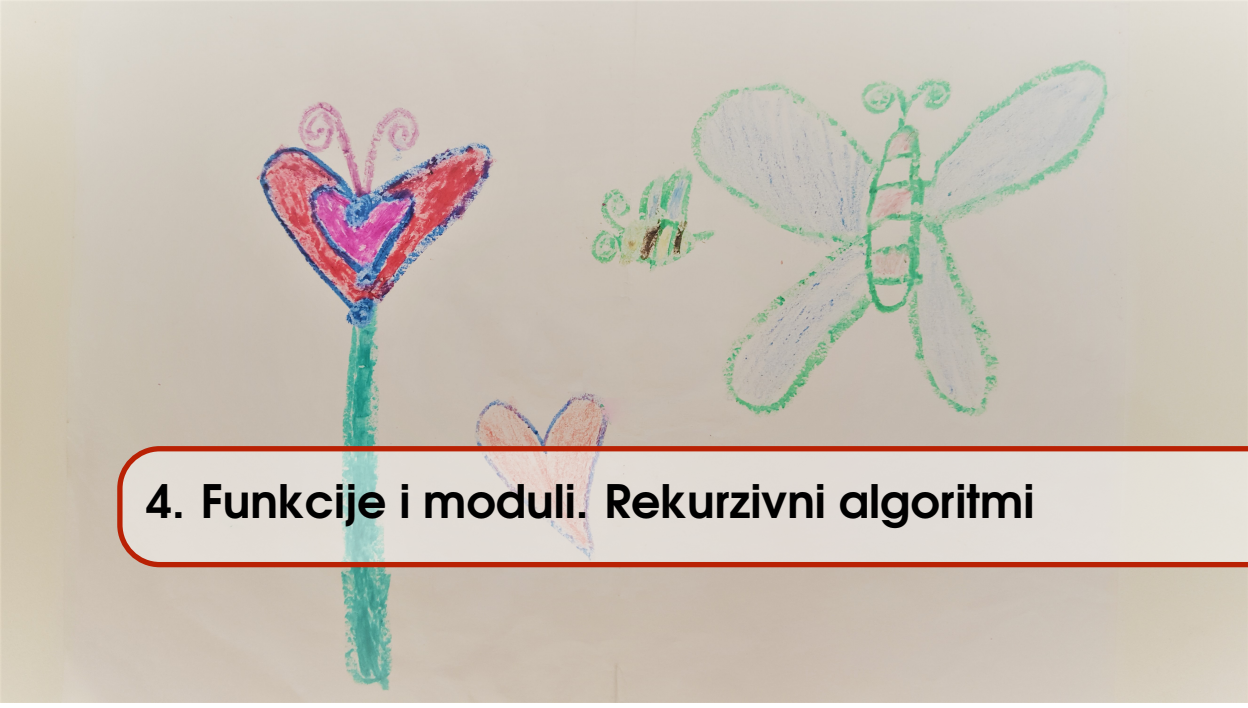


4. Funkcije i moduli. Rekurzivni algoritmi

U dosadašnjem tekstu izloženi su neophodni jezički elementi pomoću kojih se mogu opisati proizvoljni postupci izračunavanja. Međutim, kako su sistemi i procesi koji se modeluju računarskim programom često veoma složeni, uvedeni mehanizmi grananja i repetitivnosti nisu dovoljni za efikasnu realizaciju kompleksnih modela. Ova glava izučava fundamentalne pojmove računarskih nauka - *dekompoziciju* i *apstrakciju*, pomoću kojih se složeniji problemi lakše rešavaju.

Dekompozicija sistema podrazumeva njegovu podelu na *manje*, *jednostavnije* i *nezavisne* delove, koji se lakše modeluju u odnosu na celinu. Ona se, u kontekstu programiranja, može posmatrati na različitim nivoima, poput *sistemskog* (klase i objekti), *organizacionog* (biblioteke, moduli) ili *funkcionalnog* nivoa (korisnički definisane funkcije). U ovoj glavi razmatra se *funkcionalna dekompozicija*, koja podrazumeva podelu složenijih postupaka izračunavanja na manje procesne celine - *funkcije*.

Apstrakcija predstavlja pojednostavljenje sistema ili procesa, na ciljni model koji zadržava samo potrebne osobine, za dati kontekst primene. Funkcija predstavlja apstrakciju određene funkcionalnosti. Korisnik funkcije nema potrebu da zna detalje njene implementacije, već samo šta ona radi i kako se poziva.

Pored funkcija, razmatraće se i organizacija programa po *modulima* i *paketima* - *organizaciona dekompozicija*. Nešto kasnije, u glavi 10, izučavaće se *sistemska dekompozicija*, zasnovana na upotrebi korisnički definisanih klasa.

4.1 Korisničke funkcije

Izlaganje započinje primerom koji objašnjava potrebu uvođenja korisničkih funkcija. Prvo se izlaže način definisanja funkcije, a potom će biti reči o povratnim vrednostima, imenovanim i opcionim parametrima i opsegu važenja promenljivih.

4.1.1 Definisanje funkcije

Posmatra se umetnički svet ispunjenih pravougaonika. Slike se iscrtavaju linija po linija, uz pomoć simbola ispune, poput ∗ ili + (problem 3.5). Treba nacrtati sledeću sliku:

```
         ***
         ***
```

Iznad je iscrtan pravougaonik sa stranicama od dve i tri zvezdice, sa početkom u desetoj koloni. Koristeći mogućnost funkcije `print()` da nastavi ispis u *istom* redu (`end = ''`), kao i mogućnost formiranja *ponavljajućeg* teksta, sa `n * txt` (`n` - broj ponavljanja, `txt` - tekst koji se ponavlja), formiran je sledeći program:

```
# crta pravougaonik dimenzija a x b
# koristi simbol * za ispunu pravougaonika
# gornja leva * pravougaonika u k-toj koloni tekućeg reda
a = 2
b = 3
k = 10
for linija in range(a):
    print((k-1) * ' ', end='')
    print(b * '*')
```

Neka sada, usled povećanog umetničkog zanosa, treba da se nacrta sledeća slika:

```
           ***
           ***
         |||||||
         |||||||
         |||||||
         |||||||
           +++
           +++
           +++
           +++
```

Prateći logiku prethodnog rešenja, slika se može nacrtati uz pomoć tri sukcesivne petlje `for`. Svaka petlja zadužena je za iscrtavanje jednog od tri pravougaona dela tela (glava, trup i noge):

```
# pravougaona umetnost, crta Glišu
# koristi simbole *, | i + za glavu, trup i noge
# gornja leva * glave počinje  k-toj koloni tekućeg reda

# glava
a = 2
b = 3
k = 10
for linija in range(a):
    print((k-1) * ' ', end='')
    print(b * '*')

# trup
a = 4
b = 7
k = 8
for linija in range(a):
    print((k-1) * ' ', end='')
    print(b * '|')

# noge
a = 4
b = 3
k = 10
for linija in range(a):
    print((k-1) * ' ', end='')
    print(b * '+')
```

Kako je slika Gliše naišla na nepodeljeno oduševljenje kritike, autoru je ponuđena samostalna izložba u galeriji Art. Za tu priliku treba iscrtati bar još 30 umetničkih dela koja podsećaju na Glišu. Međutim, to nije nimalo lako jer treba napraviti bar 30 programa nalik na gornji. Pošto je uz veliki napor kolekcija formirana, neko iz uprave galerije moli da se zvezdice *zamene* drugim, prikladnijim simbolom (vlasnik galerije je navijač Partizana). Sada treba proći kroz sve postojeće programe i zameniti neželjene simbole, na mestu gde se koriste.

U navedenoj situaciji, kada treba iscrtati *slična* (pravougaona) dela, koristeći se *posebnim* programom za svaku sliku, uočavaju se sledeći nedostaci:

- sličan postupak (crtanje pravougaonika različitih dimenzija) *ponavlja se* više puta.
- postupak iscrtavanja složenijih slika postaje sve *kompleksniji*.
- sa porastom kompleksnosti postupka, raste *nepreglednost* potrebnog izvornog koda.
- *otežano* je modifikovanje postojećih ili uvođenje novih mogućnosti u iscrtavanju, jer treba intervenisati na *više* različitih mesta.

Navedeni nedostaci uspešno se prevazilaze primenom postupaka dekompozicije i apstrakcije. Slika iz sveta pravougaone umetnosti sastoji se od konačnog broja pravougaonika. Svaki pravougaonik definisan je visinom, širinom, simbolom ispune i pomerajem u odnosu na početak reda. Niz koraka za iscrtavanje pravougaonika navedenih osobina, može se smestiti u *korisnički definisanu funkciju*:

```
1  # linijska grafika
2
3  def pravougaonik(a, b, sim, k):
4      '''
5      crta pravougaonik dimenzija a x b
6      koristi simbol sim za ispunu pravougaonika
7      gornji levi simbol pravougaonika pocinje
8      od k-te pozicije u tekucem redu'''
9      for linija in range(a):
10         print((k-1) * ' ', end='')
11         print(b * sim)
12
13 # test primer crta Glišu
14 pravougaonik(2, 3, '*', 10) # glava
15 pravougaonik(4, 7, '|', 8)  # trup
16 pravougaonik(4, 3, '+', 10) # noge
```

Naredbom `def`, u r3, definiše se apstraktni postupak po imenu `pravougaonik`, koji iscrtava pravougaonike željenih karakteristika. Iscrtavanje je *parametrizovano* korišćenjem četiri promenljive kojima se definišu visina, širina, simbol ispune i početna pozicija. Deo programa koji poziva funkciju (r14-16) ne zna kako funkcija radi (*implementacija*), već šta radi (*namena*) i kako se poziva (redosled i značenje parametara).

Deo između ključne reči `def` i simbola dvotačke (`:`), predstavlja *potpis funkcije*. Potpis se sastoji od imena funkcije i liste *formalnih parametara*, koji se navode između zagrada. Prilikom poziva funkcije (r14-16), umesto formalnih, navode se *stvarni parametri* koji određuju kako treba iscrtati odgovarajuće pravougaonike. Umesto pojma

stvarni parametar, često se koristi pojam *argument*. U daljem tekstu, parametri će se odnositi na potpis funkcije, a argumenti, na vrednosti koje se navode prilikom njenog poziva! Argumenti mogu biti i izrazi čije se vrednosti izračunavaju pre prosleđivanja u funkciju. Redosled argumenata pri pozivu treba da odgovara, po značenju, redosledu formalnih parametara iz potpisa. Nepoštovanje redosleda može dovesti do neočekivanog rada funkcije, ili do prekida izvršavanja, uslovljenog pojavom semantičke greške.

Posle potpisa, sledi blok funkcije (r4-11), koji definiše kako funkcija realizuje svoju namenu. Neobavezni tekst, naveden između *trostrukih* apostrofa ili navodnika (r4-8), zove se još i *docstring*. On predstavlja opis funkcije koji se može dobiti po pokretanju programa, kada se u interaktivnom okruženju zada komanda `help(ime)`:

```
>>> help(pravougaonik)
Help on function pravougaonik in module __main__:

pravougaonik(a, b, sim, k)
    crta pravougaonik dimenzija a x b
    koristi simbol sim za ispunu pravougaonika
    gornji levi simbol pravougaonika pocinje
    od k-te pozicije u tekucem redu
```

Funkcija `pravougaonik()` može poslužiti za definisanje nove funkcije `gliša()` koja crta Glišu na sledeći način:

```
1  # linijska grafika
2
3  def pravougaonik(a, b, sim, k):
4      # videti prethodni listing
5
6  def gliša():
7      ''' crta Glisu fiksnih dimenzija'''
8      pravougaonik(2, 3, '*', 10) # glava
9      pravougaonik(4, 7, '|', 8)  # telo
10     pravougaonik(4, 3, '+', 10) # noge
11
12 # crta Glišu
13 gliša()
```

Iz potpisa funkcije `gliša()` uočava se odsustvo formalnih parametara, ali bez obzira na ovu činjenicu, *uvek* je potrebno navesti zagrade (r6). Korisnički definisana funkcija može pozivati proizvoljan broj drugih, korisnički definisanih funkcija, čime se postiže veća fleksibilnost u realizaciji složenijih postupaka.

Uvođenjem koncepta korisnički definisane funkcije, prevazilaze se svi problemi navedeni u listi sa početka glave:

- postupak koji se ponavlja (crtanje pravougaonika), apstrahovan je parametrizovanom funkcijom – *izbegnuto* je ponavljanja koda, u programima koji iscrtavaju pravougaonik.
- složenije slike iscrtavaju se *kombinovanjem* uvedenih funkcija – *smanjuje* se kompleksnost programa koji crta "izražajnije" slike.
- uvedene funkcije značajno *smanjuju* nepreglednost odgovarajućeg izvornog koda.
- *olakšano* je modifikovanje postojećih mogućnosti u iscrtavanju – ažururanje se obavlja samo na potrebnom mestu, unutar odgovarajuće funkcije, ili se kreira nova funkcija, u slučaju uvođenja nove funkcionalnosti.

4.1.2 Povratne vrednosti

Funkcija u Pajtonu *vraća* rezultat svoga rada putem objekta određenog tipa, koji se formira u funkciji, u toku njenog izvršavanja. Rezultat rada funkcije označava se još i kao *povratna vrednost*. Povratna vrednost *prenosi* se iz funkcije u pozivajući program, putem naredbe `return`. Naredba `return` *zaustavlja* rad funkcije i prebacuje tok izvršavanja na mesto neposredno posle poziva:

```
# konverter: km > milja i obrnuto
def konvertuj(x, km):
    ''' konvertuje kilometre u milje (km = True),
        ili milje u kilometre (km = False)'''

    c = 1.609344
    if km:
        y = x / c # km > mi
    else:
        y = x * c # mi > km

    return y

# test
print('Konverzija: km > mi')
for i in range(10):
    print(10*i, 'km = ', konvertuj(10*i, 1), 'mi')
```

Funkcija `konvertuj()` transformiše dužinu `x` u kilometrima, u odgovarajući

iznos u miljama (`km = True`), odnosno dužinu u miljama, u kilometre (`km = False`). Naredba `return`, iz r12, vraća rezultat (`y`) u pozivajući program - tok se prebacuje u poziv funkcije `print()`, gde se vraćena vrednost koristi kao ulazni argument, za formiranje ispisa (r17). Ne treba zaboraviti da funkcija zapravo vraća *objektnu referencu* na novoformirani objekat, čija vrednost u ovom slučaju, predstavlja odgovarajuću dužinu u miljama:

```
Konverzija: km > mi
0 km =  0.0 mi
10 km =  6.2137119223733395 mi
20 km =  12.427423844746679 mi
30 km =  18.64113576712002 mi
40 km =  24.854847689493358 mi
50 km =  31.068559611866696 mi
60 km =  37.28227153424004 mi
70 km =  43.495983456613374 mi
80 km =  49.709695378986716 mi
90 km =  55.92340730136005 mi
```

U primeru funkcije `pravougaonik()`, sa početka ove glave, u izvornom kodu je *izostala* naredba `return`. Kod funkcije koja ne sadrži naredbu `return`, izvršavanje se prekida kada se izvrši *poslednja* naredba u bloku funkcije, a povratna vrednost predstavljena je specijalnim objektom sa vrednošću `None`:[1]

```
>>> a = pravougaonik(2,3,'x', 5)  # nema return pa vraća None!
    xxx
    xxx
>>> print(a)
None
>>> type(a)
<class 'NoneType'>
```

Specijalna vrednost `None`, tipa `NoneType`, dodeljuje se promenljivoj ako se želi naglasiti da ona, u posmatranom tenutku, ne ukazuje ni na šta posebno. Kao što će se kasnije videti, `None` može da nosi informaciju o neuspešnom završetku nekog postupka ili o odsustvu objekta iz neke objektne kolekcije.

Funkcionalna dekompozicija koristi se kada neki složeni postupak treba razbiti na manje, jednostavnije grupe aktivnosti, što povećava čitljivost izvornog koda:

Problem 4.1 — Podniz nula u binarnom zapisu. Odrediti koji prirodan broj iz intervala $[n, m]$, ima najduži podniz nula u svom binarnom zapisu. Na primer, broj 269 (100001101), ima kraći najduži podniz nula od broja 261 (100000101). ■

[1] Engl. *None* - nijedan.

Realizacija programa podrazumeva nekoliko aktivnosti: učitavanje za n i m, prolazak kroz sve brojeve na intervalu $[n,m]$, prevođenje tekućeg kandidata u binarni zapis i izračunavanje dužine njegovog najdužeg podniza nula, kao i memorisanje onog kandidata koji ima maksimalan podniz nula u svom zapisu. Početni problem će, zbog kompleksnosti, biti dekomponovan na dva manja, jednostavnija potproblema. Sledeći prikaz odnosi se na algoritam u pseudokodu:

Algoritam 4.1 — Podniz nula u binarnom zapisu.

```
Ulaz: n i m
Ako n <= m i n > 0
       max_duzina ← -1
       ZaSvaki x iz [n, m]
             d ← naj_podniz(x)
             Ako d > max_duzina
                    max_duzina ← d,  rešenje ← x
             KrajAko
       KrajSvaki
       Prikaži rešenje, max_duzina
Inače
       Prikaži 'greška'
KrajAko
```

U algoritmu 4.1, uočava se postojanje *nezavisne* celine koja može biti realizovana kao funkcija - `naj_podniz()`. Ona izračunava dužinu *najdužeg* podniza nula u binarnom zapisu prirodnog broja. Apstrahujući ovu funkciju,[2] algoritam se svodi na pronalaženje maksimalne dužine u nizu dužina, odnosno pamćenje odgovarajućeg prirodnog broja koji sadrži maksimalni podniz (videti problem 3.2). Prvo se realizuje funkcija `naj_podniz()`, koja treba da vrati nulu, ako binarni zapis broja sadrži sve jedinice.

Prirodan broj x može se zapisati kao sledeća suma:

$$x = \sum_{i=0}^{n} a_i 2^i = \sum_{i=1}^{n} a_i 2^i + a_0, \quad a_i \in \{0,1\} \tag{4.1}$$

Sekvenca $a_n a_{n-1} \ldots a_1 a_0$ predstavlja *binarni zapis* broja x. Na primer, broj 17 može se predstaviti kao $1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$, pa je njegov binarni zapis jednak 10001. Da bi se detektovao najduži podniz nula, potrebno je izdvojiti sve koeficijente a_i iz sume (4.1). Na osnovu date jednakosti, proizilazi da je ostatak pri celobrojnom deljenju broja x sa dva, jednak koeficijentu a_0, a rezultat celobrojnog deljenja svodi se na sumu $\sum_{i=1}^{n} a_i 2^{i-1}$. Očigledno, da bi se dobio a_1, potrebno je ponoviti postupak

[2] Apstrahovanje se vrši zanemarivanjem implementacije, a uzimanjem u obzir namene i potpisa funkcije.

celobrojnog deljenja i zabeležiti novi ostatak. Postupak se prekida kada se kao rezultat celobrojnog deljenja dobije nula, a ostatak tada predstavlja koeficijent a_n:

```
1  def naj_podniz(x):
2
3      '''vraca duzinu najduzeg podniza nula u binarnom zapisu'''
4      d = 0                 # tekuća dužina podniza nula
5      maxd =  0             # dosad maks. dužina nekog podniza nula
6      prva_jedinica = True # da li će sledeća jedinica biti prva
7                            # posle podniza nula
8      while True:
9          # izdvajanje bin. cifre najmanje težine
10         bin_cifra = x % 2
11         # ako je nula uvećaj dužinu tekućeg niza
12         if bin_cifra == 0:
13             d += 1
14             prva_jedinica = True
15         # ako je prva jedinica posle nule,
16         # da li je podniz nula dosad najduži
17         elif prva_jedinica:
18             prva_jedinica = False
19             if d > maxd:
20                 maxd = d
21             d = 0
22         # priprema za sledeću cifru
23         x //= 2  # x = x // 2
24         if x == 0:
25             break
26
27     return maxd
```

U svrhu pronalaženja najdužeg podniza nula, definisane su promenljive `d` - dužina aktuelnog podniza, `maxd` - trenutna maksimalna dužina i `prva_jedinica` - promenljiva koja ukazuje da li će sledeća jedinica koja usledi, biti prva posle nule (r4-6). Kada se u zapisu pojavi nula, ažurira se dužina tekućeg podniza i priroda sledeće jedinice (r12-14). U suprotnom, ako se u zapisu naiđe na jedinicu, treba završiti sa obradom prethodnog podniza, ali samo ako je ta jedinica nastupila neposredno posle nule (r17-21). Bitnu ulogu u postupku ima logička promenljiva `prva_jedinica` koja nosi informaciju o prirodi svake izdvojene jedinice. Naredba `return`, iz r27, vraća dužinu maksimalnog podniza nula.

Već je istaknuto da funkcije vraćaju reference na objekte koji su proizašli iz njihovog rada. Kako se objektima može pristupati isključivo putem reference, u daljem tekstu će se navoditi da funkcije vraćaju željene vrednosti.

Kako se funkcije *izračunavaju* na odgovarajuće vrednosti, moguće je koristiti ih u okviru drugih, pogodnih izraza. Na primer, izraz `2 * naj_podniz(261) + 1`, ima vrednost `11`. Sledi direktna implementacija algoritma 4.1 koji pronalazi prirodan broj iz intervala, sa maksimalno dugačkim podnizom nula:

Program 4.1 — Podniz nula u binarnom zapisu.

```
1  def naj_podniz(x):
2      # videti prethodni listing
3
4  n = int(input('prirodan broj n: '))
5  m = int(input('prirodan broj m >= n: '))
6
7  if n <= m and n > 0:
8      max_duzina = -1
9      for x in range(n, m+1):
10         d = naj_podniz(x)
11         if d > max_duzina:
12             max_duzina = d
13             rešenje = x
14     print(rešenje, 'ima podniz nula dužine', max_duzina)
15 else:
16     print('greška u unosu!')
```

Rad programa, za interval $[1, 33]$, prikazan je ispod:

```
prirodan broj n: 1
prirodan broj m >= n: 33
32 ima podniz nula dužine 5
```

Povratne vrednosti različitih tipova

Iz dosadašnjeg izlaganja, poznato je da se izvršavanje funkcije prekida nailaskom na naredbu `return`, ili, ako ona nije prisutna, kada programski tok naiđe na kraj funkcijskog bloka. Međutim, izvršavanje funkcije može da se prekine na *više* mesta, pri čemu se u pozivajući program može vratiti više *različitih* tipova vrednosti:

```
# ilustruje različite izlaze i povratne vrednosti
def sve_po_malo(x):

    if x == 5:
        return 5
    elif x < 0:
        return 'negativan broj'
    elif x == 0:
        return  # ovde se vraća None

    # dolaskom u ovu tačku, vraća se None
```

```
>>> sve_po_malo(-1)
'negativan broj'
>>> sve_po_malo(5)
5
>>> sve_po_malo(0)
>>>
>>> sve_po_malo(0) == None
True
>>> sve_po_malo(36) == None
True
```

Definisanje funkcije koja vraća reference na različite tipove objekata *nije preporučljivo*, pošto takva praksa otežava razumevanje izvornog koda i potencijalni je izvor grešaka. Ipak, pored svog osnovnog tipa, funkcije često vraćaju i `None`, da ukažu na neadekvatne ulazne parametre, ili na nemogućnost izvršavanja zadatka.

4.1.3 Imenovani i opcioni parametri

Posmatra se sledeća funkcija i njeni mogući pozivi:

```
# ilustruje imenovane i opcione parametre
def kaži_zdravo(ime, koliko, jezik='sr'):

    zdravo = 'Zdravo' # podrazumevani jezik
    if jezik == 'en':
        zdravo = 'Hello'
    elif jezik == 'it':
```

```
8           zdravo = 'Ciao'
9
10      # ispis pozdrava
11      for i in range(koliko):
12          print(zdravo, ime)
```

```
>>> kaži_zdravo(koliko=2, jezik='it', ime='Miloš')
Ciao Miloš
Ciao Miloš
>>> kaži_zdravo(koliko=2, ime='Miloš')
Zdravo Miloš
Zdravo Miloš
>>> kaži_zdravo('Miloš', 1)
Zdravo Miloš
```

Argumenti funkcije mogu se, pri pozivu, *imenovati* po formalnim parametrima, što je ilustrovano prvim pozivom funkcije `kaži_zdravo()`. *Imenovani parametri* omogućavaju da se, pri njihovom navođenju, ne mora poštovati redosled definisan u potpisu funkcije, što je učinjeno u prvom pozivu.

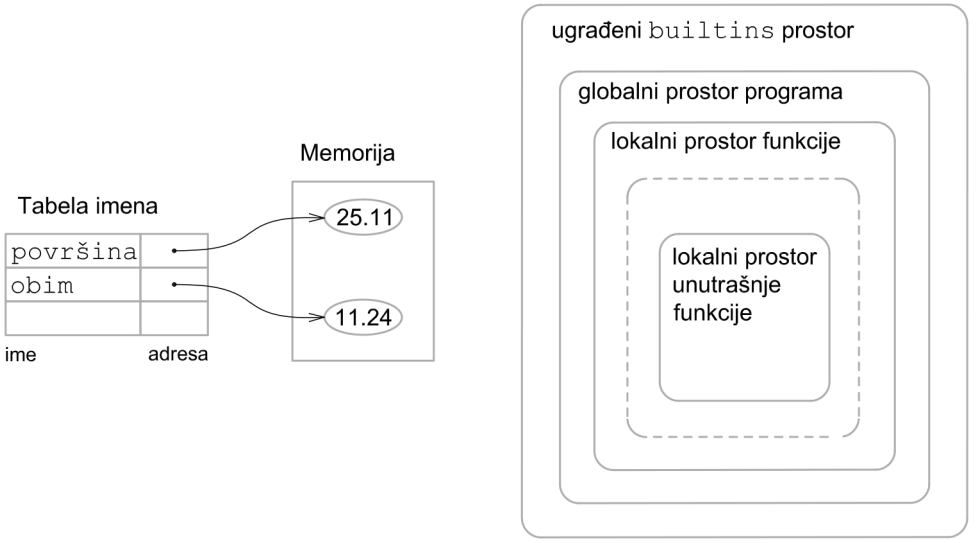
Ako se pri definisanju funkcije, uz formalni parametar pojavi i znak jednakosti (r2), onda taj parametar postaje *opcioni*. Opcioni parametri *ne moraju* se navoditi pri pozivu funkcije, kao što je učinjeno u drugom pozivu iz primera. Tada oni dobijaju vrednosti koje su definisane u potpisu funkcije. Treba obratiti pažnju da se opcioni parametri iz potpisa obavezno navode *posle* standardnih parametara. U suprotnom, interpreter će prijaviti odgovarajuću grešku. Pri navođenju izraza za vrednost opcionog parametra, treba imati u vidu da se vrednost opcije izračunava pri *definisanju potpisa*, a *ne* u trenutku poziva. Poslednji poziv ne navodi imena pa se mora poštovati redosled iz potpisa funkcije, uz korišćenje podrazumevanog jezika ispisa.

Opcioni parametri često se koriste u praksi, a podrazumevane vrednosti pažljivo su izabrane tako da pokrivaju najčešće situacije pri pozivu funkcije.

4.1.4 Imenski prostori. Opseg važenja promenljivih

Promenljive predstavljaju imena za objekte. Ime se najčešće vezuje za objekat kroz naredbu dodele vrednosti. Moguća su i druga vezivanja, poput navođenja brojača u petlji `for`, ili formalnih parametara u potpisu funkcije. Skup imena, definisanih u programu, naziva se *imenski prostor*[3] programa. Imenski prostor realizuje se *tabelom imena* u kojoj se, svakom imenu iz prostora, pridružuje adresa imenovanog objekta u memoriji (slika 4.1, levo).

[3] Engl. *Namespace*.



Slika 4.1: Imenski prostori i tabela imena: imenima promenljivih pridružuju se adrese koje označavaju objekte u memoriji (levo); hijerarhija formiranih imenskih prostora u nekom trenutku izvršavanja. Svakom prostoru pridružena je po jedna tabela imena (desno).

U toku izvršavanja programa, u memoriji istovremeno postoje *bar dva* međusobno odvojena imenska prostora. To omogućava da se, u okviru različitih delova koda, različitim objektima daju ista imena. Po pokretanju, interpreter formira imenski prostor *ugrađenog* `builtins` modula, koji sadrži predefinisana imena poput `input` ili `print`. Kada se program pokrene, kreira se *globalni* imenski prostor, a pri pozivu svake funkcije, dodatni, *lokalni* imenski prostor (slika 4.1, desno). Kasnije će biti reči i o imenskim prostorima modula i klase. Veze između pojedinih imena i objekata, sačuvane u odgovarajućoj tabeli imena, imaju svoj *životni vek*. Tabela imena *ugrađenog* `builtins` modula *postoji* sve dok je aktivna interpreterska sesija. Imenski prostor programa *opstaje* sve dok se program ne završi, a funkcije, dok traje njeno izvršavanje.

Opseg važenja promenljive odnosi se na *deo* izvornog koda u kome se promenljiva može koristiti, uz navođenje njenog imena. Pitanje životnog veka, odnosno opsega važenja, neraskidivo je povezano sa trenutno formiranim imenskim prostorima. Interpreter primenjuje jednostavno pravilo za izbor tabele u kojoj će se potražiti navedeno ime, odnosno objekat na koji ono ukazuje. Ime se prvo traži u tabeli lokalnog prostora (ako se na promenljivu referiše iz funkcije), pa ako se tu ne pronađe, ide se za po jedan korak naviše u hijerarhiji, sve dok se ne stigne do ugrađenog `builtins` prostora (slika 4.1, desno). Ako se ime ne pronađe ni u jednoj tabeli, interpreter prijavljuje grešku.

Opseg važenja promenljve definisane u nekoj funkciji `f()` (*lokalna promenljiva*), odnosi se na `f()` i sve eventualne unutrašnje funkcije. Promenljive definisane u programu (*globalne promenljive*), mogu se koristiti kako u programu, tako i u svim pozivajućim funkcijama. Sledi primer koji ilustruje prethodnu diskusiju:

```
1 def f():
2     b, c = 3, 10  # lokalni imenski prostor funkcije
3     print('u f(), a =', a, 'b =', b)
4
5 a, b = 1, 1  # globalni imenski prostor programa
6 f()
7 print('program, a =', a, 'b =', b)
8 print('program, c =', c) # greška!
```

Kada se gornji program pokrene, dobija se:

```
u f(), a = 1 b = 3
program, a = 1 b = 1
Traceback (most recent call last):
  File "C:/programi/p4_1scope.py", line 9, in <module>
    print('program, c=', c)
NameError: name 'c' is not defined
```

Globalne promenljive `a` i `b` definisane su u imenskom prostoru programa (r5). U funkciji `f()`, definisane su lokalne promenljive `b` i `c` (r2). Prilikom poziva funkcije (r6), ispisuju se vrednosti za `a` i `b` (r3). Promenljiva `a` ne nalazi se u imenskom prostoru funkcije (nije lokalna promenjiva), pa se interpreter "penje" jedan korak naviše u hijerarhiji, pronalazeći vrednost za `a` preko globalne tabele imena (=1). Slično, iako je ime `b` definisano u oba imenska prostora, promenljiva se tretira kao lokalna, pa se vrednost pronalazi preko tabele u imenskom prostoru funkcije (=3).

Posle izvršenja funkcije `f()`, njen imenski prostor prestaje da postoji, pa se u r7 ispisuju vrednosti na koje ukazuju globalne promenljive `a` i `b`. Prilikom pokušaja ispisa promenljive `c` (r8), interpreter prijavljuje grešku jer navedeno ime ne postoji ni u tabeli globalnog imenskog prostora, ni u tabeli `builtins` modula sa predefinisanim imenima (`c` očigledno nije predefinisano ime ni za jedan objekat).

Prethodni primer pokazuje da se globalno ime (`b` iz r5), ukoliko se u funkciji nađe sa *leve* strane naredbe dodele vrednosti (r2), kopira u lokalni imenski prostor, pa sada ukazuje na novi objekat, koji reprezentuje vrednost izračunatog izraza sa desne strane dodele. Kada se funkcija izvrši i kontrola toka vrati u pozivajući program, ime promenljive iz globalnog prostora ponovo postaje aktuelno, a promenljiva ukazuje na stari objekat.

Tipična početnička greška, koja nastaje zbog nerazumevanja odnosa globalne i lokalne promenljive, ilustrovana je u sledećem primeru:

```
def f():
    a = a + 1  # greška
    print(a)

a = 1  # globalna promenljiva
f()
```

```
Traceback (most recent call last):
  File "C:/programi/p4_1greska.py", line 6, in <module>
    f()
  File "C:/programi/p4_1greska.py", line 2, in f
    a = a + 1  # greška
UnboundLocalError: local variable 'a' referenced before assignment
```

Greška nastaj u r2, u toku izvršavanja `a = a + 1`. Pošto se globalno ime `a` pojavljuje na levoj strani dodele vrednosti, interpreter kopira ime u lokalni imenski prostor. Međutim, kako to ime još ne ukazuje ni na jedan lokalni objekat, prilikom pokušaja uvećavanja vrednosti za jedan, interpreter prijavljuje grešku. Da je, umesto `a = a + 1`, stajalo `b = a + 1`, onda bi sve bilo u redu. Tada bi se `a` tretirala kao globalna promenljiva, a novonastala lokalna promenljiva `b`, ukazivala bi na uvećanu vrednost.

Parametri - opseg važenja i mehanizam prenošenja

Formalni parametri imaju opseg važenja isključivo unutar bloka funkcije (tretiraju se kao lokalne promenljive), a njihov životni vek poklapa se sa njenim trajanjem. Stvarni parametri prenose se u funkciju tako što se, u njen imenski prostor, kopiraju vrednosti objektnih referenci.[4] Zapravo, objekat dobija *još jedno* ime preko koga mu se može pristupati iz funkcije, ali ako se formalni parametar nađe sa leve strane naredbe dodele vrednosti, onda se ta veza *raskida* i referenca dobija lokalni karakter. Ako parametar ukazuje na objekat *promenljivog* tipa, onda se njegova vrednost *može* promeniti u funkciji, o čemu će biti reči u glavi 5, kada se budu izučavali promenljivi tipovi. Sledi ilustrativni primer:

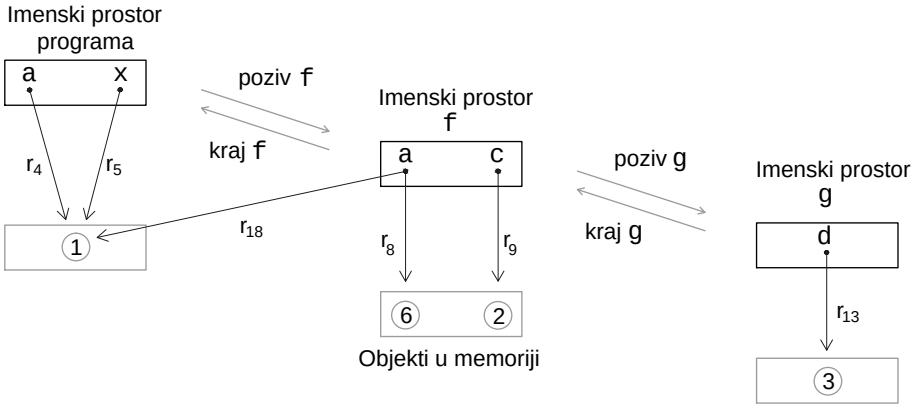
```
## Oblast važenja promenljivih

# globalne promenljive programa
a = 1
x = 1
```

[4] Engl. *Copy by object reference value*.

```
6
7  def f(a):
8      a += 5      # parametar a sakriva globalnu promenljivu
9      c = x + 1   # c je lokalna promenljiva za f
10     print('f: (a, c)', a, c)
11
12     def g():
13         d = c + x  # d je lokalna promenljiva za g
14         print('g: d', d)
15
16     g()
17
18 f(a)
19 print('program: (a,x)', a, x)
```

Globalne promenljive a i x definišu se u imenskom prostoru programa (slika 4.2). Prilikom poziva funkcije f(), memorijska adresa pridružena promenljivoj a *kopira* se, iz imenskog prostora programa, u tek kreirani imenski prostor funkcije (r18). Parametar a, koji po kopiranju ima lokalni karakter u funkciji, ukazuje na isti objekat kao i globalna promenljiva a (=1). Potom se pokušava sa uvećavanjem vrednosti objekta za pet (r8). Pošto je celobrojni tip nepromenljiv, kreira se novi objekat sa vrednošću šest, a njegova memorijska adresa smešta se u tabelu imenskog prostora funkcije. Sada parametar a ukazuje na novokreirani objekat (slika 4.2).



Slika 4.2: Prenos parametara i opseg važenja promenljive: prilikom poziva funkcije kreira se novi imenski prostor. Oznake r_x, označavaju redove u programu u kojima se obavljaju odgovarajuća imenovanja.

U r9 definiše se lokalna promenljiva c, koja po izračunavanju ukazuje na vrednost dva. Prilikom izračunavanja, bilo je neophodno pristupiti vrednosti na koju ukazuje

globalna promenljiva `x`. U ovom slučaju, interpreter je pronašao vrednost na koju ukazuje promenljiva `x`, tako što je prvo potražio ime u tabeli imenskog prostora funkcije `f()`. Pošto u tom prostoru ime nije pronađeno, interpreter pravi jedan korak unazad u putanji funkcijskih poziva i pronalazi traženo ime u imenskom prostoru programa (slika 4.2). Po pozivu funkcije `g()` (r16), izračunava se vrednost za promenljivu `d`. Kako se imena `c` i `x` ne nalaze u imenskom prostoru funkcije, interpreter pronalazi njihove vrednosti u odgovarajućim prostorima sa staze poziva. Program daje sledeći izlaz:

```
f: (a, c) 6 2
g: d 3
program: (a,x) 1 1
```

Promena vrednosti globalne promenljive iz funkcije - naredba `global`

Globalnim promenljivim može se pristupati iz svih funkcija u programu, ali se njima, kao što je već pokazano, *ne mogu* dodeljivati nove vrednosti. Ako se želi da funkcija tretira globalno ime globalnim i *posle* dodele vrednosti, onda se ono mora *deklarisati* naredbom `global`, pre eventualne dodele vrednosti:

```
1 def g():
2     global a
3     a = a + 1  # lokalni imenski prostor funkcije
4     print('u g(), a =', a)
5
6 a = 1  # globalni imenski prostor programa
7 g()
8 print('program posle g(), a =', a)
```

```
u g(), a = 2
program posle g(), a = 2
```

Parametri funkcije *ne mogu* se deklarisati naredbom `global` - interpreter bi tada prijavio grešku.

Sporedni efekat funkcije

Funkcija proizvodi *sporedni efekat*[5] ako se njena projektovana uloga ostvaruje mimo povratnog mehanizma naredbe `return`. Sporedni efekat može se desiti u bar dva slučaja: promena vrednosti promenljivih definisanih *van* opsega funkcije i emitovanje

[5] Engl. *Side effect*.

informacija u spoljni svet. U prethodnom primeru, menjanjem vrednosti globalne promenljive `a`, funkcija `g()` ostvaruje sporedni efekat. Ovaj slučaj često se smatra *lošom* praksom jer funkcija menja stanje programa mimo svog povratnog mehanizma. Tada je izvorni kod manje čitljiv pa se potencijalne logičke greške teže otklanjaju.

U slučaju kada se informacija šalje ka eksternim sistemima (ekran, datoteka, štampač, baza podataka, mreža), funkcija ostvaruje *opravdani* sporedni efekat. U takvim situacijama, funkcionalnost se ne svodi *samo* na izračnavanje, pa se ni rezultat ne može vratiti isključivo putem naredbe `return`. Primer za ovakvo ponšanje je funkcija `pravougaonik()`, iz sveta pravougaone umetnosti, sa početka glave.

Postupci koji proizvode tačan, efikasan, ali i kod koji je dobro organizovan i čitljiv, smatraju se dobrom praksom. Organizacija i čitljivost olakšavaju otklanjanje grešaka u velikim programskim sistemima. Ovo je naročito bitno ako se ima na umu da se, u toku životnog ciklusa softvera, često menjaju programeri koji ga održavaju.

4.2 Organizacija izvornog koda

Kompleksan programski sistem organizuje se po logičkim celinama tako da se izvorni kod piše brže i jednostavnije, uz korišćenje postojećih komponenti. Takvi modularni sistemi lakše se nadograđuju i održavaju. Funkcije, kao nezavisne celine, omogućavaju efikasno sprovođenje procesa dekompozicije na funkcionalnom nivou. Međutim, one nisu dovoljne da omoguće jednostavno i fleksibilno kreiranje složenijih celina, pa se zbog toga uvode više hijerarhijske podele poput *modula* i *paketa*.

4.2.1 Moduli

Izvorni kod do sada razmatranih programa bio je smešten u *samo jednu* datoteku. Sada treba pojasniti kako se funkcija, definisana u datoteci *a.py*, može pozivati u programu smeštenom u datoteci *b.py*. U tu svrhu, u Pajton se uvodi koncept *modula*. Modul predstavlja datoteku sa ekstenzijom **.py*, koja sadrži *proizvoljan* izvorni kod. On *najčešće* sadrži definicije funkcija, ali može da sadrži i *druge* naredbe, poput onih za definisanje globalnih promenljivih. Sledeći primer ilustruje koncept modula:

Problem 4.2 — Površine. Realizovati grupu funkcija koje računaju površine paralelograma, trapeza i kruga. Testirati funkcije u posebnom programu, u kome se unosi tip geometrijske slike i neophodni elementi za računanje površine. ■

Grupa funkcija realizovana je kao modul `površine`, smešten u datoteku *površine.py*:

Program 4.2 — Površine.

```
# modul površine (datoteka površine.py)
paralelogram = 1
trapez = 2
krug = 3

def p_paralelogram(a, b):
    '''računa površinu paralelograma sa stranicama a i b'''
    return a * b

def p_trapez(a, b, h):
    '''računa površinu trapeza sa osnovama a, b i visinom h'''
    return h * (a + b)/2

def p_krug(r):
    '''računa površinu kruga poluprečnika r'''
    return 3.14 * r**2
```

Program za testiranje modula `površine`, smešten je u datoteku glavnog programa:

```
# unosi tip slike (ceo broj), pa elemente za računanje površine
# unos se prekida kada se unese broj različit od:
# 1 (paralelogram), 2 (trapez), 3 krug
import površine # čini funkcije i promenljive iz povrsine.py
                # vidljivim u programu
while True:
    slika = int(input('1 paralelogram, 2 trapez, 3 krug  '))
    if slika == površine.paralelogram:
        a = float(input('a '))
        b = float(input('b '))
        print('P=', površine.p_paralelogram(a, b))
    elif slika == površine.trapez:
        a = float(input('a '))
        b = float(input('b '))
        h = float(input('h '))
        print('P=', površine.p_trapez(a, b, h))
    elif slika == površine.krug:
```

```
18          r = float(input('r '))
19          print('P=', površine.p_krug(r))
20      else:
21          break
```

Naredba `import` *uvodi* modul u program (r4).[6] Uvođenje obuhvata učitavanje naredbi modula u operativnu memoriju i kreiranje novog imenskog prostora za modul. Funkcije i globalne promenljive iz ovog prostora postaju *vidljive* u programu, ako im se pristupa korišćenjem *notacije sa tačkom*, oblika `<ime_modula>.<ime_u_modulu>` (r8, 11, 12, 16, 17, 19). Notacija sa tačkom sprečava moguće *preklapanje* imena definisanih u programu i u uvedenom modulu, pa je dozvoljeno koristiti dva ista imena u obe celine. Ako se želi izbeći korišćenje notacije sa tačkom, može se koristiti i sledeća forma: `from površine import <ime>`. Na ovaj način, imena iz modula postaju *direktno* vidljiva, ali u slučaju preklapanja, imena definisana u programu *sakrivaju* imena u uvedenom modulu. Pored primenjenog oblika naredbe `import` (r4), ponekad je korisno upotrebiti opciju `as`, iza koje sledi kraće, alternativno ime za modul. Na primer, `import površine as p`, omogućava da se funkcija može pozvati sa `p.p_paralelogram(a, b)`, umesto `površine.p_paralelogram(a, b)`.

Programi koji su *u celosti* smešteni u jednoj datoteci nazivaju se još i *skripta*. Skripta se, van IDLE okruženja, pokreće u komandnom režimu operativnog sistema, zadavanjem komande koja poziva interpreter, uz navođenje njene pune putanje. Kako je svaka skripta ujedno i modul, ona se, unutar interaktivne sesije u IDLE-u, može pokrenuti i sa `import <ime_skripta>`.[7] Prilikom uvođenja u drugi modul, inicijalno se izvršavaju sve naredbe iz bloka uvedenog modula (`površine`, r2-4). Da bi se povećala čitljivost glavnog programa, u modulu `površine` definisane su promenljive koje ukazuju na tip slike čija se površina traži. Datoteka sa izvornim kodom zvaće se modulom, *samo* ako nije namenjena za samostalno pokretanje, već služi kao *biblioteka* funkcija za realizaciju složenijih programa.

Naredbe bloka modula izvršavaju se, zbog uštede resursa, *samo* pri prvom uvođenju, u okviru jedne interpreterske sesije. Ukoliko se modul modifikuje, promene *neće* biti vidljive sve dok se interpreterska sesija ne restartuje (<ctrl> + <F6>) i modul ponovo ne učita u memoriju!

U daljem razmatranju, koristiće se funkcije iz različitih dostupnih modula, u okviru standardne instalacije Pajtona. Na primer, upotrebljavaće se modul koji sadrži često korišćene matematičke funkcije – `math`. Sledi primer korišćenja modula `math` i funkcije `dir()`, koja prikazuje imena definisana u traženom modulu:

[6] Engl. *Importing*.

[7] Važi pod uslovom da se direktorijum, koji je sadrži, nalazi u skupu putanja koje IDLE pretražuje kada učitava module u operativnu memoriju.

```
>>> import math as m
>>> m.sin(m.pi)
1.2246467991473532e-16
>>> m.sqrt(16)
4.0
>>> dir(m)  # lista imena definisana u modulu
['__doc__', '__loader__', '__name__', '__package__', '__spec__',
'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil',
'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp',
'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum',
'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf',
'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf',
'nan', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan',
'tanh', 'trunc']
```

Uočiti da, za ugao od 180 stepeni, sinus iznosi 1.2246467991473532e-16 (eksponencijalni zapis, e-16 = 10^{-16}). Ovo se dešava jer se najveći broj funkcija izračunava *približno*, upotrebom konačnog broja osnovnih aritmetičkih operacija. Osim toga, već je diskutovano da se realni brojevi, u opštem slučaju, *ne mogu* tačno predstaviti u brojnom sistemu sa osnovom dva.

4.2.2 Paketi

Paketi organizuju veći broja modula u jedinstvenu imensku hijerarhiju. Pretpostaviti da na disku postoji direktorijum `slike` koji obuhvata datoteke (module) `gif.py`, `jpeg.py` i `png.py`. Svaki od navedenih modula sadrži funkcije koje obrađuju slike istoimenog formata. Funkcionalnosti iz paketa `slike` mogu se, po uvođenju (`import slike`), koristiti u proizvoljnom programu P. Na primer, iz modula `jpeg` postaje dostupna funkcija `skaliraj()`: `slike.jpeg.skaliraj(0.5)` – umanjuje sliku za 50% po širini i dužini. Uvođenjem paketa, kao organizacione celine, omogućena je upotreba različitih biblioteka koje sadrže module sa *istim* imenom. Program P može dodatno koristi funkcionalnosti iz istoimenog modula `jpeg.py`, ali u okviru paketa `prepoznavanje`: `prepoznavanje.jpeg.lice()` – prepoznaje da li se na slici u jpeg formatu nalazi lice osobe.

Da bi se moduli, smešteni u nekom direktorijumu *d*, tretirali kao jedinstveni paket, potrebno je da *d* sadrži specijalno imenovanu datoteku `__init__.py`. U okviru ove datoteke, koja se izvršava prilikom prvog uvođenja paketa, nalazi se izvorni kod kojim se paket inicijalizuje – na primer postavljanje globalnih promenljivih paketa na početne vrednosti. Ukoliko inicijalizacija nije neophodna, datoteka može biti i prazna, ali *mora* da postoji!

Složenije biblioteke mogu biti realizovane kao paketi sa većim brojem potpaketa. Potpaketima odgovara po jedan direktorijum sa `__init__.py` datotekom. Poziv oblika `paket.potpaket.modul.funkcija()`, moguć je pošto se paket uvede u program.

4.2.3 Funkcije i moduli kao objekti

Pajton tretira sve funkcije i module kao objekte tipa `function`, odnosno `module`. Na ovaj način, moguće je imenovati ih različitim promenljivim, uz pomoć naredbe dodele vrednosti, ili prosleđivati kao stvarne parametre u druge funkcije. U sledećem primeru ilustrovana je njihova objektna priroda:

```
>>> import math
>>> print(id(math), type(math))
1967655099432 <class 'module'>
>>> sin = math.sin
>>> sin(0)
0.0
>>> print(id(sin), type(sin))
1967655155752 <class 'builtin_function_or_method'>
>>>
```

Po uvođenju modula `math` u imenski prostor interaktivne sesije, prikazani su identitet (memorijska adresa) i tip objekta koji reprezentuje modul. Funkcija `sin()`, iz modula `math`, *dodeljuje* se promenljivoj `sin` iz imenskog prostora interaktivne sesije. Dodela je moguća jer se funkcije, kao i moduli, tretiraju kao objekti. Sada se funkcija može pozivati preko novog imena, bez navođenja notacije sa tačkom. Objektna priroda funkcije potvrđena je ispisivanjem njenog identiteta i tipa.

Ponekad se algoritam za rešavanje određene klase problema ne razlikuje za pojedine probleme iz klase, sem u jednom nezavisnom, izolovanom delu. Ako nepromenljivi deo algoritma "vidi" izolovanu celinu uvek na isti način, kroz unapred poznato ponašanje, onda je moguće da se celina unese u algoritam u obliku funkcije, pored ostalih ulaznih veličina. Koncept se ilustruje sledećim primerom:

Problem 4.3 — Tablica funkcije. Napisati funkciju koja ispisuje sve vrednosti za proizvoljnu funkciju $f(x)$, na intervalu $[a,b]$, sa korakom *delta*. Testirati funkciju za $y = cos(x)$ i $y = e^x$. ■

Ispisivanje tablice vrednosti za *proizvoljnu* funkciju, na intervalu $[a,b]$, podrazumeva *fiksni* iterativni postupak u kome se, sa zadatim korakom, vrši odabir vrednosti nezavisne promenljive. U postupku ispisivanja, *varijabilna* je jedino funkcija za koju se tablica formira. Pošto se funkcije tretiraju kao objekti, moguće je *preneti* ih u druge funkcije, kao *ulazne* parametre:

Program 4.3 — Tablica funkcije.

```
1 def tablica_vrednosti(a, b, delta, f):
2     ''' Ispisuje tablicu vrednosti f-je f(x)
```

```
3           na intervalu [a, b], sa korakom delta.
4           Podrazumeva se da je f(x) definisana na intervalu'''
5
6       if a <= b and delta > 0:
7           x = a
8           while x <= b:
9               print(x, f(x))
10              x += delta
11      else:
12          print('mora biti a < b i delta > 0')
13
14  import math
15  print('cos(x)')
16  tablica_vrednosti(0, 2 * math.pi, math.pi / 6, math.cos)
17  print('\nexp(x)') # \n specijalni karakter za prelaz u novi red
18  tablica_vrednosti(0, 5, 0.5, math.exp)
```

Parametar `f`, iz potpisa funkcije `tablica_vrednosti()` (r1), predstavlja prenetu funkciju čije se vrednosti ispisuju u r9. Po uvođenju modula `math` (r14), prvo se ispisuje tablica za $cos(x)$ (r16), pa potom i za e^x (r18). Izabrana funkcija prenosi se jednostavnim navođenjem imena kao stvarnog parametra. U primeru se koristi notacija sa tačkom, pošto su funkcije koje se prenose, definisane unutar imenskog prostora uvedenog modula.

Uočiti da se u tekstu pod navodnicima, u funkciji `print()`, pojavljuje simbol kome prethodi \ (ovde \n). Radi se o *kontrolnom karakteru*[8] koji proizvodi određeni efekat pri ispisu. Na primer, kontrolni karakter \n, označava prelazak u novi red.

```
cos(x)
0 1.0
0.5235987755982988 0.8660254037844387
1.0471975511965976 0.5000000000000001
1.5707963267948966 6.123233995736766e-17
2.0943951023931953 -0.4999999999999998
2.617993877991494 -0.8660254037844385
3.1415926535897927 -1.0
3.6651914291880914 -0.866025403784439
4.1887902047863905 -0.5000000000000004
4.71238898038469 -1.8369701987210297e-16
5.235987755982989 0.5000000000000001
5.759586531581288 0.8660254037844388
```

[8] Engl. *Escape character*.

```
exp(x)
0 1.0
0.5 1.6487212707001282
1.0 2.718281828459045
1.5 4.4816890703380645
2.0 7.38905609893065
2.5 12.182493960703473
3.0 20.085536923187668
3.5 33.11545195869231
4.0 54.598150033144236
4.5 90.01713130052181
5.0 148.4131591025766
```

4.3 Rekurzivni algoritmi

U praksi se često definiše značenje različitih pojmova koji se odnose na materijalni ili apstraktni svet. Ako se tom prilikom koristi *sam pojam*, kao sastavni deo objašnjenja, onda se radi o *rekurzivnoj* definiciji. Na primer, validno ime za promenljivu u Pajtonu može se definisati na sledeći način:

1. slovo, ili donja crta (_), je validno ime za promenljivu.
2. validno ime iza koga sledi slovo, broj ili donja crta, predstavlja validno ime.

U definiciji validnog imena promenljive, stav 2, primećuje se rekurzivna priroda formulacije: ako je niz simbola validno ime, onda je to i isti niz proširen za slovo, broj ili donju crtu. Uz pomoć stava 2, mogu se generisati proizvoljno dugačke sekvence, ali i dalje nije jasno čime treba da započne validno ime. Prvi stav rešava problem početnog simbola – svako slovo, ili donja crta, predstavlja validno ime za promenljivu. Sada se, primenom drugog stava, može sagraditi bilo koje validno ime. Prvi stav naziva se još i *bazni slučaj*. Treba primetiti da bi, bez baznog slučaja, definicija postala *nepotpuna*, pa samim tim i neupotrebljiva.

4.3.1 Rekurzija na delu

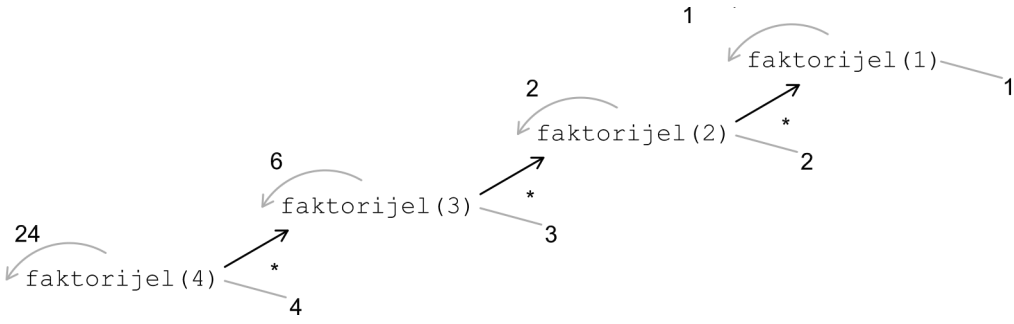
Rekurzivni način rešavanja problema odnosi se na specijalnu klasu algoritama iz porodice *podeli pa vladaj* (glava 1.3.1). Neka se problem, po svojoj složenosti, može okarakterisati prirodnim brojem n, pri čemu složenijem problemu odgovara veće n. Ako rešenje problema složenosti n, zavisi od rešenja *istog problema* složenosti k, pri čemu je $n > k$, te ako je poznato bazno rešenje istog problema (složenost 1), onda se u rešavanju može primeniti *rekurzivni algoritam*. Sledi ilustrativni primer:

Problem 4.4 — Drugovi Faktorijel i Fibonači. Napisati rekurzivne funkcije koje, za zadati prirodni broj n, izračunavaju $n!$ i Fibonačijev broj $F(n)$, i smestiti ih u modul `rekurzija`. Funkcije testirati za prvih 10 prirodnih brojeva, u posebnom programu. ■

Program 4.4 — Drugovi Faktorijel i Fibonači.

```
1  def faktorijel(n):
2      '''Izračunava n! u rekurzivnom postupku, n>=0'''
3
4      if n == 1 or n == 0:      # bazni slučaj
5          return 1
6      else:                     # rekurzija
7          return n * faktorijel(n - 1)
8
9  def fibonaci(n):
10     '''Izračunava Fib(n) u rekurzivnom postupku, n>0'''
11
12     if n == 1:                # bazni slučaj
13         return 0
14     elif n == 2:              # bazni slučaj
15         return 1
16     else:                     # rekurzija
17         return fibonaci(n - 1) + fibonaci(n - 2)
```

Faktorijel prorodnog broja n, definiše se sa $n! = n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1$, pri čemu je $0! = 1$. Očigledno, može se zapisati i $n! = n \cdot (n-1)!$, pa se računanje svodi na *manje složenu* verziju *istog* problema (r7). Bazni slučaj nastupa kada je n jedanko jedan (r4, 5). Uslov iz r4 proširen je tako da funkcija vraća tačan rezultat, za specijalni slučaj kada je $n = 0$. Rekurzivni postupak za 4! ilustrovan je *stablom rekurzije*[9] (slika 4.3).



Slika 4.3: Stablo rekurzije: prilikom svakog poziva funkcije `faktorijel()`, kreira se novi imenski prostor. Na dnu stabla poziva prikazan je bazni slučaj (n=1). Rezultati propagiraju unazad, sve dok se ne formira konačni rezultat (4! = 24).

[9] O pojmu stabla bilo je reči u glavi 3, kada je razmatrana heuristika pohlepne pretrage.

U slučaju izračunavanja n-tog člana Fibonačijevog niza, direktno se primenjuje rekurzivna definicija po kojoj je sledeći broj jednak zbiru prethodna dva: $F(n) = F(n-1)+F(n-2)$ (r17). Za razliku od iterativne verzije rešenja, o kojoj je bilo reči u problemu 3.6, ovde se za prva dva člana uzimaju brojevi 0 i 1, a potom se definišu dva bazna slučaja (r12-15). Program, kojim se testiraju novokreirane funkcije, smešten je u odvojenu datoteku:

```
1  # testira rekurzivne implementacije za n! i Fib(n)
2  import rekurzija
3
4  fak = rekurzija.faktorijel
5  fib = rekurzija.fibonaci
6
7  print('Faktorijeli:')
8  for n in range(1, 11):
9      print(fak(n), end=' ')
10
11 print('\n\nFibonacijevi brojevi:')
12 for n in range(1, 11):
13     print(fib(n), end=' ')
```

Program prvo učitava modul `rekurzija`, koji stavlja na raspolaganje dve prethodno opisane rekurzivne funkcije (r2). Kako su funkcije po svojoj prirodi objekti, one se mogu dodeliti promenljivima `fak` i `fib` (r4-5). Ovo je učinjeno kako bi se izbeglo navođenje punog imena funkcije koje uključuje ime modula u notaciji sa tačkom. Treba se podsetiti da sekvenca `\n` prouzrokuje prelazak u novi red (r11), dok se sa `end=' '`, u funkciji `print()` nastavlja ispis u istom redu, posle jednog blanko mesta (r9, 13):

```
Faktorijeli:
1 2 6 24 120 720 5040 40320 362880 3628800

Fibonacijevi brojevi:
0 1 1 2 3 5 8 13 21 34
```

Problem 4.5 — Broj redova u sali. Petar stoji na bini pozorišne sale koja je ispunjena do poslednjeg mesta. Kako on može saznati broj redova u sali, a da se pri tom ne posluži brojanjem? Pretpostaviti da redovi nisu numerisani. ■

Petar bi mogao pretpostaviti da je poslednji red numerisan brojem jedan, onaj do njega brojem dva i tako redom. Tada bi prvi red do bine bio numerisan brojem koji označava broj redova u sali. Sledi algoritam za rešavanje problema u prirodnom jeziku:

Algoritam 4.2 — Broj redova u sali.

Ako neko pita osobu X za njen broj reda onda:

1. X se okreće i pita osobu Y, iza sebe, za njen broj reda, pa kada dobije odgovor, uvećava ga za jedan i rezultat saopštava na glas.
2. ako X, po okretanju, nema iza sebe ni jedan red, onda odgovara na glas – "jedan"!

Petar pita osobu, koja sedi u redu do bine, za njen broj reda. Osoba će mu, uz izvesno kašnjenje, saopštiti koliko ima redova u sali (njen broj reda)!

Treba zapaziti da umesto Petra, kome je brojanje zabranjeno, to čine gledaoci u sali. Pri rešavanju problema, često je potrebno *promeniti perspektivu* (ugao gledanja na problem), kako bi se isti rešio!

Pažljivom analizom algoritma 4.2, uočava se rekurzivna priroda rešenja, kojom se problem određivanja broja reda, svodi na svoju jednostavniju verziju: stav 1 formuliše određivanje broja reda, R, preko određivanja broja reda iza R. Ako je taj broj poznat, onda se R numeriše za jedan većim brojem. Redukovanje kompleksnosti problema, putem rekurzije, *obavezno* treba da se završi u nekom od baznih slučajeva. Tada se rešenje dobija trivijalnim postupkom u kome su poznati svi neophodni elementi za rešavanje – stavka 2. Rekurzivno rešavanje problema, po svojoj prirodi, podseća na izvođenje dokaza putem matematičke indukcije. Sledi program koji simulira brojanje na opisani način, uz učinjenu pretpostavku za nepoznati broj redova u sali:

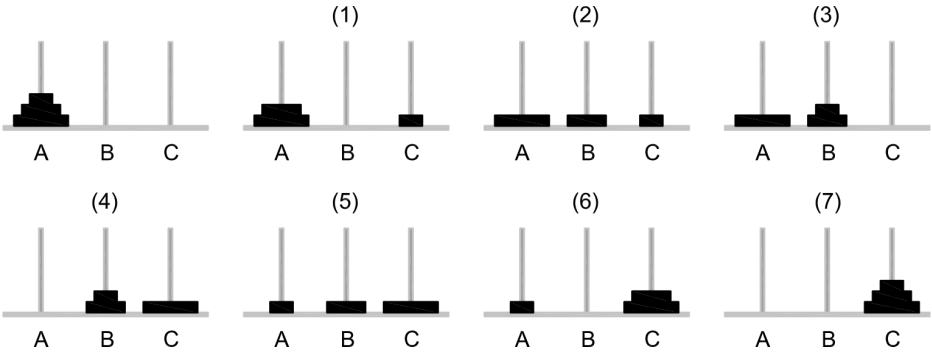
Program 4.5 — Broj redova u sali.

```
1  # Koliko ima redova u sali, prvi red je najdalje od bine
2
3  def pitaj_za_broj_reda(n):
4      '''n: nepoznati broj tekućeg reda, vraća br. tekućeg reda'''
5
6      if n == 1:  # bazni slučaj - pri okretanju, iza nema nikoga!
7          return 1
8      else:
9          return 1 + pitaj_za_broj_reda(n-1)
10
11 # ukupan broj redova, pravimo se da je nepoznat
12 nepoznati_broj_redova = int(input('Stvaran broj redova? '))
```

```
# Petar postavlja pitanje osobi iz prvog reda do bine
odgovor = pitaj_za_broj_reda(nepoznati_broj_redova)
print('Sala ima', nepoznati_broj_redova, 'redova.')
print('Algoritam pronalazi', odgovor, 'redova.')
```

```
Stvaran broj redova? 50
Sala ima 50 redova.
Algoritam pronalazi 50 redova.
```

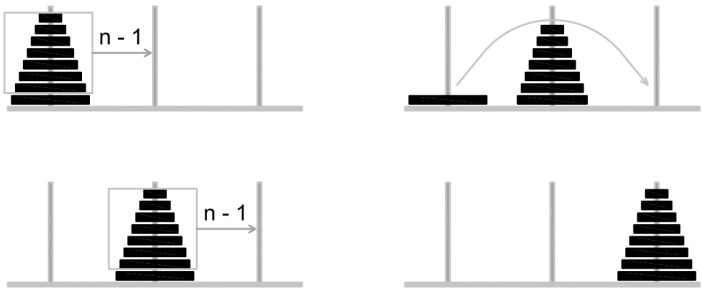
Problem 4.6 — Hanojske kule - Edouard Lucas, 1883. U dvorištu budističkog manastira u Hanoju, Vijetnam, postavljena su tri tanka drvena štapa, A, B i C. Na štapu A poslagano je n drvenih diskova koji su bušni u sredini, tako da mogu da se navuku na štapove. Diskovi su, u početnom položaju, poslagani od najvećeg ka najmanjem (slika 4.4 - gore levo). Potrebno je prebaciti diskove sa A na C, u svakom koraku po jedan, tako da budu poslagani na ciljni štap, isto kao i u početnom položaju. Pri prebacivanju se koriste svi štapovi, ali tako da, ni u jednom trenutku, veći disk ne može stajati preko manjeg. Napisati funkciju koja ispisuje poteze za rešavanje problema u najmanje koraka. Koliko vremena je potrebno da se reši problem sa n diskova, ako prebacivanje diska, sa štapa na štap, traje jednu sekundu? ■



Slika 4.4: Hanojske kule: sekvenca rešavanja u 7 poteza, za n=3.

Uočiti da rešavanje problema Hanojskih kula *ne zavisi* od numeracije štapova – ako je poznat postupak za prebacivanje sa A na C, onda se isti postupak može primeniti i na bilo koji drugi par štapova, pošto se izvrši renumeracija (A će uvek označavati početni, a C krajnji štap). Problem se može rešiti primenom rekurzije, tako što se verzija sa n diskova, svodi na rešavanje dve verzije sa $n-1$ diskova (slika 4.5).

U svakom koraku rekurzije, problem se svodi na rešavanje verzije sa jednim diskom manje. Na ovaj način, stiže se do trivijalnog baznog slučaja, kada više nema diskova za prebacivanje! Uočiti da predloženi postupak vodi ka minimalnom broju poteza, pošto



Slika 4.5: Hanojske kule: rekurzivni postupak za n diskova. Prvo se, sa polaznog (A), na pomoćni štap (B), prebacuje $n-1$ disk. Potom se, sa A, najveći disk prebacuje na ciljni štap C. Konačno, treba prebaciti $n-1$ disk, sa B na C. Prebacivanje $n-1$ diskova, sa A na B, odnosno sa B na C, vrši se na isti način, s tim da polazni, ciljni i pomoćni štap menjaju oznake.

se u svakom rekurzivnom koraku, najveći disk prebacuje u samo jednom potezu, sa polaznog na ciljni štap, a bazni slučaj ne zahteva nijedan potez! Tvrđenje se može jednostavno dokazati matematičkom indukcijom.

Program 4.6 — Hanojske kule - Edouard Lucas, 1883..

```
1  # Hanojske kule
2
3  def hanoj(n, A, B, C):
4      ''' n - broj diskova,
5          A - oznaka stapa SA koga treba prebaciti diskove
6          B - oznaka pomocnog stapa
7          C - oznaka stapa NA koji treba prebaciti diskove'''
8      if n > 0:
9          hanoj(n-1, A, C, B)
10         print('disk', n, 'sa', A, 'na', C) #najv. disk sa A na C
11         hanoj(n-1, B, A, C)
12
13 hanoj(3, 'A', 'B', 'C')
```

```
disk 1 sa A na C
disk 2 sa A na B
disk 1 sa C na B
disk 3 sa A na C
disk 1 sa B na A
disk 2 sa B na C
disk 1 sa A na C
```

Funkcija, u svakom rekurzivnom koraku, osim u baznom, ispisuje redni broj diska koji se prebacuje (r10). Diskovi su numerisani od 1 do n, pri čemu je, sa 1, označen najmanji disk.

Da bi se odredio minimalni broj poteza za rešavanje problema sa n diskova, potrebno je neznatno preraditi funkciju `hanoj()`, tako da vraća ukupan broj koraka načinjenih pri svakom pozivu. U svakom pozivu, osim u baznom slučaju, pored poteza koji prebacuje najveći preostali disk, treba pribrojati i poteze za prebacivanje preostalih diskova, u dva rekurzivna poziva. U baznom slučaju, funkcija treba da vrati nulu. Program koji određuje minimalni broj poteza, za nekoliko vrednosti broja n, dat je u sledećem prikazu:

```
# Hanojske kule, vraća broj poteza

def hanoj2(n, A, B, C):
    ''' n - broj diskova,
        A - oznaka stapa SA koga treba prebaciti diskove
        B - oznaka pomocnog stapa
        C - oznaka stapa NA koji treba prebaciti diskove'''
    if n > 0:
        return 1 + hanoj2(n-1, A, C, B) + hanoj2(n-1, B, A, C)
    else:
        return 0

print('n    B(n)')
for n in range(1, 6):
    print(n, '    ', hanoj2(n, 'A', 'B', 'C'))
```

```
n               B(n)
1               1
2               3
3               7
4               15
5               31
```

Na osnovu dobijenih rezultata, može se pretpostaviti zavisnost između n i ukupnog minimalnog broja poteza: $B(n) = 2^n - 1$. Dokaz se sprovodi matematičkom indukcijom. Za $n = 1$, tvrđenje očigledno važi. Pretpostavlja se da tvrđenje važi za $n = k$. U slučaju sa $k+1$ diskova, a na osnovu slike 4.5, za pomeranje k diskova, sa A na B, odnosno sa B na C, potrebno je po $B(k)$ poteza. Za pomeranje najvećeg diska, sa A na C, potreban je jedan potez. Otuda je $B(k+1) = 2B(k)+1$. Kako po pretpostavci važi $B(k) = 2^k - 1$,

sledi $B(k+1) = 2(2^k - 1) + 1 = 2^{k+1} - 1$, što je i trebalo dokazati. Na primer, za 32 diska i vreme od jedne sekunde po potezu, potrebno je $2^{32} - 1$ sekundi, ili nešto više od 136 godina da bi se problem rešio!

Primer Hanojskih kula ilustruje poteškoće sa kojima se mozak susreće prilikom interpretacije brzine rasta eksponencijalne funkcije. Na primer, za varijantu sa 64 diska i vreme prebacivanja od jedne sekunde po disku, potrebno je da protekne približno 585 milijardi godina, da bi se problem rešio! Problemi čije vreme rešavanja raste po eksponencijalnom zakonu, u odnosu na ulazne parametre, predstavljaju izuzetan izazov, čak i za najbrže računare.

4.3.2 Prednosti i mane rekurzivnog pristupa

Na kraju izlaganja o rekurziji, izlažu se potencijalne prednosti i mane, u odnosu na iterativne postupke iz glave 3. Rekurzivno rešenje često se ostvaruje *kraćim* izvornim kodom, pa se smanjuje mogućnost za potencijalne greške. Povećana je i čitljivost (elegancija) koda, što povoljno utiče na jednostavnije održavanje programa. Osim toga, postoje praktični problemi u kojima bi iterativno rešenje bilo dosta složenije, pa je rekurzija prirodan izbor (na primer, problem obilaska stabla po dubini).

Pri svakom rekurzivnom pozivu, kreira se *novi* imenski prostor sa pripadajućom tabelom imena, pa je utrošak memorijskih resursa veći u odnosu na ekvivalentno iterativno rešenje. U slučaju velikog broja rekurzivnih poziva, program bi mogao da prekine sa radom, usled prekomerne potrošnje memorijskih resursa. Sa stanovišta vremenske efikasnosti, rekurzivno rešenje obično je *sporije* od iterativnog, pošto se mora potrošiti izvesno režijsko vreme za prenos parametara, kreiranje imenskog prostora pri pozivu i uklanjanje na izlasku iz funkcije, te konačno vraćanje rezultata u pozivajuću celinu. Jednostavan primer kojim se izračunava suma prvih n prirodnih brojeva, realizovan u obe varijante, ilustruje tipičan problem koji se odnosi na utrošak memorijskih resursa:

```
1  def f(n):      # iterativna suma
2      s = 0
3      for i in range(n+1):
4          s += i
5      return s
6
7  def g(n):      # rekurzivna suma
8      if n > 0:
9          return n + g(n-1)
10     else:
11         return 0
12
```

```
13 print('iterativno rešenje:', f(1000))
14 print('rekurzivno rešenje:', g(1000))
```

```
iterativno rešenje: 500500
Traceback (most recent call last):
--------------- IZOSTAVLJENO ------------------
RecursionError: maximum recursion depth exceeded in comparison
```

Interpreter, zbog mogućeg prekomernog utroška memorijskih resursa, ograničava broj rekurzivnih poziva. U gornjem primeru, program prekida sa radom jer je dostigao *granični broj* rekurzivnih poziva. Ovaj broj *može* se eksplicitno povećati na *sopstveni* rizik. Na kraju, treba istaći da je moguće *transformisati* sva rekurzivna rešenja, u odgovarajuće iterativne varijante i obrnuto, ali ovde o tome neće biti reči.

ŠTA JE NAUČENO

- Funkcijom se, pod jednim imenom, apstrahuje niz algoritamskih koraka za obavljanje nekog posla.
- Funkcije omogućavaju dekompoziciju složenih postupaka na jednostavnije procesne celine.
- Funkcije obično zavise od ulaznih parametara, kojima se parametrizuje rešavanje problema.
- Funkcija obavezno vraća rezultate svoga rada, putem naredbe `return`, ili ako se obave sve naredbe iz bloka funkcije - tada je rezultat predstavljen objektom tipa `None`.
- Prilikom poziva funkcije, pozivajuća celina nema potrebu da zna kako funkcija radi, već šta radi i kako se poziva. Argumenti se u pozivu navode po redosledu iz potpisa funkcije, ili po proizvoljnom redu, ako se imenuju po formalnim parametrima iz potpisa.
- Opcioni parametri funkcije mogu se pri pozivu izostaviti i tada dobijaju podrazumevane, predefinisane vrednosti.
- Funkcije mogu pozivati druge, kao i definisati unutrašnje funkcije.
- Moduli i paketi predstavljaju više hijerarhijske celine za organizaciju kompleksnih programskih sistema.

- Moduli sadrže srodne funkcije, koje se mogu koristiti u različitim aplikacijama, a paketi organizuju module, u nezavisne softverske komponente, za određenu namenu.
- Promenljiva definisana u modulu ima globalni karakter - može joj se pristupati iz svih funkcija unutar modula, kao i iz onih modula u koje je definišući modul uveden, putem naredbe `import`.
- Ista imena, u različitim modulima, dostupna su uz korišćenje notacije sa tačkom oblika `ime_modula.ime_objekta`.
- Lokalna promenljiva definiše se unutar funkcije i može joj se pristupati samo unutar iste.
- Pri svakom pozivu funkcije kreira se novi imenski prostor u kome se, unutar tabele imena, čuvaju preslikavanja između lokalnih imena i objekata na koje ta imena (promenljive) ukazuju.
- Argumenti funkcije prenose se po sistemu kopiranja objektnih referenci. Otuda se, nepromenljivi objektni tipovi, poput brojeva i teksta, kada su preneti kao argumenti, ponašaju u funkciji kao lokalne promenljive.
- Funkcije imaju objektnu prirodu: mogu se dodeljivati promenljivim i prenositi u druge funkcije kao argumenti.
- Rekurzivni algoritmi rešavaju problem tako što ga, u seriji funkcijskih poziva, redukuju na jednostavnije verzije istog problema, sve dok se početni problem ne svede na bazni slučaj, u kome je rešenje trivijalno. Rešenje baznog slučaja propagira se unazad, sve do prvobitno pozvane funkcije, koja dobija sve elemente za traženo rešenje.
- Rekurzivna rešenja mogu biti memorijski veoma skupa jer se, prilikom svakog rekurzivnog poziva, kreira novi imenski prostor sa svojim kompletom lokalnih promenljivih.