\beta')$ translacijska forma. Neterminali iz γ i γ' egzaktno su udruženi kao što su udruženi i u pravilu. Neterminali iz α i β udruženi su s takvim neterminalima iz α' i β' u novoj translacijskoj formi egzaktno kao i u staroj.

Pišemo $(\alpha A\beta, \alpha' A\beta') \Rightarrow (\alpha\gamma\beta, \alpha'\gamma'\beta')$ što čitamo "izravno izvodi". Slično kao i kod izvođenja rečeničnih formi, $s \Rightarrow^*$ će biti označen niz od k izvođenja translacijskih formi, za $k \geq 0$, pa je prevođenje definirano s T, označeno s $\tau(T)$, skup parova:

$$\tau(T) = \{ (x,y) \mid (S, S) \Rightarrow^* (x, y), x \in \Sigma^*, y \in \Delta^* \}$$

Implementacija SDT u Pythonu dana je u proceduri SDT():

SDT.py

```
def SDT (X) :
  A = (X.replace (' ', '')).split('\n')
  Y = {'start' : A[1][0]}
  for a in A :
     if not a : continue
     b = a.split('->'); N = b[0]
     b = b[1].split(',');
     Y [N] = (tuple (b[0].split('|')),
              tuple (b[1].split('|')))
  return Y
```

gdje je X ulazno-izlazna gramatika jezika koji se prevode, s produkcijama oblika *A -> I, O* gdje su *I* alternative ulaznog, a *O* izlaznog jezika. Na primjer, za prevođenje rimskih u arapske brojeve može se definirati gramatika RA:

```
RA = """
R -> MA| CB| XD| I,  MA| CB| XD| I
M -> m| mm| mmm,      1| 2| 3
A -> ε| CB| XD| I,  000| CB| 0XD| 00I
C -> c| cc| ccc| cd| d| dc| dcc| dccc| cm,
     1|  2|   3|  4| 5|  6|   7|    8|  9
B -> ε| XD| I,       00| XD| 0I
X -> x| xx| xxx| xl| l| lx| lxx| lxxx| xc,
     1|  2|   3|  4| 5|  6|   7|    8| 9
D -> ε| I,            0| I
I -> i| ii| iii| iv| v| vi| vii| viii| ix,
     1|  2|   3|  4| 5|  6|   7|    8|  9
"""
```

Da bi se ulazni niz, rimski broj, pretvorio u arapski, neophodno je nekim od postupaka parsiranja izvesti stablo sintaksne analize (niz izvođenja) [Dov2013]. Ovdje je to izostavljeno, pa pokažimo na primjeru triju generiranih rečeničnih formi shemu prevođenja rimskih brojeva u arapske:

```
T = SDT (RA);  frm  = "(%s, %s)"
for i in range (3) :
  x = y = T['start']
  print (frm % (x, y), end = ' ')
  while not x.islower() :
    for s in x :
      if s.isupper() :
        a, b = T[s]
        z = ''.join(sample (a, 1))
        i = a.index(z); z = z *(z != 'ε')
        x = x.replace (s, z)
        a = b[i];  y = y.replace (s, a)
        print ('\t-->', frm % (x, y))
        break
  print ()
```

```
>>>
(R, R)  --> (XD, XD)
        --> (xD, 1D)
        --> (x, 10)
(R, R)  --> (MA, MA)
        --> (mmA, 2A)
        --> (mmI, 200I)
        --> (mmvi, 2006)
(R, R)  --> (CB, CB)
        --> (dB, 5B)
        --> (dXD, 5XD)
        --> (dxcD, 59D)
        --> (dxc, 590)
```

PREDPROCESORI

Predprocesori su vrsta prevodilaca koji programski jezik visoke razine prevode u neki drugi, ciljni jezik također visoke razine. Proces prevođenja sadrži sve faze kao i kod formalnih jezika samo što se ovdje umjesto znakova promatraju riječi.

Sredinom sedamdesetih godina prošlog stoljeća Dijkstra je zaključio da tada nije postojao odgovarajući jezik za prikaz algoritama i u svojoj je monografiji „A Discipline of Programming“, [Dij1976], definirao sintaksu i semantiku jednog svog jezika. Glavne odlike su mu:

- stroga definicija semantike,
- jednostavna sintaksa (mali broj naredbi),
- zadovoljenje svih poznatih principa strukturnog programiranja

S takvom definicijom taj je jezik bio „kao stvoren“ za analize i realizaciju u teoriji formalnih jezika i prevođenju. Prvi predprocesor, [Dov1982], realiziran je u FORTRANu, potom 1985. godine u Pascalu.

Ali, tek je 30 godina kasnije dobiveno mnogo bolje rješenje primjenom Pythona, jezik DDH. Pokazalo se da je i Python „kao stvoren“ za realizaciju prevodilaca, i kao ciljni jezik, [Dov2013]. Klasa `array`, dio je tog rješenja, prikazana u ovom poglavlju, kao i nekoliko prikazanih algoritama dobivenih prijevodom u Python dio su tog rješenja. Svi programi su napisani u Pythonu 2.7 pa mi se oni, koji se budu bavili proučavanjem predprocesora, mogu javiti na e-mail danom u predgovoru da im pošaljem novu verziju.

Kao primjer originalnog programa napisanog u jeziku DDH i njegovog prijevoda u Python dajemo algoritam Eratostenovog sita (podebljali smo rezervirane riječi):

```
01 PROGRAM Eratos;
02
03 BEGIN
04   PRIVAR P, Q, X, S, N, i;
05   P VIR int =   2;  Q VIR int = 0;
     N VIR int = 100;  X VIR int = 0;
06   i VIR int =   0;
07   S VIR int array = (1);
08   DO[i <> N] i = i+1; S: hiext(i) OD;
09   BEGIN
10     GLOVAR P, Q, X, S; GLOCON N;
       PRIVAR m, r;
11     m VIR int = 0;  r VIR int = 0;
12     DO [ P*P <= N ] Q = P;
13       DO [ P*Q <= N ] X = P*Q;
14         DO [ X <= N ] S : (X) = 0;
                         X = P*X OD;
15         m = Q+1;
16         DO [ S(m) == 0 ] m = m+1 OD;
17         Q = S(m)
18         OD;
19       r = P+1;
20       DO [ S(r) == 0 ] r = r+1 OD;
21       P = S(r)
22       OD
23     END;
24   i = 2;
25   DO [ i <= N ]
26     IF [ S(i) <> 0 ] Write (S(i))
27       ![ S(i) == 0 ] SKIP
28       FI;
29     i = i+1
30     OD
31   END.
```

Generirani prijevod u Pythonu:

```
# -*- coding: cp1250 -*-
# PROGRAM Eratos
from arr import *
P=2
Q=0
N=100
X=0
i=0
S=array ('int', (1, ))
while True :
  if i != N :
    i=i+1
    S.hiext (i)
  else : break
def BEGIN_1 () :
  global P,Q,X,S
  global N
  m=0
  r=0
  while True :
    if P*P <= N :
      Q=P
      while True :
        if P*Q <= N :
          X=P*Q
          while True :
            if X <= N :
              S.alt (X, 0)
              X=P*X
            else : break
          m=Q+1
          while True :
            if S.ind (m) == 0 :
              m=m+1
            else : break
          Q=S.ind (m)
        else : break
      r=P+1
      while True :
        if S.ind (r) == 0 :
          r=r+1
        else : break
      P=S.ind (r)
    else : break
BEGIN_1 ()
i=2
while True :
  if i <= N :
    if S.ind (i) != 0 :
      print ( S.ind (i),end = ' ')
    elif S.ind (i) == 0 :
      pass
    i=i+1
  else : break
```

```
Program je upamćen pod imenom DDH_ERATOS.py
```

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53
59 61 67 71 73 79 83 89 97
```

Obrada prirodnih jezika

Danas je i sve veća uporaba teorije formalnih jezika u obradi prirodnih jezika koje ima dodirne točke s umjetnom inteligencijom. Veliki doprinos tome je i uporaba Pythona.

MODUL `nltk`

Za obradu prirodnih jezika dosta je popularan modul `nltk`. Možete ga instalirati sa:

```
pip install NLTK
```

Taj je modul kup alata za analizu tekstova prirodnog jezika. Opis metoda dan je na stranici:

https://www.nltk.org/

No, nedostatak tih programa je što rade na granici beskontekstnih jezika, što znači da se ne bave semantikom prirodnih jezika.

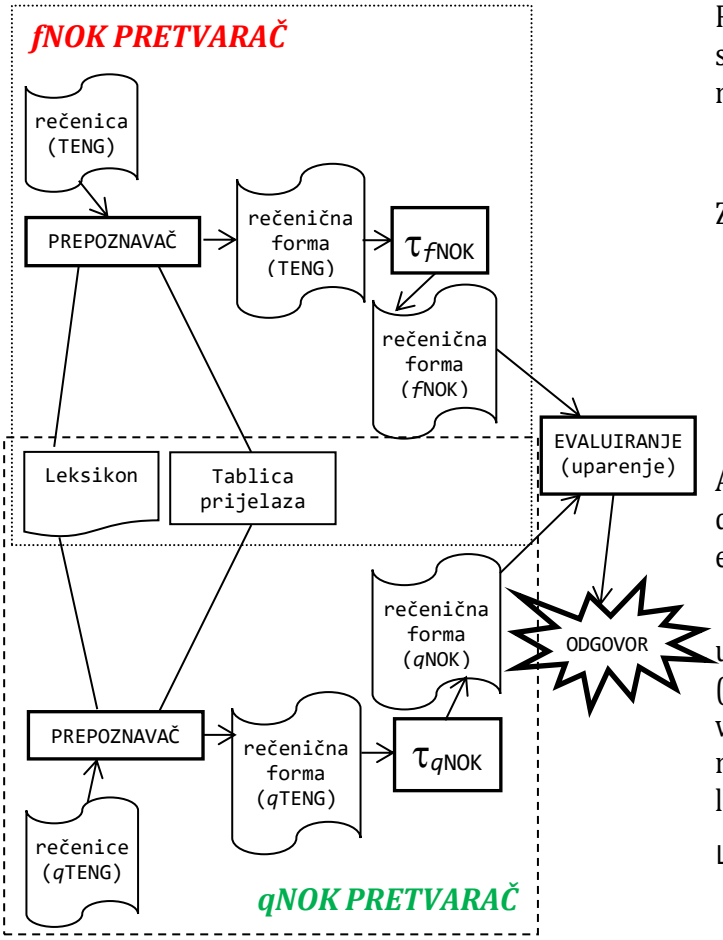
Sve je više obrade prirodnih jezika, u umjetnoj inteligenciji, kao što je sustav QANOK, u strojnom učenju, ili primjene u nekim teorijama, kao što je na primjer teorija baza podataka. Prikazat ćemo sustav QANOK i jedan doktorski rad iz teorije baza podataka.

SUSTAV QANOK

U radovima [Jak2014] i [Pav2014] razvijen je sustav QANOK, realiziran u Pythonu, u kojem je dan prijedlog rješenja problema pretraživanja teksta s jednostavnim rečenicama engleskog jezika (TENG), postavljanjem jednostavnih pitanja (*q*TENG). Sustav QANOK spada u područje „question answering" sistema (skraćeno QA). Sastoji se od dva pretvarača:

1) *f*NOK pretvarača koji ulaznu rečenicu napisanu u jeziku TENG prevodi u rečenicu u jeziku *f*NOK.
2) *q*NOK pretvarača koji ulaznu rečenicu (pitanje), napisanu u jeziku *q*TENG prevodi u rečenicu u jeziku *q*NOK.

i evaluatora koji na temelju prevedenog teksta i pitanja u *f*NOK i *q*NOK notaciju i njihovog uparivanja pronalazi odgovor. Ovdje nam ograničeni prostor ne dopušta prikaz modula i glavnog programa sustava QANOK. Model sustava predočen je sljedećom slikom:



Aplikaciju QANOK.py možete dobiti zahtjevom na e-mail dan u predgovoru. Slijedi djelomični opis i prikaz nekih rezultata.

Iz datoteke ulaznih rečenica, TENG.txt, izdvojili smo:

Girls are on the beach.
Julia writes a letter to a friend.
Tom drove Mery's red car on Monday.
I talk about the solution with my colleague.
Several girls sing.
Tom has two cars.
A boy gives an apology to the girl.
The student reads a wonderful book.
This girl sings.
Tom swims.
Julia swims fast on the pool.
Tom's brother swims.
Student's pencil writes.

Rečenice u jeziku TENG definirane su na ograničenom broju riječi iz skupa:

N Noun	*imenica*	**V** Verb	*glagol*
A Adjective	*pridjev*	**R** Adverb	*prilog*
S Adposition	*prijedlog*	**M** Numeral	*broj*

Riječima su pridruženi atributi, [i]. Atributi uvode semantička svojstva pojedinih riječi. Na primjer, imenice imaju 15 atributa:

N (**p** | [**cpx**][**mfn**][**0sp**][**01ma**][**cv**])

Značenje atributa je:

c	opća	**p**	vlastita	**x**	zbirna
m	muški rod	**f**	ženski rod	**n**	srednji rod
0	nebrojiva	**s**	jednina	**p**	množina
0	neživo	**1**	živo	**m**	gradivna
a	apstraktno	**c**	suglasnik	**v**	samoglasnik

Atributi označeni crvenom bojom su dodani definiciji danoj u [Erj2010]. Planirani su za buduće obrade engleskog jezika.

Leksikon sustava QANOK implementiran je mapom u kojoj je ključ riječ, a sadržaj *n*-torka: osnovni oblik (nominativ za imenice ili infinitiv za glagole), oznakom vrste riječi i atributima, oznakom upitnih riječi i njihovim kodom. Evo nekoliko izdvojenih riječi iz leksikona:

```
Lx = {
  'a'       : ('a',     'Di',      '',  0),
  'about'   : ('about', 'Sp',    'QSp', 1),
```

```
'an'      : ('a',      'Di',        '', 0),
'apology' : ('apology','Ncns',   'QNc', 3),
'are'     : ('be',     'Vmip-p', 'QVm',17),
'drive'   : ('drive',  'Vmip-p', 'QVm',17),
'drives'  : ('drive',  'Vmip3s', 'QVm',17),
'drove'   : ('drive',  'Vmis',   'QVm',17),
'fast'    : ('fast',   'Af',     'QA5', 9),
'girl'    : ('girl',   'Ncns',   'QN1', 2),
'girls'   : ('girl',   'Ncnp',   'QN1', 2),
'is'      : ('be',     'Vmip3s', 'QVm',17),
'Julia'   : ('Julia',  'Npfs',   'QNp', 2),
'sing'    : ('sing',   'Vmip-p', 'QVm',17),
'sings'   : ('sing',   'Vmip3s', 'QVm',17),
'two'     : ('two',    'Mc',   'QSpec',14),
'writes'  : ('write',  'Vmip3s', 'QVm',17),
'write'   : ('write',  'Vmip-p', 'QVm',17),
```

Prvo FNOK pretvarač („transducer“) učitava rečenice napisane na engleskom jeziku TENG, analizira ih i ako su sintaktički korektne, prevodi ih rečeničnu formu (RF) TENG, potom *ƒ*NOK jezika, [Pav2015],[Jak2014].

Sintaksna je analiza realizirana prepoznavačem, [Dov2012b] i [Dov2013], a prevođenje pretvaračem u kojem je tablica prijelaza dana sa (iz dva dijela):

		Af	*Dd*	*Di*	*Dg*	*Ds*	*Mc*	*Nc*	*Np*	*Pg*	*Pp*
Net	*q*	1	2	3	4	5	6	7	8	9	10
Np	0	4	1	1	9	9		11	11	11	11
N'	1	4						11	11	11	11
	2	4									
	3	4									
	4	4						5	5		
	5						6	6	6		
	6							5^{17}			
	7		8	8	9	9	6	11	11	11	
	8	10	1	1	1	1	6	11			11
	9	4						11	11	11	11
	10							11			
VP	11										
NP'	12	16	13	13	22	22	6	20	20	20	
N'	13	16						12	20		20
	14	16									
	15	16									
	16	16						17	17		
	17						18	18	18		
	18							17			
	19		21	21	22	22	18	20	20	20	
	20	16	21	21	22	22					20
	21	23					18	20			
	22	16						20			
	23							20			

		Px	*Rm*	*Rs*	*Sp*	*Vm*	*Nc* 's	*Np* 's	*Pg* 's	.
Net	*q*	11	12	13	14	15	16	17	18	19
Np	0		3	2			9	9	9	
N'	1		3	2						
	2		3							
	3									
	4									
	5		3	2						
	6		6		7	12				0
	7				7	12				
	8				7					
	9									
	10		3	2						
VP	11					12				
NP'	12					12^{3}				
N'	13	20	15	14	19		16	16	16	0
	14		15	14						
	15		15							
	16									0
	17		15	14	19					0
	18		18		19					0
	19				19					0
	20		18		19					0
	21		15		19					0
	22									
	23		15	14						

Evo primjera prevođenja rečenica iz datoteke TENG.txt:

```
Girls are on the beach          ulazna rečenica
Nc Vm Sp Dd Nc                  RF TENG
Vm(QN1 Nc, QSp Sp QNc Dd Nc)    RF fNOK
are("who?" girls, "where?" on "what?" the
beach)                          ƒNOK

Julia writes a letter to a friend
Np Vm Di Nc Sp Di Nc
Vm(QNp Np, QNc Di Nc QSp Sp QNc Di Nc)
writes("who?" Julia, "what?" a letter "where?"
to "what?" a friend)

Tom drove Mery's red car on Monday
Np Vm Np's Af Nc Sp Np
Vm(QNp Np, QNc Nc(QDs Np's, QA4 Af) QS1 Sp QNp
Np)
drove("who?" Tom, "what?" car("whose?" Mery's,
"which?" red) "when?" on "who?" Monday)

I talk about the solution with my colleague
Pp Vm Sp Dd Nc Sp Ds Nc
Vm(QPp Pp, QSp Sp QNc Dd Nc QSp Sp QDs Ds Nc)
talk("who?" I, "where?" about "what?" the
solution "where?" with "whose?" my colleague)
```

```
Several girls sing
Af Nc Vm
Vm(QN1 Nc(QA6 Af))
sing("who?" girls("how_many?" several))

Tom has two cars
Np Vm Mc Nc
Vm(QNp Np, QSpec Mc QNc Nc)
has("who?" Tom, "how_many?" two "what?" cars)

A boy gives an apology to the girl
Di Nc Vm Di Nc Sp Dd Nc
Vm(QN1 Di Nc, QNc Di Nc QSp Sp QN1 Dd Nc)
gives("who?" a boy, "what?" an apology "where?"
to "who?" the girl)

The student reads a wonderful book
Dd Nc Vm Di Af Nc
Vm(QNp Dd Nc, QNc Di Nc(QA3 Af))
reads("who?" the student, "what?" a
book("what?" wonderful))

This girl sings
Dg Nc Vm
Vm(QN1 Nc(QDg Dg))
sings("who?" girl("which?" this))

Tom swims
Np Vm
Vm(QNp Np)
swims("who?" Tom)

Julia swims fast on the pool
Np Vm Af Sp Dd Nc
Vm(QNp Np, QA5 Af QSp Sp QNc Dd Nc)
swims("who?" Julia, "how?" fast "where?" on
"what?" the pool)

Tom's brother swims
Np's Nc Vm
Vm(QN1 Nc(QDs Np's))
swims("who?" brother("whose?" Tom's))

Student's pencil writes
Nc's Nc Vm
Vm(QNc Nc(QDs Nc's))
writes("what?" pencil("whose?" student's))
```

Tablici prijelaza *f*NOK rečenica dodan je dio za postavljanje pitanja s oznakama riječi:

		Qw	Q0	Q2	Q3	Qh	Qm	Qk	?	Wt	Qf
Net	q	20	21	22	23	24	25	26	27	28	30
➔	0	51				63					96

Prijelazi i kodovi akcija dani su u modulu koji sadrži mapu TP, od stanja 51 do 119.

TP.py

```
TP = { # TABLICA PRIJELAZA
( 0, 1):  4, (0, 2):  1, (0, 3):  1, (0, 4): 9,
( 0, 5):  9, (0, 7): 11, (0, 8): 11, (0, 9):11,
( 0,10): 11, (0,12):  3, (0,13):  2, (0,14):67,
( 0,15): 94, (0,16):  9, (0,17):  9, (0,18): 9,
( 0,20): 51, (0,24): 63, (0,30): 96,
# ... dalje v. tablicu fNOK
(23, 7): 20, (23, 12): 15, (23,13): 14,
# tablica prijelaza pitanja
(51, 1): 115, (51, 7): 58, (51, 8): 58,
(51,15):  52, (51,21): 53, (51,22): 78,
(51,23):  55, (52, 1): 75, (52, 2): 60,
(52, 3):  60, (52, 5): 57, (52, 7): 66,
(52, 8):  66, (52,10): 54, (52,11): 66,
(52,12):  75, (52,13): 65, (52,14): 70,
(52,17):  61, (52,27):  0, (53, 1): 87,
(53,10):  54, (53,14): 70, (53,15): 66,
(53,21):  52, (54,12): 75, (54,15): 59,
(54,21):  52, (54,27):  0, (55, 1): 92,
(55, 2):  56, (55, 3): 56, (55, 4): 57,
(55, 5):  57, (55, 8): 54, (55, 9): 54,
(55,10):  54, (55,16): 56, (55,17): 56,
(55,18):  56, (56, 1): 57, (56, 7): 53,
(56,13):  65, (57, 1): 92, (57, 7): 53,
(57,12):  75, (57,21): 66, (58,15): 59,
(59, 2):  60, (59, 3): 60, (59, 6): 104,
(59, 8):  66, (59,27): 52,
(60, 1):  62, (60, 7): 52, (61, 1): 62,
(61, 7):  52, (62, 7):  52, (63, 1): 64,
(63,23):  55, (63,25): 107, (64,15): 59,
(65, 1):  92, (65,12):  75, (66,27):  0,
(67,20):  68, (68, 7): 110, (68,10): 69,
(68,23):  55, (69,15):  66,

(70, 2):  72, (70, 3):  72, (70, 5): 71,
(71, 7):  66, (72, 7):  73, (73, 1): 66,
(73,12):  66, (73,14):  74, (73,21): 66,
(73,27):   0, (74, 5):  71, (75, 1): 92,
(75,14):  76, (75,27):   0, (76, 2): 77,
(76, 3):  77, (76,14):  70, (77, 7): 102,
(78, 1): 100, (78, 2): 100, (78, 3): 100,
(78, 8):  79, (79,15):  80, (79,21): 83,

(80, 4):  81, (80, 5):  81, (80,16): 81,
(80,17):  81, (81, 1):  82, (82, 7): 85,
(83,14):  84, (84, 1):  82, (84, 2): 86,
(84, 3):  86, (84, 4):  81, (84, 5): 81,
(84,16):  81, (84,17):  81, (85,14): 86,
(85,27):   0, (86, 7):  66, (86, 8): 66,
(87, 7):  88, (88,21):  66, (88,14): 89,
(89,14):  90,

(90, 2):  91, (90, 3):  91, (91, 1): 92,
(92, 7):  93, (92,27):   0, (93,12): 75,
(93,14):  86, (93,15):  66, (93,21): 103,
(94, 2):  97, (94, 3):  97, (94, 7): 95,
(95,14):  76, (96,20):  51, (97, 7): 98,
(98,13):  99, (99, 1):  66, (98, 1): 66,

(100,1): 101, (100, 7):102, (101,7): 102,
(102,21):103, (102,27):  0, (103,6): 104,
(103,14): 86, (103,27):  0, (104,7): 105,
```

```
(105,14):106, (106,10): 66, (106,14): 70,
(107, 7):108, (108,15):109, (109, 8): 66,

(110,10):111, (111,15):112, (112,14): 113,
(113, 2):114, (114, 7): 66, (115, 7): 116,
(116,22):117, (117, 8):118, (117,10): 118,
(118,15):119, (119,14): 86                }
```

Poslije prevođenja u *q*NOK notaciju i postupak evaluiranja, odgovora se na postavljena pitanja. Pitanja su:

Are girls on the beach?
How does Julia swim?
What did Tom do with Mery's red car on Monday?
What do I do with my book?
What do several girls do?
What does Julia do for a friend?
What does Julia do on the pool?
How many cars has Tom?
What does Tom do?
What does Tom's brother do?
What does a boy do to the girl?
What does a boy do?
What does student's pencil do?
What does this girl do?
When did Tom drove Mery's red car?
Where are girls?
Who swim?

Evo odgovora:

```
Are girls on the beach?
  YES  (Girls are on the beach)

How does Julia swim?
  fast on the pool (Julia swims fast on the
                    pool)

What did Tom do with Mery's red car on Monday?
  drove   (Tom drove Mery's red car on Monday)

What do I do with my book?  I DO NOT KNOW

What do several girls do?
  sing   (Several girls sing)

What does Julia do for a friend?
  GREŠKA  52 30 for 52
  *** Syntax error!

What does Julia do on the pool?
  swims   (Julia swims fast on the pool)

How many cars has Tom?
  two   (Tom has two cars)

What does Tom do?
  drove   (Tom drove Mery's red car on Monday)
  has     (Tom has two cars)
  swims   (Tom swims)

What does Tom's brother do?
  swims   (Tom's brother swims)

What does a boy do to the girl?
  gives an apology  (A boy gives an apology to
                     the girl)

What does the student's pencil do?
  GREŠKA  56 16 student's 56
  *** Syntax error!

What does this girl do?
  sings   (This girl sings)

When did Tom drove Mery's red car?
  on Monday   (Tom drove Mery's red car on
               Monday)
Where are girls?
  on the beach   (Girls are on the beach)

Who sing?
  several girls   (Several girls sing)
  this girl   (This girl sings)

Who swim?
  Tom   (Tom swims)
  Julia fast on the pool   (Julia swims fast on
                            the pool)
  Tom's brother   (Tom's brother swims)
```

TEORIJA BAZA PODATAKA

Python sve više nalazi mjesto i u teoriji baza podataka. Zapažena je primjena Pythona u disertaciji Sabrine Šuman *„Sustav za prevođenje poslovnih opisa u model podataka entiteta i veza“*, [Šum2019]. Istodobno je to doprinos rješavanju problema obrade teksta i umjetnoj inteligenciji.

Polazeći od proširenog leksikona skupa engleskih rečenica za opis poslovanja kojima je dodano nekoliko atributa u radu se, uz leksičku, sintaksnu i semantičku analizu, generiraju rečenice u jeziku u kojem su ekstrahirani entiteti, atributi, kardinalnosti i relacije. Sve se generirane rečenice pamte i u Excelu. Na primjer, ulazna rečenica:

A bookstore sells books and software to students.

poslije leksičke, sintaksne i semantičke analize prevedena je u četiri rečenice entiteta i veza:

```
L_En        _R      R_C   R_En
bookstore   sells   M     books

L_En    L_C   _R        R_C   R_En
Books   M     sold to   M     students

L_En        _R      R_C   R_En
bookstore   sells   M     software

L_En      L_C  _R        R_C  R_En
Software  M    sold to   M    students
```

Proširimo granice

Na kraju ovog završnog poglavlja dajemo nekoliko nestandardnih modula Pythona koji vam mogu korisno poslužiti u mnogim širim područjima programiranja. Na internetu ćete naći beskrajan niz paketa, modula, pa čak i zasebnih pythonskih distribucija koje su razvijene da podrže specijalna područja od interesa kao što su znanost i obrada jezika. Veliki je doprinos mnogih „pythonovaca“ koji nesebično prikazuju svoja rješenja i dijele ih širom svijeta. Možda ćete i vi biti među njima, doprinijeti toj zajednici i pomoći da Python bude još bolji!

GUI

Na stranici:

https://wiki.python.org/moin/GuiProgramming

dan je veliki broj GUI platforma koje podržavaju Python. Ipak, preporučujemo Qt, odnosno **PyQt5**. Na stranici:

https://build-system.fman.io/pyqt5-tutorial

dane su upute za intaliranje modula PyQt5, kao i kratki opis desetak njegovih komponenti. PyQt5 ima i svoj dizajner koji se može skinuti na:

https://build-system.fman.io/qt-designer-download

Poslije dizajniranja postoji poseban program za generiranje koda s komponentama. Nedostatak je što se izmjene u kodu ne mogu prenijeti u dizajner. Drugi nedostatak je što se generira kôd s velikim brojem linija. Na primjer, autor je ove knjige u jednoj aplikaciji 1750 generiranih linija sveo, uvođenjem svojih procedura, na 350. Dobiveni kôd je bio pregle-niji i lakši za „ručne“ izmjene.

Druga GUI platforma je **wxPython**. Može se instalirati sa:

```
pip install wxPython
```

Uvozi se sa `>>> import wx`. Upute se mogu naći na:

https://www.tutorialspoint.com/wxpython/index.htm

wxPython ima i svoj dizajner, **wxFormBuilder**. Može se potražiti na internetu i instalirati.

GRAFIKA

U standardnim Pythonovim modulima grafiku smo mogli rabiti u `Canvas` komponenti tkintera i u kornjačinoj grafici.

Postoji jedan nestandarni modul pod imenom **matplotlib**, sveobuhvatna biblioteka za stvaranje statičkih, animiranih i interaktivnih vizualizacija u Pythonu. Modul možemo dodati na uobičajeni način:

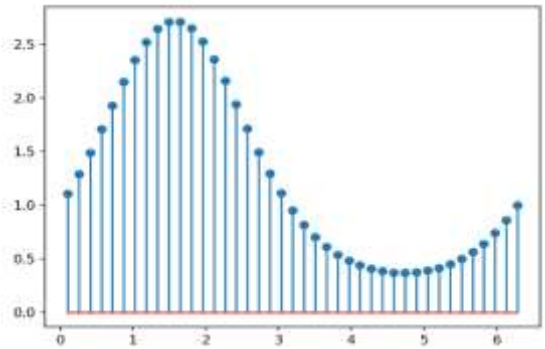
```
pip installing -U matplotlib
```

Na stranici:

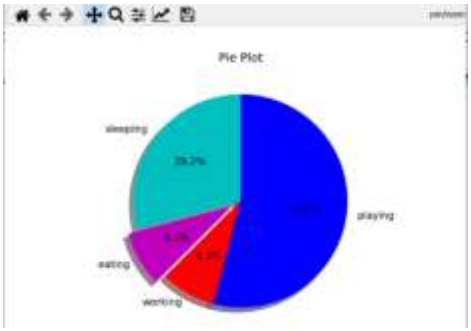
https://matplotlib.org/stable/index.html

trenutno je u uporabi inačica 3.4. To je moćan program čija dokumentacija obuhvaća oko 3500 stranica u PDF-u! No, ne treba nas to obeshrabriti. Na istoj ćemo stranici naći dosta primjera, programa, koji prikazuju primjene *matplotliba* u mnogim područjima, u 2D i 3D grafici koje možemo kopirati i koristiti. Za dosta primjena je dovoljno koristiti modul **pyplot**. Pogledajmo dva primjera koje smo izabrali iz te velike ponude:

```
# Stem_plot.py
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0.1, 2 * np.pi, 41)
y = np.exp(np.sin(x))
plt.stem(x, y);  plt.show()
```




```
# pyplot1.py
import matplotlib.pyplot as plt
days       = [1,2,3,4,5]
sleeping   = [7, 8, 6, 11, 7]
eating     = [2, 3, 4, 3, 2]
working    = [7, 8, 7, 2, 2]
playing    = [8, 5, 7, 8, 13]
slices     = [7, 2, 2, 13]
activities = ['sleeping','eating',
              'working','playing']
cols       = ['c','m','r','b']
plt.pie(slices,
  labels = activities, colors = cols,
  startangle = 90, shadow = True,
  explode = (0,0.1,0,0),
  autopct = '%1.1f%%')
plt.title ('Pie Plot');  plt.show()
```

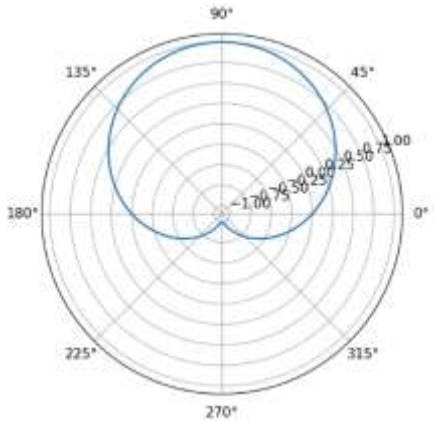


Grafovi u klasičnom koordinatnom sustavu nisu jedino što Pythom *Matplotlib* može ostvariti. Dosta česti alternativni prikaz podataka je pomoću polarnih grafova. Koristeći ovaj library njima se izrazito lako pristupa. Podaci se definiraju na potpuno jednak način kao za grafove u klasičnim koordinatnim sustavima. Polarni graf kreiramo naredbom *axes()*.

```
pyplot.axes (polar = True)
```

Parametar `polar` je inicijalno jednak `False` i ako ga se promijeni dobiva se polarni graf. Na primjer, sljedeći program crta sinusoidu na intervalu $[0, 2\pi]$ u pravokutnom koordinatnom sustavu:

```
# Polarni_graf.py
import numpy
import matplotlib.pyplot as pyplot
pyplot.axes (polar = True)
x = numpy.linspace (0, 2*numpy.pi, 101)
y = numpy.sin(x)

pyplot.plot (x, y);  pyplot.show ()
```



Rezultat je iznenađujući, ponajviše jer većina nije naviknuta na ovakvu prezentaciju podataka.

BAZE PODATAKA

Pythonove strukture podataka pružaju mogućnosti za obradu podataka. Ali, od verzije 2.5 postoji standardni modul **`sqlite3`** koji predstavlja implementaciju DB-API 2.0 koji koristi bazu SQLite 3.x. To je dobar izbor, ali nedostatak je što ne radi u mreži. Python podržava sve poznate baze podataka, kao što su **MySQL**, **PostgreSQL** i **Oracle**. Za instaliranje modula za rad u Oraclu napišite: `pip install cx_Oracle`

Za instaliranje baze PostgreSQL, koju preporučujemo, informacije se mogu naći na:

https://www.postgresqltutorial.com/postgresql-python/connect/

VEZA S DRUGIM DATOTEKAMA

Python može generirati i uvoziti dokumente iz drugih formata, kao što su XML, Word, PDF, Excel itd. Više o tome može se naći na internetu. Na primjer, za rad s Excelom može se uvesti modul **`xlrd`** koji ima metode za čitanje i pisanje Excel datoteka:

```
pip install xlrd
```

PROGRAMIRANJE IGRICA

Programi igrica dani u prethodnim i ovom poglavlju napisani su uz pomoć standardnih modula. Za naprednije stvaranje Python postoji **`pygame`** nestandarni modul kojeg možete instalirati s:

```
pip install pygame
```

Ovaj modul uključuje snažnu podršku za multimedijsko programiranje, sadrži grafiku i zvuk, kao i interakciju s tipkovnicom, miševima, džojstikom i ostalim perifernim uređajima. Na internetu se nalazi dovoljan broj uputa za rad u tom modulu i veliki broj primjera.

PYTHON U ZNANOSTI

Prvo moramo napomenuti da modul random ima funkcije koje su korisne u znanstvenim istraživanjima. To su funkcije koje se odnose na statistiku.

U sljedećoj tablici dani su nestandardni moduli koji sadrže veliki broj funkcija i metoda za primjene u matematici i znanosti. Sve ih se može dodati vašem Pythonu sa:

```
pip install Modul.
```

Modul	*Opis*
`scipy`	Skup alata za obavljanje znanstvene i numeričke analize u Pythonu.

scikit	Zbirka znanstvenih i numeričkih modula za obradu koja se općenito oslanja na SciPy, ali nije dio formalnog SciPy paketa.
numpy	Paket unutar SciPy-a koji pruža napredne alate za numeričku obradu.
sympy	Paket unutar SciPy-a za simboličku matematiku.

Matplotlib, ima sve veću primjenu kao potpora za znanstvena istraživanja u mnogim prirodnim znanostima.

Na primjer, na stranicama od 212 do 217 knjige prof. dr. sc. Slobodana Jankovića, [Jan1998], dan je matematički model gibanja prodora ose projektila kroz ravninu okomitoj na brzinu leta koje je sadržano u kompleksnom broju ξ.

Realni dio tog kompleksnog broja β predstavlja kut između brzine leta i projekcije osi projektila na krovnu ravan kroz brzinu leta, a imaginarni dio α predstavlja kut između osi projektila i njezine projekcije na krovnu ravan kroz brzinu leta.

$$\tilde{\xi}_h = K_1 e^{i\phi_1} + K_2 e^{i\phi_2}$$

gdje su (za $j = 1, 2$)

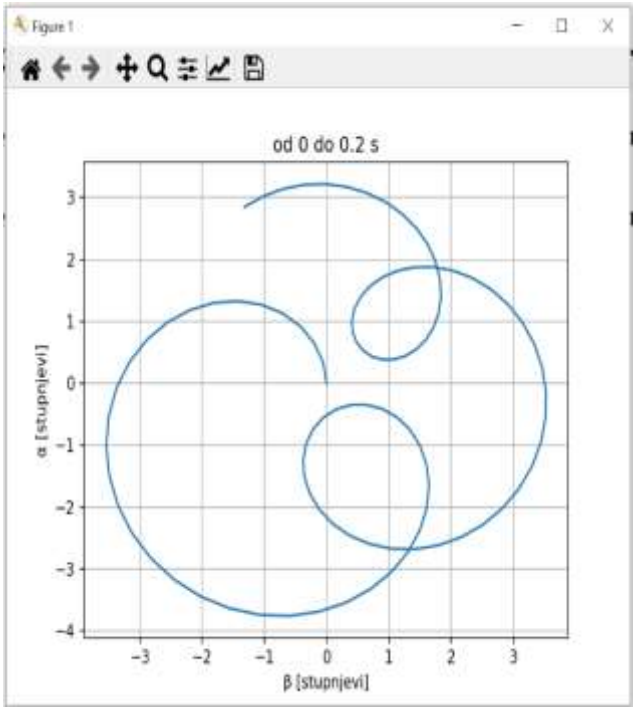
$$K_j = K_{j0} e^{\lambda_j \bar{s}}$$

$$\phi_j = \phi_{j0} + \phi'_j \bar{s}$$

To je idealan primjer za prikaz rada s kompleksnim vrijednostima i primjenu matplotliba. Rješenje za zadane parametre, s izborom imena varijabli koje su usklađene s formulama, dano je u sljedećem programu:

```
# Projektil.py
import matplotlib.pyplot as plt
from   cmath import (exp as e,
                     pi  as π)
# α β λ ξ π φ
λ1 = -0.000559; λ2  = -0.000023
φ1p = 0.0186;   φ2p = 0.0044
φ10 = 0;        φ20 = π;     s = 180/π
V   = 693;      d   = 0.122; i = 1j
K10 = K20 = 0.0372; β = []; α = []

for k in range (101) :
  t  = 2 *k/1000;  sc = V *t/d
  φ1, φ2 = φ10 +φ1p*sc, φ20 +φ2p*sc
  K1, K2 = K10*e(λ1 *sc), K20*e(λ2 *sc)
  ξ      = K1 *e(φ1*i) +K2 *e (φ2*i)
  β.append (ξ.real *s)
  α.append (ξ.imag *s)

fig, ax = plt.subplots()

ax.plot (β, α)

ax.set  (xlabel = 'β [stupnjevi]',
         ylabel = 'α [stupnjevi]',
         title  = 'od 0 do 0.2 s')
ax.grid ()

plt.show()
```

Izvršenjem je dobiven graf:



Reference

[Dij1976] Dijkstra, E.W.: *A Discipline of Programming*, Prentice-Hall, 1976.

[Dov1982] Dovedan, Z.: *Sinteza i realizacija interaktivnog jezika s formalno definiranom semantikom*, mag. rad, Elektrotehnički fakultet, Zagreb, 1982.

[Dov1995] Dovedan, Z.: *PASCAL i programiranje*, don, Zagreb 1995.

[Dov2011] Dovedan, H. Z.: *PASCAL S TEHNIKAMA PROGRAMIRANJA*, VVG, Velika Gorica, 2011.

[Dov2012a] Dovedan, H. Z.: *FORMALNI JEZICI I PREVODIOCI • regularni izrazi, gramatike, automati*, Element, Zagreb, 2012.

[Dov2012b] Dovedan, H. Z.: *FORMALNI JEZICI I PREVODIOCI • sintaksna analiza i primjene*, Element, Zagreb, 2012.

[Dov2013] Dovedan, H. Z.: *FORMALNI JEZICI I PREVODIOCI • prevođenje i primjene*, Element, Zagreb, 2013.

[Erj2010] Erjavec, T. (2010b). MULTEXT-East Version 4: Multilingual Morphosyntactic Specifications, Lexicons and Corpora. *Proceedings of the LREC 2010*. Malta: European Language Resources Association. 2544-2547

[Jak2014] Jakupović, A., Pavlić, M., & Dovedan, H. Z. (2014). *Formalisation method for the text expressed knowledge*. Expert systems with applications., 41(11), 5308–5322.

[Jan1998] Janković, S. *"MEHANIKA LETA PROJEKTILA"*, Sveučilište u Zagrebu, Fakultet strojarstva i brodogradnje, 1998, str. 212-217

[Pav2013] Pavlić, M., Jakupović, A., & Meštrović, A. (2013a). *Nodes of knowledge method for knowledge representation*. *Informatologia*. 46 (3), 206-214.

[Pav2014] Pavlić, M., Dovedan, H. Z. & Jakupovic´, A. (2015). *Question answering with a conceptual framework for knowledge-basedsystem development "Node of Knowledge".* Expert systems with applications., 42(3), 5264–5286.

[Šum2019] Šuman, S. (2019). *Sustav za prevođenje poslovnih opisa u model podataka entiteta i veza*. Doktorski rad. Sveučilište u Rijeci, Odjel za informatiku.

[Wir1976] WIRTH, Niklaus: *Algorithms + Data Structures = Programs.* Prentice-Hall, Inc, 1976.

URL

[1] Popularity of programming language, (online), https://pypl.github.io/PYPL.html

[2] The python standard library, (online), https://docs.python.org/2/library

[3] Eulerova formula, (online), https://en.wikipedia.org/wiki/Euler%27s_formula

[4] Timsort https://en.wikipedia.org/wiki/Timsort

[5] Eight queens puzzle https://en.wikipedia.org/wiki/Eight_queens_puzzle

[6] Permutations and the N Queens Problem http://jrwalsh1.github.io/posts/permutations-and-the-n-queens-problem/

Programi po poglavljima

Kazalo

sintaksne kategorije

Bilješka o autoru

chanson d'amour et d'amitié
chanson d'un vieux routier
de la vieille rengaine

chanson des rues et des pavés
perdue ou retrouvée
sur le bord de la seine

chanson qui vit
dans ma mémoire
et vient dans ma guitare
me jouer la chansonnette

Prof
chanson des nappes de papiers
chanson qui fait rêver
musique un peu simplette... -

Georges Moustaki

Prof. dr. sc. Zdravko Dovedan, od 2010. **Han** (1952), umirovljeni je sveučilišni profesor u trajnom zvanju. Diplomirao je (1975) politehniku (aerodinamiku). Magistrirao je (1982) i doktorirao (1992) s temama iz područja računalnih znanosti, discipline formalni jezici i prevodioci.

Počeo je programirati u FORTRANu još za vrijeme studija politehnike, 1972. godine. Od 1977. godine bio je asistent na predmetima programiranja (BASIC i FORTRAN) i *Matematičkih principa programiranja*, a od 1979. godine predavač iz predmeta *Strukturno programiranje* (Pascal). Od 1984. godine predaje *Jezične procesore*, a od 1990. godine uvodi predmet *Formalni jezici i prevodioci* kojeg od 2005. godine čine tri kolegija: *Uvod u formalne jezike i automate*, *Teorija sintaksne analize i primjene* i *Teorija prevođenja i primjene*. Predavao ih na preddiplomskom, diplomskom i doktorskom studiju na Odsjeku za informacijske i komunikacijske znanosti Filozofskog fakulteta Sveučilišta u Zagrebu. Također je predavao *Algoritme i strukture podataka* i *Objektno i vizualno programiranje*, prvo u Pascalu (Delphiju), a potom u Pythonu na preddiplomskom studiju istog odsjeka. Na Veleučilištu Velika Gorica predavao je od 2003. do 2018. godine *Uvod u programiranje*, *Algoritme i strukture podataka*, prvo u Pascalu, potom od 2012. godine u Pythonu.

Autor je niza znanstvenih i stručnih radova te članaka iz područja informacijskih i računalnih znanosti, posebno teorije formalnih jezika i programiranja. Vodio je tri znanstvena projekta MZO iz područja obrade i razumijevanja prirodnih jezika. Tvorac je nekoliko informacijskih sustava koji su u razdoblju od 1988. do 2021. godine bili instalirani u preko pedeset tvrtki diljem Hrvatske.

Kao autor objavio je devet, a kao koautor četiri knjige, od kojih su najpoznatije *BASIC*, Ljubljana (1986); *FORTRAN 77 s tehnikama programiranja*, Ljubljana (1987); *PASCAL i programiranje*, Ljubljana (1989); *GW-BASIC*, Zagreb (1990); *FORMALNI JEZICI – sintaksna analiza*, Zagreb (2003); *PASCAL s tehnikama programiranja*, Velika Gorica (2011); tri sveučilišna udžbenika: *FORMALNI JEZICI I PREVODIOCI – regularni izrazi, gramatike, automati, FORMALNI JEZICI I PREVODIOCI – sintaksna analiza i primjene*, Zagreb (2012) i *FORMALNI JEZICI I PREVODIOCI – prevođenje i primjene (2013)*.

Hobi su mu pjevanje francuskih šansona, posebno šansonjera – kantautora: Georges Moustakija, Georges Brassensa, Yves Duteila, Gilbert Becauda, Pascal Danela, Herve Vilarda, Alain Barriera i Christophea. Zainteresirani ih mogu naći upisom „Zdravko Dovedan (Han)“ u tražilicu youtuba.