Zdravko Dovedan Han

PYTHON

i

programiranje

Zdravko Dovedan Han

PYTHON

i

programiranje

Zdravko Dovedan Han

Ozalj, 2025.

Recenzenti

Boris Čulina
Davor Lauc
Mirko Smilevski

Lektorica

Vjera Lopina

Slog i prijelom

Zdravko Dovedan Han

Dizajn ovitka

Zdravko Dovedan Han

Nakladnik

Vlastita naklada: Zdravko Dovedan Han
Ozalj, 2025.

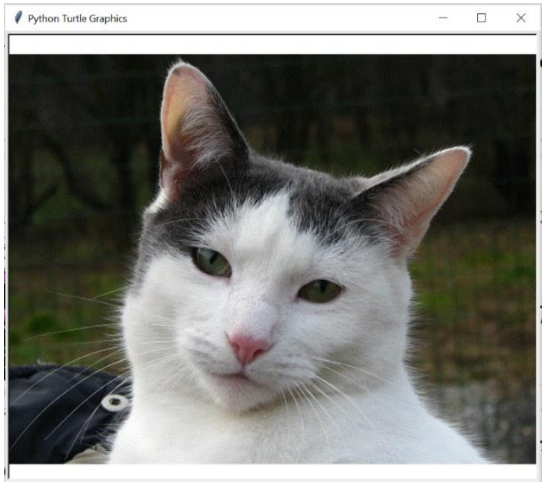
Tisak

PRENSA j.d.o.o
Kaštel Kambelovac

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

ISBN 978-953-49798-2-2

Sadržaj



Jacques

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 3.13.1.

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 ožujku 2025. godine rezultati su prikazani u sljedećoj listi (samo prvih sedam jezika). Python je prvom mjestuNa prvom mjestu je od siječnja 2018. do danas!

US, Mar 2025 :

Rank	Change	Language	Share	1-year trend
1		Python	33.19 %	+1.1 %
2		Java	13.75 %	-0.0 %
3	↑↑	C/C++	9.14 %	+1.7 %
4	↓	R	8.1 %	+0.3 %
5	↓	JavaScript	6.93 %	-0.6 %
6		C#	5.52 %	-0.6 %
7	↑	Rust	4.87 %	+1.1 %

Kad se danas kaže „Python“ misli se na Python 3.x koji se počeo razvijati od inačice 3.0 2010. godine do 3.13 2024. 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 publicirane 20.4.2020. godine.

„PYTHON i programiranje“

Druga je knjiga o Pythonu od istog autora!? Nastala je kao skraćena inačica knjige:

Dovedan, H. Z.: „progovorimo pythonski“, Zdravko Dovedan Han, Zagreb, 2021.

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

MALE TAJNE PROGRAMIRANJA

P R O G R A M I

Strukturom teksta određen je opći sadržaj poglavlja: uvod (motivacija), sintaksa i semantika naredbi, 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 „male tajne programiranja" 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 i 11 glavnih poglavlja. Knjiga sadrži preko 150 programa iz 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?

Ova je knjiga, prije svega, namijenjena gimnazijama, ekonomskim i tehničkim srednjim školama i naprednim učenicima viših razreda osnovne škole.

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 srednjim školama:

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

Knjiga je namijenjena i onima (od 13 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.

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

Onima koji žele saznati mnogo više o Pythonu i primjenama Pythona preporučujemo našu prvu knjigu „progovorimo pythonski". Informacije se mogu dobiti slanjem e-maila.

Podbrežje kod Ozlja, travnja 2025.

A u t o r

0. Osnove

Izučavanje jezika za programiranje, algoritama, tipova i struktura podataka te programiranja zahtijeva strog, formalni pristup, s opisom njegove sintakse (pravila pisanja ili „gramatike) i semantike (ili značenja). Također pretpostavlja solidno predznanje iz matematike, posebno diskretne matematike: teorije skupova i matematičke logike. Ovdje, veoma sažeto, dan je pregled matematičke logike, definicija jezika za programiranje, algoritama i prevodilaca.

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×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×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, ρ⊆N×M, može biti

ρ = { (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∈N, y∈M, da im je zbroj manji ili jednak 4. Isto svojstvo može se napisati kao xρy 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 "¬*a*", njezino djelovanje na sud *a* dano je u sljedećoj tablici:

a	¬*a*
F	T
T	F

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

Ako su *a* i *b* sudovi, onda je *a*∧*b* novi složeni sud ili konjunkcija sudova *a* i *b*. Znak "∧" 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:

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

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 prika-zivanja 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:

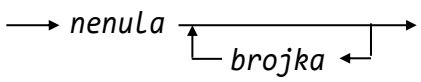
```
<naredba>  <ime>  <broj>  <izraz>
```

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

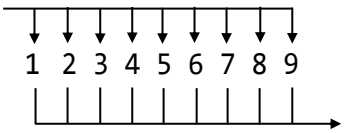
- sintaksne kategorije bit će pisane nakošenim malim slovima
- ako je ime sintaksne kategorije sačinjeno od više riječi, riječi će biti razdvojene donjom crtom (podcrtom)
- simboli jezika bit će pisani normalnim slovima ili nizovima znakova u boji ili ne

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:

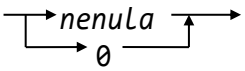
prirodni_broj:



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:

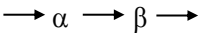
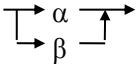
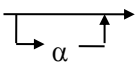
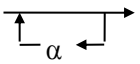
1) Kompaktnim konačnim pravilom - sintaksnim dijagramom - moguće je opisati veliki skup kombinacija u generiranju dane naredbe.
2) Pravila predočena dijagramima brže se pamte i uče, jer većina ljudi bolje pamti vizualno.
3) Promatranjem svih naredbi jezika moguće je uočiti dijelove koji su jednaki. Uvođenjem 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*
$\alpha\ \beta$	slijed	
$\alpha \mid \beta$	alternativa (α ili β)	
$[\alpha]$	α izostavljeno ili napisano jedanput	
$\{\alpha\}$	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!

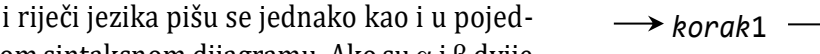
OSNOVNI ALGORITAMSKI KONSTRUKTI

Oblikovanje algoritma zahtijeva poznavanje nekoliko algoritamskih konstrukata.

Tri su osnovna konstrukta, čijom se kompozicijom može oblikovati algoritam kao rješenje zadanog problema: slijed (sekvenca), gra-nanje (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



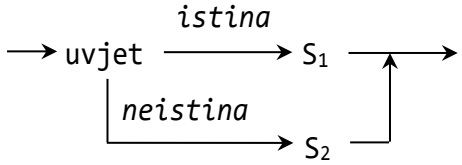
Na primjer, algoritam

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 × f(n-1), n≥1
```

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.

Interaktivni (Shell) môd

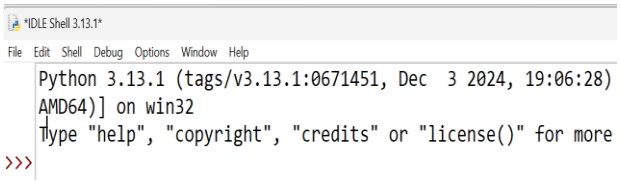
*Počet ćemo s opisom Pythona u njegovom interaktivnom ili **Shell** modu koji će nam pomoći da, uvodeći brojeve, izraze, stringove, standardne funkcije i procedure, već poslije pola sata učenja rješavamo neke brojčane probleme. Otpočetka uvodimo notaciju za prikaz pravila tvorbe („sintaksu“) pojedinih dijelova („rečenica“) Pythona i opisujemo njihovu značenje ili „semantiku“. Podrazumijeva se da sve vježbe i programe izvršavate na svom računalu.*

Uvod

Instalirajte Python 3.13 na svoje računalo s adrese:

http://www.python.org/.

Ljuska (engl. *shell* - ljuska, školjka) je termin koji se u računarstvu općenito doživljava kao dio softvera koji pruža sučelje za korisnika na neki drugi softver ili operacijski sustav. Poslije instalacije Pythona 3.10 u *START* aplikacijama vaših Windowsa Pythonovu ljusku s imenom IDLE (Python 3.10 64-bit) prenesite u taskbar. Poslije njegova poziva imali biste sljedeći sadržaj ekrana:



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

Options* -> *Configure IDLE

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. Sa **Size** izaberite veličinu znakova, na primjer 16. Na kraju sve izmjene potvrdite s **Ok**.

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 poslije simbola „`>>>`“ koji se nalazi u lijevom stupcu i ne može se mijenjati.

Napomena: Ako radite u inačici 3.9 ili manjoj promt „`>>>`“ je bio dio tekuće linije!

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. Pozivamo Python i upisujemo:

```
>>> Dobar dan. <Enter>
    SyntaxError: invalid syntax
```

Dojavljena je opća sintaksna pogreška. Dalje pišemo:

```
>>> Python <Enter>
    NameError: name 'Python' is not defined
```

Opet je dojavljena sintaksna pogreška s objašnjenjem da ime `'Python'` nije poznato (definirano).

Brojevi i izrazi

Probajmo unijeti brojeve:

```
>>> 1 <Enter>      1
>>> 101 <Enter>    101
>>> 123456789999234987654321000155 90 <Enter>
    123456789999234987654321000155 90
```

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. To je cijeli (dekadski) broj. Piše se prema pravilu:

dekadski_broj : $[0]^+ \mid [1\text{-}9]\,[0\text{-}9]^*$

Na primjer:

```
00000  1234500001  99999999998888877777
```

Ispred cijelog broja može se napisati jedan ili više predznaka. Predznak nije dio broja. Može ga slijediti jedan ili više razmaka i ima značenje unarne operacije. Rezultirajući broj imat će predznak jednak „-“, ako je napisan neparni broj minusa, „+” ako je napisan parni broj minusa. Predznak „+” je neutralan i izostavlja se u interpretaciji (prikazu vrijednosti) broja. Primjeri:

```
>>> -657                  -657
>>> -     2147483648      -2147483648
>>> - - - 20074854        -20074854
>>> --4                   4
>>> +++---+++ 678         -678
```

Osim cijelobrojnih vrijednosti Python ima i realne (decimalne) vrijednosti. Ime realnog tipa je „float”.

Realni brojevi mogu biti napisani u normalnom (uobičajenom) i eksponencijalnom obliku. U normalnom obliku je cijeli od decimalnog dijela odvojen točkom. Evo pravila:

normalni_realni_broj:
$[0\text{–}9]^+$. $[0\text{-}9]^+ \mid [0\text{–}9]^+$. | . $[0\text{-}9]^+$

Broj se piše bez razmaka. Za razliku od cijelih brojeva, koji 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.

1.1 realni brojevi

```
>>> 1.000     1.0          >>> 001.          1.0
>>> 00.5      0.5          >>> .5           0.5
>>> .123      0.123        >>> 2.           2.0
>>> 12.5012.5              >>> 3.14         3.14
>>> 999999999999999.0          99999999999999.0
>>> 9999999999999991.0      9999999999999992.0
>>> 9999999999999993.0      9999999999999992.0
>>> 9999999999999995.0      9999999999999996.0
>>> 9999999999999997.0      9999999999999996.0
>>> 9999999999999998.0      9999999999999998.0
>>> 9999999999999998.9      9999999999999998.0
>>> 9999999999999999.0                   1e+16
```

Uočavamo neočekivane rezultate (aproksimacije) kod velikih brojeva koji sadrže više od petnaest znamenki cijelog dijela interpretiraju s približnim vrijednostima.

Broj 9999999999999999.0 je prikazan u eksponencijalnom obliku također aproksimiran, jer je to 10**16, o čemu će riječi biti kasnije.

Realni brojevi mogu biti predznačeni na jednak način kao i cijeli brojevi. Na primjer:

```
>>> -123.5    -123.5   >>> - -+++ 55.    55.0
>>> - 3.14    -3.14    >>> + 1.618       1.618
```

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.

```
>>> # Ovo je komentar
>>>         # i ovo je komentar, ne mora
>>>    # biti napisan otpočetka
```

```
>>> 98765432100099999999999 # veliki broj
    98765432100099999999999
>>> # Cijeli brojevi su bez ograničenja
```

Brojčani izrazi

Brojčani izrazi mogu biti jednostavni ili složeni. Cijeli i realni brojevi, s predznakom ili bez njega, primjeri su jednostavnih brojčanih izraza koji se pišu prema pravilu:

```
jed_br_izraz : predznak broj
  predznak   : [+-]*
  broj       : cijeli_broj|realni_broj
```

Osim predznaka, kao unarne operacije, nad cijelim i relnim brojevima definirani su binarni operatori. Ako su *x* i *y* dva jednostavna brojčana izraza, simboli i značenja operatora su:

+	zbrajanje, *x+y*
-	oduzimanje, *x-y*
*	množenje, *x*y*
/	realno dijeljenje, *x/y*
**	potenciranje, *x**y* = x^y

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. Značenje operacija zbrajanja, oduzimanja, množenja i dijeljenja jednako je kao u matematici. Pogledajmo nekoliko primjera i rezultata izračunavanja u sljedećim vježbama:

›››1.2 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
```

›››1.3 množenje

```
>>> # cjelobrojno množenje
>>> -15 *12      -180    >>> -15 *-12     180
>>> 10 *---- 2   20
>>> 97767075765556445 * 78765650007756453221
    770068727203162675184375132867755934 5
>>> # realno množenje
>>> 9776766757655564445 *
    787656500077564453221.0
    7.700733886409659e+39
>>> 9776766757655564 *1.0
    9776766757655564.0
```

›››1.4 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
>>> 10 /0 ZeroDivisionError: division by zero
```

Vidimo da se u realnom dijeljenju može dobiti približan rezultat. Operacija potenciranja ima značenje kao u matematici.

›››1.5 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 # drugi korijen iz 2
    1.4142135623730951
```

➧ *U matematici je potenciranje* `0⁰` *neodređeno. U Pythonu je* `0**0` *jednako* 1. □

SINTAKSA

Sada možemo definirati pravila pisanja brojčanog izraza:

```
br_izraz : operand { operator operand }
operand  :
  predznak ( broj | ( br_izraz ) )
operator : + | - |* | / | **
```

Ovo je najkompaktniji prikaz sintakse brojčanog izraza ali, na prvi pogled, može izazvati nerazumijevanje zbog uporabe rekurzije: u definiciji operanda pojavljuje se brojčani izraz.

Uvedimo sljedeće oznake za pojedine sintaksne kategorije:

```
BI - br_izraz
O  - operand        o  - operator
p  - predznak       b  - broj
c  - cijeli_broj    r  - realni broj
```

Evo dva primjera pravilno napisanih brojčanih izraza:

```
BI → O
   → p ( BI )          → - ( BI )
   → - ( O o O )       → - ( p b o O )
   → - ( b o O )       → - ( c o O )
   → - ( 101 o O )     → - ( 101 - O )
   → - ( 101 - p b )   → - ( 101 - b )
   → - ( 101 - c )     → - ( 101 - 55 )
```

```
BI
→ O o O o O o O          o O o O o O
→ O o O o O o ( O o O ) o O o O o O
→ c + c * c + ( c + c ) / c - r   ** c
→ 1 + 2 * 3 + ( 4 + 5 ) / 3 - 2.5 ** 2
```

Pisanjem brojčanih izraza u interaktivnom modu Pythona upozorava zvučnim signalom ako je pogrešno napisana otvorena ili zatvorena zagrada. Na primjer:

```
>>> 2*(1+2))        SyntaxError: unmatched ')'
```

Poslije druge zatvorene zagrade ignoriran je zvučni signal i pritisnuto <Enter> pa je dojavljena pogreška.

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 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.

››› 1.6 tip izraza

```
>>> type (12 +4)          <class 'int'>
>>> type (12 /4)          <class 'float'>
>>> type (12 -4)          <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

Na primjer, pogledajmo kako će biti evaluiran izraz:

```
1 +2*3 +(4 + 5)/3 -2.5**2
1 +2*3 +(4 + 5)/3 -2.5**2
   ↓
1 +6 +(4 + 5)/3 -2.5**2
↓
7 +(4 + 5) /3 -2.5**2
```

```
7 +(4 + 5) /3 -2.5**2
   ↓
7 +9 /3 -2.5**2
   ↓
7 +3.0 -2.5**2
↓
10.0 -2.5**2
      ↓
10.0 -6.25
↓
3.75
```

››› 1.7 izračunavanje izraza

```
>>> 1 +2*3        7     >>> (1 +2)*3         9
>>> -2 **3        -8    >>> (1 +2)*(2+4)     18
>>> - (101 -55 ) -46
>>> 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 ×2 ×3 × ×n

››› 1.8 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* → Library reference → 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)

››› 1.9 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
```

„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* }

Značenje zareza jest niz vrijednosti dobivenih izračunavanjem izraza. U ispisu će biti prikazani u okruglim zagradama. Značenje točka-zareza jest završetak jednog izraza čija će vrijednost biti ispisana u sljedećem redu i početak narednog izraza. Pogledajte sljedeću vježbu:

››› 1.10 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. Redovi komandne linije označeni su s tri točke.

››› 1.11 nastavak komandne linije

```
>>> 1; \    <Enter>
     2, \   <Enter>
       3    <Enter>
    1
    (2, 3)
>>> 12 +      \     <Enter>
        13 * 14     <Enter>              194
>>> round (1.5555,  <Enter>
                 3 )  <Enter>          1.556
>>> 100\ <Enter>
         1  <Enter>
    SyntaxError: invalid syntax
>>> (  100  <Enter>
      +200   +   300 ) *5   <Enter>     3000
>>> (  100   <Enter>
      +200   <Enter>
      +300 ) *5 +10 *( 11 **2  <Enter>
                    +12 **2)  <Enter>  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 pravokutnika sa stranicama a=6.25 i b=12.5.

Opseg je $2(a+b)$, a površina ab, pa je rješenje kao što slijedi.

››› 1.12 opseg i površina pravokutnika

```
>>> 2 *(6.25 + 12.5)  # Opseg:          7.5
>>> 6.25 *12.5        # Površina:     78.125
```

Nedostatak je ovoga rješenja što smo na dva mjesta morali napisati vrijednost stranica *a* i *b*. 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.

››› 1.13 opseg i površina pravokutnika (2)

```
>>> a = 6.25            >>> b = 12.5
>>> a            6.25   >>> b         12.5
>>> 2 *(a +b)    37.5   >>> a *b     78.125
```

JEDNOSTAVNO PRIDRUŽIVANJE

U prethodnoj smo vježbi sa `a=6.25` i `b=12.5` 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 pišu prema sljedećem pravilu:

```
ime : ( slovo| _ )( slovo| _| brojka )*
```

Ili, riječima, ***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***, neograničeni broj puta. Karakteristika je Pythona, od inačice `3.0`, da slovo 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 Pythona `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, `a=6.25` i `b=12.5` iz vježbe **›››1.13**, bile definirane dvije varijable u dijelu radne memorije, s imenima `a` i `b`. Varijabli s imenom `a` pridružena je realna vrijednost `6.25`, a varijabli s imenom `b` također realna vrijednost `12.5`.

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

operand **:**
predznak **(** *broj* **|** *br_varijabla* **|**
(*br_izraz* **)** **)**

Provjerimo vrijednosti i tipove varijabli `a` i `b`:

```
>>> a          6.25      >>> b       12.5
>>> type (a)   <class 'float'>
>>> type (b)   <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.

››› 1.14 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 i tip pridruže 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 svojom „starom” vrijednošću. Poslije evaluiranja izraza `i+5` dobivena vrijednost bit će pridružena varijabli `i`. „Nova” vrijednost varijable `i` bit će `15`. U sljedećem 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.

››› 1.15 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
>>> _                                   125
```

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

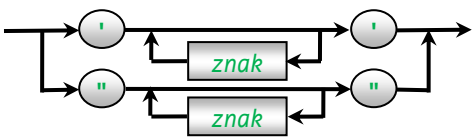
```
>>> _ = 101
>>> _                                   101
>>> 2**8                                256
>>> _                                   101
```

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"
```

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

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

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"
    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.

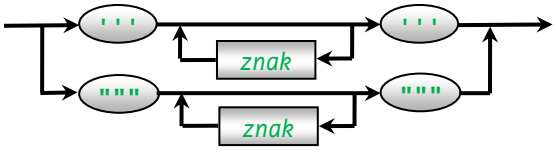
››› 1.16 znakovni nizovi

```
>>> "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:



››› 1.17 tekst

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

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:

››› 1.18 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.

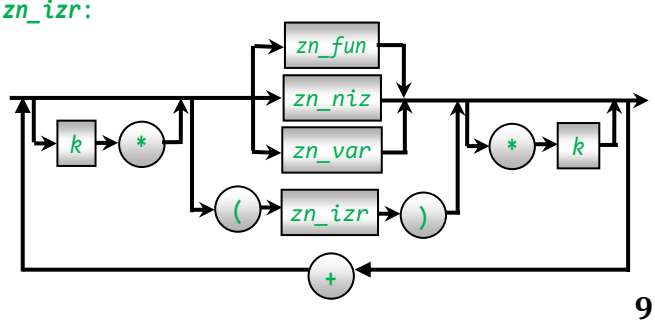
››› 1.19 ord()

```
>>> 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'
```

››› 1.20 znakovni izrazi

```
>>> '='*10                          '=========='
>>> 'ha, '*3 +'   '          '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 sa *zn_var*.

››› 1.21 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 = '3.13.1';  P +' ' +v
    'Python 3.13.1'
```

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)

1.22 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
```

1.23 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).

1.24 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".

1.25 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.

1.26 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.

1.27 '\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.\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:

››› 1.28 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 **›››1.28** :

››› 1.29 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.

›››1.30 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'`.

MALE TAJNE PROGRAMIRANJA

Svako će poglavlje sadržavati poseban dio pod nazivom „Male tajne programiranja" 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 veći od nule i `Y` decimalni dio od `X`, to možemo uraditi na dva načina:

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

```
>>> 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
```

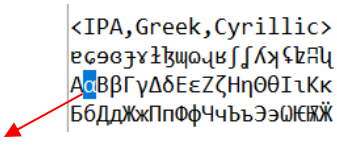
Poneki programeri biraju „duhovita” imena varijabli i konstanti. Možda je u tome najviše pretjerao Jean-Pierre, ljubitelj mačaka i svega što je francusko. U svojim programima koristi isključivo svoje ime i imena:

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

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

U šestom poglavlju 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 + 2/12

Zadatak 1.4

*Izračunati broj kombinacija 7 od 35 brojeva igri **LOTO 7/35** i 5 od 50 i 2 od 12 brojeva u igri **EUROJACKPOTa**.*

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

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

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

$$Loto = \frac{35 * 34 * 33 * 32 * 31 * 30 * 29}{1 * 2 * 3 * 4 * 5 * 6 * 7}$$

$$Eurojackpot = \frac{50*49*48*47*46}{1*2*3*4*5} * 6 * 11$$

››› LOTO 7/35 i EUROJACKPOT 5/50 + 2/12

```
>>> 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
    n = 50; m = 1; i = n; j = 1
    exec (n_m *4); Euro = n //m *6*11
    print ('Broj kombinacija:')
    print ('  Loto',Loto,' Euro', Euro)"""
```

```
>>> exec (LOTO_EURO)
    Broj kombinacija:
      Loto 6724520  Euro 139838160
```

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
    815915283247897734345611269596115894272
    000000000

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

```
70
119785716699698917960727837216890987364589381425464258575553628646280095827898453196800000000000000000
```

„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].

A rješenje je jednostavno! Kazaljke se 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:

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

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
>>>                      # KRAJ POGLAVLJA #
```

2.

Programski môd

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

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`. 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. Kasnije je uvedena je „Unicode" standardizacija.

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

Rječnik

Nad alfabetom Pythona definirane su sljedeće klase riječi ili simbola:

- cijeli brojevi
- realni 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]+
```

U prethodnom smo poglavlju radili s cijelim dekadskim brojevima. Iz njegove sintakse sada vidimo da je 0 ili neograničeni broj nula cijeli broj čija je vrijednost jednaka 0. Ali, ne smije početi s nulom, dok binarni, oktalni i heksadecimalni cijeli brojevi moraju početi s nulom iza koje slijede slova B, O ili X kao oznaka vrste broja.

››› 2.1 Cijeli dekadski brojevi

```
>>> 0                                  0
>>> 0000000                            0
>>> 01234
    SyntaxError:
>>> 1 0 0 1
    SyntaxError: invalid syntax
>>> 1001                               1001
>>> 12345678909876543217532498573498573323
    12345678909876543217532498573498573323
```

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

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

gdje su b_i, i=0, 1, , n, znamenke binarnih, oktalnih ili heksadecimalnih brojeva, u kojima je značenje heksa-decimalnih 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:

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

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

››› 2.2 Binarni, oktalni i heksadecimalni brojevi

```
>>> 0b10101010101010101010101          592405
>>> 0B11111111111111111111111         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
```

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 skupa rezerviranih riječi Pythona korištenih u ovoj knjizi, napisanih alfabetski:

```
and  as  break  class  continue  def
del  elif  else  except  False  for
from  if  import  in  lambda  None
not  or  pass  return  True  try  while
```

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 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 posebnih simbola sačinjenih od dva ili tri znaka:

```
!=  *=  """  +=  <=  %=  ==  -=  >=
'''  //  //=  /=  |=  **  **=
```

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.

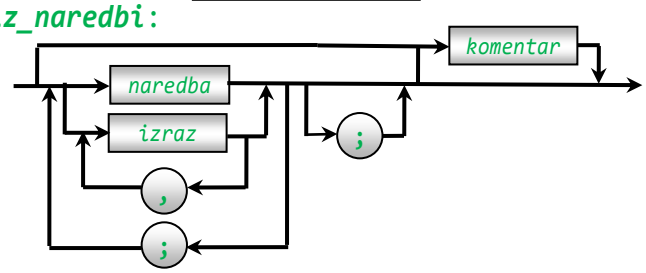
SINTAKSNA STRUKTURA

Osnovna sintaksna struktura Pythona dana je sljedećim pravilima:

program: *blok*

blok:

niz_naredbi:



naredba:

pridruživanje | *LAMBDA_funkcija* |
naredba_EXEC | *naredba_DEL* |
ispis | *uvoz_modula*

Python ima nekoliko složenih naredbi od kojih u ovoj knjizi opisane selekcija, u četvrtom, *WHILE* i *FOR* petlja u petom poglavlju.

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.

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`“.

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 primitivan ili složeni.

Od primitivnih tipova podataka u ovoj knjizi opisani su cjelobrojni, realni i logički, a od složenih znakovni nizovi (stringovi), nizovi (n-torke i liste), skupovi i rječnici.

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.

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`.

››› 2.3 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   >>> 12//-5         -3
```

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`

››› 2.4 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.* □

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 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
```

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
```

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:/   /_ PROGRAMI/02/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:

››› 2.5 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 želje-

nom 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 vrijed-nosti **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` `'12345678'`
		`"%i" % 12345678` `'12345678'`
		`"%u" % 12345678` `'12345678'`
		`"%d" % -12345678.999` `'-12345678'`
		`"%i" % 0xFFF` `'4095'`
		`"%d" % 0b1111` `'15'`
		`"%d" % 0017` `'15'`
		`"%d" % 0xF` `'15'`

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

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 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***	`"%12.4f" % 2**0.5` `'  1.4142'`
		`"%12.4f" % 300**0.5` `'  17.3205'`
		`"%12.2s" % 'abcdef'` `'  ab'`
		`"%12.3s" % 'abcdef'` `'  abc'`
		`"%12.4s" % 'abcdef'` `'  abcd'`
		`"%12.5s" % 'abcdef'` `'  abcde'`
		`"%12.6s" % 'abcdef'` `'  abcdef'`
		`"%12.12s"% 'abcdef'` `'  abcdef'`

d	*Značenje*	*>>> Primjeri*
	znakova u polju duljine ***m***.	`"%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

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 vjetar m/s koliko je to km/h?

>>> Vjetar

```
>>> m_s  = eval (input ('Zadajte '
                 'vjetar 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 vjetar 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 **›››2.5** 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 môd.

```
 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 dio njegovog sadržaja.

>>> 2.6 math

```
>>> import math;  dir (math)
['__doc__', ..., 'acos', 'acosh', 'asin',
'asinh',  'atan',  ...,  'cos',  'cosh',
'degrees', 'dist', 'e', ..., 'exp', ...,
'factorial', 'floor', ..., 'log', 'log10',
'log1p', 'log2', ..., 'pi', 'pow', 'prod',
'radians', ..., 'sin', 'sinh', 'sqrt', ...,
'tan', 'tanh', 'tau', 'trunc', 'ulp']
```

Ovdje, već po nazivu, prepoznajemo neke od trigonometrijskih, 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.

2.7 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]	*r* Logaritam po bazi b (`log(x)/log(b)`)	Prirodni logaritam (po bazi `e` *x*>0 *b*>0 `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.

2.8 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 $[a,b]$, $a \le b$.

2.9 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.

››› 2.10 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.

››› 2.11 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()`**.

››› 2.12 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

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:

$Id \leftarrow$ id(vrijednost (i))
$v_1 \rightarrow Id \quad v_2 \rightarrow Id \quad v_n \rightarrow Id$

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:

››› 2.13 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

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

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

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 .

2.14 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.

2.15 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.

2.16 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.

››› 2.17 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

››› 2.18 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 programi-ma. 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.

››› 2.19 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
```

```
>>> 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)
```

MALE TAJNE PROGRAMIRANJA

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 DVIJU VRIJEDNOSTI

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 imali smo 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
    43466557686937456435688527675040625802564660517371780402481729089536555417949051890403879840079255169295922593080322634775209689623239873322471161642996440906533187938298969649928516003704476137795166849228875
```

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, osim „MALIH TAJNI PROGRAMIRANJA", sadržavati i odjeljak PROGRAMI u kojem će biti dani primjeri programskih rješenja mnogih problema matematike, fizike, kemije, obrade teksta, prevođenja itd. primjenjujući dotad uvedene naredbe, tipove i strukture podataka. Neki će programi biti „usavršavani" u narednim poglavljima, kad budu uvedene složene naredbe i složeni tipovi podataka.

PLAĆANJE RAČUNA S NAJMANJIM BROJEM APOENA



Dani iznos računa C u eurima bez centi treba platiti s najmanjim brojem apoena. Apoeni su 200, 100, 50, 20, 10, 5, 2 i 1. Postupak ćemo započeti s iznosom c = C dijeleći ga s najvećim apoenom, 200, 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"*8. Novčanice kojih nije bilo u iznosu računa imat će 0 u ispisu.

Plaćanje_računa.py

```
# PLAĆANJE RAČUNA U EURIMA S NAJMANJIH
# BROJEM APOENA
d = divmod; C = c = int (eval (input (
    'Iznos računa, u eurima ')))
_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"*8 %
(_200, _100, _50, _20, _10, _5, _2, _1))
eur = ("%4d"*8 % (200,100,50,20,10,5,2,1))
print ("Iznos %d kn platiti sa:" % C)
print (X);  print ("   x"*8)
print (eur)
```

```
Iznos računa, u eurima 1259
Iznos 1259 eura platiti sa:
   6   0   1   0   0   1   2   0
   x   x   x   x   x   x   x   x
 200 100  50  20  10   5   2   1
Iznos računa, u eurima 388
Iznos 388 eura platiti sa:
   1   1   1   1   1   1   1   1
   x   x   x   x   x   x   x   x
 200 100  50  20  10   5   2   1
```

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
99999888888881231312357659999
Broj 99999888888881231312357659999 ima 28
znamenki. Zbroj znamenki = 175
```

TABLICA

Evo 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
```

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
```

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  = 0b1111111011111100000001
    x // y = 0b1000000
```

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

   Koordinate vrha A: -1, -1
   Koordinate vrha B: 3, 1
   Koordinate vrha C: 0, 4
   P = 8.99328867513853
```

3.
Logički tip podataka

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.

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

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`.

››› 3.1 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`, `''` ili `False`. Inače, vraća `True`.

››› 3.2 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 ("neistina")         True
>>> bool (" ")                True
>>> bool ('')                 False
```

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:

log_izraz : *log_operand*
{ *log_operator log_operand* }
| *rel_izraz*
log_operand : *log_negacija*
(False | True | *log_varijabla* |
log_funkcija | (*log_izraz*))
log_operator : and | or
log_negacija : (not)*

Relacijski izraz

Relacijski izrazi važna su podklasa logičkih izraza. Pišu se prema pravilu:

rel_izraz : *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.* □

S uspoređivanjem dvaju stringova zasad se nemojte zamarati. Objasnit ćemo ih u šestom poglavlju.

3.3 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
>>> 1.5 > True                      True
>>> 1 == True                       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.

››› 3.4 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 pisanju logičkih izraza pojavljuju se dvije binarne logičke operacije, and i or. Sljedeća je vježba prikaz logičkih operacija na pregledniji način od onoga u vježbi **›››3.3.**

››› 3.5 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.

››› 3.6 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.

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).

››› 3.7 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
```

››› 3.8 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 3.1

Usluga parkiranja u jednom danu naplaćuje se 2 Eura po satu za prva tri sata, poslije toga još po 1 Euro za svaki započeti sat. Izračunati cijenu usluge parkiranja ako se vrijeme dolaska i odlaska unese kao realni broj S.MM.

››› 3.9 Usluga parkiranja (1)

```
>>> Parking = """
T1, T2 = eval (input(
   'Vrijeme dolaska i odlaska u formatu '
  'S.MM? '))
   T  = int (round (abs (T2 -T1) +0.491))
EUR = 2*T*(T <= 3) \
    +(2*3 +1*(T -3)) *(T > 3)
print (
"Platiti %d EUR (%d sati parkiranja)"
 % (EUR, T)) """

>>> exec (Parking)
    Vrijeme dolaska i odlaska u formatu
    S.MM? 9.09, 14.10
    Platiti 12 EUR (6 sati parkiranja)
```

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:

››› 3.10 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.

››› 3.11 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'
```

››› 3.12 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*.

››› 3.13 Usluga parkiranja (2)

```
>>> Parking2 = """
T1,  T2  = eval  (input  ("Vrijeme  dolaska  i
odlaska u formatu S.MM? "))
T  = int (round (abs (T2 -T1) +0.491))
EUR = 2 *T if T <= 3 else 2*3 +1*(T -3)
print ("Platiti %d EUR "
"(%d sati parkiranja)" % (EUR, T)) """
>>> exec (Parking2)
    Vrijeme dolaska i odlaska u formatu
    S.MM? 9.09, 14.10
    Platiti 12 EUR (6 sati parkiranja)
```

Ova se inačica razlikuje od one dane u vježbi ›››3.9, u načinu izračuna ukupnog iznosa `Eura`:

```
# >>> 3.9:
>>> EUR = 2*T*(T <= 3) \
        +(2*3 +1*(T -3))*(T > 3)
# >>> 3.13:
>>> EUR = 2*T if T<=3 else 2*3 +1*(T -3)
```

LAMBDA funkcija

LAMBDA funkcija uvedena u prethodnom poglavlju može, osim brojčanog, imati logički ili uvjetni izraz:

izraz : *brojčani_izraz*| *logički_izraz*| *uvjetni_izraz*

Ovisno o tipu izraza reći ćemo da je LAMBDA funkcija cjelobrojna, realna ili logička.

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". Rješenje je sadržano u sljedećoj vježbi:

›››3.14 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)
```

MALE TAJNE PROGRAMIRANJA

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. Prije svega, proširimo **Moj_modul.py** sa:

```
Input = lambda s : eval (input (s))
ok = lambda g : g >= 1583
PG = lambda g : ok (g) and (g %400 == 0 or
   g %4   == 0 and g %100 != 0)
```

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)  1.41421... >>> 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) 1.41421...    >>> 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)

a > 10 and a < 20
        ↓
10 < a and a < 20    → 10 < a < 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; m = S *60/11.0; M = int (m)
s = round ((m-M)*60, 2) # s sekunde
print (T, ':', M, ':', s)
```

```
>>>
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

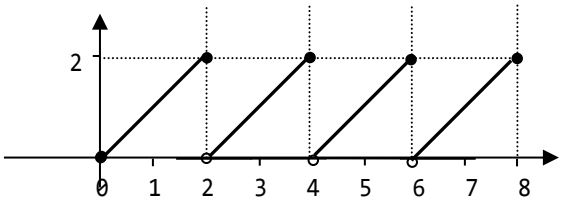
Prije svega, uočimo da iz semantike uvjetnih izraza slijedi definicija logičkih operacija:

```
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 3.3

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.

3.15 "Pilasta funkcija"

```
>>> # "Pilasta funkcija" f(x) na int. [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
```

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 desetom 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 100-tog člana je trenutan! Evo rezultata:

```
100-ti član  : 354224848179261915075
```

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. U nekoliko je programa pokazana uporaba uvjetnih izraza u rješavanju problema iz prakse, kao i rješenje programa iz prethodnih poglavlja na drugi način.

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	la gano 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 test iz predmeta
# "Programiranje u Pythonu".
# Prolaznost je 90%, od čega su
# vjerojatnosti dobivenih ocjena:
# 2 25%, 3 45%, 4 20% i 5 10%
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
```

CIJENA PARKIRANJA U ZRAČNOJ LUCI ZAGREB

Pretpostavimo da je „Cjenik parkirališta za putnike i posjetitelje" u Zračnoj luci Zagreb 2025. godine bio:

Kategorije (vrijeme zadržavanja)	**Cijene u EUR uključivo PDV**
do **1** sat	3,50 €
2 sata	7,00 €
3 sata	10,50 €
od **3 do 6** sata	14,00 €
od **6 do 12** sati	17,50 €
od **12 do 24** sata	21,00 €
od **1**. dana svaki sljedeći dan	11,00 €

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 = 3.5  if T <=  1.0  else  \
```

```
    7.0   if T <=  2.0  else  \
    10.5  if T <=  3.0  else  \
    14.0  if T <=  6.0  else  \
    17.5  if T <= 12.0  else  \
    21.0  if T <= 24.0  else  \
    21.0 +( 0 if d <= 1 else 11.0 *(d-1))
print ( "Usluga parkiranja %0.2f €" % C)
```

```
>>>
Trajanje parkiranja (Dana, Sati, Minuta)? 0,
0, 59
Usluga parkiranja 3.50 €
>>>
Trajanje parkiranja (Dana, Sati, Minuta)? 0,
2, 15
Usluga parkiranja 10.50 €
>>>
Trajanje parkiranja (Dana, Sati, Minuta)? 0,
3, 28
Usluga parkiranja 14.00 €
>>>
Trajanje parkiranja (Dana, Sati, Minuta)? 7,
5, 11
Usluga parkiranja 98.00 €
```

ZBROJ ZNAMENKI PRIRODNOG BROJA (2)

Treba zbrojiti znamenke (brojke) velikog prirodnog broja. Rješenje je sadržano u programu

Zboj_znamenki.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_znamenki.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
   848029428374283742389742384763245676543785643785634786543785634875643876537485674385634875643786534785634785634875634875328765387265843756348756374865348765473865473856743825438725638479
   986
```

4. 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 "iznimke" 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, funkcija `math.sqrt()` (drugi korijen) za $x \geq 0$. 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`:

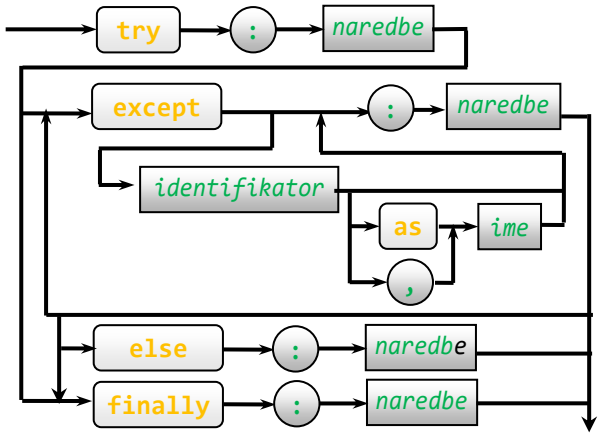
```
>>> '123'**2
    TypeError: unsupported operand type(s)
    for ** or pow(): 'str' and 'int'
```

U pokušaju dijeljenja s nulom bila bi dojavljena pogreška:

```
>>> 12/0
    ZeroDivisionError: integer division or
    modulo by zero
```

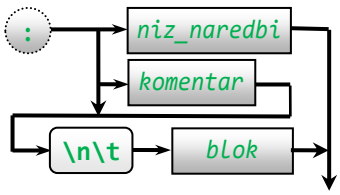
Poslije svake dojave pogreške prekida se daljnje izvršavanje programa. Ako ne želimo da se to dogodi, postoji posebna naredba, *naredba TRY*, koja nam dopušta provjeru izvršivosti pojedinih naredbi (tamo gdje očekujemo da do toga može doći), dojavimo pogrešku i eventualno tražimo unos valjanih podataka.

naredba_TRY:



Naredba TRY ili iznimke prva je složena naredba koju ćemo definirati. Sastoji se iz glavnog dijela, koji započinje rezerviranom riječju `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!
```

Niz poziva funkcije `sqrt()` ovdje je zamijenjen logičkim izrazom:

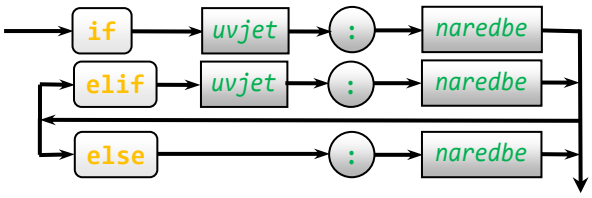
```
s > 0 and s-a > 0 and s-b > 0 and s-c > 0
```

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:



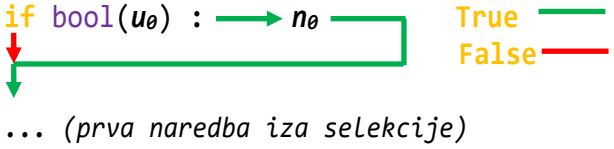
Kontekstna pravila pisanja selekcije jednaka su konteksnim pravilima pisanja *naredbe TRY*. Rezervirane riječi `if`, `elif` i `else` pišu se u istoj koloni (jer su dio istog bloka, osnovnog ili na nižoj razini). Dijelovi koji počinju s `elif` i `else` zvat ćemo *ELIF* i *ELSE grane* selekcije. Primjeri:

```
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)

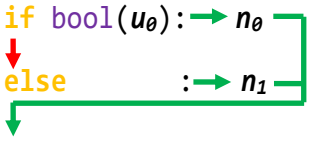


Prvo se izračuna `bool(`u_0`)`. Ako je jednak `True`, izvršit će se naredbe n_0 i nastaviti na prvoj naredbi iza selekcije. Ako je jednak `False`, naredbe n_0 neće biti izvršene, već se izvršavanje programa nastavlja na prvoj naredbi iza selekcije. Na primjer:

```
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)

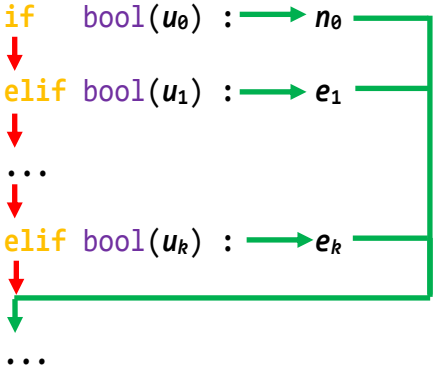


Prvo se izračuna `bool(`u_0`)`. Ako je jednak `True`, izvršit će se naredbe n_0 i nastaviti na prvoj naredbi iza selekcije. Ako je jednak `False`, bit će n_1, naredbe *ELSE grane* i izvršavanje programa će se nastaviti na prvoj naredbi iza selekcije. Na primjer:

```
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)



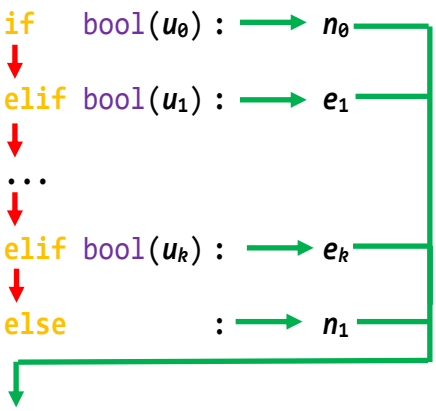
Opet se kreće od glavne grane i računa se U_0 = `bool(`u_0`)`. Ako je U_0 jednako `True`, izvršit će se naredbe n_0 i nastaviti na prvoj naredbi iza selekcije. Ako je U_0 jednako `False`, prelazi se na prvu *ELIF granu* i računa U_1. Ako je jednako `True`, bit će izvršene naredbe e_1, i nastavit će se 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, 3)
if   i == 1 : B = 'jedan'
elif i == 2 : B = 'dva'
elif i == 3 : B = 'tri'
```

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*.

MALE TAJNE PROGRAMIRANJA

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.py`.

```
>>> 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:

```
>>> Input1 = """
    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 Input1) """
>>> exec (Input1)
    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:

```
>>> Input2 = """
    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 Input2)  """
>>> exec (Input2)
    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 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.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 konj. kompleksna')
      x1 = a1 +a2*1j; x2 = a1 -a2*1j
      print ('x1 =', x1, ' x2 =', x2)
else : print ("Nije kvad. jedn. (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 je rješenje 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*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 = 1/2
>>>
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
```

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,..., 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 MALE TAJNE PROGRAMIRANJA i PROGRAMI to je zorno prikazano u velikom broju primjera.

Uvod

U prethodnom smo poglavlju pokazali kako se mogu provjeravati ulazni podaci i kako ponoviti unos ako nisu korektni. 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 = 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
```

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 = 5
```

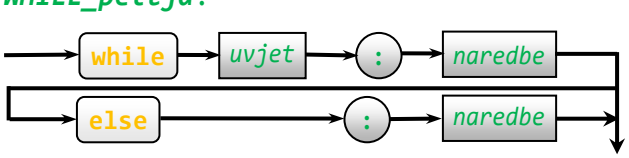
```
 i       i**2       i**0.5
-------------------------
 1          1       1.0000
 2          4       1.4142
 3          9       1.7321
 4         16       2.0000
 5         25       2.2361
```

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:



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)' )
```

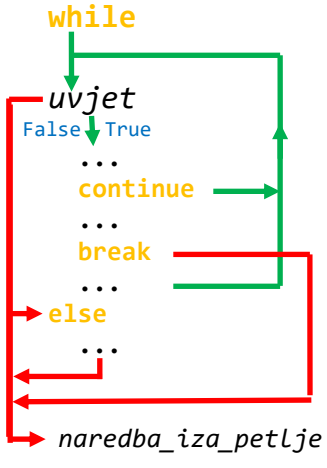
```
>>>
0 1 2 3 4 5 6 7 8 9
 *** (izvan WHILE petlje)
```

```
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) )
```

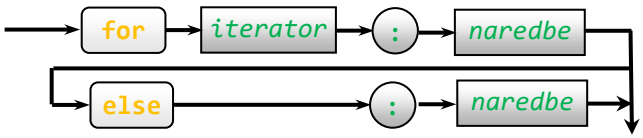
```
>>>
Ispisujem tablicu od 1 do n = 5

 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

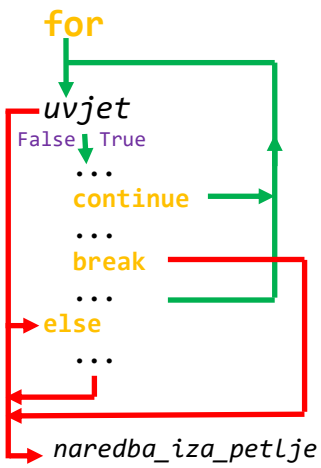
```
from Moj_modul import Input, NL
while 1 :
  try : I = Input('Zadaj domenu iteriranja,'
    +' range([od,] do [,korak]) '); break
  except : print(
      'Pogreška! Ponovi unos.' )
print( I, ' ', len(I), ' iteriranja',
       NL, 'WHILE petlja:', sep = '' )
i, do, k = I.start, I.stop, I.step
while 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 zna-čenje 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
```

MALE TAJNE PROGRAMIRANJA

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!
```

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_5.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! =
933262154439441526816992388562667004907159
682643816214685929638952175999932299156089
414639761565182862536979208272237582511852
109168640000000000000000000000000

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 1200
a1200 =
272698844554062701579916153136421987050007
799929177258211805028949747264763730268094
825092845623100311701723801276272144935976
167438564430160399722058474059176346607504
749145618796567632686585280921957156260732
482240677942538091322190563829391639184000
```

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.py

```
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
```

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)
```

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 progra-mima 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
```

Znamo da je funkcija ord(*Ch*), gdje je *Ch* znak, inverzna 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 'ććđšžĆĆĐŠŽ' :
    print (c, ord(c), end = ' ')
```

```
ć 263 ć 263 đ 273 š 353 ž 382 Ć 262 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 )
```

```
>>>
Ruski alfabet:

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

Primjer:
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 *c*1 i *c*2 dva znaka, vrijedi

```
c1 < c2
```

ako i samo ako je `ord(c1) < ord(c2)`.

6.1 usporedba znakova

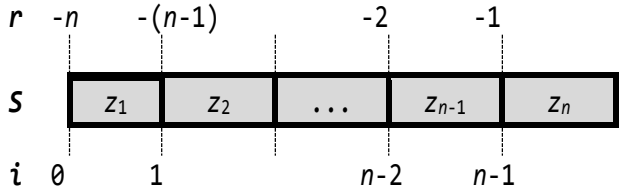
```
>>> 'A'<'B' True    >>> '0'<'1'   True
>>> 'a'<'A' False   >>> '9'<'0'   False
>>> '9'<'+' False   >>> '+'<'-'   True
>>> 'š'<'T' False   >>> '9'<'A'   True
>>> 'a'<'b'<'c'                   True
```

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 (2)

Znakovna varijabla, ili varijabla sa strukturom znakovnog niza, ime je kojem je pridružena vrijednost znakovnog izraza ili znakovni niz *funkcijom INPUT* (v. prvo poglavlje). Sada možemo dodati da se višestruko pridruživanje može upotrijebiti ako se ista vrijednost pridružuje dvijema ili više znakovnih varijabli.

Operatorsko pridruživanje znakovnim varijablama, koje moraju prethodno biti definirane, postiže se pravilom:

```
operatorsko_pr :
  znakovna_var ( += znakovni_izraz |
                 *= cjelobrojni_izaz )
```

6.4 operatorsko pridruživanje

```
>>> P = 'Python'; P += ' je najbolji jezik'
>>> P                  'Python je najbolji jezik'
>>> T = '.'; T *= 10; T                ..........
```

Pristup pojedinoj komponenti znakovne varijable bit će sa:

nv [i]

gdje je ***nv*** varijabla znakovnog tipa. Ponekad ćemo reći da je „stringovna varijabla“. Ako je ***n*** duljina niza znakova sadržanih u ***nv***, ***i*** je cjelobrojni izraz čija je vrijednost (indeks i relativni indeks) iz intervala:

$$-n \le i \le n-1$$

U sljedećoj smo vježbi znakovni niz pridružili varijabli Py i potom ga ispisali, znak po znak.

6.5 ispis znakova znakovnoga niza

```
>>> Py = 'Python'
>>> for i in range (len (Py)) :
    print( Py[i], end = ' ' )
P y t h o n
```

RELACIJE SA ZNAKOVNIM NIZOVIMA

Nad znakovnim nizovima definirane su standardne relacije kao i nad primitivnim tipovima:

relacijski_izraz:
zn_izraz relacija zn_izraz

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)`

6.6 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*.

6.7 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          (True, False)
>>> 'ab' in S, 'bc' in S         (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.

6.8 *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.

6.9 iteriranje

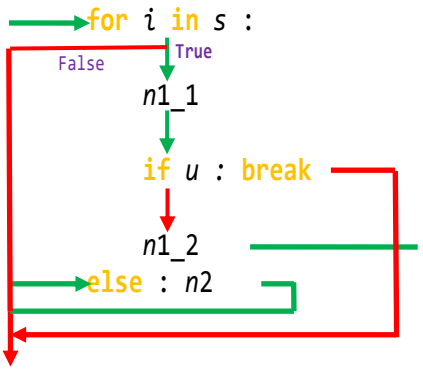
```
>>> A = 'Python'
>>> for x in A :
    print( x, end = ' ' );  A += x
    P y t h o n
>>> x                                   'n'
>>> A                              'PythonPython'
```

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, a 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

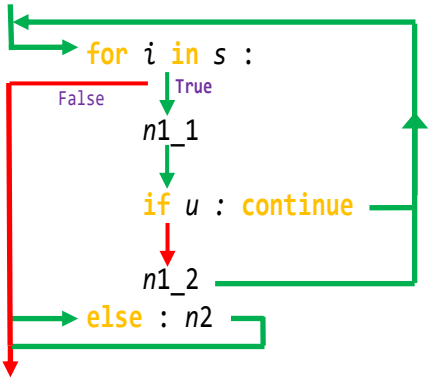
```
S      = input ('Učitaj rečenicu:\n')
IntPun = " .:,;!?-()+-*/'\""; i = 0
while True :
  w = ''
  for s in S[i:] :
    i += 1
    if s in IntPun :
      if w       : print (w, end = ' ')
      if s != ' ': print (s, end = ' ')
      break
    w += s
  else :
    if w : print ( w )
    break
```

```
>>>
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

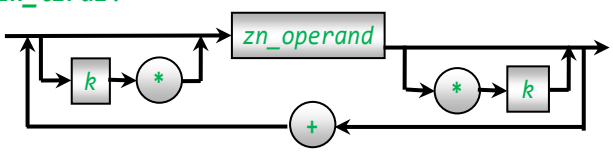
```
S = input( 'Učitaj rečenicu:\n' )
w = ''
for s in S :
  if s == ' ' : continue
  w += s
print( w )
```

```
>>>
Učitaj rečenicu:
Ova rečenica    ima          puno    razmaka.
Ovarečenicaimapunorazmaka.
```

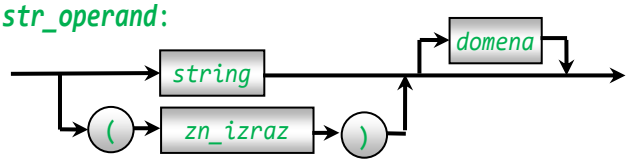
ZNAKOVNI IZRAZI

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

6.10 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]	**= $c_m c_{m+o} \ldots c_{m+i*o}$, i=1,...,(n-m-1)/o** ako je **o>0 and m<n** **= $c_m c_{m+o} \ldots c_{m+i*o}$, 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.

6.11 stringovne funkcije

```
>>> bin (2**16)          '0b10000000000000000'
>>> bin (11111)            '0b10101101100111'
>>> bin (0b111**4)           '0b100101100001'
>>> oct (2**16)                    '0o200000'
>>> 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',
    'swapcase',..., '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 'čćžšđ' :
    print( x.capitalize(), end = ' ' )
    Č 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.5'.lower()                   'abc.5'
>>> 'ČŽĐ'.lower()                     'ćžšđ'
>>> 'abc.123'.lower()               'abc.123'
>>> gr_alf = ('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(gr_alf ); print (gr_alf.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( 'ćšđžČĆŠĐŽ'. swapcase() )
    ČĆŠĐŽćšđž
>>> print( 'ČćĆžŽšŠĐđ'. swapcase() )
    čĆćĆžŽšŠđĐ
```

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'
>>> 'žš'. upper()                        'ŽŠ'
```

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() )       ͼ 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.

6.12 int() i float()

```
>>> int ('1'+'23')                        123
>>> int ('1'+'2'*3)                      1222
>>> int ('1' +'0'*10)              10000000000
>>> int ('12.5')          ValueError: ...
>>> int ('1'+'2*3')       ValueError: ...
>>> 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.

6.13 modul string

```
>>> import string
>>> dir (string)
    [ ..., 'ascii_letters',
    ascii_lowercase', 'ascii_uppercase',
    'capwords', digits', 'hexdigits',
    'octdigits', 'printable', 'punctuation',
    'whitespace']
```

6.14 konstante

```
>>> from string import *
>>> ascii_letters
    'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLM
    NOPQRSTUVWXYZ'
>>> ascii_lowercase
    'abcdefghijklmnopqrstuvwxyz'
>>> ascii_uppercase
    'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> digits                          '0123456789'
>>> hexdigits       '0123456789abcdefABCDEF'
>>> octdigits                        '01234567'
>>> punctuation
    '!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
>>> whitespace                 ' \t\n\r\x0b\x0c'
```

MALE TAJNE PROGRAMIRANJA

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
...
0370–03FF Greek and Coptic
0400–04FF Cyrillic ...
>>> for i in range (0x0370, 0x03FF +1) :
    print( chr (i), end = ' ')
    ͰͱͲͳ ʹ ͵ Ͷͷ 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
>>> ϵ = ϵ_kn = .54124; 125 *ϵ          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 翩 翼 翻 翱 翹 翸 翷 翱 鵡 翵 翳 ...
    羟 粮 義 羨 羧 羦 羟 群 羣 戟 羨 羭 羟 羞 羝
>>> chr (k -1)                                  '翿'
>>> chr (k -255)                                '缁'
>>> '缁' .isalpha()                            True
>>> '胜' .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.
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', 'A')
>>> ord ('A'), ord ('A')          (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 palin-drom. 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

Šest primjera programa danih u ovom poglavlju u dovoljnoj mjeri 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 )
```

```
>>> int ('282ba4aab', 13)          2147483648
>>> int ('282ba4aab', 20)         61604836211
>>> int ('123456789', 36)        984619134745
>>> 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'
```

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

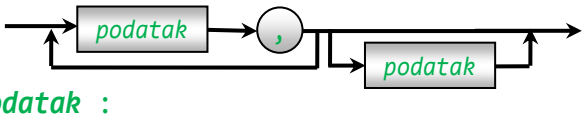
Python ima dvije standardne strukture podataka, n-torke i liste, koje predstavljaju kolekcije uređenih nizova podataka. S obzirom na to da te dvije strukture podataka imaju dosta zajedničkih svojstava, opisujemo ih zajedno u ovom poglavlju. Razlika između njih je što su n-torke nepromjenljiv („immutable"), a liste promjenljiv tip (klasa) podataka.

Znajući da su nizovi važni u programiranju, posebno pružaju „elegantna" rješavanja mnogih problema, dali smo veliki broj programa.

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*

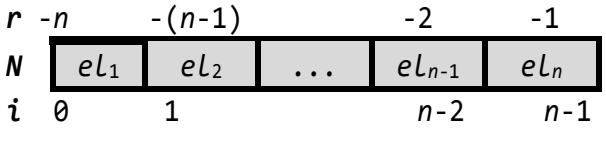
Dva su tipa (klase) podataka sa strukturom niza: *n*-torke i liste:

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:



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']
>>> # "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', 2 )
>>> Y; print( Y )
   (1, False, ('a', 'b'), 3.15, 'python', 2)
   (1, False, ('a', 'b'), 3.15, 'python', 2)
```

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 biti 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, $o \neq 0$, 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*.
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)
```

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)
```

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

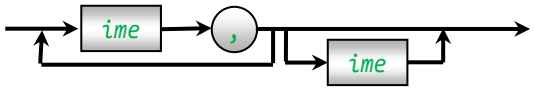
```
>>> A = (1,2,3); B = ('a','b','c')
>>> A; B
    (1, 2, 3)
    ('a', 'b', 'c')
>>> del B
>>> A; B
    (1, 2, 3)
NameError: name 'B' is not defined
>>> L = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> L                [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> del L[1 : len(L)+1 : 2]
>>> L                             [1, 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]
```

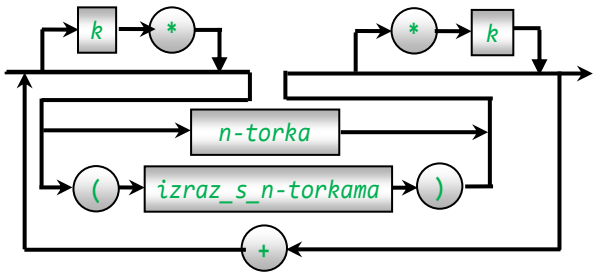
7.9 Konkurentno pridruživanje

```
>>> 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
```

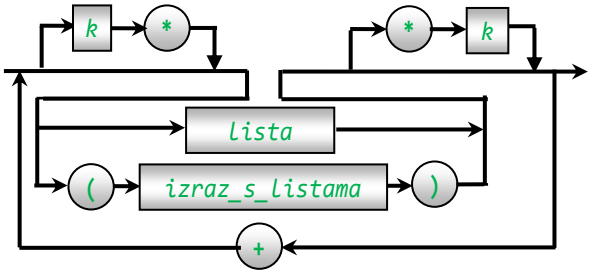
```
>>> Prvi                                  1
>>> Srednji                               []
>>> Zadnji                                100
>>> Y = (1, 50, 100)
>>> Prvi, *Srednji, Zadnji = Y
>>> Prvi                                  1
>>> Srednji                               [50]
>>> Zadnji                                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.

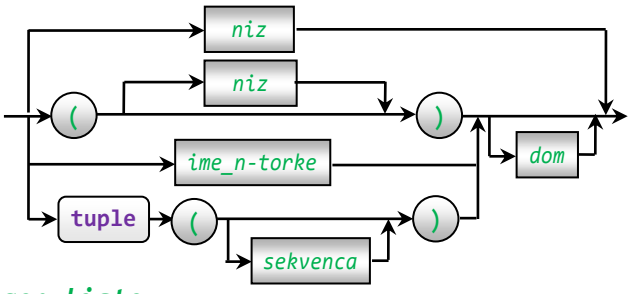
Ako su *X* i *Y* nizovi istoga tipa, značenje operacije + jest nastavljanje niza X nizom Y. Ako treba proširiti n-torku X, moramo rabiti naredbu za pridruživanje u kojoj će izraz s *n*-torkama sadržavati *n*-torku X kao operand. Ako je X prvi operand možemo rabiti operatorsko pridruživanje += i *=.

```
>>> (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)
```

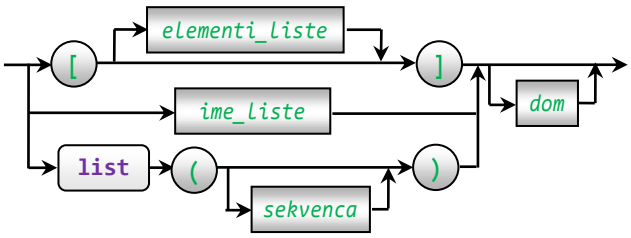
GENERIRANJE NIZA

Potpuna pravila generiranja niza, *n-torke* i liste, dana su sljedećim sintaksnim dijagramima:

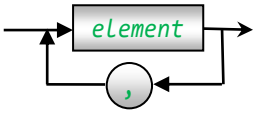
gen_n-torke:



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`.

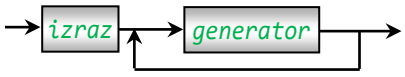
7.12 Generiranje niza

```
>>> 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 NIZA ZNAKOVA IZ STRINGA

Generiranje niza (n-torke ili liste) znakova (ili pretvorba) iz stringa postiže se funkcijom `tuple()` ili `list()` sa stringom kao argumentom:

```
( tuple | list ) ( string ) [ domena ]
```

7.16 Pretvorba stringa u niz znakova

```
>>> tuple ('abcdef'); list ('abcdef')
    ('a', 'b', 'c', 'd', 'e', 'f')
    ['a', 'b', 'c', 'd', 'e', 'f']
>>> tuple ('0123456789') [::2]
    ('0', '2', '4', '6', '8')
>>> list ('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()	Sortirana lista ***L***, u rastućem nizu.

sort(reverse = True**)**	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*** = len(***T***)
ili [$i_1, i_2, \ldots, i_n$] = ***T***, ***n*** = 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 cjel. var.
>>> # 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
```

MALE TAJNE PROGRAMIRANJA

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* ≤ *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 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 '
```

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 = \
    234324234238542383289574328957489047280437280957340895743890573485743534999999999921777777777777777777777777771233166666
>>>
>>> # 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]
```

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)
    30414093201713378043612608166064768844
    77645689605120000000000000
```

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 eurima bez centi. Ideja je bila da se zadani iznos dijeli, počevši od najvećeg apoena (`200 eura`), 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 eurima s najmanjim
# brojem apoena
while 1 :
  C = EU = int (input ('Zadaj vrijednost u '
                       'eurima bez centi '))
  if EU >= 1 : break
N = (200, 100, 50, 20, 10, 5, 2, 1); PL = ()
for n in N :
    x, EU = divmod( EU, n ); PL += (x, )
X  = "%4d"*8 % PL
EU = "%4d"*8 % N
print( "Iznos %d eura platiti sa:" % C )
print( X );  print( "   x"*8 ); print( EU )
```

```
>>>
Zadaj vrijednost u eurima bez centi 288
Iznos 1888 eura platiti sa:
   1   0   1   1   1   1   1   1
   x   x   x   x   x   x   x   x
 200 100  50  20  10   5   2   1
```

Placanje_3.py

```
while 1 :
  EU = int (input ('Zadaj vrijednost '
            'u eurima bez centi '))
  if EU >= 1 : break
print( 'Iznos od %d eura bit će plaćen'
       ' sa:\n' % EU )
N = (200, 100, 50, 20, 10, 5, 2, 1); PL = ''
for n in N : x, EU = divmod (EU, n); \
    PL += "%4d x %d\n"               \
    % (x, n) if x else ''
print ( PL )
```

```
>>>
Zadaj vrijednost u eurima bez centi 288
Iznos od 288 eura bit će plaćen sa:
   1 x 200
   1 x 50
   1 x 20
   1 x 10
   1 x 5
   1 x 2
   1 x 1
>>>
Zadaj vrijednost u eurima bez centi 1001
Iznos od 1001 eura bit će plaćen sa:

 200 x 5
   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 2025. godini. Računa se po 8 sati za sve dane osim subota i nedjelja (4-ti i 5-ti dan u tjednu). Zna se da je 1.1.2025. bila srijeda. Sati će biti generirani u listi RS s indeksom 1 do 12 po mjesecima. k je redni broj dana u godini, od 1 do 31+1, za i=1, od 32+28+1, za i=2, sve do sum(M[:12])+1 i 365+1, za i=12. Zbrajaju se svi dani koji nisu djeljivi s 4 niti s 5.

Radni_sati_2025.py

```
# Radni sati po mjesecima u 2025. godini
M  = [0, 31, 28, 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 2025. godini' )
```

```
>>>
mjesec    :    1   2   3   4   5   6   7
   8   9  10  11  12
rad. sati :   184 160 168 176 176 168 184
 168 176 184 160 184
Ukupno 2088 sati u 2025. 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 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 2025. godine

Znajući da će `1.1.2025.` bila srijeda, za zadani datum `2025.` godine treba ispisati koji je to dan u tjednu i njegov redni broj u godini.

dan_2025.py

```
# Redni broj dana i dan u tjednu datuma
# u 2025. godini
G = 2025; M = [0, 31, 28, 31, 30, 31, 30,
                  31, 31, 30, 31, 30, 31]
D = ('uto', 'sri', 'čet', 'pet',
     'sub','ned','pon')
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 2025. godini"
  % (Datum +(D[d %7], d)) )
```

```
>>>
Upiši dan i mjesec 1, 3
1.3. sub, 60. dan u 2025. 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 $n \le 10$, odnosno kao `J[j]`, gdje je `j` sufiks (jedinice) dvoznamenkastog broja, $0 \le j \le 9$. 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] 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 (4)

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:

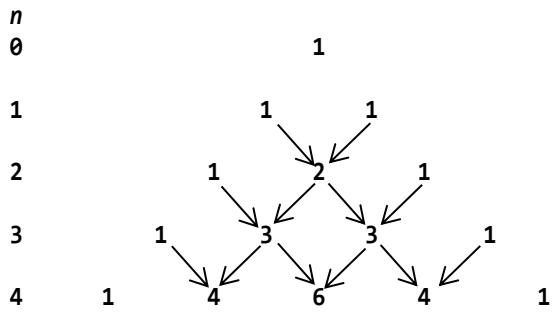
Fibonacci_4.py

```
# 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

 (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]
```

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 ... 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 ... 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 50
2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47
```

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!

DAN U TJEDNU ZADANOG DATUMA

U literaturi se mogu naći formule (programi) za određivanje dana u tjednu za zadani datum. Evo jedne:

```
if M < 3 : M += 12; G -= 1
B = (D +2*M +int (0.6*(M+1)) +G
     +G // 4 -G // 100 +G // 400 +2)/7
n = (int ((B -int(B))*7 + 0.5) +6) % 7
```

gdje su: `M` - mjesec, `G` - godina i `D` - dan zadanoga datuma, `n` - redni broj dana u tjednu (`0` - nedjelja,...).

Dan_datuma.py

```
# Dan_datuma.py
from Moj_modul import *
Mj  = [0, 31, 28, 31, 30, 31, 30,
          31, 31, 30, 31, 30, 31]
Dan = ('ned', 'pon', 'uto', 'sri',
       'čet', 'pet', 'sub')
while True :
  X = input (
      'Zadaj datum u obliku dan, '
      'mjesec, godina ')
  if X == '' : break
  D, M, G = eval (X)
  if G < 1583 :
     print ('Pogrešna godina!'); continue
  Mj [2] = 28 +PG (G) # +1 za prijest.god.
  if 1 <= M <= 12 and 1 <= D <= Mj[2] :
    if M < 3 : M += 12; G -= 1
    B = (D +2*M +int (0.6*(M+1)) +G
         +G // 4 -G // 100 +G // 400 +2)/7
    n = (int ((B -int(B))*7 + 0.5) +6) % 7
    print (Dan [n])
  else : print ('Pogrešan datum!')
```

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.

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.

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')
```

Potom upišimo nekoliko redova funkcijom write() dodavajući `'\n'` na kraj svakog upisa:

```
>>> Poruka = True
>>> 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')
```

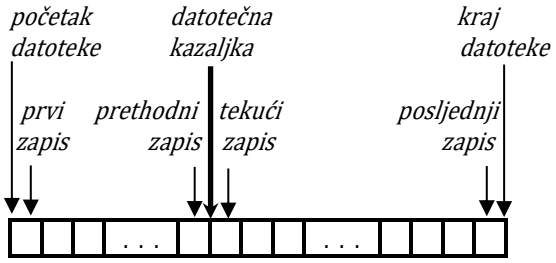
```
Upiši rečenice (Enter za prekid):
1. red                    7
2. red                    7
3. red                    7
<Enter>
```

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 *ƒ* datoteka, postoji sedam atributa nad njom danih u nastavku.

ƒ.closed

Logička vrijednost koja označuje trenutno stanje datoteke. Ovo je atribut samo za čitanje; metoda `close()` mijenja vrijednost.

ƒ.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.

ƒ.errors

Obrađivač pogrešaka Unicode koji se koristi zajedno s kodiranjem.

ƒ.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.

ƒ.name

Naziv datoteke, ako je ime pridruženo stvarnoj datoteci stvoren pomoću open ().

ƒ.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
```

„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 sek-

vence. 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
police = shelve.open ( ime_datoteke )
police ['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
```

Naredba TRY

Često ćemo pokušati učitati podatke s neke datoteke, oformiti ih, ako datoteka ne postoji, ili prekinuti daljnje izvršavanje. Za to ćemo koristiti *naredbu TRY* prema shemi:

```
try    :
  ime = open ('Datoteka', 'r')
  # učitavanje ...
except :
  # upisivanje sadržaja ili poruka
  # i prekid
```

Inače, *naredba TRY* je složena naredba sa znatno širim značenjem. Njezinu sintaksu i semantiku možete naći u on-line Pythonovoj dokumentaciji (`F1->Index...`). Detaljno je opisana i u [Dov2021].

MALE TAJNE PROGRAMIRANJA

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
```

Pjesma

Evo pjesme (šansone) kao još jedan primjer tekstualne datoteke i centriranja 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')
```

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 MALE TAJNE PROGRAMIRANJA i PROGRAMI pokazano je kako se skupovi i mape mogu koristiti u implementaciji nekih algoritama i rješavanju problema iz matematike i kemije.

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 = ' ' )
```

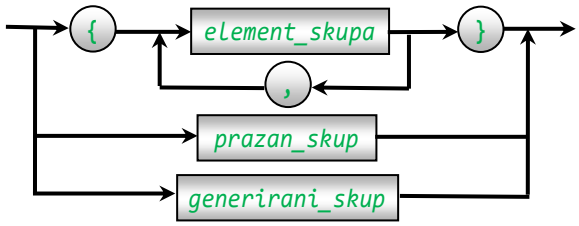
```
>>>
Upiši znakovni niz:
Prebroj sve znakove ove rečenice!

! -->  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

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

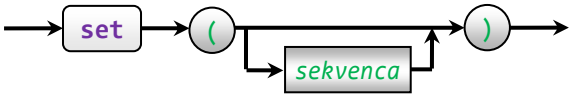
```
>>> X = {1, 2, 3, 4};  Y = {3, 4, 5, 6}
>>> A = set();  A                    set()
>>> ∅ = set(); print( ∅ )            set()
>>> Jezici = {'Python', 'Pascal', 'BASIC',
              'Delphi'}
>>> Jezici
    {'Delphi', 'Pascal', 'Python', 'BASIC'}
```

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( 'abc' )   # string  {'c', 'a', 'b'}
>>> 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, 6 ))        {1, 2, 3, 4, 5}
>>> set(range( 10, 0, -1 ))
    {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
>>> set(range( 2, 11, 2 ))  {2, 4, 6, 8, 10}
```

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    >>> S1 > S2     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
>>> max ({1, 30, -2})                   30
>>> min ({1, 'Python', -5})
    TypeError: '<' not supported between
    instances of 'str' and 'int'
```

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'
```

SKUPOVI, *n*-TORKE I LISTE

Uvođenjem skupova moguće su konverzije u svim smjerovima između ove tri strukture podataka.

```
>>> 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'}
```

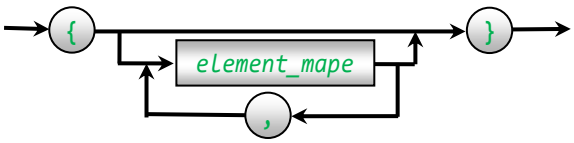
Pretvorbom liste (ili *n*-torke) koja ima dva ili više jednaka elementa u skup, duplikati će biti izbačeni. Na primjer:

```
>>> 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:

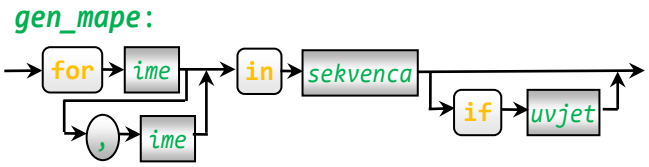
››› 9.12 Generiranje iz sekvence parova

```
>>> 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:



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')]
```

MALE TAJNE PROGRAMIRANJA

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'
```

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__', ...,
    '__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__', ..., '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
```

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.

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
```

PROGRAMI

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')
```

n-TI ČLAN FIBONACCIJEVOG NIZA (5)

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_5.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] )
```

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 izvršava akcija definirana u tablici akcija, mapi Ta. Tablicom prijelaza izbjegnut je veliki broj selekcija u sintaksnoj analizi ulaznog znakovnog niza, a tablica akcija je generator izlaznog izraza.

```
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
```

GEM U TENISU

Na kraju ovog poglavlja dajemo program za simulaciju jednog gema u tenisu. Na početku gema igrači imaju po 0 poena. Dalje su prvi, drugi i treći poen označeni s 15, 30, 40 i Gem ili Igra (Game). Ako su oba igrača dobili po tri poena, rezultat je ***Izjednačenje (Deuce)***. Nakon Izjednačenja rezultat je ***Prednost (Advantage)*** za igrača koji dobije sljedeći poen. Ako isti igrač dobije i sljedeći poen, dobiva gem. Ako, međutim protivnik dobije sljedeći poen, rezultat je ponovo Izjednačenje (Deuce). Igrač mora dobiti dva poena za redom nakon Izjednačenja da bi dobio Gem.

U našoj simulaciji jednog gema igraju igrači X i Y. Dodali smo „rang" igrača X, a to je vjerojatnost njegove pobjede koja može biti u intervalu od 0.4 do 0.6. Ako je označimo s R, vjerojatnost pobjede igrača Y je 1-R. Mapa Poen preslikava dobiveni poen, 1, 2 i 3, u '15', '30'i '40'. Ako jedan od igrača dosegne 4 poena, dobio je igru ako ima više od jednog poena od drugog igrača. Prednost ima igrač s oznakom poena s 'A '. Kod ponovnog izjednačenja igračima je ponovo pridruženo po 3 poena.

Gem.py

```
# Gem u tenisu
from random import random
Poen = { 0: '0 ', 1: '15', 2: '30',
         3: '40', 4: 'A '}

while 1 :
  R = eval (input ('Rang igrača X '
                   '(od 0.4 do 0.6) = '))
  if 0.4 <= R <= 0.6 : break
  print ('Pogrešan rang! Ponovi upis')

X = Y = 0
print ('X  : Y'); print ('0  : 0')
while True :
  if random () < R : X += 1
  else             : Y += 1
  if (X>= 4 or Y>= 4) and abs (X-Y)> 1:
     break
  if X == Y == 4 : X = Y = 3
  print (Poen[X], ':', Poen[Y])
print ('pobjeda X!' if X > Y else
       'pobjeda Y!')
```

```
Rang igrača X (od 0.4 do 0.6) = 0.5
X  : Y
0  : 0
0  : 15
0  : 30
0  : 40
pobjeda Y!
```

```
Rang igrača X (od 0.4 do 0.6) = 0.5
X  : Y
0  : 0
15 : 0
30 : 0
30 : 15
40 : 15
40 : 30
40 : 40
A  : 40
40 : 40
A  : 40
pobjeda X!
```

```
Rang igrača X (od 0.4 do 0.6) = 0.6
X  : Y
0  : 0
0  : 15
0  : 30
15 : 30
30 : 30
40 : 30
40 : 40
A  : 40
pobjeda X!
```

```
Rang igrača X (od 0.4 do 0.6) = 0.6
X  : Y
0  : 0
15 : 0
30 : 0
30 : 15
30 : 30
40 : 30
pobjeda X!
>>>
```

10. Potrogrami i moduli

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 MALE TAJNE PROGRAMIRANJA sadrži dosta analiza i preporuka. Dio PROGRAMI *sadrži rješenja nekih problema s maksimalnom primjenom potprograma.*

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.

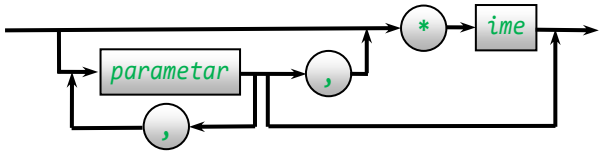
Procedure i funkcije

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".

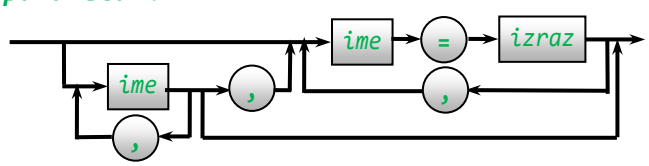
Karakteristika je tih naredbi, osim *LAMBDA funkcija*, da su bile izvršne. Ni potprogram, kojeg sada uvodimo, nije izvršni dio glavnog programa. Definiranje potprograma dano je sljedećim pravilima:

potprogram:
  def *ime_def* ([*parametri* | ***ime*]) :
      naredbe

ime_def : *ime*
parametri:

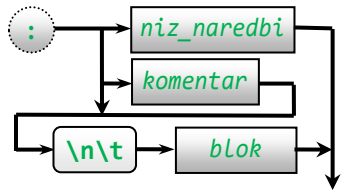


parametar:

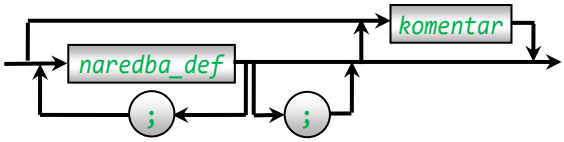


Sintaksnu kategoriju *naredbe* definirali smo ranije kao

naredbe:



niz_naredbi:



naredba_def : *naredba* | *naredba_RETURN*
naredba_RETURN :
return [*izraz* {, *izraz*}]

iz čega zaključujemo da tijelo potprograma ima jednaku strukturu 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, $k \geq 0$.

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.

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); E (1, 2)
    ...
    TypeError: E() missing 1 required
    keyword-only argument: 'y'
    ...
    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

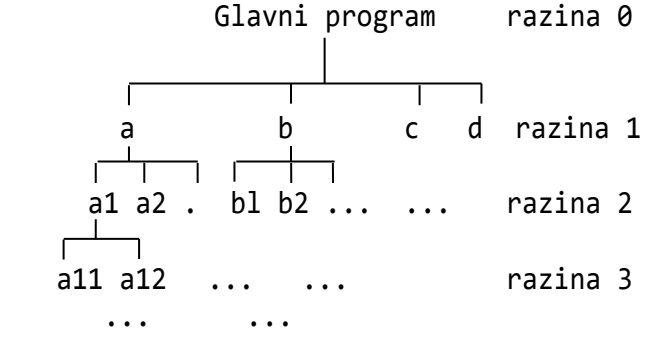
Imena upućuju (pokazuju ili referiraju) na objekt u memoriji. To znači da se imenu pridružuje identifikacija objekta (instance). Na primjer,

```
>>> 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.

KONTEKSTNI ASPEKTI POZIVANJA POTPROGRAMA



Strukturu programa općenito predočavamo stablom gdje čvorovi predstavljaju potprograme. Hijerarhijskom strukturom programa definirani su tekstualni

dosezi potprograma, a istodobno i kontekstni aspekti ili pravila pozivanja potprograma. Pravila pozivanja su 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 ()
 >>>
  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 fun-kcije (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.

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 funkcij*e 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() (2)

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
    ...
    NameError: name 'L' is not defined
```

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', ...,
    'calendar', 'datetime', ...,
    '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__', ..., '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 mje-secu). 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(2022, 10, 22, 19, 43,
    7, 297340)
>>> Datum = datetime.now(); Datum
    datetime.datetime(2022, 10, 22, 19, 43,
    50, 617344)
>>> print (Datum)
    2022-10-22 19:43:50.617344
```

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 (2022, 11, 9);
>>> print (Y.day, Y.month, Y.year)
    9 11 2022
```

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" )
    '22 10 2022 19 43 50 617344'
```

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 = 2022-10-22
```

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: 2022
>>> print ("mjesec:", Danas.month)
    mjesec: 10
>>> print ("dan:", Danas.day)
    dan: 22
>>> x = date.today();  x
    datetime.date(2022, 10, 23)
>>> print (x)                          2022-10-23
>>> 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',
    ..., 'tzname', 'utcoffset']
>>> t = time (15, 16, 29); t; print (t)
    datetime.time(15, 16, 29)
    15:16:29
>>> 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
```

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()`.

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(
    "https://www.google.co")
    True
>>> Url =\
"https://translate.google.hr/?hl=en&tab
=wT#hr/hr/"
>>> webbrowser.open (Url)
    True
```

MALE TAJNE PROGRAMIRANJA

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] )
```

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
```

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
```

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 3. i 10. 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 = 2025; print ('POSLJEDNJE NEDJELJE \n')
for g in range (G, G+2) : Nedjelja(g); print()
```

```
>>>
  POSLJEDNJE NEDJELJE
  2025.  30. 3.  26. 10.
  2026.  29. 3.  25. 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)
```

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 uporabu potprograma i obuhvaćaju gotovo sve dosad opisane naredbe, tipove i strukture podataka.

ISPIS BROJA OD 1 DO 99 SLOVIMA

Sljedeći program ispisuje slovima zadani broj iz intervala [1, 99].

Slovima_2.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>
>>>
```

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

```
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
```

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 2025. do 2030. godine
from calendar import *
G = 2025
print ('CRNI PETAK \n')
for g in range (G, G+6) :
  print ( str (g) +'. ', end = ' ')
  for m in range (1, 13) :
    if weekday (g, m, 13) == 4 :
       print (
       '  13.', str(m) +'.', end = ' ')
  print ()
```

```
>>>
CRNI PETAK

2025.    13. 6.
2026.    13. 2.   13. 3.   13. 11.
2027.    13. 8.
2028.    13. 10.
2029.    13. 4.   13. 7.
2030.    13. 9.   13. 12.
```

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 = 3.5   if T <=  1.0  else  \
    7.0   if T <=  2.0  else  \
    10.5  if T <=  3.0  else  \
    14.0  if T <=  6.0  else  \
    17.5  if T <= 12.0  else  \
    21.0  if T <= 24.0  else  \
    21.0 +( 0 if d <= 1 else 11.0 *(d-1))
dan = 'dana' if D != 1 else 'dan'
sat = 'sati' if S != 1 else 'sat'
print (("Usluga parkiranja za %d " +dan
       +" %d " +sat +" i %d min... %0.2f €")
       % (D, S, M, C) )
```

```
Početak:
dan, mjesec, godina, sat, minuta 12, 3,
2025, 16, 38
Kraj:
dan, mjesec, godina, sat, minuta 12, 3,
2025, 19, 39
Usluga parkiranja za 0 dana 3 sati i 1
min... 14.00 €

Početak:
dan, mjesec, godina, sat, minuta 12, 3,
2025, 16, 38
Kraj:
dan, mjesec, godina, sat, minuta 13, 3,
2025, 16, 39
Usluga parkiranja za 1 dan 0 sati i 1
min... 32.00 €
```

Arapsko-rimsko-arapski rječnik

Napišimo svoj modul za prevođenje arapskih u rimske brojeve ili obrnuto.

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)
```

Prijevodi će biti zapamćeni u mapi `'ARA.dat'`. Modul A_R_A.py možemo uvoziti u programe koji rade s rimskim i arapskim brojevima. Slijede dva programa koji to čine.

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 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>
```

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.

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__
    ', ...,'__xor__', 'bit_length', ...,
    '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 proširivljivijim. 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)
```

MALE TAJNE PROGRAMIRANJA

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 bi, 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.

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'
```

Zbrajanje vektora

Problem izračunavanja rezultante dviju sila riješit ćemo svodeći ga na zbrajanje dvaju vektora, 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.
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.

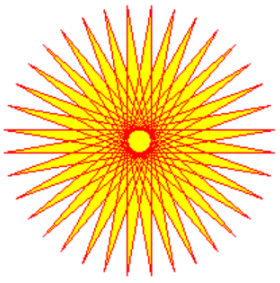
Pythonov modul `turtle` *sadrži jezik kornjačine grafike čije mogućnosti znatno nadilaze navedene prijašnje implementacije. Pruža grafičke primitive kornjača, kako na objektno orijentirani, tako i na proceduralni način, te na strukture i tipove podataka Pythona, na čijim temeljima je postavljena implementacija jezika kornjačine grafike. Kao i dosad, a u ovom poglavlju posebno, pretpostavka je da se sve naredbe i skripte paralelno 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

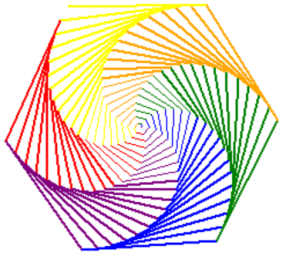
Pythonova kornjačina grafika sadržana je u standardnom modulu `turtle`. Sa svojim procedurama i funkcijama daleko nadilazi Logo i Turbo Pascal. Pogledajte najprije demostraciju nekih mogućnosti otvaranjem i izvršavanjem modula `turtle` (`Alt+M` iz glavnog menu-a editora).

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`

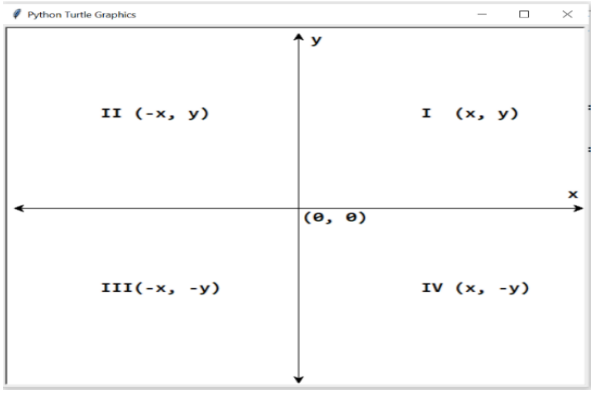
Kornjačin modul **`turtle`** možemo uvesti sa:

```
>>> import turtle
>>> dir (turtle)   [..., 'xcor', 'ycor']
```

Modul **turtle** sadrži veliki broj klasa, funkcija i procedura. 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

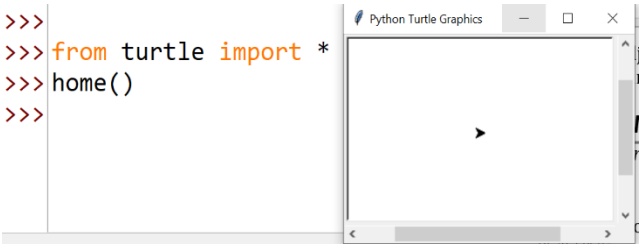
Crta "kornjača" koja, krećući se naprijed, nazad ili pod nekim kutem, ostavlja svoje "tragove" na posebnom zaslonu (prozoru) za crtanje. Zaslon se može zamisliti kao koordinatni sustav s četiri kvadranta, s ishodištem u sredini, kao što je prikazano u sljedećem crtežu.




```
>>> 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*)

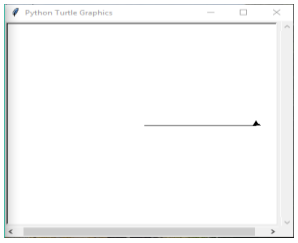
Pomiče pero naprijed za ***p*** *piksela, ako je* ***p>0****, inače unazad* ***p*** *piksela ako je* ***p<0****.*

```
>>> 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****.*

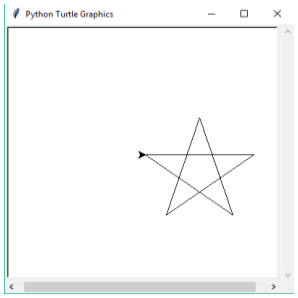
```
>>> forward (150); right (144)
```



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): rt (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
```

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.

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
```

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 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:

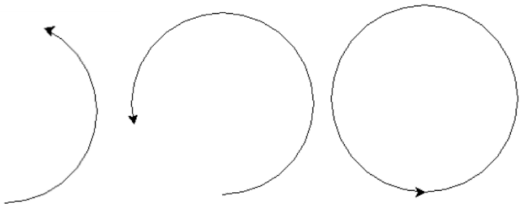
```
>>> color ('red', 'green')
>>> color ()                  ('red', 'red')
>>> fillcolor ()                    'green'
```

circle (*radius, extend* = None, *steps* = None)

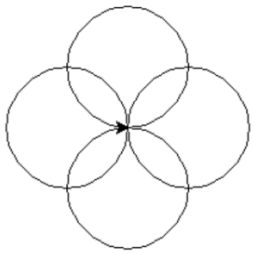
Crta kružnicu s danim radijusom. Zaokreće pero ulijevo (suprotno smjeru okretanja kazaljke na satu).

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.

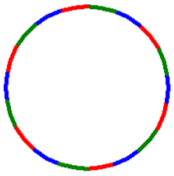
```
>>> circle (100)
```



```
# 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.

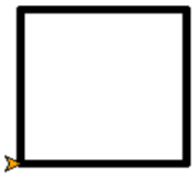
begin_fill ()

Pamti početnu točku poligona koji će biti ispunjen.

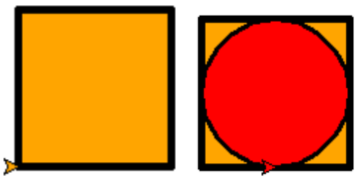
end_fill ()

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)
```



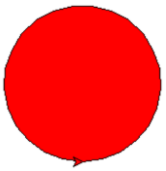
```
>>> end_fill ()
```



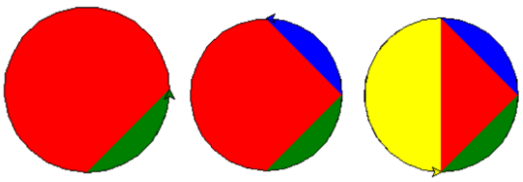
```
>>> fillcolor ('red'); pensize (3)
>>> up (); goto (50, 0); pd ()
>>> begin_fill (); circle (50)
>>> end_fill()
```

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:*

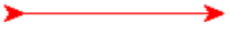
```
  size = max (pensize() +4, 2 *pensize() )
>>> from turtle import *
>>> home ()
>>> dot (); fd (50); dot (20, "blue")
>>> fd(50); position()          (100.00,0.00)
```

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)`.

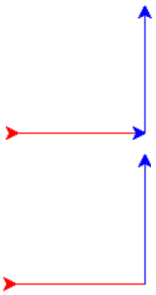
```
>>> color ('red'); stamp ()  11
>>> fd (100)
```



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!

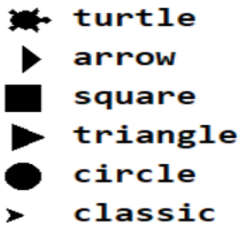
Sada ćemo konačno pokazati kako se pero može promijeniti u lik kornjače i/ili neki drugi standardni ili vlastito dizajnirani lik. I dalje ćemo u tekstu koristiti riječ „kornjača“ umjesto pero, posebno kad se radi o komandama za kretanje (crtanje). Parametri kornjačine grafike (ili varijable) dani su 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
```

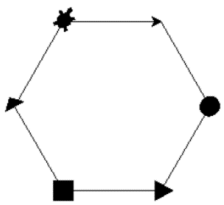
Oblik s imenom mora postojati u rječniku oblika kornjače. U početku postoje sljedeći (standardni) oblici čije su slike prikazane u nastavku.



"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)

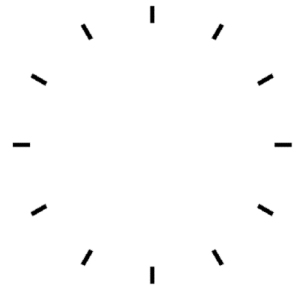
```
>>> from turtle import *
>>> s = Shape ( "compound" )
>>> p = ((0,0),(10,-5),(0,10),(-10,-5))
```

```
>>> 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()
```



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
```

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()
```

UNOS PODATAKA

Jezik kornjačine grafike ima dvije metode za unos podataka: `textinput()` i `numinput()`.

textinput (*title, prompt*)

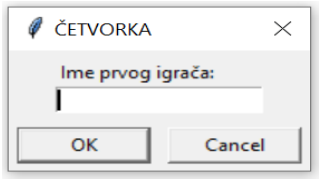
Otvara dijaloški prozor i vraća uneseni znakovni niz. Oba parametra su stringovi. Ako je dijalog otkazan, vraća **None**.

- *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

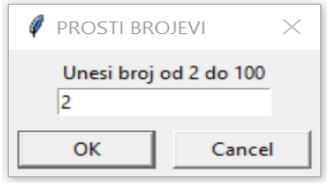
(*title*, *prompt*, default = None, minval = None, maxval = None)

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)
```



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.*

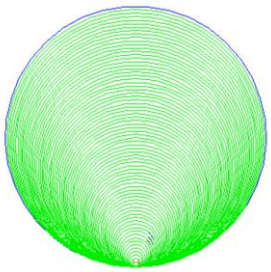
```
>>> home(); delay() # inicijalno          10
>>> delay (100);     delay()              100
>>> delay (10.999);  delay()              10
>>> delay(0);        delay()              0
>>> delay (-10);     delay()              -10
```

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. 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)

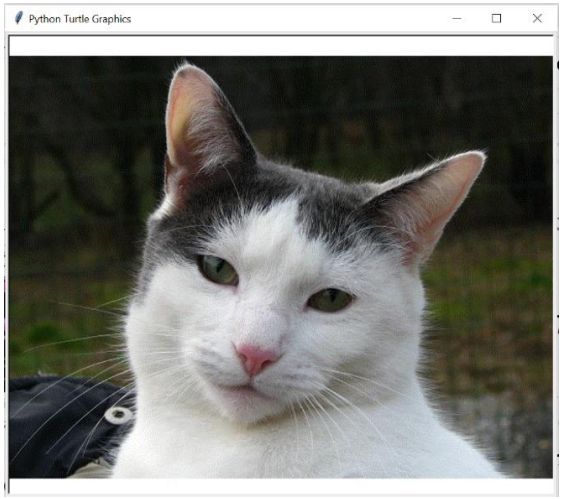
Postavlja pozadinsku sliku ili vraća ime trenutne pozadinske slike.

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 ("Jackues.gif"); print (bgpic ())
```

```
nopic
Jacques.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 ()

Briše sve tragove kretanja kornjače i vraća zaslon na početno stanje.

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
```

MALE TAJNE PROGRAMIRANJA

U ovom smo završnom 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. Kornjačina grafika nas navodi 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.

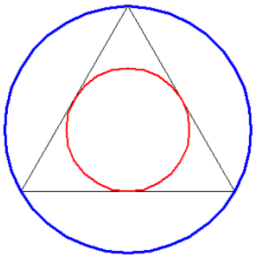
KRUŽNICE TROKUTA

Mnogo je primjera primjene kornjačine grafike u matematici, posebno geometriji. Na primjer u crtanju upisane i opisane kružnice jednakostraničnog tkrokuta zadane stranice.

Kružnice_trokuta.py

```
# Opisana i upisana kružnica
# jednakostraničnog trokuta
from turtle import *;_1 = 50
a  = eval (input ('Zadaj stranicu trokuta '))
a *= _1
for i in range (3) : fd (a); lt(120)
h = a*3**0.5/2
fd (a/2);  width (2); color ("red")
circle (h/3)
up(); goto(a/2, -h/3); pd(); width (3)
color ("blue"); circle (2*h/3)
ht()
```

```
Zadaj stranicu trokuta 5
```

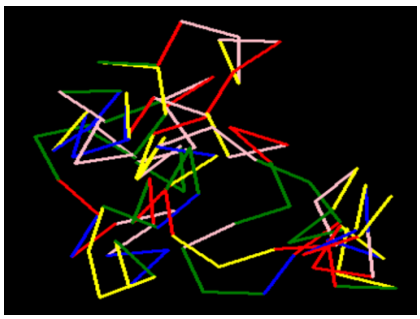


CRTEŽI

Evo nekoliko primjera koji pokazuju kako se s jednostavnim programima mogu dobiti interesantni crteži.

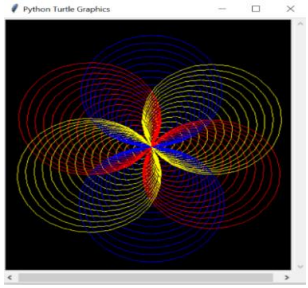
Škrabotina.py

```
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)
```



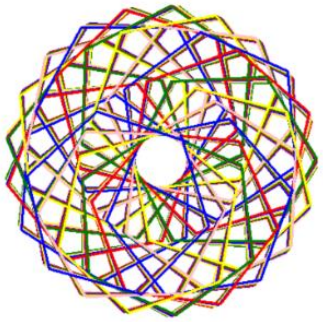
Slika.py

```
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

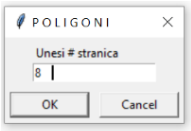
Yin_yang.py

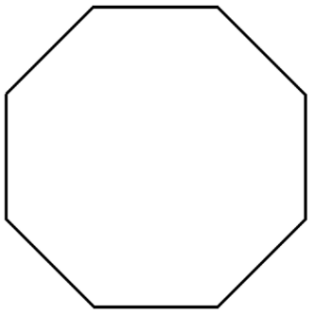
```
from turtle import *
r = 100; r2 = r/2; r3 = r/3
c = circle; width (3)
# crta kružnicu, polumjera r
# "pola" je crno
begin_fill(); c (r, 180); lt(180)
c (-r2, 180); c (r2, 180)
end_fill()
lt (180); c (-r, 180); pu (); ht ()
# crta manje krugove (kao točke)
goto (0, 3*r2); dot (r3)
color ('white')
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 )
```

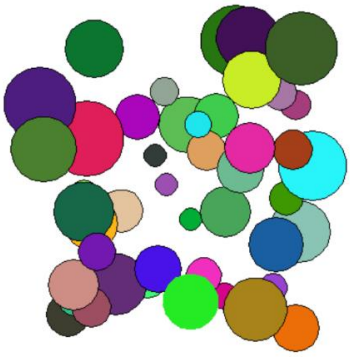




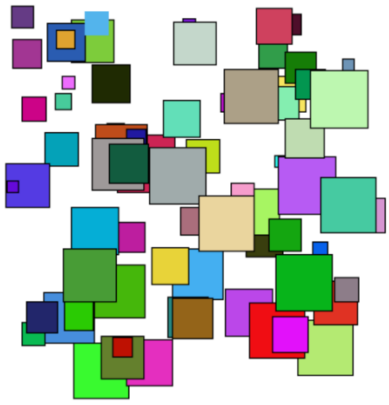
Likovi.py

```
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 stranica; duljine d
  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, ypos = randint(-a,a), 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

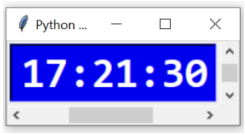
```
from datetime import *
def Pauza (sec) :
  t0 = datetime.now()
  while 'pauza' :
    t  = datetime.now(); T = t -t0
    dT = (T.seconds +T.microseconds/
                     10**6)
    if dT > sec : break
  return
```

DIGITALNI SAT

U sljedećem smo programu pokazali kako se može simulirati digitalni sat 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

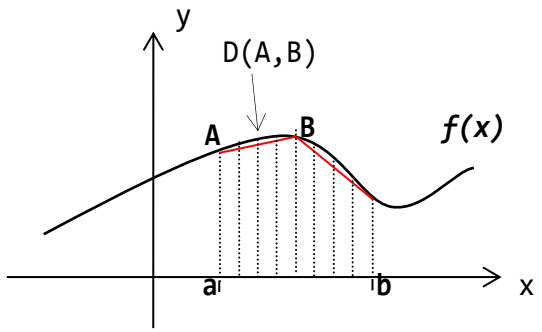
```
from turtle import *
home(); delay(0)
fnt = ("Consolas", 16, "bold")
X, Y = 310, 260
def Označi (a, b, c, d, e, f) :
  goto (xcor() +a, ycor() +b); pd()
  write (c, font = fnt); pu(); goto (d, e)
  pd(); write (f, font = fnt); pu()
for i in range (4) :
  if i in [0, 2] : fd (X)
  else           : 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

U priloženim programima rabljene su gotovo sve metode modula `turtle`. Nastojali smo maksimalno primijeniti i sve ono što smo naučili u prethodnim poglavljima.

Duljina krivulje

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).

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.

Duljina_krivulje.py

```
# Duljina_krivulje.py
from math   import *
from turtle import *
_1 = 50
y  = lambda x : eval (fx)
fx = input ('Upiši f(x) = ')
while 1 :
  a, b =   eval (input ('interval a, b '))
  try :
    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    : (x, y(x))
D = lambda A, B : ((A[0] -B[0])**2
                   +(A[1] -B[1])**2)**0.5

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 += D(A, B); A = B
    x += d
  if L != L0 and abs(L-L0) < P : break
  L0 = L; n *= 2
print ('d =', L)
```

U programu su inicijalne vrijednosti: broj intervala, `n = 2`, duljina krivulje, `L0 = 0`, i preciznost, `P = 1e-5`. Parovi točaka su n-torke `A` i `B` između kojih se računa udaljenost s funkcijom `D(A,B)`. Postupak okončava kad je postignuta preciznost `P`.

```
Upiši f(x) = sqrt (25 -x**2)
interval a, b -5, 6
Pogreška!
interval a, b -5, 5
d = 15.707961285384032
n # broj segmenata
16384
abs(5*pi -L) # odstupanje u odnosu na
  točnu vrijednost (poluopseg, r*π)
1.9825649335558637e-06
```

KRIŽIĆ – KRUŽIĆ

Popularna igra križić-kružić (tic-tac-toe) može nam poslužiti kao pregled mogućnosti primjene strukture stringa. Ako polja matrice 3x3 označimo brojevima:

```
123
456
789
```

možemo definirati string S koji će sadržavati oznake redova, stupaca i dijagonala te matrice:

```
S = '123 456 789 147 258 369 159 357'
```

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

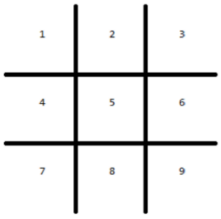
```
3*I in S
```

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.

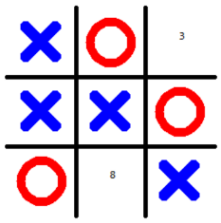
Sada možemo dodati grafiku za ovu popularnu igricu. Prvo igra plavi. Pritišću se brojevi od 1 do 9.

Križić_kružić.py

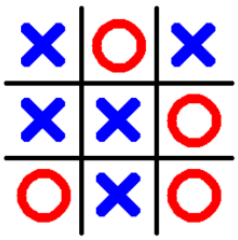
```
from turtle import *
Križ = Shape ( "compound" )
p1   = ((-1,1),(24,26),
        (26, 24),(1, -1), (-1, 1))
p2   = ((24,-1),(-1,24),
        (1, 26),(26, 1), (24, -1))
Križ . addcomponent (p1, "blue", "blue")
Križ . addcomponent (p2, "blue", "blue")
register_shape ( "križić", Križ )
T = [1, 2]; k = 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

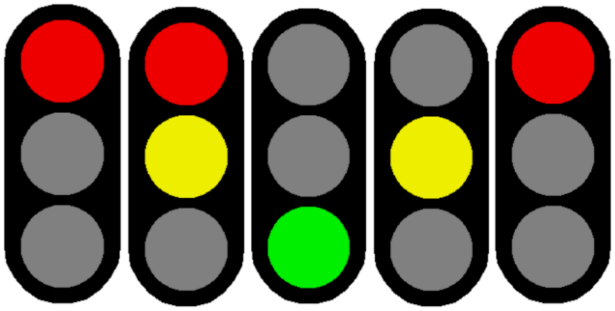


ili neodlučno:



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.

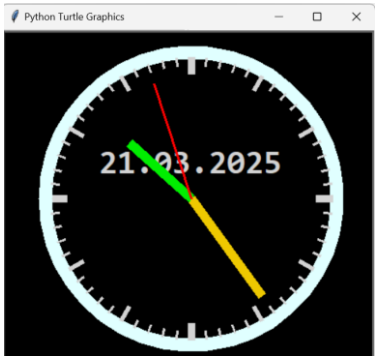


Semafor.py

```
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

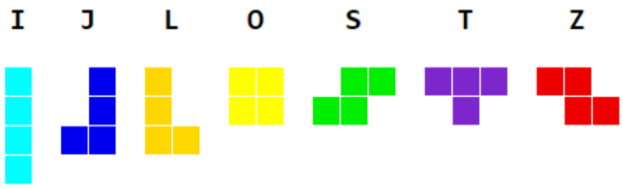
```
from turtle   import *
from datetime import datetime
from winsound import Beep
mode ('logo') # !!!
def Kazaljka (ime, r, d, k, b) :
  X = Turtle(); s = Shape ("compound")
  p = ((d,0),(d, r), (-d,r),
       (-d, 0), (d,0))
  s . addcomponent (p, b, b)
  register_shape (ime, s)
  X.shape (ime); X.degrees (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.tiltangle( m )
         S += m/60; _S.tiltangle( S )
       _s.tiltangle( 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. tiltangle( m0 ); _m.st()
_S. tiltangle( S0 ); _S.st()
_s. tiltangle( s0 ); _s.st()
end_fill(); tracer(1)
onkey   (bye, "Escape")
listen  ()

# Pokreni sat
while True : Otkucavaj ()
```

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.

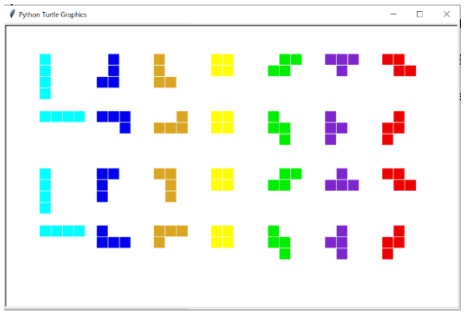


Modul TETRADE.py sadrži sve tetrade u osnovnom obliku i tri rotacije ulijevo za 90 stupnjeva.

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()
```

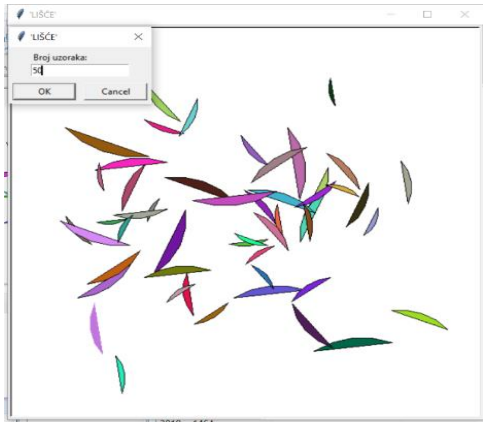


Generiranje "lišća"

Sljedeći program generira „lišće“ u različitim (slučajno odabranim) bojama.

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.

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.

Igrači pritišću brojeve od 0 do 7 koji označuju stupce koji će se popunjavati. 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:

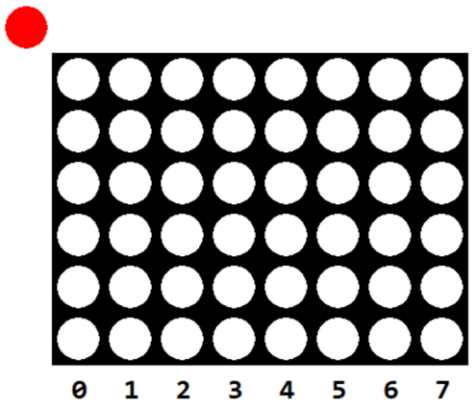
```
Proc = """
def _C () :
  global OK
  if OK : Igra (I, 'C')
onkey (_C, "C")
"""
for C in '01234567' :
  proc = Proc.replace ('C', C)
  exec (proc)
```

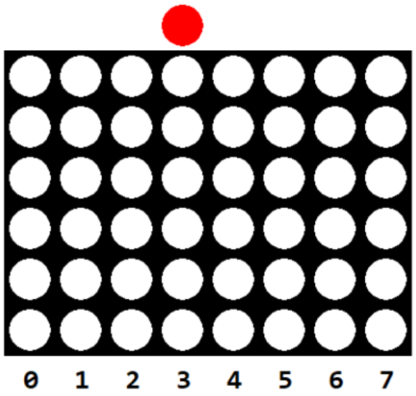
Voilà! Preostaje da priložimo cijeli kôd.

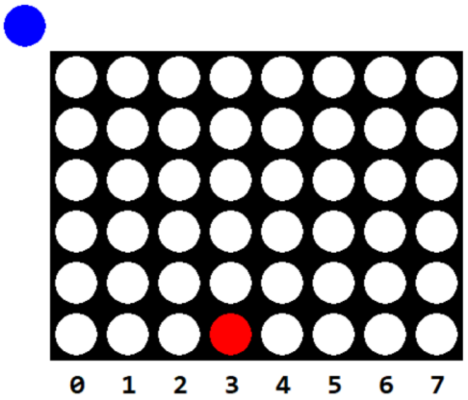
Četvorka.py

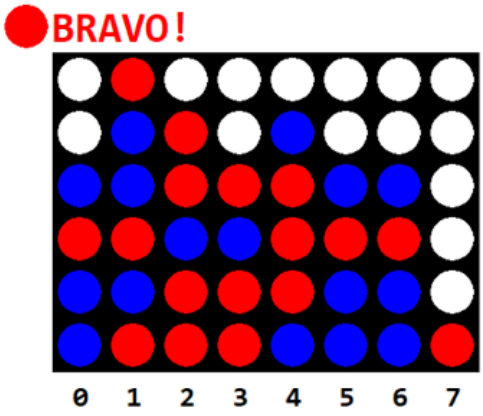
```
# IGRA ČETVORKE
from turtle import *
from time   import sleep
from random import randint
global ST
# parametri
# setup (500, 500, 0, 500)
d = 50
X0, Y0 = -4*d, -2*d
x0, y0 = X0 -d/2, 4*d +d/2
M = 6;  N = 8;  St = [0]*N
T = {}; Q = {}
for m in range (M) : T[m] = list (' '*N)
for n in range (N) : Q[n] = 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.

Tenis

U devetom smo poglavlju simulirali jedan gem tenisa. Sada prilažemo program koji će simulirati kompletnu igru tenisa do 2 ili tri dobivena seta. Set se igra do 6 dobivenih gemova pod uvjetom da je razlika dobivenih gemova veća od 1, odnosno do 7 ako je jedan od igrača poslije rezultata 6:5 dobio sedmi set. Ako je rezultat bio 6:6, igra se sedma igra ili Tie-break.

Tijekom tie-breaka poeni se broje 0, 1, 2, 3, itd. Prvi igrač koji dobije sedam poena dobiva Gem i Set ukoliko njegov protivnik ima barem dva poena manje. Ukoliko to nije slučaj, tie-break se nastavlja sve dok se ne postigne razlika od dva poena između igrača.

Igrač koji je na redu za servirati servira prvi poen tie-breaka. Sljedeća dva poena servira njegov protivnik. Nakon toga, igrači naizmjenično serviraju po dva poena do kraja tie-breaka.

Tenis.py

```
# Igraju X (A) i Y (B). Rang igrača je
# vjerojatnost dobivanja poena.
from turtle import *
from random import *; rnd = random
P = { 0: '0 ', 1: '15', 2: '30', 3: '40',
      4: 'A '}
X  = Y = A = B = a = b = 0
d  = 0.05 # prednost na servisu RX +d
          # ili RX -d ako servira igrač Y
S  = 0    # odigranih setova
Tie = False; Mach = True; End = False
Ix = 'X'; Iy = 'Y' # Igraju X i Y
title ("TENIS"); setup (600,200, 0,0); ht ()
def Semafor (x,a, T1,T2, c0,c1, O = "black"):
    tracer (0)
    begin_fill(); width(2); color (O)
    pu (); setpos (x, 0);  pd()
    for i in range(2):
      fd(a); lt(90); fd(60); lt(90)
    fillcolor (c0); end_fill()
    if x == -120 : x = -135
    color (c1); font = ("Consolas", 20,
                        "bold")
    pu(); setpos (x+20, 30); pd()
    write (T1, False, "left", font)
    pu(); setpos (x+20,  0); pd()
    write (T2, False, "left", font); pu()
Semafor (-250, 130,  Ix,  Iy, "white",
         "black")
Semafor (-120,  20, '*', ' ', "white",
         "orange")
Semafor (-100,  50, '0', '0', "green",
         "white")
Semafor ( -50,  50, '0', '0', "dark violet",
         "white")
Semafor ( 0, 50, '0', '0', "white", "black")
def Igra ():
  global X, Y, x, A, B, a, b, d, S, Tie, \
         Mach, End
  if not Mach:
     if not End : End = True
     return
 def Set () :
    global A, B, X, Y, S, Mach
    if A > B : X += 1
    else     : Y += 1
    S += 1; Mach = X != N and Y != N
    Semafor (-100,       50, str(X), str(Y),
             "green", "white")
    Semafor (-100 +S*50, 50, str(A), str(B),
             "blue",  "white")
    if not Mach :
       Semafor (-120,       20, ' ', ' ',
                "white", "white")
       Semafor (-48 +S*50, 100, ' ', ' ',
                "white", "white", "white")
       return
    Semafor ( -50 +S*50,   50, '0',    '0',
             "dark violet", "white")
    Semafor (   0 +S*50,   50, '0',    '0',
             "white",       "black")
  def Servis (d) : # promjena serviranja
      d1 = '*' *(d > 0); d2 = '*' *(d < 0)
      Semafor (-120,  20, d1, d2, "white",
               "orange")
  p = rnd(); q = 'X' *(d>0) +'Y'*(d<0)
  if p < RX +d : a += 1; Q = 'X'
  else         : b += 1; Q = 'Y'
  if (not Tie and ((a >= 4 or b >= 4) and
      abs (a-b) > 1)) \
      or  (Tie and ((a >= 7 or b >= 7) and
      abs (a-b) > 1)) :
     A += 1*(a > b); B += 1*(b > a)
     a = b = 0;
     if Tie and abs(A-B) == 1 or (not Tie
        and ((A >= 6 or B >= 6) and
        abs (A-B) > 1)) :
        Set ()
        if not Mach : return
        A = B = 0
     if not Tie :
        Semafor (-50 +S*50, 50, str(A),
             str(B), "dark violet", "white")
        Semafor (0   +S*50, 50, '0', '0',
             "white", "black")
        d = -d; Servis (d)
     Tie = False
  if not Tie and (A == B == 6) :
     Tie = True;
     a = b = 0 # tie break
  if not Tie :
     if a == b == 4 : a = b = 3
     Semafor (S*50, 50, P[a], P[b], "white",
              "black")
  else :
     Semafor (S*50, 50, a, b, "white",
              "red")
     if (a+b) % 2 : d = -d; Servis(d)
# Poen se dobiva poslije "serviranja",
# s "Enter".
onkey (Igra, "Return")
```

```
while True :
  N = int (textinput ("TENIS",
                      'Igra se do 2 ili 3'))
  if N in {2, 3} : break

# RX rang prvog igrača od 0.2 do 0.8. Drugi
igrač ima ranga 1 -RX
while True :
  RX = float (textinput ("TENIS",
          'Rang prvog igrača (0.2 - 0.8) '))
  if 0.2 <= RX <= 0.8 : break

listen(); mainloop()
```

Poslije zadavanja do koliko se dobivenih setova igra (2 ili 3) i zadavanja „ranga“ Rx igrača X (rang igrača Y je 1 -Rx) evo nekoliko poena u prvom gemu prvog seta:

X	*	0	0	0
Y		0	0	0

X	*	0	0	15
Y		0	0	0

X	*	0	0	15
Y		0	0	40

X		0	0	0
Y	*	0	1	0

X		0	5	40
Y	*	0	4	30

Stanje u 10-tom gemu, igrač Y ima prednost:

X		0	5	40
Y	*	0	4	A

Igrač X je izjednačio i ima prednost:

X		0	5	A
Y	*	0	4	40

Igrač X je iskoristio gem-set loptu i poveo s 1:0:

X	*	1	6	0	0
Y		0	4	0	0

Pogledajmo stanje u osmom gemu drugog seta. Igrač X ima tri set lopte (na servis igrača Y)!:

X		1	6	5	40
Y	*	0	4	2	0

X		2	6	6
Y		0	4	2

Pobjeda igrača X u dva dobivena seta sa 6:4 i 6:2!

U drugom „polufinalnom“ meču igrači su izjednačeni i poslije reziltata 6:6 u prvom setu, igraju tie-break:

X	*	0	6	0
Y		0	6	0

Igrač X ima tri set lopte:

X		0	6	6
Y	*	0	6	3

i dobiva prvi set poslije treće lopte. Igrač Y je dobio drugi set i igra se deveti gem trećeg seta:

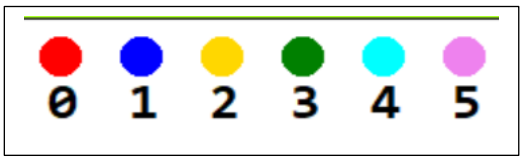
X		1	7	4	3	30
Y	*	1	6	6	5	40

Igrač Y je pobijedio sa rezultatom 7:6, 4:6 i 3:6!

X		1	7	4	3
Y		2	6	6	6

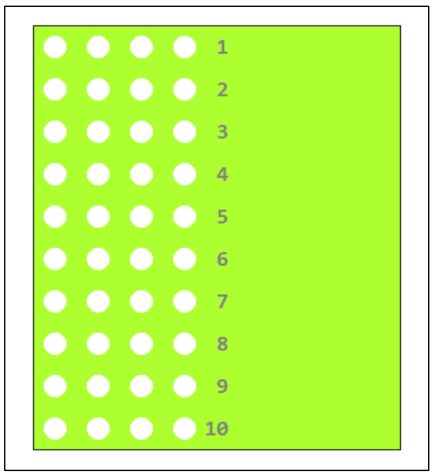
MasterMind

Mastermind („glavni um“) igra je pogađanja kombinacija za dva igrača. Modernu igru s čiodama izmislio je, 1970. godine, Morcedal Meirowitz, upravnik pošte i ekspert u oblasti telekomunikacija. Treba pogoditi boju i mjesto čioda koje postavlja prvi igrač, a nevidljivo je za drugog igrača. Ima ukupno šest boja. Označili smo ih brojkama 0 do 5:

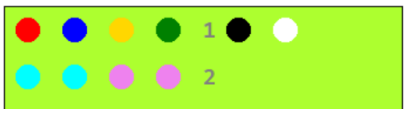


Boje se mogu ponavljati (npr. sve četiri jednake ili dvije jednake, a dvije različite itd.). Drugi igrač postavlja u rupice redom po četiri čiode u izabranim bojama. Poslije postavke prvi igrač mu daje obavijest koliko ima pogođenih i na pravom mjestu, a koliko pogođenih boja, ali na pogrešnom mjestu. Za to se koriste crna čioda za pravo mjesto i bijela za pogrešno.

Na primjer, prvi igrač je postavio jednu zelenu i tri žute čiode, nevidljive za drugog igrača.



Drugi igrač je postavio redom boje 0 1 2 3 i dobio odgovor jedna crna (boja na pravom mjestu) i jedna bijela, a poslije 4 4 5 5 nema pogođene boje.



Slijedi zaključak da su postavljene dvije boje, po dvije od svake ili tri od jedne i jedna od druge. Poslije provjere boja 0 i 1 zaključujemo da su u igri boje 2 i 3. Pitamo se koliko je boja 2 pa u redu 4 postavljamo četiri takve boje. Iz odgovora da su 3 na pravom mjestu zaključujemo da tražimo tri boje 2 (žute) i jednu 3 (zelena).

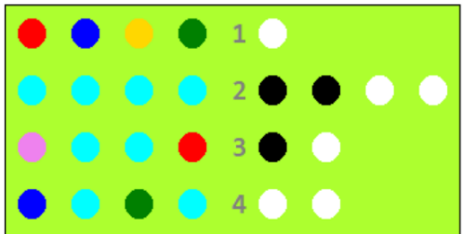
Postavljamo četiri žute i poslije odgovora da su 3 na pravom mjestu i iz reda 1 zaključujemo da su to treće i četvrto mjesto, a treća je na prvom ili drugom mjestu. Iz petog reda zaključujemo da je na drugom mjestu, a zelena na prvom i pogađamo!

Mastermind.py

```
# Mastermind.py
from   turtle import *
from   random import *
global I, J, Z, B, END, b, T, O, Q
# parametri
setup (500, 1000, 0, 0)
d = 40
X0, Y0 = -4.5*d, 6*d
x0, y0 = X0 -d/2, 4*d +d/2
M = 11;  N = 4
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)]
b = ('red', 'blue', 'gold', 'green',
     'cyan', 'violet')
def Prikaz (I, J, k) :
    color (b[k])
    goto (X[I], Y[J]); stamp()
def _Y () :
   for i in range(4): Prikaz (i, 0, Z[i])
def _X () :
   for i in range(4): color ('white'); \
     goto (X[i], Y[0]); stamp()
def Write (x, y, s, c = 'black', f = 20) :
    pu(); goto (x, y); color (c); pd()
    write(s, False, "left", ("Consolas",
          f, "bold"))
    pu()
def Igra (k) :
  global I, J, B, Z, END, b, T, O, Q
  if END : return
  if I < 4 :
    B.append (k)
    shape ('circle'); shapesize (1, 1)
    Prikaz (I, J, k)
```

```
    I += 1
    if I > 3 :
      for i in range (4) :
        y = B[i]
        if y == Z[i] : T += 1
        elif y in Z  : O += 1
      P = [] +['black']*T +['white']*O
      for i in range (len (P)) :
          color (P[i])
          goto (X[3] +d/2 +d*(i+1), Y[J]);
          stamp()
      if T == 4 :
        Write (X[3] +d, Y[0]-d/2,
               'BRAVO!', 'red', 30)
        for i in range(4) : Prikaz(i, 0, Z[i])
        END = True
      Q += 1
      if Q == 10 :
         _Y ()
         Write (X[0] -2*d, Y[0] +d,
           'Nisi pogodi(o/la) iz 10 '
           'pokušaja!', 'red', 18)
         END = True
      I = 0; J += 1; B = []; T = O = 0

def Ploča () :
    pu(); goto (4*d, Y0-d); pd()
    tracer(0); fillcolor ("green yellow")
    begin_fill()

    for _ in range(2) :
      rt(90); fd ( 10 *d); rt(90); fd (8.5 *d)
    end_fill()
    pu (); goto (d, Y0)
    color ('white'); pu()
    shape ('circle'); shapesize (1, 1)

    for i in range (N) :
      for j in range (M) :
          goto (X[i], Y[j]); stamp()
    up(); color ('black')
    x = X0 +4*d; y = Y0-1.8*d

    for j in range (1, 11):
      Write(x, y, "%2s" % str(j), 'gray', 15)
      y -= d
    pu (); x = X0 +2*d
    shape ('circle'); shapesize (1, 1)

    for i in range (6) :
        color (b[i])
        goto (x +i*d, -5.5*d); stamp()
        Write (x -5 +i*d, -6.5*d, str(i))
    up()

Proc = """
def _C () :
  global I, J, B, Z, END, b, T, O
  Igra (C)

onkey (_C, "C")
"""
for C in '012345' :
  proc = Proc.replace ('C', C)
  exec (proc)

onkey (_Y, 'Y'); onkey (_X, 'X')
END = False
I = 0; J = 1; B = []; T = O = Q = 0
pu (); ht(); Ploča ()
Ok = textinput ("MASTERMIND", 'Zadaješ boje'
                              ' (d/n)')
if Ok[0].upper() == 'D' :
    Z = eval (textinput
        ("MASTERMIND",
         'Zadaj boje (b1, b2, b3, b4)'))
else : Z = [randint (0, 5)
            for i in range (4)]
listen(); mainloop()
```

Na kraju, analizirajmo sljedeći primjer:



Da li se iz prva četiri reda moglo jednoznačno zaključiti koje je rješenje (peti red)?

Nazivi svih boja

Na kraju dajemo nazive svih boja koje se mogu koristiti kao stringovi u aplikacijama. Primjeri:

```
'deep sky blue' 'green3' 'green yellow'
```

snow	deep sky blue	gold	seashell3	SlateBlue2	LightBlue3
ghost white	sky blue	light goldenrod	seashell4	SlateBlue3	LightBlue4
white smoke	light sky blue	goldenrod	AntiqueWhite1	SlateBlue4	LightCyan2
gainsboro	steel blue	dark goldenrod	AntiqueWhite2	RoyalBlue1	LightCyan3
floral white	light steel blue	rosy brown	AntiqueWhite3	RoyalBlue2	LightCyan4
old lace	light blue	indian red	AntiqueWhite4	RoyalBlue3	PaleTurquoise1
linen	powder blue	saddle brown	bisque2	RoyalBlue4	PaleTurquoise2
antique white	pale turquoise	sandy brown	bisque3	blue2	PaleTurquoise3
papaya whip	dark turquoise	dark salmon	bisque4	blue4	PaleTurquoise4
blanched almond	medium turquoise	salmon	PeachPuff2	DodgerBlue2	CadetBlue1
bisque	turquoise	light salmon	PeachPuff3	DodgerBlue3	CadetBlue2
peach puff	cyan	orange	PeachPuff4	DodgerBlue4	CadetBlue3
navajo white	light cyan	dark orange	NavajoWhite2	SteelBlue1	CadetBlue4
lemon chiffon	cadet blue	coral	NavajoWhite3	SteelBlue2	turquoise1
mint cream	medium aquamarine	light coral	NavajoWhite4	SteelBlue3	turquoise2
azure	aquamarine	tomato	LemonChiffon2	SteelBlue4	turquoise3
alice blue	dark green	orange red	LemonChiffon3	DeepSkyBlue2	turquoise4
lavender	dark olive green	red	LemonChiffon4	DeepSkyBlue3	cyan2
lavender blush	dark sea green	hot pink	cornsilk2	DeepSkyBlue4	cyan3
misty rose	sea green	deep pink	cornsilk3	SkyBlue1	cyan4
dark slate gray	medium sea green	pink	cornsilk4	SkyBlue2	DarkSlateGray1
dim gray	light sea green	light pink	ivory2	SkyBlue3	DarkSlateGray2
slate gray	pale green	pale violet red	ivory3	SkyBlue4	DarkSlateGray3
light slate gray	spring green	maroon	ivory4	LightSkyBlue1	DarkSlateGray4
gray	lawn green	medium violet red	honeydew2	LightSkyBlue2	aquamarine2
light grey	medium spring green	violet red	honeydew3	LightSkyBlue3	aquamarine4
midnight blue	green yellow	medium orchid	honeydew4	LightSkyBlue4	DarkSeaGreen1
navy	lime green	dark orchid	LavenderBlush2	SlateGray1	DarkSeaGreen2
cornflower blue	yellow green	dark violet	LavenderBlush3	SlateGray2	DarkSeaGreen3
dark slate blue	forest green	blue violet	LavenderBlush4	SlateGray3	DarkSeaGreen4
slate blue	olive drab	purple	MistyRose2	SlateGray4	SeaGreen1

SpringGreen2	DarkGoldenrod1	brown4	pink3	purple1	gray26	gray64
SpringGreen3	DarkGoldenrod2	salmon1	pink4	purple2	gray27	gray65
SpringGreen4	DarkGoldenrod3	salmon2	LightPink1	purple3	gray28	gray66
green2	DarkGoldenrod4	salmon3	LightPink2	purple4	gray29	gray67
green3	RosyBrown1	salmon4	LightPink3	MediumPurple1	gray30	gray68
green4	RosyBrown2	LightSalmon2	LightPink4	MediumPurple2	gray31	gray69
chartreuse2	RosyBrown3	LightSalmon3	PaleVioletRed1	MediumPurple3	gray32	gray70
chartreuse3	RosyBrown4	LightSalmon4	PaleVioletRed2	MediumPurple4	gray33	gray71
chartreuse4	IndianRed1	orange2	PaleVioletRed3	thistle1	gray34	gray72
OliveDrab1	IndianRed2	orange3	PaleVioletRed4	thistle2	gray35	gray73
OliveDrab2	IndianRed3	orange4	maroon1	thistle3	gray36	gray74
OliveDrab4	IndianRed4	DarkOrange1	maroon2	thistle4	gray37	gray75
DarkOliveGreen1	sienna1	DarkOrange2	maroon3		gray38	gray76
DarkOliveGreen2	sienna2	DarkOrange3	maroon4		gray39	gray77
DarkOliveGreen3	sienna3	DarkOrange4	VioletRed1		gray40	gray78
DarkOliveGreen4	sienna4	coral1	VioletRed2		gray42	gray79
khaki1	burlywood1	coral2	VioletRed3		gray43	gray80
khaki2	burlywood2	coral3	VioletRed4		gray44	gray81
khaki3	burlywood3	coral4	magenta2		gray45	gray82
khaki4	burlywood4	tomato2	magenta3		gray46	gray83
LightGoldenrod1	wheat1	tomato3	magenta4		gray47	gray84
LightGoldenrod2	wheat2	tomato4	orchid1		gray48	gray85
LightGoldenrod3	wheat3	OrangeRed2	orchid2		gray49	gray86
LightGoldenrod4	wheat4	OrangeRed3	orchid3		gray50	gray87
LightYellow2	tan1	OrangeRed4	orchid4		gray51	gray88
LightYellow3	tan2	red2	plum1		gray52	gray89
LightYellow4	tan4	red3	plum2		gray53	gray90
yellow2	chocolate1	red4	plum3		gray54	gray91
yellow3	chocolate2	DeepPink2	plum4		gray55	gray92
yellow4	chocolate3	DeepPink3	MediumOrchid1		gray56	gray93
gold2	firebrick1	DeepPink4	MediumOrchid2		gray57	gray94

Reference

[Dij1976] Dijkstra, E.W.: *A Discipline of Programming*, Prentice-Hall, 1976.
[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.
[Dov2021] Dovedan, H. Z.: *progovorimo pythonski*, Zdravko Dovedan Han, Zagreb, 2021.

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] Timsort https://en.wikipedia.org/wiki/Timsort

Programi po poglavljima

12

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) na TVA Zagreb. 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 je 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 pet knjiga, 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)
FORMALNI JEZICI I PREVODIOCI – regularni izrazi, gramatike, automati, Zagreb (2012)
FORMALNI JEZICI I PREVODIOCI – sintaksna analiza i primjene, Zagreb (2012)
FORMALNI JEZICI I PREVODIOCI – prevođenje i primjene, Zagreb (2013)
progovorimo pythonski, Zagreb (2021)

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.