4-ProgramiranjeS

Isti matematički problem je često moguće riješiti na različite načine, putem algoritama iza kojih stoje sasvim različiti načini razmišljanja. Razložimo to na primjeru funkcije faktorijel.

```
factorial(7)
```

```
5040
```

Klasični način programiranja, upotrebljavan od dana prvih računala, je tzv. proceduralno (ili imperativno) programiranje kod kojeg na program gledamo kao na niz naredbi koje obično postupno mijenjaju vrijednosti nekih varijabli pohranjenih u memoriji računala:

```
def factorialProc(n):
    fac = 1
    i = 1
    while i<n:
        i = i + 1
        fac = fac * i
    return fac
factorialProc(7)
```

```
5040
```

Zanemarite zasad preciznu sintaksu komande while. Ovdje je očito riječ o standardnom algoritmu kakvog bismo isprogramirali u bilo kojem proceduralnom jeziku poput Fortrana ili C-a: Imamo petlju koja se prolazi n puta i svaki puta množimo rezultat prošlog prolaza sa za 1 većim brojem.

Međutim, postoje i drugačiji pristupi. Tako je kod funkcionalnog programiranja naglasak na izvrijednjavanju funkcija tj. primjeni funkcija na izraze. Faktorijel prirodno zamišljamo kao funkciju koja daje "umnožak svih brojeva od 1 do zadanog broja".

Kao prvo, treba nam modul 'operator' koji implementira funkcije koje odgovaraju standardnim matematičkim operacijama *, +, -,...

```
import operator
```

```
import operator
(operator.mul(2,3), operator.add(2,3))
```

```
(6, 5)
```

Zatim koristimo funkciju reduce(), koja kumulativno primjenjuje prvi argument (koji mora biti funkcija) na elemente drugog argumenta:

```
f = function('f')
reduce(f, range(1,5))
```

```
f(f(f(1, 2), 3), 4)
```

```
reduce(operator.mul, range(1,8))
```

```
5040
```

Primijetite da nam ovdje nisu trebale pomoćne varijable (poput i i fac iz factorialProc), ali nam je trebalo više memorije jer smo trebali pohraniti čitavu listu [1, 2, ...] koju daje range().

Proceduralno programiranje

Osnovna paradigma proceduralnog programiranja je da se kreće od nekog definiranog stanja programa (vrijednosti varijabli i nizova varijabli u memoriji računala). To stanje se specificira početnim naredbama pridruživanja (x=0, i=1) (assignment) i onda se algoritam izvodi primjenom raznih grananja, petlji i potprograma koji svi mijenjaju to stanje i vode na konačno stanje u kojem vrijednosti nekih varijabli odgovaraju rješenju problema.

if-then grananje

Sintaksa je očita iz primjera:

```
if 2>3:
    print "veći je"
elif 2==3:
    print "jednak je"
else:
    print "manji je"
```

```
manji je
```

Ovdje treba paziti na uvlačenje blokova koda (tamo gdje bi u C-u bile vitičaste zagrade).

U uvjetima se mogu koristiti standardni logički operatori and, or, not, ...

```
if 2>3 or 2<3:
    print "nisu isti"
```

```
nisu isti
```

Petlje

Do-petlje ne postoje, ali postoje standardne while- i for-petlje. Primjer while petlje se vidi u primjeru factorialProc() na početku ovog odjeljka. For petlja ima specifičnu sintaksu

```
for x in <iterable>:
    <body>
```

gdje je <iterable> lista, tuple, skup, riječnik, ... Za prijevremeno iskakanje iz petlje postoji komanda "break". Npr. slijedeći algoritam pronalazi prvi cijeli broj čiji rastav sadrži više od pet različitih prostih faktora.

```
%time
for i in range(1,1e5):
    if len(factor(i))>5:
        break
print i
```

```
30030
CPU time: 0.50 s,  Wall time: 0.50 s
```

Ovo je relativno neelegantan program za pronalaženje najmanjeg broja koji ima više od pet različitih faktora. Bolje bi bilo

```
%time
i = 1
while len(factor(i))<6:
    i +=1
print i
```

```
30030
CPU time: 0.41 s,  Wall time: 0.41 s
```

a najbolje

```
reduce(operator.mul, primes_first_n(6))
```

```
30030
```

ali ovo zadnje spada već u funkcionalno programiranje.

Zadatak 4.1: Isprogramirajte proceduralno funkciju fibProc(n) koja računa n-ti član Fibonaccijevog niza (niz kod kojeg je svaki član definiran kao zbroj prethodna dva, a javlja se pri analizi idealizirane populacije zečeva).

Zadatak 4.2: Konstruirajte funkciju brown(n) koja daje listu $[(x_0, y_0), (x_1, y_1), \ldots]$ pozicija čestice koja se giba u ravnini slučajnim gibanjem koje je definirano rekurzijama $x_i = x_{i-1} + r$ i $y_i = y_{i-1} + s$ gdje su r i s slučajni brojevi između -1 i 1. Nacrtajte odgovarajuću putanju čestice.

Naputak:

- Inicijalizirajte listu pozicija tako da sadrži (0,0) kao prvi vektor pozicije i putem petlje dodajte u svakom koraku toj listi novi slučajni vektor.
- Za crtanje možete koristiti list_plot(<točke>, plotjoined=True)

Zadatak 4.3: Konstruirajte funkciju walk1D() tako da simulira tzv. jednodimenzionalnog nasumičnog šetača. Svaki šetačev korak treba biti iste, jedinične duljine, ulijevo ili udesno. Dakle, umjesto vektora (r, s) iz gornjeg Brownovog gibanja trebamo slučajan odabir koraka +1 ili -1. Provjerite ispravnost tvrdnje da je očekivana udaljenost nasumičnog jednodimenzionalnog šetača od ishodišta nakon n koraka $\sqrt{n}$.

Funkcionalno programiranje

Funkcionalno programiranje je stil programiranja koji stavlja naglasak na izvrijednjavanje izraza, a ne na sukcesivno izvršavanje komandi. Kod čistih funkcionalnih programa obično nema pomoćnih varijabli i nepotrebnih

popratnih efekata izvrijednjavanja funkcija. U tom smislu funkcionalno programiranje je pomalo slično radu s tabličnim kalkulatorom (npr. MS Excel): redosljed izračunavanja ćelija je
nebitan tj. očekujemo automatsku konzistenciju. Isto, vrijednosti ćelija su dane izrazima, a ne slijedovima komandi. (Misli se na Excel, a ne Sage ćelije.)
(Više informacija o funkcionalnom programiranju nudi Functional Programming FAQ ili WWW stranice Haskell funkcionalnog jezika.)

Takav stil programiranja često rabi nekoliko specijalnih funkcija (mogli bismo ih zvati "meta-funkcije") čija je uloga upravljanje primjenivanjem drugih funkcija koje dolaze kao argumenti ovih meta-funkcija. Možda najvažnija takva funkcija je map():

```
f = function('f')
map(f, range(4))
```

```
[f(0), f(1), f(2), f(3)]
```

Ukoliko funkcija prima više argumenata, može se map()-irati na više listi:

```
map(f, range(4), range(3,7))
```

```
[f(0, 3), f(1, 4), f(2, 5), f(3, 6)]
```

Recimo da sad želimo ispisati tablicu s nizom prirodnih brojeva i njihovih faktorijela. Možemo prvo postupiti tako da prvo definiramo pomoćnu funkciju koja generira jedan red tablice i onda je pomoću map() primijenimo na niz brojeva:

```
def auxfun(k):
    return (k, factorial(k))
```

```
map(auxfun, range(7))
```

```
[(0, 1), (1, 1), (2, 2), (3, 6), (4, 24), (5, 120), (6, 720)]
```

Međutim, baš kao i pomoćne varijable, tako ni pomoćne funkcije nisu u duhu funkcionalnog programiranja. Zato postoje tzv. lambda-funkcije koje su bezimene i definiraju se i upotrebljavaju na slijedeći način:

```
map(lambda k: (k, factorial(k)), range(7))
```

```
[(0, 1), (1, 1), (2, 2), (3, 6), (4, 24), (5, 120), (6, 720)]
```

```
matrix(_)  # čitljiviji ispis bez uptrebe formatirajućih
stringova
```

```
[  0   1]
[  1   1]
[  2   2]
[  3   6]
[  4  24]
[  5 120]
[  6 720]
```

Treba uočiti kako je često upotrebu funkcije map() i lambda-funkcija moguće izbjeći korištenjem obuhvaćanja liste. Npr, gornji primjer se elegantnije realizira ovako:

```
[(k, factorial(k)) for k in range(7)]
```

```
[(0, 1), (1, 1), (2, 2), (3, 6), (4, 24), (5, 120), (6, 720)]
```

Kako su sage/python funkcije punopravni objekti, možemo i sami definirati funkcije koje primaju druge funkcije kao argumente (to smo već radili u prošlom odjeljku). Evo funkcije koja implementira kompoziciju funkcija:

```
def compose(f, n, x):
    """Uzastopno komponiranje funkcije sa samom sobom n
puta."""
    if n <= 0:
        return x
    x = f(x)
    for i in range(n-1):
        x = f(x)
    return x
```

Npr. tzv. zlatni omjer $(\sqrt{5}+1)$ se može izraziti kao beskonačni razlomak

$$\frac{\sqrt{5}+1}{2} = 1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1+\ddots}}} = 1.618034\ldots$$

Takav razlomak možemo izvrijedniti na slijedeći način:

```
gr = compose(lambda x: 1+1/x, 5, x); gr
```

```
1/(1/(1/(1/(1/x + 1) + 1) + 1) + 1) + 1
```

```
gr.subs(x=1).n()
```

```
1.62500000000000
```

Ili, uz povećanje točnosti:

```
print "Golden ratio =\n%.14f" % n((1+sqrt(5))/2)
compose(lambda x: 1+1/x, 32, x).subs(x=1).n()
```

```
Golden ratio =
1.61803398874989
1.61803398874986
```

Primjer razvoja funkcionalnog programa

Promotrimo razvoj programa koji ispisuje tablicu frekvencija pojavljivanja elemenata u listi. Načinimo prvo jednostavnu testnu listu

```
lst = ['a', 'c','b', 'a', 'c', 'a', 'd', 'b', 'c','d', 'c']
```

Metoda liste koja daje broj pojavljivanja nekog elementa je count()

```
lst.count('c')
```

```
4
```

Da bismo ispisivali tablicu trebamo listu oblika [(element1, frekvencija elementa1), (element2, frekvencija elementa2),...]. Element te liste (red tablice) se može dobiti ovako:

```
def el(x):
    return (x, lst.count(x))
el('c')
```

```
('c', 4)
```

Tablicu frekvencija za različite elemente dobijemo korištenjem lambda-funkcije analogne el() i njenom primjenom pomoću funkcije map() na popis elemenata:

```
map(el, ['a', 'b', 'c', 'd'])
```

```
[('a', 3), ('b', 2), ('c', 4), ('d', 2)]
```

```
map(lambda x: (x, lst.count(x)), ['a', 'b', 'c', 'd'])
```

```
[('a', 3), ('b', 2), ('c', 4), ('d', 2)]
```

To je sad to, jedino još želimo izbjeći da sami moramo ručno identificirati koji se sve elementi pojavljuju u listi. To nam može raditi funkcija uniq() koja izbacuje duplikate iz liste:

```
res = map(lambda x: (x, lst.count(x)), uniq(lst)); res
```

```
[('a', 3), ('b', 2), ('c', 4), ('d', 2)]
```

Kao zadnju stvar, poželjno je listu sortirati po frekvencijama. Funkcija sorted() je već definirana tako da zna sortirati liste različitih objekata, no ovdje bi po defaultu sortirala parove po prvom elementu i to po abecednom redu, ...

```
sorted(res)
```

```
[('a', 3), ('b', 2), ('c', 4), ('d', 2)]
```

... a nama treba sortiranje po drugom elementu i to po veličini. Stoga moramo putem opcionalnog argumenta 'key' funkciji sorted() proslijediti funkciju koja će vraćati onaj dio elementa liste po kojem želimo sortiranje:

```
def keyfun(item):
    """Return last element of item."""
    return item[-1]
```

```
sorted(res, key=keyfun, reverse=True)
```

```
[('c', 4), ('a', 3), ('b', 2), ('d', 2)]
```

(Opcionalni argument reverse daje silazno sortiranje umjesto defaultnog uzlaznog.)

Naravno, i ovu pomoćnu funkciju keyfun() možemo zamijeniti lambda funkcijom:

```
sorted(res, key=lambda it: it[-1], reverse=True)
```

```
[('c', 4), ('a', 3), ('b', 2), ('d', 2)]
```

I to je to. Sad definiramo kompletnu funkciju:

```
def frequencies(lst):
    """Elements of list lst with their frequencies."""
    return sorted(map(lambda x: (x, lst.count(x)), set(lst)),
           key=lambda it: it[-1], reverse=True)
```

```
for it in frequencies(lst):
    print "%s: %d" % it
```

```
c: 4
a: 3
b: 2
d: 2
```

Primijenimo to na frekvenciju pojavljivanja slova u Shakespeareovom Mletačkom trgovcu (zapravo koristimo engleski original The Merchant of Venice, skinut sa WWW stranica projekta Gutenberg)

```
load('http://www.phy.hr/~kkumer/sp/venice.py')
```

```
--- Učitan Mletački trgovac kao string u varijablu 'venice'.
```

```
print venice[:300]
len(venice)
```

```
***The Project Gutenberg's Etext of Shakespeare's First Folio***
**********************The Merchant of Venice**********************

The Merchant of Venice

Actus primus.

Enter Anthonio, Salarino, and Salanio.

  Anthonio. In sooth I know not why I am so sad,
It wearies me: you say it wearies you;
B
117687
```

```
for it in frequencies(venice)[:10]:
    print "%s:  %d" % it
```

```
 :  20958
e:  12267
o:  7277
t:  7241
a:  6244
h:  5570
n:  5534
s:  5326
r:  5241
i:  5215
```

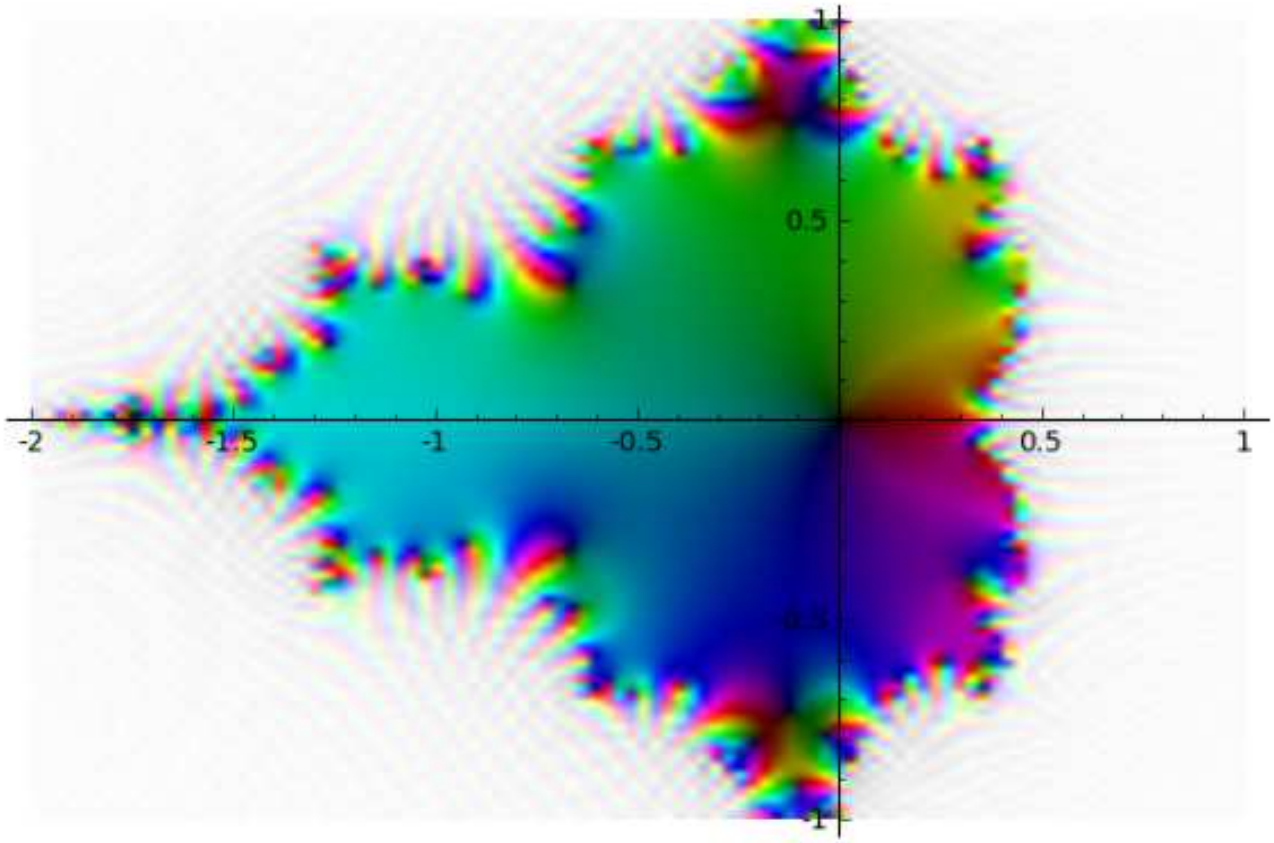
Vidimo da je "e" daleko najčešće slovo (poslije razmaka), činjenica vrlo važna pri dešifriranju engleskih tekstova.

Kao netrivijalniji primjer upotrebe gore definirane funkcije compose() možemo nacrtati tzv. Mandelbrotov skup, definiran kao skup svih točaka c kompleksne ravnine za koje iteracija

$$z_0 = 0\ ; \qquad z_{n+1} = z_n^2 + c$$

ne divergira.

```
complex_plot(compose(lambda z: z^2+x, 8, x), (-2, 1), (-1, 1))
```



(Analizirajte sami kako radi ovaj "program".)

Zadatak 4.4: Definirajte funkciju allequal() koja testira da li su svi elementi neke liste jednaki i iskoristite je da pronađete neprekidni niz od 6 istih znamenki u prvih 1000 decimala broja π. (Za pretvaranje broja u listu znamenaka možete iskoristiti funkciju str().) Koristite ili obuhvaćanje liste ili map() tj. nije dopušteno korištenje petlji (for, while, ...) osim eventualno unutar funkcije allequal(), gdje to isto nije nužno (Hint: all()).

Zadatak 4.5: Logističko preslikavanje je dano iteracijskom formulom $x_{n+1} = \lambda x_n(1 - x_n)$. Za male vrijednosti parametra λ, to preslikavanje za veliki n konvergira k jednoj vrijednosti. Kad λ raste, negdje blizu $\lambda = 3$, dolazi do bifurkacije i iteracije više ne konvergiraju već skaču između dvije vrijednosti. S daljnjim rastom λ dolazi do nove bifurkacije i sustav se za veliki n ponavlja s periodom četiri, itd. Rezultat se može prikazati kao tzv. Feigenbaumov bifurkacijski dijagram (vidi sliku ovdje). Nacrtajte ga tako da prvo definirate funkciju composelist() koja je analogna gornjoj funkciji compose(), ali ne vraća samo kranji rezultat kompozicije funkcija već i listu

sa svom međukoracima. Nakon toga iskoristite tu funkciju za iteriranje logističkog preslikavanja. Eliminirajte prvi dio liste (dok se iteracije ne stabiliziraju) i nacrtajte drugi dio pomoću list_plot(...., pointsize=1). Uočite da svaka vrijednost apscise lambda() traži posebno iteriranje.

Zadatak 4.6(*): Odredite najmanji prirodni broj koji se *ne može* dobiti iz brojeva 2, 3, 4 i 5 korištenjem operacija zbrajanja, oduzimanja i množenja, a gdje se svaki broj smije koristiti najviše jednom. (Npr. 1=3-2, 2=3-5+4, ..., 6=2*3, ..., 14=4*5-3*2, ... Naputak: Od koristi se možda može pokazati već ranije korišteni modul 'operator', te funkcije permutations() i combinations().

Zadatak 4.7: Prim-brojevi blizanci su parovi prim-brojeva koji se razlikuju za dva, poput (3,5) ili (17,19). V. Brun je 1919. dokazao da suma recipročnih vrijednosti prim-brojeva blizanaca konvergira

$$B = \left(\frac{1}{3} + \frac{1}{5}\right) + \left(\frac{1}{5} + \frac{1}{7}\right) + \left(\frac{1}{11} + \frac{1}{13}\right) + \left(\frac{1}{17} + \frac{1}{19}\right) + \cdots$$

Ovako preciznu vrijednost je vrlo teško dobiti, no napišite funkciju brun(n) koristeći listu od prvih n prim-brojeva. Inače, zanimljivo je da je upravo računalno određivanje ove konstante ukazalo na bug u prvoj generaciji Pentium procesora.

Literatura:

- Functional programming for Mathematicians
- Opsežniji tekst koji i kritizira upotrebu funkcionalnog programiranja u pythonu.