Računarstvo i praktikum / Uvod u python

 [1]

Maro Cvitan

1. veljače 2019.

Sadržaj

Uvod

Ovaj dokument služi kao pomoć pri učenju osnova programskog jezika python u okviru kolegija Računarstvo i praktikum na prvoj godini istraživačkog smjera studija fizike na Fizičkom odsjeku te studija geofizike na Geofizičkom odsjeku Prirodoslovno-matematičkog fakulteta Sveučilišta u Zagrebu. Pretpostavlja se osnovno znanje C-a. Pretpostavlja se da se primjeri izvršavaju na Unix-like operativnom sustavu (linux, OSX). Primjeri u pythonu i C-u će raditi i na Windows-ima, ali će ih trebati pokrenuti na odgovarajući način.

1.1 Interpreter

- Jezik python napravljen je s namjerom da olakša (=ubrza) pisanje programa. Na primjer, za razliku od C-a, nema deklariranja varijabli unaprijed, a dopušteno je miješanje tipova (npr. dopušteno je `a=1` i nakon toga `a="xy"`).
- Programe napisane u C-u potrebno je prvo prevesti u strojni jezik, a zatim izvršiti. Python umjesto toga koristi "simultano prevođenje" tj. program se prevodi u strojni jezik malo po malo tijekom izvršavanja. Programi koji prevode sve odjednom nazivaju se *compiler*i (npr. `gcc`), a programi koji "simultano prevode" nazivaju se *interpreter*i (npr. `python`). Više na http://en.wikipedia.org/wiki/Interpreter_(computing).
- 2 načina korištenja:
  - interaktivna ljuska: ljusku pokrenemo s `python` (ili `python3`), upisujemo/izvršavamo naredbu po naredbu, iz ljuske izađemo pomoću CTRL-d — ovakav način nam omogućuje brzo isprobavanje dijelova programa. Primjere u kojima se koristi ljuska možete prepoznati po oznaci >>> koju ljuska ispisuje kad očekuje unos naredbe.
  - samostalni programi: upišemo cijeli program pomoću tekst-editora i snimimo ga npr. pod imenom `prg.py` zatim pokrenemo pomoću `python prg.py` (ili `python3 prg.py`).

    Napomena: Ako postavimo da je datoteka `prg.py` izvršna, pomoću `chmod u+x prg.py`, i ako je prva redak programa `prg.py` standardno zaglavlje (vidi *Uobičajeno zaglavlje*, *Zadaća 6 (itertools.groupby())*), tada se program može pokrenuti samo s `./prg.py`.

- Postoje 2 glavne verzije python-a. Mogu se preuzeti (besplatne su) s https://www.python.org. To su:
  - python 2 (npr. python 2.6, python 2.7)
  - python 3 (npr. python 3.6, python 3.7)

  Pozor: Python 3 **nije** *kompatibilan unazad* s pythonom 2 (to znači da programe za python 2 treba prepraviti ako ih se želi izvršiti pomoću pythona 3).
- Kako se čini da python 2 neće biti podržan nakon 2020 godine (https://www.python.org/dev/peps/pep-0373/), u nastavku ćemo koristiti python 3, iako će većina primjera, uz sitne popravke, raditi i na pythonu 2.

1.2 Kako učiti

1. S obzirom da već imate praksu u C-u prilično je lako naučiti osnove pythona. Za tu svrhu možete koristiti primjere s ovih stranica ili neki od tutoriala:
   - https://docs.python.org/3/tutorial/

- https://www.tutorialspoint.com/python3/python_variable_types.htm

Također mogu biti korisni i sljedeći tutoriali za python 2:

- http://cs.stanford.edu/people/nick/python-in-one-easy-lesson/
- https://developers.google.com/edu/python/

2. Početi koristiti python. Pri tome se korisno pomagati već gotovim receptima:

- Npr. ako tražimo *python print string as bytes*, google nas uputi na http://stackoverflow.com/questions/12214801/print-a-string-as-hex-bytes gdje se nalazi gotov recept koji omogućuje ispis pojedinih byteova koji čine string (općenito stranica http://stackoverflow.com je dosta korisna).
- Moguće da će neka rješenja biti vezana za python 2 i bit će ih potrebno prebaciti u verziju 3.

Napomena: Za prebacivanje u verziju 3 u najjednostavnijim primjerima samo će biti potrebno pretvoriti `print a, b` u `print(a,b)`, kao i `a/b` u `a//b` ako se želi cjelobrojno dijeljenje. (U pythonu 3 vrijedi `a//b==floor(a/b)`)

Također prebacivanje iz 2 u 3 olakšava program `2to3` koji dolazi s pythonom. Na webu se mogu naći razne upute za ručno prebacivanje npr. :

– http://python-future.org/compatible_idioms.html

1.3 Dodaci za python

Dodaci (tj. dodatni paketi/biblioteke) za python sadrže funkcije za razne namjene. Dodaci koji se isporučuju s pythonom nazivaju se standardni dodaci. Nestandardni dodaci dobavljaju se iz drugih izvora. Spomenimo neke:

1. Za crtanje grafova u pythonu, koristan je dodatak `matplotlib`

- http://matplotlib.org/users/pyplot_tutorial.html

2. Za rad s poljima brojeva brojeva, koristan je dodatak `numpy`

- http://wiki.scipy.org/Tentative_NumPy_Tutorial

3. Gotove funkcije za numeriku, nalaze se u dodatku `scipy`

- http://docs.scipy.org/doc/scipy/reference/tutorial/index.html

4. U službenim uputstvima za python mogu se naći sve specifikacije za python i njegove standardne dodatke:

   - https://docs.python.org/3/
   - https://docs.python.org/3/library/index.html

   Standardnih dodataka ima dosta. Oni omogućuju programerima pristup sustavu, te donose gotova rješenja za uobičajene probleme. Primjerice, dodatak za tzv. *regular expressions* omogućuje traženje po tekstu. Najjednostavniji oblik *regular expression*a ste već vidjeli prilikom korištenja `scanf()` - npr. `scanf("%[^\n]",s)` učitava u `s` sve do znaka za novi red. Proučavanje detalja tog paketa, naravno, izlazi van okvira kolegija, ali korisno je znati da postoji. Također treba uočiti da *regular expressions* ne postoje isključivo u pythonu nego predstavljaju standard za traženje po tekstu koji koriste i drugi jezici i programi.

   - https://developers.google.com/edu/python/regular-expressions
   - http://www.tutorialspoint.com/python/python_reg_expressions.htm
   - https://docs.python.org/3/howto/regex.html

5. Popisi popularnih dodataka za python korisnih za znanstvene primjene

- https://gist.github.com/sebp/58da862b779489998e8e6088908fbfa5#file-python-scientific-computing-md
- https://medium.com/activewizards-machine-learning-company/top-20-python-libraries-for-data-science-in-2018-2ae7d1db8049

Napomena: S pythonom i alatima numpy, scipy, matplotlib ćete se opet susresti u kolegijima Simboličko programiranje, Klasična mehanika, Kvantna fizika, ..., tako da trud koji uložite sada nije uzalud. Ipak, uputstava ima jako puno i ideja je proučiti detaljnije samo manji dio npr. primjere koji slijede ili neki od navedenih tutoriala, a ostale materijale koristiti kad i ako zatrebaju — kao rječnik ili telefonski imenik.

1.4 Druge implementacije pythona

Gore spomenuti python interpreteri (oni s https://www.python.org) predstavljaju jednu (standardnu) implementaciju jezika python, napisanu u C-u koja se naziva CPython. Također postoje i RPython (podskup pythona) i Cython (http://cython.org, prevoditelj iz pythona u C) kojima je glavni cilj ubrzavanje izvršavanja. Postoje i implementacije jezika python pisane u drugim jezicima: java (Jython), RPython (PyPy), C# (IronPython) itd. Korištenje neke druge implementacije omogućuje lagan pristup bibliotekama jezika u kojem je ta implementacija pisana (ali u tom slučaju neke standardne (CPython-ove) biblioteke mogu biti nepodržane). U nastavku ćemo se baviti samo standardnom, CPython, implementacijom.

Usporedba osnovnih elemenata C-a i python-a

2.1 Pregled često korištenih tipova podataka

Sljedeća tablica prikazuje često korištene tipove podataka u pythonu i referencu na odjeljak gdje se navedeni tip detaljnije diskutira.

opis	*mutable*	*immutable*
jednostavni tipovi, brojevi		`int`, `float`, `complex`, `bool`, `NoneType` (vidi *Osnovni tipovi podataka*)
niz byteova	`bytearray`	`bytes` (vidi *Binarne datoteke*, *Unicode*)
niz unicode znakova (=string)	nema, ali `io.StringIO`, `array.array`, `bytearray`, `list` mogu poslužiti	`str` (*Stringovi i polja (liste)*, *Rad sa stringovima*, detalji u *Unicode*)
niz objekata	`list`	`tuple`
niz objekata određenog tipa ("homogeno" polje)	`array.array`	
niz objekata bez ponavljanja	`set`	`frozenset`
rječnik	`dict` (vidi *Rječnik (Dictionary)*)	

Mutable varijable se mogu mijenjati, *immutable* varijable se ne mogu mijenjati (konstante su). Treba primijetiti da su u C-u primitivni tipovi podataka `int`, `float`, ... *mutable*, a u pythonu su *immutable*. Razlika između *mutable* i *immutable* diskutira se u odjeljku *Pridruživanje lista*.

Osnovne primjere upotrebe možete naći u raznim tutorialima npr. :

- http://www.diveintopython3.net/native-datatypes.html

2.2 Operatori

Operatori su slični kao u C-u. Postoje razlike, npr. postoji poseban operator `//` za cjelobrojno dijeljenje kao i za potenciranje `**`.

Python	C
`and`	`&&`
`or`	||
`a**b`	`pow(a,b)`
`a/b`	`((double)a)/b` (dijeljenje s decimalama)
`a//b`	`a/b` ako su oba cjelobrojni, inače `floor(a/b)`
`not`	`!`
`id(x)`	`&x` (adresa od x)

U pythonu postoji operator `in` koji pretražuje postoji li element u skupu. Npr. u sljedećem primjeru provjeravamo jesu li `1`, `'1'`, `''` elementi liste `a`:

```
>>> a = [1, 2, 3, 'x', '']
>>> print( 1 in a, '1' in a, '' in a )
True False True
```

Direktan rad s memorijom u pythonu nije podržan što znači da ne postoji operator za očitavanje vrijednosti na adresi kao `*` u C-u. Ne postoji operator za očitavanje adrese objekta ali postoji funkcija `id()` koja očitava "identitet" objekta. U CPythonu se kao "identitet" objekta koristi adresa objekta.

Popis operatora s jednostavnim primjerima:

- https://www.tutorialspoint.com/python/python_basic_operators.htm

2.3 Korisne kratice

Umnožak stringa i broja slijepi više primjeraka tog stringa npr. `'xy'*3` daje `'xyxyxy'`. Slično za liste i tupleove: `[1,2]*3` daje `[1,2,1,2,1,2]`.

Korisne kratice su i:

Python	C
`a<b<c`	`a<b && b<c`
`a,b,c = 1,2,3`	`a=1; b=2; c=3;`

Na sličan način funkcija može vratiti "dvije vrijednosti":
`def f(x):`
`  return x+2, x*2`
`a, b = f(10)`
`print( a, b )`
`12 20`
Također, (generatorske) funkcije mogu vratiti više vrijednosti, ali jednu po jednu, što ima prednost da nisu sve odjednom u memoriji (vidi npr. *Pridruživanje iteratora varijablama*).
	U C-u se to obično radi koristeći polja ili strukture što je manje elegantno.

2.3.1 Zadavanje vrijednosti tipa `tuple`, `list`, `set`

Povratnu vrijednost u gornjem primjeru mogli smo pridružiti i jednoj varijabli.

```
def f(x):
  return x+2, x*2
a = f(10)
print( a, type(a) )
```

```
(12, 20) <class 'tuple'>
```

Vidimo da je vrijednost te varijable `(12, 20)` a tip `tuple`.

Napomena: Nizovi vrijednosti tipa `tuple` zadaju se pomoću okruglih zagrada, nizovi tipa `list` pomoću uglatih, nizovi tipa `set` (kao i `dict`) pomoću vitičastih.

U slučaju tipa `tuple` zagrade se u nekim slučajevima mogu izostaviti. To smo vidjeli kod naredbe `return` (npr. `return a,b,c` je isto što i `return (a, b, c)`). Naredbe koje to dopuštaju su npr. `yield`, `for` i pridruživanje.

```
a = (1, 2, 3, 2, 3)
b = [2, 3, 4]
c = frozenset( a )
d = {2, 3, 4}
e = {}
f = set()
g = {'a':1, 'b':2, 'c':3}

print( a, type(a) )
print( b, type(b) )
print( c, type(c) )
print( d, type(d) )
print( e, type(e) )
print( f, type(d) )
print( g, type(g) )
print()

b.extend( a )
d.update( a )

print( b, type(b) )
print( d, type(d) )
```

```
(1, 2, 3, 2, 3) <class 'tuple'>
[2, 3, 4] <class 'list'>
frozenset({1, 2, 3}) <class 'frozenset'>
{2, 3, 4} <class 'set'>
```

```
{} <class 'dict'>
set() <class 'set'>
{'a': 1, 'b': 2, 'c': 3} <class 'dict'>

[2, 3, 4, 1, 2, 3, 2, 3] <class 'list'>
{1, 2, 3, 4} <class 'set'>
```

2.4 Osnovni tipovi podataka

Python	Komentar	C

`bool`
`>>> a = True`
`>>> print(a, type(a))`
`True <class 'bool'>`
	Logički tip podataka `True`, `False`. U Pythonu `None`, razne nule (`0`, `0.0`), prazne liste `[]`, stringovi `""`, itd. kad se koriste kao logički uvjet znače `False`.	U starijem standardu C-a ne postoji, od C99 nadalje postoji i naziva se `_Bool`. I u starim i u novim verzijama C-a tip `int` se može svugdje koristiti umjesto `_Bool` (na računalima u F26 `_Bool` zauzima 4 bytea što možemo vidjeti pomoću `sizeof(1==0)` koji vrati broj 4).
		Kad je potrebno držati u memoriji polje `boolean` vrijednosti, radi štednje memorije možemo koristiti npr. `char` umjesto `int` ili radi još veće uštede koristiti pojedinačne bitove unutar veće varijable kao ovdje ili ovdje.

`int`
`>>> a = -1000`
`>>> print(a, type(a))`
`-1000 <class 'int'>`
`>>> a = 12345678901234567890123456789O`
`>>> print(a, type(a))`
`12345678901234567890123456789O <class 'int'>`
	Pythonov `int` drži cijele brojeve proizvoljnog broja znamenaka.	C nema ugrađen takav tip. Slični tipovi su `int`, `long`, samo oni ne drže proizvoljan broj znamenki.

`float`
`>>> a = 1.0e-300`
`>>> print(a, type(a))`
`1e-300 <class 'float'>`
	Python-ov `float` je implementiran pomoću C-ovog `double`-a	`double`

`complex`
`>>> a = 1 + 1.0e-300j`
`>>> print(a, type(a))`
`(1+1e-300j) <class 'complex'>`
	Noviji C prevoditelji podržavaju kompleksne brojeve. Primjeri na: http://stackoverflow.com/questions/6418807/how-to-work-with-complex-numbers-in-c.	`double complex`

`NoneType`
`>>> a = None`
`>>> print(a, type(a))`
`None <class 'NoneType'>`
	Varijabla se postavi na ovaj tip kad je definirana ali nije inicijalizirana. Primjer je funkcija koja zaboravi vratiti vrijednost:
	`def fun():`
	`  print("funkcija samo ispisuje")`
	`a = fun()`
	`print(a, type(a))`
	`funkcija samo ispisuje`
	`None <class 'NoneType'>`
		Podsjeća na `void` u C-u, samo u C-u nisu postojale vrijednosti tipa `void`, postojao je samo tip `void`. U pythonu postoji samo jedna moguća vrijednost tipa `NoneType` a to je `None`.

Literatura

- https://docs.python.org/3/library/stdtypes.html#numeric-types-int-float-long-complex
- http://www.tutorialspoint.com/python/python_variable_types.htm
- http://www.python-course.eu/sequential_data_types.php

2.5 Grananje i petlje

Neke očite sličnosti i razlike C-a i python-a vidimo u sljedećoj tablici.

Python	C
`break`	`break;`
`continue`	`continue;`

`if uvjet:`
`  radnja`
	`if(uvjet){`
	`  radnja;`
	`}`

`if uvjet:`
`  radnja1`
`else:`
`  radnja2`
	`if(uvjet){`
	`  radnja1;`
	`}`
	`else{`
	`  radnja2;`
	`}`

`while uvjet:`
`  radnja1`
	`while(uvjet){`
	`  radnja1;`
	`}`

2.6 Osnovni elementi programa

Python	C
Indentacija	Vitičaste zagrade

Napomena: U pythonu nema vitičastih zagrada, a blok naredbi je definiran time što je uvučen u desno, **relativno na prethodni red**, pomoću razmaka ili tabova. Kako broj razmaka koje tab predstavlja nije fiksan (tj. može se namjestiti u editoru po želji) bitno je to indentiranje raditi uniformno. Preporuča se koristiti isključivo razmake za uvlačenje (npr. 2 ili 4). Da bi se olakšalo upisivanje obično je moguće namjestiti u editoru da pritisak na tab generira npr. 4 razmaka `' '` (a ne tab `'\t'`).

Napomena: U C-u se lako moglo definirati prazan blok pomoću `;` ili {}. Za tu svrhu u pythonu postoji naredba `pass` koja isto tako ne radi ništa.

Svaka naredba u svom redu ili `;`	`;`

`'xx'` ili `"xx"` ili `"""xx"""` (trostruki navodnici mogu uključivati više redaka teksta)
Kod unosa stringova u pythonu postoje još neke mogućnosti koje postignu dodavanjem slova (tzv. prefiksa) ispred prvih navodnika.
Prefiks `r` omogućuje upis znaka \ bez potrebe da se piše \\, tj. dovoljno je samo `r"\"`. Naravno u takvom zapisu ne možemo upisati znak za novi red `\n` jer je `r"\n"` isto što i `"\\n"`. Ovakav zapis je koristan kad je potrebno upisivati stringove koji sadrže veliki broj znakova \.
Prefiks `f` omogućuje lagano ubacivanje vrijednosti varijabli u string, te formatiranje. (Više na https://realpython.com/python-f-strings/)
`a = 1`
`print(f"{a}+{a}={a+a}")`
`print(f"{a:5}+{a:5}={a+a:5}")`
`1+1=2`
`1+ 1= 2`
Prefiks `u` se koristio u pythonu 2. U pythonu 3 se podrazumijeva da su stringovi nizovi unicode znakova.
	`"xx"`

Nema tip podataka znak (char). Umjesto toga koristi string duljine 1: `"x"` (ili niz byteova duljine 1: `b"x"`).	`'x'`
Python izvršava kôd naredbu po naredbu. Ne pregledava točnost unaprijed kao C nego tek u trenutku izvršavanja. To znači da ako se program u pythonu uspije pokrenuti, ne znači da u njemu nema tipfelera.	C sve provjerava unaprijed.
Case sensitive (tj. razlikuje velika i mala slova)	Također case sensitive

Heksadecimalni brojevi unose se koristeći prefiks `0x`.
Dodatno, radi preglednosti, grupe znamenki se mogu razdvajati znakom _. Npr.:
`a = 12_345.6`
`print(f"{a:,} + {a:,} = {a+a:,}")`
`print(f"{a:_} + {a:_} = {a+a:_}")`
`12,345.6 + 12,345.6 = 24,691.2`
`12_345.6 + 12_345.6 = 24_691.2`
	Također `0x`.

2.7 `for` petlja

Python

```
for i in range(pocetak,kraj,korak):
  radnja
```

`range()` je oznaka za interval sa zadanim granicama i iznosom koraka. U pythonu 2 rezultat poziva `range(...)` je lista brojeva, u pythonu 3 rezultat je objekt tipa `range`.

```
a = range(1,26,5)
print( a, type(a) )
```

```
range(1, 26, 5) <class 'range'>
```

Tip `range` predstavlja niz podataka (tj. *sequence type*) kao `tuple` i `list`. Može se lako pretvoriti u listu:

```
a = range(1,26,5)
print( a[0], a[1] )
print( list(a) )
```

```
1 6
[1, 6, 11, 16, 21]
```

Za potrebe for petlje možemo koristiti bilo koji tip koji je niz ili koji se može iterirati. Ne mora nužno biti `range`. O generatorima/iteratorima vidi napomenu u *Generatori/iteratori*.

```
for i in ['a', 1, 100]:
  print(i)
```

```
a
1
100
```

Pozor: Bitno je uočiti da su elementi liste uvijek manji od granice `kraj`. (npr. u gornjem primjeru broj 26 nije u listi)

C

```
for(i=pocetak; i<kraj; i+=korak){
  radnja;
}
```

2.8 Formatiranje (formatirani ispis)

Formatiranje stringova i brojeva.

Savjet: Korisna mogućnost pythona je operator `%` koji radi više manje isto što i `sprintf` iz C-a

Savjet: U pythonu nije moguće direktno zbrojiti string i broj nego broj treba pretvoriti u string: `"broj = " + str(6)`

```
a = 6
b = "broj = %d" % a
d = "x=%f,y=%f" % (1.0,2.2)
print(b)
print(d)
```

```
broj = 6
x=1.000000,y=2.200000
```

```
int a=6;
char b[32];
char d[32];
sprintf(b,"broj = %d",a);
sprintf(d,"x=%f,y=%f", 1.0, 2.2);
printf("%s\n",b);
printf("%s\n",d);
```

```
broj = 6
x=1.000000,y=2.200000
```

Python podržava i druge načine formatiranja. Pogledati primjere na:

- https://docs.python.org/3/tutorial/inputoutput.html
- https://cito.github.io/blog/f-strings/
- https://docs.python.org/3/library/string.html#format-examples
- https://docs.python.org/3/library/string.html#template-strings
- https://www.python.org/dev/peps/pep-0498/
- https://realpython.com/python-f-strings/

2.9 Stringovi i polja (liste)

Duljina stringa

```
len(a)
```

```
strlen(a)
```

Duljina polja

```
L=[1,2,3]
print(len(L))
```

```
3
```

```
int L[]={1,2,3};
//u nekim situacijama može se
//očitati duljina polja pomoću
printf("%lu\n",sizeof(L)/sizeof(L[0]));
```

```
3
```

Ispis elemenata polja

```
L=[1,2,30,40]
for i in range(len(L)):
  print(L[i])
```

```
1
2
30
40
```

```
int L[]={1,2,30,40};
int i;
for(i=0; i<4; i++)
  printf("%d\n",L[i]);
```

```
1
2
30
40
```

Savjet: Ispis elemenata polja na način tipičan za python

```
L=[1,2,30,40]
for a in L:
  print(a)
```

```
1
2
30
40
```

2.9.1 Zbrajanje i nadodavanje

Zbrajanje stringova = nadodavanje stringa na string

```
a="str"
b="ing"
c=a+b
print(c)
```

```
string
```

```
char a[32], b[32], c[32];
strcpy(a,"str");
strcpy(b,"ing");
strcpy(c,a);
strcat(c,b);
printf("%s\n",c);
```

```
string
```

Zbrajanje lista = nadodavanje liste na listu

```
a=[1,2,3]
b=[21,1]
c=a+b
print(c)
```

```
[1, 2, 3, 21, 1]
```

```
int a[16],b[16],c[16];
a[0]=1; a[1]=2; a[2]=3;
b[0]=21; b[1]=1;
memcpy(c,a,3*sizeof(int)); //kopirati a u c
memcpy(c+3,b,2*sizeof(int)); //kopirati b na c+3
int i;
for(i=0;i<5;i++)printf("%d ",c[i]);
printf("\n");
```

```
1 2 3 21 1
```

Nadodavanje elemenata na listu

```
a = [10,20]
a.append(30)
a.append(40)
print(a)
```

```
[10, 20, 30, 40]
```

```
int a[16] = {10,20};
int na = 2; // programer mora voditi evidenciju
            // o tome gdje je kraj liste
a[na++] = 30;
a[na++] = 40;

int i;
for(i=0;i<na;i++)printf("%d ",a[i]);
printf("\n");
```

```
10 20 30 40
```

Zbrajanje lista moguće i za liste s elementima raznih tipova

```
a = [1,2,"a"]
b = ["w",[4,1]]
c = a+b
print(c)
```

```
[1, 2, 'a', 'w', [4, 1]]
```

U C-u je ovo komplicirano. Trebalo bi raditi s listom struktura koji sadrže: informacije o tipu podataka + pointer na same podatke i takve strukture nadodavati s jedne liste na drugu.

Napomena: Python omogućuje i razne druge operacije nad listama - pogledati https://docs.python.org/3/tutorial/datastructures.html

2.10 Rad sa stringovima

mijenjanje stringova

u pythonu nije dopušteno mijenjati stringove, iako to ne dolazi do izražaja jer je dopušteno:

```
a=a+"3"
```

gornja naredba kreira **novi** string a zaboravi stari. nakon izvršavanja gornje naredbe `a` pokazuje na novi string. memoriju zauzetu starim stringom oslobodi u nekom trenutku python-ov *garbage collector*, ako više nije potrebna.
u žargonu se kaže da su u pythonu stringovi *immutable*. (a liste su *mutable*)

Napomena: pogledati *Pridruživanje lista*

u C-u je promjena dopuštena.

```
strcat(a,"3")
```

gornja naredba je promijenila sadržaj na koji `a` pokazuje, ali sam pointer `a` je ostao isti.
za razliku od pythona programer mora paziti da se ne prekorači rezervirana duljina od `a`.

mala u velika slova

```
a="string"
print(a.upper())
```

```
STRING
```

treba primijetiti da je sam `a` ostao isti tj. `"string"`

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

void upper( char* s ){
  while( (*s++ = toupper(*s)) );
}

int main(){
  char a[32];
  strcpy(a, "string");
  upper(a);
  printf("%s\n",a);
}
```

```
STRING
```

u C-u je lakše napraviti tako da se sadržaj od `a` promijeni (u žargonu: *in-place* operacija)

traženje podstringa u stringu

```
a="string"

pos = a.find("tri")
if pos >= 0:
  print(pos)
else:
  print("nije pronađen")

pos = a.find("dva")
if pos >= 0:
  print(pos)
else:
  print("nije pronađen")
```

```
1
nije pronađen
```

```
char a[32];
char* pos;
strcpy(a, "string");

pos = strstr(a,"tri");
if(pos)
  printf("%ld\n",pos-a);
else
  printf("nije pronađen");

pos = strstr(a,"dva");
if(pos)
  printf("%ld\n",pos-a);
else
  printf("nije pronađen");
```

```
1
nije pronađen
```

pristup pojedinim znakovima u stringu

```
a[1]
```

```
a[1]
```

pristup podstringovima

```
a="string"
b=a[1:4]
print(b)
```

```
tri
```

```
char a[]="string";
char b[32];
strncpy(b,a+1,3);
b[3]='\0';
printf("%s\n",b);
```

```
tri
```

Napomena: Popis funkcija (preciznije, metoda) za rad sa stringovima na https://docs.python.org/3/library/stdtypes.html#string-methods

2.11 Dijelovi stringa/liste

radnja	kôd

sve osim prvog	`>>> a="string"`
	`>>> a[1:]`
	`tring`

prva 3 znaka/elementa	`>>> a = "string"`
	`>>> a[:3]`
	`str`

zadnja 3 znaka/elementa	`>>> a = "string"`
	`>>> a[-3:]`
	`ing`

odbaciti prvi i zadnji	`>>> a = ["string",2,3,5j,"4"]`
	`>>> a[1:-1]`
	`[2, 3, 5j]`

Pridruživanje lista

I u C-u i u pythonu postoje sličnosti pri radu sa stringovima i sa listama (poljima). Najveća razlika između stringova i lista u pythonu dolazi od toga da su stringovi u pythonu nepromjenjivi (*immutable*):

```
a = "100"
#a[2] = "1" ovo bi izbacilo grešku
```

Napomena: U pythonu postoji i *immutable* verzija liste i taj tip podataka se naziva *tuple*. Za razliku od liste koja se zadaje uglatim, tuple se zadaje okruglim zagradama npr. `a = (1, 2, 3)`. O razlici lista i tupleova na: http://stackoverflow.com/a/1708538

3.1 Pridruživanje vs. shallow copy

Pridruživanjem lista pomoću operatora pridruživanja (=) ne kopira se sadržaj liste. Samo se varijabli slijeva pridruži postojeća lista navedena desno od operatora pridruživanja. U sljedećem primjeru vidimo da nakon pridruživanja `a` i `b` pokazuju na isti objekt što znači da ako se naknadno promijeni `a[0]`, promijeni se i `b[0]`.

```
>>> a=[1,2,3]
>>> b=a
>>> a[0]=11
>>> a
[11, 2, 3]
>>> b
[11, 2, 3]
```

U sljedećem primjeru pomoću `a[:]` cijeli sadržaj liste je kopiran i ta nova kopija je pridružena varijabli `b`. Sad možemo mijenjati `a` neovisno o `b`.

```
>>> a=[1,2,3]
>>> b=a[:]
>>> a[0]=11
>>> a
[11, 2, 3]
>>> b
[1, 2, 3]
```

3.2 Shallow copy vs. deep copy

U sljedećem primjeru kopiramo listu koja u sebi sadrži referencu na drugu listu. Za kopiranje koristimo *shallow-copy* način iz prethodnog primjera. U prethodnom primjeru, *shallow-copy* je omogućio da liste `a` i `b` budu neovisne. Ovdje vidimo da to još uvijek vrijedi za prvi nivo liste. Sad je jasno zašto se ovo zove plitko kopiranje (shallow copy): kopiran je samo prvi nivo liste, drugi nivo je kopiran kao pokazivač, pa u listi `b` još uvijek "živi" isti objekt `c` koji živi u `a`.

```
>>> c=[4,5,6]
>>> a=[1,2,c]
>>> b=a[:]
>>> a[0]=11
>>> c[0]=44
>>> a
[11, 2, [44, 5, 6]]
>>> b
[1, 2, [44, 5, 6]]
```

Funkcija `copy.deepcopy()` služi da se kopiraju svi nivoi liste, tako da su u ovom primjeru `a` i `b` potpuno neovisni.

```
>>> import copy
>>> c=[4,5,6]
>>> a=[1,2,c]
>>> b=copy.deepcopy(a)
>>> a[0]=11
>>> c[0]=44
>>> a
[11, 2, [44, 5, 6]]
>>> b
[1, 2, [4, 5, 6]]
```

3.3 Razne operacije koje sve funkcioniraju kao shallow copy

```
>>> import copy
>>> c=[4,5,6]
>>> a=[1,2,c]
>>> b=copy.copy(a)
>>> a[0]=11
>>> c[0]=44
>>> a
[11, 2, [44, 5, 6]]
>>> b
[1, 2, [44, 5, 6]]
```

```
>>> c=[4,5,6]
>>> a=[1,2,c]
>>> b=list(a)
>>> a[0]=11
>>> c[0]=44
>>> a
[11, 2, [44, 5, 6]]
>>> b
[1, 2, [44, 5, 6]]
```

```
>>> c=[4,5,6]
>>> a=[1,2,c]
>>> b=a[:]
>>> a[0]=11
>>> c[0]=44
>>> a
[11, 2, [44, 5, 6]]
>>> b
[1, 2, [44, 5, 6]]
```

```
>>> c=[4,5,6]
>>> a=[1,2,c]
>>> b=a+[]
>>> a[0]=11
>>> c[0]=44
>>> a
[11, 2, [44, 5, 6]]
>>> b
[1, 2, [44, 5, 6]]
```

3.4 Usporedba immutable i mutable veličina

Sve varijable u pythonu su interno pointeri. Python u pravilu ne omogućuje očitavanje adresa iako je u nekim implementacijama pythona (npr. u uobičajenoj implementaciji CPython) adresu moguće očitati pomoću `id(a)`.

Immutable

```
a = "123" # a pokazuje na "123"
b = a     # b pokazuje na isti taj "123"
a = a + "4" # a pokazuje na novi string "1234"
   # b pokazuje još uvijek na onaj stari "123"
print(a)
print(b)
```

```
1234
123
```

Treba primijetiti da se `b` nije promijenio jer se stringovi ne mogu mijenjati, samo može nastati novi string.

Mutable

```
a = [1,2,3] # a pokazuje na objekt [1,2,3]
b = a       # b pokazuje na isti taj objekt [1,2,3]
a.append(4) # objekt se promijenio na [1,2,3,4]
print(a)    # svi koji pokazuju na taj
print(b)    #   objekt vide promjenu
```

```
[1, 2, 3, 4]
[1, 2, 3, 4]
```

Treba primijetiti da se i `b` promijenio zato što je pokazivao na listu koja se u međuvremenu promijenila.

Mutable (*inplace* operacija)	Mutable (operacija koja kopira)

`a = [1,2,3]`
`b = a`
`a += [4]`
`print(a)`
`print(b)`
	`a = [1,2,3]`
	`b = a`
	`a = a + [4]`
	`print(a)`
	`print(b)`

`[1, 2, 3, 4]` `[1, 2, 3, 4]`	`[1, 2, 3, 4]` `[1, 2, 3]`
Isto kao primjer s `list.append()` (ostale moguće operacije u tablici o operacijama nad *mutable* tipovima *Mutable Sequence Types*).	Iako je tip mutable, kad se umjesto `+=` koristi `+`, rezultat je stvaranje novog neovisnog objekta, a ne proširivanje postojećeg.

Literatura

- https://www.python-course.eu/python3_deep_copy.php
- http://en.wikipedia.org/wiki/Object_copy
- http://stackoverflow.com/questions/184710/what-is-the-difference-between-a-deep-copy-and-a-shallow-copy
- https://docs.python.org/3/library/copy.html
- http://stackoverflow.com/questions/17246693/what-exactly-is-the-difference-between-shallow-copy-deepcopy-and-normal-assignm
- na google-u tražiti: *shallow deep copy python*

Rječnik (Dictionary)

Dictionary (rječnik) je tip podataka sličan polju, samo što index nije isključivo cjeli broj nego bilo koji *immutable* tip podataka kao što su *string*ovi ili brojevi. To znači da su dopušteni izrazi `a['rezultat'] = a['prvi pribrojnik'] + a['drugi pribrojnik']` ili `a['rezultat'] = a[('pribrojnik',1)] + a[('pribrojnik',2)]`, za razliku od polja gdje je elementima moguće pristupiti samo pomoću brojčanog indeksa: `a[2] = a[0] + a[1]`.

4.1 Kreiranje

```
dana = dict()    #ili dana = {}
dana['siječanj'] = 31
dana[2] = 28
print(dana)
print(dana[2])
print(dana['siječanj'])
```

```
{'siječanj': 31, 2: 28}
28
31
```

```
dana = dict(sijecanj=31, mj2=28)
print(dana)
print(dana['mj2'])
print(dana['sijecanj'])
```

```
{'sijecanj': 31, 'mj2': 28}
28
31
```

```
dana = {1:"trideset jedan", "2":28}
print(dana)
print(dana["2"])
print(dana[1])
```

```
{1: 'trideset jedan', '2': 28}
28
trideset jedan
```

4.2 Očitavanje i mijenjanje

```
>>> dana = dict(sijecanj=31, mj2=28)
>>> print(dana)
{'sijecanj': 31, 'mj2': 28}
>>> print(dana['mj2'])
28
>>> print(dana['sijecanj'])
31
>>> print(list(dana.keys()))
['sijecanj', 'mj2']
>>> print(list(dana.values()))
[31, 28]
>>> print(list(dana.items()))
[('sijecanj', 31), ('mj2', 28)]
```

```
dana = dict(sijecanj=31, mj2=28)
dana.update({
  'veljača':28,
  'travanj':30})
dana['ožujak'] = 31
for k in list(dana.keys()):
  print(k, dana[k])
```

```
sijecanj 31
mj2 28
veljača 28
travanj 30
ožujak 31
```

```
dana = {'siječanj':"trideset jedan", 2:28}
print('siječanj' in dana)
print('svibanj' in dana)
```

```
True
False
```

4.3 Primjer: brojanje ponavljanja riječi u listi

```
L = ["kruška","jabuka","jabuka","limun","kruška","kruška"]
brojac = {}
for i in L:
  if i in brojac:
    brojac[i] += 1
  else:
    brojac[i] = 1

for k in brojac:
  print(k, brojac[k])
```

```
kruška 3
jabuka 2
limun 1
```

U gornjem primjeru smo pazili da ne pokušamo očitati vrijednost `brojac[i]` ako ključ `i` već nije u rječniku `brojac`. Da bi se ta provjera izbjegla, moguće je zadati default vrijednosti nekom rječniku koristeći funkciju `dict.setdefault()` ili klasu `collections.defaultdict`:

```
from collections import defaultdict

dana = defaultdict( lambda: 31 )
dana["veljača"]=28
dana["travanj"]=30
L = ["siječanj", "veljača", "ožujak", "travanj", "bilo što"]
for i in L:
  print(i, dana[i])
```

```
siječanj 31
veljača 28
ožujak 31
travanj 30
bilo što 31
```

Brojanje riječi se, prema tome, može riješiti i ovako:

```
from collections import defaultdict

L = ["kruška","jabuka","jabuka","limun","kruška","kruška"]
brojac = defaultdict( lambda: 0 )
for i in L:
  brojac[i] += 1

for k in brojac:
  print(k, brojac[k])
```

```
kruška 3
jabuka 2
limun 1
```

Dodatni primjeri na https://docs.python.org/3/library/collections.html#defaultdict-examples.

Datoteke

5.1 Tekstualne datoteke

Primjer koji upisuje neka slova i brojeve u datoteku.

```
with open("txt1.txt","w") as f:
  print("1.red: 123", 456, file=f)
  print("2.red 1230", 4560.123, file=f)
```

Rezultat je datoteka:

```
1.red: 123 456
2.red 1230 4560.123
```

U C-u je običaj štedljivo učitavati ono što je potrebno broj po broj slovo po slovo. U pythonu je običaj učitati odjednom sve a nakon toga analizirati.

```
with open("txt1.txt","r") as f:
  s = f.read()
print(s)
```

```
1.red: 123 456
2.red 1230 4560.123
```

Sadržaj učitan u jedan veliki string.

```
with open("txt1.txt","r") as f:
  L = f.readlines()
print(L)
```

```
['1.red: 123 456\n', '2.red 1230 4560.123\n']
```

Sadržaj učitan kao lista stringova: jedan red = jedan string.

```
with open("txt1.txt","r") as f:
  L = f.readlines()
for red in L:
  L1 = red.split()
  print(L1[:-1], float(L1[-1]))
```

```
['1.red:', '123'] 456.0
['2.red', '1230'] 4560.123
```

Sadržaj učitan kao lista stringova. Zatim svaki red rastavljen u listu stringova. Zadnji element liste pretvaramo u `float` i ispisujemo. Ostatak liste ispisujemo kako jest.

Napomena: Datoteke se zatvaraju u trenutku napuštanja `with` bloka, tj. nakon izvršenja zadnje naredbe u `with` bloku.

5.1.1 Tekstualni format JSON

JSON omogućuje lagano prebacivanje iz kombinacije dictionary+list+osnovni tipovi u string **i obratno**. Time se lako mogu iz datoteke učitati npr. ulazni podaci za program.

```
>>> import json
>>> čestice = [{'rb':1, 'm':10, 'x':0.0, 'y':0.0, 'fiksna':True}, {'rb':2, 'm':20, 'x':1.0, 'y':1.1, 'fiksna':False}]
>>> print(čestice[0]['rb'])
1
>>> print(čestice[0]['x'])
0.0
>>> print(čestice[0]['y'])
0.0
>>> print(čestice[1]['rb'])
2
>>> print(čestice[1]['x'])
1.0
>>> print(čestice[1]['y'])
1.1
>>> print(repr(json.dumps(čestice)))
'[{"rb": 1, "m": 10, "x": 0.0, "y": 0.0, "fiksna": true}, {"rb": 2, "m": 20, "x": 1.0, "y": 1.1, "fiksna": false}]'
>>> print(json.dumps(čestice,indent=4))
[
    {
        "rb": 1,
```

```
        "m": 10,
        "x": 0.0,
        "y": 0.0,
        "fiksna": true
    },
    {
        "rb": 2,
        "m": 20,
        "x": 1.0,
        "y": 1.1,
        "fiksna": false
    }
]
```

U gornjem primjeru smo pomoću `json.dumps()` prebacili sadržaj varijable u string. Pomoću `json.dump()` sadržaj varijable upisujemo u datoteku.

```
import json
data = {'datum':'2018-09-01','popis':[(1,2),5,10,15]}
with open("json1.txt","w") as f:
  json.dump(data,f)
```

U datoteci je upisano:

```
{"datum": "2018-09-01", "popis": [[1, 2], 5, 10, 15]}
```

Učitavanje iz stringa radimo pomoću `json.loads()` a iz datoteke pomoću `json.load()`.

```
import json
with open("json1.txt","r") as f:
  data = json.load(f)
print(data)
print(data['popis'])
```

```
{'datum': '2018-09-01', 'popis': [[1, 2], 5, 10, 15]}
[[1, 2], 5, 10, 15]
```

5.2 Binarne datoteke

5.2.1 Strukture

C interno u memoriji drži samo gole podatke koje je stoga lako direktno prepisati iz memorije u datoteku. Python interno u memoriji drži puno više dodatnih informacija pa je potrebno prije upisa u datoteku izvući same podatke u obliku **liste byteova**. To se radi pomoću funkcije `struct.pack()`.

```
>>> import struct
>>> print(struct.pack('10s',b"tekst"))
b'tekst\x00\x00\x00\x00\x00'
>>> print(struct.pack('20s',b"tekst"))
b'tekst\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
>>> print(struct.pack('i',10))
b'\n\x00\x00\x00'
>>> print(struct.pack('d',0.1))
b'\x9a\x99\x99\x99\x99\xb9?'
```

Slijedi primjeri koji upisuju i čitaju riječ, `int` i `double` u datoteku.

```python
import struct
L=[b"tekst",10,0.1]
byteovi = struct.pack('10sid',*L)
print(len(byteovi))
with open("bin1.bin","wb") as f:
  f.write(byteovi)
```

```
24
```

Oznaka `10sid` sastoji se od `10s`, `i`, `d`. To znači da će se sastaviti struktura u kojoj će string zauzimati 10 byteova nakon kojih slijede int i double. Pri tome se ubacuje isti *alignment* kao u C-u. Više o oznakama za kodiranje strukture na https://docs.python.org/3/library/struct.html#byte-order-size-and-alignment i na https://docs.python.org/3/library/struct.html#format-characters.

Pomoću programa `hexdump` možemo vidjeti sadržaj binarnih datoteka. U terminal upišemo `hexdump bin1.bin`

```
0000000 74 65 6b 73 74 00 00 00 00 00 00 00 0a 00 00 00
0000010 9a 99 99 99 99 99 b9 3f
0000018
```

Vidimo da je sadržaj datoteke `bin1.bin` napravljene iz pythona isti kao datoteke `bin2.bin` napravljene iz C-a.

```c
#include <stdio.h>
typedef struct {
  char s[10];
  int i;
  double d;
} zapis;

int main()
{
  zapis z = {"tekst", 10, 0.1};
  printf("%lu\n",sizeof(z));
  FILE* f = fopen("bin2.bin","wb");
  if(!f)return 1;
  if( fwrite(&z, sizeof(z), 1, f) != 1 )return 1;
  if( fclose(f) != 0 )return 1;
  return 0;
}
```

```
24
```

U terminal upišemo `hexdump bin2.bin`.

```
0000000 74 65 6b 73 74 00 00 00 00 00 00 00 0a 00 00 00
0000010 9a 99 99 99 99 99 b9 3f
0000018
```

Učitavanje iz datoteke:

```python
import struct
with open("bin1.bin","rb") as f:
  byteovi = f.read()
print(struct.unpack('10sid',byteovi))
```

```
(b'tekst\x00\x00\x00\x00\x00', 10, 0.1)
```

```c
#include <stdio.h>
typedef struct {
  char s[10];
  int i;
  double d;
} zapis;

int main()
{
  zapis z;
  FILE* f = fopen("bin2.bin","rb");
  if(!f)return 1;
  if( fread(&z, sizeof(z), 1, f) != 1 )return 1;
  if( fclose(f) != 0 )return 1;
  printf("%s; %d; %f\n",z.s, z.i, z.d);
  return 0;
}
```

```
tekst; 10; 0.100000
```

5.2.2 Homogena polja

Python omogućuje i homogena polja tj. ona koja sadrže elemente istog tipa. Prilikom inicijalizacije potrebno je navesti tip podataka za elemente.

```
>>> from array import array
>>> z = array( 'l', [1, 2, 3, 4, 5] )
>>> print(z)
array('l', [1, 2, 3, 4, 5])
>>> print(z.itemsize)
8
>>> print(len(z))
5
>>> print(len(z.tobytes()))
40
```

Byteove koje definiraju polje možemo dobiti pomoću `array.tobytes()` i nakon toga ih možemo upisati u datoteku. Također je moguće direktno upisivanje iz polja u datoteku pomoću `array.tofile()`.

```
from array import array
z = array( 'l', [1, 2, 3, 4, 5] )
with open("bin_array1.bin","wb") as f:
  z.tofile(f)
  #ili f.write( z.tobytes() )
print(z.itemsize, len(z))
```

```
8 5
```

```
#include <stdio.h>

int main()
{
  long z[] = {1, 2, 3, 4, 5};
  size_t l = sizeof(z[0]);
  size_t N = sizeof(z)/l;

  FILE* f = fopen("bin_array2.bin","wb");
  if(!f)return 1;
  if( fwrite(&z, l, N, f) != N )return 1;
  if( fclose(f) != 0 )return 1;

  printf( "%lu %lu\n", l, N );
  return 0;
}
```

```
8 5
```

```
0000000 01 00 00 00 00 00 00 00 02 00 00 00 00 00 00 00
0000010 03 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00
0000020 05 00 00 00 00 00 00 00
0000028
```

```
0000000 01 00 00 00 00 00 00 00 02 00 00 00 00 00 00 00
0000010 03 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00
0000020 05 00 00 00 00 00 00 00
0000028
```

Vidimo da su rezultati prethodnih programa isti.

Byteove koji definiraju elemente polja možemo interpretirati kao elemente i dodati ih u postojeće polje pomoću `array.frombytes()`. Također je moguće byteove elemenata zapisanih u datoteku pročiti i dodati u postojeće polje pomoću `array.fromfile()`.

```
from array import array
z = array( 'l' )
with open("bin_array1.bin","rb") as f:
  z.fromfile( f, 5 )
  #ili z.frombytes( f.read() )
print(z)
```

```
array('l', [1, 2, 3, 4, 5])
```

```
#include <stdio.h>

int main()
{
  long z[16];
  size_t l = sizeof(z[0]);
  size_t N = 5;

  FILE* f = fopen("bin_array2.bin","rb");
  if(!f)return 1;
  if( fread(&z, l, N, f) != N )return 1;
  if( fclose(f) != 0 )return 1;

  for(int i=0; i<N; i++ )
    printf("%ld ", z[i]);
  printf("\n");

  return 0;
}
```

```
1 2 3 4 5
```

Sortiranje

Primjer: sortiranje polja stringova tako da se prvi znak u stringu ignorira.

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

char L[32][32];

int usporedba( char* a, char* b )
{
  return strcmp( a+1, b+1 ); //sortiranje ignorirajući
}                            //prvi znak

main()
{
    strcpy( L[0], ".kruska" );
    strcpy( L[1], "-jabuka" );
    strcpy( L[2], "*limun" );

    qsort( L, 3, 32,
      (int(*)(const void*,const void*))usporedba );

    int i;
    for( i=0; i<3; i++ )
        printf( "%d %s\n", i, L[i] );
}
```

```
0 -jabuka
1 .kruska
2 *limun
```

```python
from functools import cmp_to_key

def cmp( a, b ):
  return (a>b) - (a<b)

def usporedba( a, b ):
  return cmp(a[1:],b[1:])

L = [".kruska", "-jabuka", "*limun" ]
L.sort( key=cmp_to_key(usporedba) )

for i,s in enumerate(L):
  print(f"{i} {s}")
```

```
0 -jabuka
1 .kruska
2 *limun
```

Pythonskiji način bi bio koristiti funkciju koja generira ključ po kojem će se sortirati:

```python
def kljuc( a ):
  return a[1:]

L = [".kruska", "-jabuka", "*limun" ]
L.sort( key=kljuc )

for i,s in enumerate(L):
  print(f"{i} {s}")
```

```
0 -jabuka
1 .kruska
2 *limun
```

`sorted()` vs. `list.sort()`

```python
def kljuc( a ):
  return a[1:]

L = [".kruska", "-jabuka", "*limun" ]

print(sorted(L))
print(sorted(L, key=kljuc))
print(sorted(L, reverse=True))
print(sorted(L, key=kljuc, reverse=True))
print(L)
```

```
['*limun', '-jabuka', '.kruska']
['-jabuka', '.kruska', '*limun']
['.kruska', '-jabuka', '*limun']
['*limun', '.kruska', '-jabuka']
['.kruska', '-jabuka', '*limun']
```

Vidimo da za razliku od metode `list.sort()` (vidi tablicu *Mutable Sequence Types*) iz prvog primjera, funkcija `sorted()` nije promijenila početnu listu `L`. (`list.sort()` dakle radi *in-place* sort, a `sorted()` stvara još jednu listu.) Također vidimo kako opcija `reverse` može poslužiti za promjenu redoslijeda.

Literatura

- https://wiki.python.org/moin/HowTo/Sorting
- http://www.pythoncentral.io/how-to-sort-a-list-tuple-or-object-with-sorted-in-python/

- https://docs.python.org/3/howto/sorting.html#sortinghowto
- https://docs.python.org/3/library/functions.html#sorted

Detalji

Ovo poglavlje je tu radi potpunosti. Izlazi van okvira kolegija, te nije ga potrebno detaljno proučavati.

7.1 Uobičajeno zaglavlje

Za kodiranje slova s kvačicama preporuča se koristiti UTF-8 kodiranje. U pythonu 2, potrebno je dodati zaglavlje `# -*- coding: utf-8 -*-`. U pythonu 3, UTF-8 je default. (Više na https://stackoverflow.com/questions/14083111/should-i-use-encoding-declaration-in-python-3)

Da bi se program u pythonu mogao pokretati kao samostalan program (tj. direktno, kao što se pokreće `./a.out`) osim postavljanja prava izvršavanja (`chmod u+x ...`) potrebno je **na samom početku** dodati još i zaglavlje koje označava da je python interpreter program pomoću kojeg se naš program treba pokrenuti. To se može pomoću zaglavlja `#!/usr/bin/env python`.

```
#!/usr/bin/env python

print("samostalni program")
```

```
$ chmod u+x zag.py
$ ./zag.py
samostalni program
```

Zaglavlje `#!/usr/bin/env python` nalaže da se koristi naredba python iz nekog od direktorija upisanih u varijablu okruženja `PATH`. Umjesto toga, može se u zaglavlje staviti putanja na konkretni python interpreter, npr: `#!/usr/bin/python2.7`. Ako se radi na ovaj način moguće je poslati argumente pythonu: `#!/usr/bin/python2.7 -tt`. Opcija `-tt` uzrokuje prekid izvršavanja ako interpreter primijeti u programu miješanje razmaka i tabova.

U terminalu vrijednost varijable `PATH` doznamo pomoću:

```
$ echo $PATH
/usr/bin:/usr/local/bin
```

Može se dogoditi da je instalirano više programa istog naziva (npr. više pythona). Zbog toga redoslijed direktorija u varijabli `PATH` može biti bitan: kad u terminalu upišemo npr. `python` sustav traži postoji li program tog naziva **redom** u direktorijima u varijabli `PATH`. S tim u vezi korisna je naredba `which` pomoću koje možemo doznati iz kojeg direktorija iz `$PATH` potječe program koji pokrećemo.

```
$ which python
/usr/bin/python
```

Više o načinima kako promijeniti vrijednost varijable `PATH` na https://www.java.com/en/download/help/path.xml

Literatura

- http://stackoverflow.com/questions/2429511/why-do-people-write-usr-bin-env-python-on-the-first-line-of-a-python-script
- http://en.wikipedia.org/wiki/Shebang_(Unix)
- http://stackoverflow.com/questions/1352922/why-is-usr-bin-env-python-supposedly-more-correct-than-just-usr-bin-pyt

7.2 Argumenti (parametri) programa

Slično kao u C-u, i u pythonu za vrijeme izvršavanja programa postoji polje u kojem se nalazi naziv programa kao i argumenti s kojima je pokrenut program.

```c
#include <stdio.h>

int main( int argc, char** argv )
{
    int i;
    for( i=0; i<argc; i++ )
        printf( "%d %s\n", i, argv[i] );
    return 0;
}
```

```
$ gcc op1_arg1.c && ./a.out 1 2 3
0 ./a.out
1 1
2 2
3 3
```

```python
import sys

for i in range(len(sys.argv)):
  print("%d %s" % (i, sys.argv[i]))
```

```
$ python op1_arg1.py 1 2 3
0 op1_arg1.py
1 1
2 2
3 3
```

```c
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char** argv )
{
    int i;
    double suma = 0;
    for( i=1; i<argc; i++ )
        suma += atof(argv[i]);
    printf("%f\n",suma);
    return 0;
}
```

```
$ gcc op1_arg.c && ./a.out 1 2 3
6.000000
```

```python
import sys

suma = 0
for i in sys.argv[1:]:
    suma += float(i)
print(suma)
```

```
$ python op1_arg.py 1 2 3
6.0
```

```python
import sys

print( sum( map( float, sys.argv[1:] ) ) )
```

```
$ python op1_arg2.py 1 2 3
6.0
```

Python sadrži i standardni paket za obradu argumenata programa koji olakšava rukovanje argumentima (olakšava analizu unesenih argumenata, obradu grešaka, ispis uputstava za argumente):

- https://docs.python.org/3/howto/argparse.html
- https://docs.python.org/3/library/argparse.html#module-argparse
- https://docs.python.org/3/faq/programming.html#faq-argument-vs-parameter

7.3 Automatsko oslobađanje memorije

Python "shvati" kad se objekt više ne koristi i nakon toga ga može "uništiti" (tj. oslobodi memoriju koju taj objekt zauzima). Kaže se da python sadrži *garbage collector*. Primjer:

```python
a=[1,2,3]
#a pokazuje na objekt [1,2,3]
b=a
#a i b pokazuju na objekt [1,2,3]
a[1] = 2
#a i b pokazuju na objekt koji se u međuvremenu promijenio i sad glasi [2,2,3]
a = [4,5,6]
#sad samo b pokazuje na [2,2,3]
b="a"
#sad više nitko ne pokazuje na [2,2,3] i garbage collector će ga u nekom trenutku izbrisati iz memorije
```

7.4 Unicode

Slova/znakovi su predstavljeni u memoriji pomoću brojeva. To pridruživanje se naziva kodiranje karaktera ili kodna stranica. Primjer je ASCII kôd koji definira kodove od 0-127 (npr. slovo 'A' ima kôd 65, slovo 'B' 66, itd.)

Slova s kvačicama nisu na tom popisu. Kroz povijest su se slovima s kvačicama dodjeljivali kodovi na više načina:

1. umjesto nekih znakova ASCII kôda (0-127), npr. `ČĆĐŠŽ` umjesto `^]\[@`, a `čćđšž` umjesto `~}|{\`. (postoje i druge nacionalne varijante http://en.wikipedia.org/wiki/ISO/IEC_646)

2. umjesto nekih znakova proširenog ASCII kôda (128-255). (vidi http://en.wikipedia.org/wiki/ISO/IEC_8859)
3. unicode: kôd duljine 4 *byte*a pridružen svakom znaku/slovu iz bilo kojeg pisma (http://en.wikipedia.org/wiki/ISO_10646)

Mana ovog prvog pristupa je da se nije moglo u istom dokumentu imati istovremeno znakove kao \, [, { i slova s kvačicama.

Situaciju donekle ispravlja drugi pristup u kojem također programi moraju znati koji skup kodova (kodna stranica) se koristi.

Treći način (unicode) omogućuje da se za sva pisma koristi ista kodna stranica. Međutim i tu postoji više varijanti zapisa. Razlikuju se po tome koliko se byteova koristi za zapis, kojim redoslijedom, itd., pa postoje tzv.: UCS-2, UCS-4, UTF-16, UTF-32, UTF-8, ...

Trenutno je najpopularniji način kodiranja UTF-8, koji koristi 1-6 byteova za kodiranje jednog znaka (broj byteova varira od znaka do znaka), a unazad je kompatibilan s ASCII standardom (to znači da je svaka ASCII tekst datoteka ujedno i UTF-8 datoteka). Također nikad ne koristi byte nula, tako da se neće nikad umjetno pojaviti null-terminator, pa će stoga funkcije iz C-a kao `strcpy()` raditi ispravno. Također, vrijednosti kodova 0-127 se ne koriste za znakove koji zahtijevaju višebyteni zapis što znači da se npr. znak za novi red ne može slučajno pojaviti a to znači da editor koji ne zna da se radi o UTF-8 može točno odrediti koliko ima redaka teksta.

To su sve dobra svojstva u odnosu na ostale načine kodiranja, iako treba biti svjestan da prikaz znaka u UTF-8 (i u ostalim unicode kodiranjima) nije jednoznačan. Npr. slovo č posjeduje vlastiti kôd (NFC normalizacija), ali se č može prikazati kao kôd za c + kôd za kvačicu (NFD normalizacija). Više o normalizacijama na http://stackoverflow.com/a/7934397, http://www.unicode.org/reports/tr15. Dodatna komplikacija je da postoje i nizovi znakova koji *kao niz* imaju svoj kôd (npr. tri točke), pa neka pretvaranja iz normalizacije u normalizaciju mogu biti ireverzibilna.

Više na:

- http://www.dabeaz.com/python3io/MasteringIO.pdf
- http://en.wikipedia.org/wiki/UTF-8
- http://www.joelonsoftware.com/articles/Unicode.html
- https://docs.python.org/3/howto/unicode.html
- https://www.pythonsheets.com/notes/python-unicode.html

Da bi se omogućio rad s UTF-8 u pythonu potrebno je:

1. u editoru to odabrati kao opciju.
2. u samim python 2 programima dodati liniju koja označava da je kodiranje koje se koristi UTF-8. u pythonu 3 UTF-8 je default pa to nije potrebno.

U pythonu 3 nazivi varijabli mogu biti unicode znakovi, u pythonu 2 ne mogu. (Više na https://stackoverflow.com/a/46001544/2866169)

Sadržaj varijabli može biti niz unicode znakova i u pythonu 2 i u pythonu 3, ali postoje razlike:

Sadržaj niza	Python 2	Python 3	Ispis
Byteovi	Tip `str`; unosi se pomoću "..."; dopušteno "č"	Tip `bytes`; unosi se pomoću b"..." (potreban prefiks b); zabranjeno b"č"	Prilikom ispisa python 2 pretvara niz byteova u niz unicode znakova. Python 3 ne pretvara niz byteova u unicode znakove nego samo ispisuje byteove.
Unicode znakovi	Tip `unicode`; unosi se pomoću u"..." (potreban prefiks u); dopušteno u"č"	Tip `str`; unosi se pomoću "..."; dopušteno "č"	U obje verzije pythona ispisuju se kao unicode znakovi

U verziji 3 razlikuju se nizovi byteova od nizova unicode znakova. U pythonu 2 ta distinkcija nije bila toliko izražena jer se (kao što vidimo u tablici) prilikom ispisa byteovi pokušaju ispisati kao unicode znakovi. U pythonu 2 zbunjuje što je i u unicode stringove i u nizove byteova dopušteno upisati unicode znakove. **U pythonu 3 je situacija pojednostavljena: nizovi byteova su uvijek byteovi, stringovi su uvijek nizovi unicode znakova.** Da bi se izbjegle moguće zabune u pythonu 3 je zabranjeno u nizove byteova direktno upisivati unicode znakove ako ih se nije prethodno kodiralo pomoću `str.encode()`.

U pythonu 3 stringovi su nizovi unicode kodova. Za određeni string i kodnu stranicu, pomoću `str.encode()` možemo dobiti niz byteova koji taj string predstavlja. Za određeni niz byteova i kodnu stranicu, pomoću `bytes.decode()` možemo dobiti niz unicode znakova koji je predstavljen tim byteovima.

```
a = b'\xc4\x8d'
b = "č"
c = a.decode('utf8')
d = b.encode('utf8')
print(a, b, c, d, sep="  |  ")
```

```
b'\xc4\x8d'  |  č  |  č  |  b'\xc4\x8d'
```

Razna kodiranja istog unicode znaka daju različite nizove byteova:

```
print('utf-8      ', "č".encode('utf-8'))
print('cp852      ', "č".encode('cp852'))
print('cp1250     ', "č".encode('cp1250'))
print('ISO-8859-2 ', "č".encode('ISO-8859-2'))
print('ISO-8859-16', "č".encode('ISO-8859-16'))
print('UTF-32     ', "č".encode('UTF-32'))
print('UTF-16     ', "č".encode('UTF-16'))
```

```
utf-8       b'\xc4\x8d'
cp852       b'\x9f'
cp1250      b'\xe8'
ISO-8859-2  b'\xe8'
ISO-8859-16 b'\xb9'
UTF-32      b'\xff\xfe\x00\x00\r\x01\x00\x00'
UTF-16      b'\xff\xfe\r\x01'
```

Umjesto `'č'.encode('utf8')` mogli smo koristiti `bytes('č','utf8')` ili samo `'č'.encode()` jer je `'utf8'` default za `str.encode()` u pythonu 3.

Popis kodiranja koje python podržava: https://docs.python.org/3/library/codecs.html#standard-encodings

Ako radimo s tekstualnom datotekom a ona nije u UTF-8 kodiranju, u pravilu je potrebno otvoriti takvu datoteku navodeći argument `encoding=...` prilikom pozivanja funkcije `open()`. (Drugi način bi bio pročitati je kao binarnu a zatim koristiti `bytes.decode()`.)

Na nekim operativnim sustavima nazivi datoteka su nizovi byteova (linux), a na nekim su nizovi unicode znakova (Windows). U slučaju da je naziv datoteke niz byteova koji se ne može "ispravno" dekodirati u niz unicode znakova (jer je kodiran u drugoj kodnoj stranici, ili je to proizvoljni niz byteova koji ne predstavlja unicode znakove) unicode zapis se komplicira (vidi https://www.python.org/dev/peps/pep-0383). Tada postoji mogućnost otvaranja datoteke pomoću naziva predstavljenog nizom byteova `open(b'naziv.dat',...)`. Za detalje pogledati: http://www.dabeaz.com/python3io/MasteringIO.pdf

Modul `unicodedata` nam može dati podatke o pojedinom unicode znaku:

```
import unicodedata
for c in "čćžšđ":
  print(c, 'U+%04x' % ord(c), "%-16s" % c.encode(), unicodedata.category(c), unicodedata.name(c))
```

```
č U+010d b'\xc4\x8d'      Ll LATIN SMALL LETTER C WITH CARON
ć U+0107 b'\xc4\x87'      Ll LATIN SMALL LETTER C WITH ACUTE
ž U+017e b'\xc5\xbe'      Ll LATIN SMALL LETTER Z WITH CARON
š U+0161 b'\xc5\xa1'      Ll LATIN SMALL LETTER S WITH CARON
đ U+0111 b'\xc4\x91'      Ll LATIN SMALL LETTER D WITH STROKE
```

Promjenom normalizacije možemo prebaciti prikaz npr. sa č na c + kvačica:

```
import unicodedata
for c in "č":
  print('U+%04x' % ord(c), "%-16s" % c.encode(), unicodedata.category(c), unicodedata.name(c))
print()
for c in unicodedata.normalize("NFD","č"):
  print('U+%04x' % ord(c), "%-16s" % c.encode(), unicodedata.category(c), unicodedata.name(c))
```

```
U+010d b'\xc4\x8d'      Ll LATIN SMALL LETTER C WITH CARON

U+0063 b'c'             Ll LATIN SMALL LETTER C
U+030c b'\xcc\x8c'      Mn COMBINING CARON
```

Na sličan način možemo prebaciti npr. znak № u slova od kojih se on sastoji N i o:

```
import unicodedata
for c in "№":
  print('U+%04x' % ord(c), "%-16s" % c.encode(), unicodedata.category(c), unicodedata.name(c))
print()
for c in unicodedata.normalize("NFKC","№"):
  print('U+%04x' % ord(c), "%-16s" % c.encode(), unicodedata.category(c), unicodedata.name(c))
```

```
U+2116 b'\xe2\x84\x96'  So NUMERO SIGN

U+004e b'N'             Lu LATIN CAPITAL LETTER N
U+006f b'o'             Ll LATIN SMALL LETTER O
```

7.5 Generatori/iteratori

7.5.1 Generatorske funkcije

Spomenuli smo da `for` petlja može "ići" ne samo po elementima liste (ili drugih *sequence* tipova) nego po bilo kojem *iteratoru*. *Iteratori* su objekti koji definiraju niz omogućavanjem dohvaćanja sljedećeg elementa iz niza. Time se definira niz jer se ponavljanjem te radnje prođu tj. dohvate svi elementi niza. Iterator dakle pamti koliko elemenata je već prošao te sadrži recept kako dobiti sljedeći element. Pisanje općenitog iteratora obično nije praktično. Jednostavnije je koristiti *generatorske funkcije*. Za razliku od običnih funkcija koje vraćaju jednu vrijednost (koja može biti lista ili `tuple`) generatorske funkcije vraćaju više vrijednosti koristeći princip iteratora. Sama generatorska funkcija izgleda isto kao obična osim što u sebi sadrži naredbu `yield`. `yield` dohvaća sljedeću od više vrijednosti koje generatorska funkcija vraća. Ona je slična naredbi `return` jer kao i `return` izlazi van funkcije i vrati vrijednost. Međutim, kad se zatraži sljedeća vrijednost, generatorske funkcije će se nastaviti izvršavati dalje (a ne od početka) do sljedeće naredbe `yield`. Pogledajmo primjere:

```
def f():
  print("prije prvog yielda")
  yield 1
  print("poslije prvog yielda")
  yield 2
  print("poslije drugog yielda")
  yield 3
  print("poslije trećeg yielda")
  return 4

i = f()
print(i)
print(1)
print('next(i) daje', next(i))
print(2)
print('next(i) daje', next(i))
print(3)
print('next(i) daje', next(i))
try:
  print(4)
  print('next(i) daje', next(i))
except StopIteration as e:
  print('return je vratio',e.value)
```

```
<generator object f at 0x106ec9480>
1
prije prvog yielda
next(i) daje 1
2
poslije prvog yielda
next(i) daje 2
3
poslije drugog yielda
next(i) daje 3
4
poslije trećeg yielda
return je vratio 4
```

Nakon što pozivatelj zatraži sljedeću vrijednost funkcija se izvršava do naredbe `yield`. Kad pozivatelj ponovno zatraži sljedeću vrijednost, izvršavanje funkcije se **nastavlja** od naredbe koja slijedi prethodnoj naredbi `yield` do sljedeće `yield`. Ako pozivatelj zatraži sljedeću vrijednost a izvršavanje funkcije dođe do kraja ili do naredbe `return` to izazove grešku `StopIteration`. Ta greška služi kao oznaka pozivatelju da nema više vrijednosti koje funkcija treba vratiti.

Obično je pozivatelj petlja:

```
def f():
  print("prije prvog yielda")
  yield 1
  print("poslije prvog yielda")
  yield 2
  print("poslije drugog yielda")
  yield 3
  print("poslije trećeg yielda")
  return 4

for x in f():
  print('u petlji',x)
  print('.')
```

```
prije prvog yielda
u petlji 1
.
poslije prvog yielda
u petlji 2
.
poslije drugog yielda
u petlji 3
.
poslije trećeg yielda
```

U gornjem primjeru petlja interno obrađuje grešku tako da nam ovdje detalji obrade pogrešaka nisu bitni. Nešto više detalja o `try/except` je dano u *Zadaća 8 (try ... except ... else, raise)*.

7.5.2 Generator comprehensions

Ako se `for .. in` stavi u uglate zagrade rezultat je lista.

```
i = [x+100 for x in range(0,3)]
print(i)
```

```
[100, 101, 102]
```

Ako se `for .. in` stavi u okrugle zagrade rezultat je iterator.

```
i = (x+100 for x in range(0,3))
print(i)
print('next(i) daje', next(i))
print('next(i) daje', next(i))
print('next(i) daje', next(i))
```

```
<generator object <genexpr> at 0x10d8d5390>
next(i) daje 100
next(i) daje 101
next(i) daje 102
```

Pogledati primjere u odjeljku *Zadatak 3b*.

Više na:

- https://docs.python.org/3/glossary.html#index-20
- https://djangostars.com/blog/list-comprehensions-and-generator-expressions/

7.5.3 Pretvaranje liste u iterator

Koristeći `iter()` možemo prebaciti razne tipove u iterator npr. liste:

```
i = iter([1,2,3])
print(i)
print('next(i) daje', next(i))
print('next(i) daje', next(i))
print('next(i) daje', next(i))
```

```
<list_iterator object at 0x104ea4978>
next(i) daje 1
next(i) daje 2
next(i) daje 3
```

7.5.4 Pretvaranje iteratora u listu

Koristeći `list()` možemo prebaciti razne tipove u listu npr. iteratore:

```
def f():
  yield 1
  yield 2
  yield 3

print(list(f()))
```

```
[1, 2, 3]
```

7.5.5 Pridruživanje iteratora varijablama

Može biti koristan i ovaj način pozivanja generatorskih funkcija:

```
def f():
  yield 1
  yield 2
  yield 3

a, b, c = f()
print(a,b,c)
```

```
1 2 3
```

7.6 Pozivanje funkcija

Python omogućuje raznovrsne načine pozivanja funkcija:

- argumentima prema poziciji (kao u C-u)
- argumentima prema ključnoj riječi
- postavljanjem default vrijednosti za argumente
- pretvaranjem pozicijskih argumenata u tuple i obratno (*)
- pretvaranjem argumenata s ključnim riječima u dictionary i obratno (**)

Pogledajmo primjere:

```
>>> def f(a,b=20000,c=30000,d=40000): print(a,b,c,d)
>>>
>>> f(100,2,3,4) #svi navedeni pozicijski
100 2 3 4
>>>
>>> f(101,2,d=3,c=4) #svi navedeni: neki pozicijski a neki preko ključnih riječi
101 2 4 3
>>>
>>> f(102,**{'c':4,'b':5}) #pomoću ** dict se pretvori u klj. rij. a za one argumente koji nisu poslani postoji default vrijednost
102 5 4 40000
>>>
>>> f(**{'c':4,'a':103}) #slično kao prije
103 20000 4 40000
>>>
>>> #f(b=1) #zabranjeno jer a nije naveden a nema default vrijednost
```

```
>>> def f( *args, **kwargs ): print(f"args={args} kwargs={kwargs}")
>>>
>>> f(100,2) # pretvaranje iz pozicijskih u tuple args
args=(100, 2) kwargs={}
>>>
>>> f(a=101,b=2) #pretvaranje iz argumenata s ključnim riječima u dict
args=() kwargs={'a': 101, 'b': 2}
>>>
>>> f(102,2,a=3,b=4) #kombinacija prethodna dva primjera
args=(102, 2) kwargs={'a': 3, 'b': 4}
>>>
>>> #f(103,a=2,3,b=4) #zabranjeni pozicijski (3) poslije ključnih riječi (a=...)
>>>
>>> f(104,2,*(3,4),a=5,b=6) #pretvaranje iz tuple (3,4) u poz., a zajedno s prva dva pozicijska u tuple args
args=(104, 2, 3, 4) kwargs={'a': 5, 'b': 6}
>>>
>>> f(105,2,*(3,4),a=5,b=6,**{'c':7,'d':8}) #pretvaranje iz dict u klj. rij., zatim sve klj. rij. a=5,b=6,c=7,d=8 u dict kwargs
args=(105, 2, 3, 4) kwargs={'a': 5, 'b': 6, 'c': 7, 'd': 8}
```

Moguće je i kombinirati:

```
def f( a, b=20000, *args, **kwargs ):
  print(f"a={a} b={b} args={args} kwargs={kwargs}")

f(100,q=11,w=22,e=33)
f(100,200,1,2,3,q=11,w=22,e=33)
```

```
a=100 b=20000 args=() kwargs={'q': 11, 'w': 22, 'e': 33}
a=100 b=200 args=(1, 2, 3) kwargs={'q': 11, 'w': 22, 'e': 33}
```

Jednostavna primjena u odjeljku *Rješenje 4* u rješenjima zadaća.

Više na:

- https://docs.python.org/3/glossary.html#term-parameter
- https://docs.python.org/3/glossary.html#term-argument
- https://docs.python.org/3/reference/expressions.html#calls
- https://stackoverflow.com/questions/36901/what-does-double-star-asterisk-and-star-asterisk-do-for-parameters

Rješenja nekih zadataka, koje ste rješavali u C-u, pomoću pythona

Ideja je u pythonu riješiti zadatke koje ste već rješavali u C-u i time produbiti znanje pythona.

8.1 Zadaća 1 (`map()`, `lambda`, `math.sin()`, `sum`, `[ ... for ... in ...]`, `( ... for ... in ...) )`

8.1.1 Zadatak 3b

```
import math
N = int(input())
L = range(0,N+1)
L = (x*math.pi/float(N) for x in L)
L = map( math.sin, L )
print(sum(L))
```

Pokrenemo program i unesemo npr. 10, rezultat je:

```
6.313751514675044
```

U interaktivnom modu možemo pratiti izvršavanje:

```
>>> import math
>>> N = 10
>>> L = list(range(0,N+1))
>>> L = [x*math.pi/10. for x in L]
>>> list(map( lambda x: "%.2f" % x, L ))  # ispis samo 2 decimale
['0.00', '0.31', '0.63', '0.94', '1.26', '1.57', '1.88', '2.20', '2.51', '2.83', '3.14']
>>> L = list(map( math.sin, L ))
>>> list(map( lambda x: "%.2f" % x, L ))  # ispis samo 2 decimale
['0.00', '0.31', '0.59', '0.81', '0.95', '1.00', '0.95', '0.81', '0.59', '0.31', '0.00']
>>> sum(L)
6.313751514675044
```

Pozor: Za razliku od samostalnog programa gdje smo svaki međurezultat koristili samo jednom u interaktivnom primjeru smo npr. `L` iz 4. retka koristili jednom u 5. a drugi put u 6. retku. To je važno uočiti jer u slučaju da nam neka lista treba samo jednom možemo koristiti tzv. iteratore koji **ne zauzimaju toliko memorije** koliko liste i računaju se tek kad zatreba. Mana iteratora je da ih se može koristiti samo jednom. Također treba uočiti da je rezultat od `map()` iterator, kao i rezultat od `( ... for ... in ...)`. Naprotiv, rezultat od `[ ... for ... in ...]` je lista. Više o generatorima u odjeljku *Generatori/iteratori*.

Kako iteratore možemo upotrijebiti samo jednom, u drugom primjeru smo morali koristiti `list()` i pretvoriti iteratore u liste da bismo ih mogli koristiti više puta.

8.2 Zadaća 2 (str.join(), *args)

8.2.1 Zadatak 1b

Rješenje 1

```
L = [x * 0.1 for x in range(31,41)]

print("Tablica mnozenja 1b:")
for i in L:
  print("%5.2f" % (i*L[0]), end='')
  for j in L[1:]:
    print("|%5.2f" % (i*j), end='')
  print()
```

Rješenje 2

Moguće je koristiti direktan upis stringa u `stdout` pomoću `sys.stdout.write`.

```
import sys
L = [x * 0.1 for x in range(31,41)]

print("Tablica mnozenja 1b:")
for i in L:
  sys.stdout.write( "%5.2f" % (i*L[0]) )
  for j in L[1:]:
    sys.stdout.write( "|%5.2f" % (i*j) )
  print()
```

Rješenje 3

Za rješavanje ovog zadatka može poslužiti funkcija `str.join()`.

```
print("-između-".join(["100","200"," ","abc"]))
```

```
100-između-200-između- -između-abc
```

Počnemo tako da generiramo tablicu množenja kao dvodimenzionalnu listu `L2`.

```
L = [x * 0.1 for x in range(31,41)]
L2 = [["%5.2f"%(i*j) for j in L] for i in L]
print(L2[0])
print("...")
print(L2[-1])
```

```
[' 9.61', ' 9.92', '10.23', '10.54', '10.85', '11.16', '11.47', '11.78', '12.09', '12.40']
...
['12.40', '12.80', '13.20', '13.60', '14.00', '14.40', '14.80', '15.20', '15.60', '16.00']
```

Zatim povežemo unutarnju listu znakom `"|"`, a vanjsku listu znakom za novi red `"\n"`.

```
L = [x * 0.1 for x in range(31,41)]
L2 = ["|".join(["%5.2f"%(i*j) for j in L]) for i in L]

print("Tablica mnozenja 1b:")
print("\n".join(L2))
```

Rješenje 4

```
L = [1,2,3]
print(L)
print(*L)
print(*L,sep='//')
```

```
[1, 2, 3]
1 2 3
1//2//3
```

Vidimo da ako je `L=[1,2,3]` tada je `print(L)` isto što i `print([1,2,3])` (ispis liste), a `print(*L)` isto što i `print(1,2,3)` (ispis 3 broja).

```
L = [x * 0.1 for x in range(31,41)]

print("Tablica mnozenja 1b:")
for x in L:
  print( *["%5.2f" % (x*y) for y in L], sep='|' )
```

```
Tablica mnozenja 1b:
 9.61| 9.92|10.23|10.54|10.85|11.16|11.47|11.78|12.09|12.40
 9.92|10.24|10.56|10.88|11.20|11.52|11.84|12.16|12.48|12.80
10.23|10.56|10.89|11.22|11.55|11.88|12.21|12.54|12.87|13.20
10.54|10.88|11.22|11.56|11.90|12.24|12.58|12.92|13.26|13.60
10.85|11.20|11.55|11.90|12.25|12.60|12.95|13.30|13.65|14.00
11.16|11.52|11.88|12.24|12.60|12.96|13.32|13.68|14.04|14.40
11.47|11.84|12.21|12.58|12.95|13.32|13.69|14.06|14.43|14.80
11.78|12.16|12.54|12.92|13.30|13.68|14.06|14.44|14.82|15.20
12.09|12.48|12.87|13.26|13.65|14.04|14.43|14.82|15.21|15.60
12.40|12.80|13.20|13.60|14.00|14.40|14.80|15.20|15.60|16.00
```

Više o pozivima funkcija, `*args` i `**kwargs` u odjeljku *Pozivanje funkcija*.

8.3 Zadaća 3 (`zip`, `input`, `math.sqrt()`)

8.3.1 Zadatak 4

Promotrimo ovaj primjer:

```
>>> list(zip( [100,200], [3,4] ))
[(100, 3), (200, 4)]
>>> [i+j for i,j in zip([100,200],[3,4])]
[103, 204]
```

Vidimo kako je moguće jednostavno raditi operacije nad dvjema listama, tako da se uzima prvi (100) s prvim (3), drugi (200) s drugim (4), itd. To iskoristimo za rješavanje zadatka.

```
import math
N = int(input())
a = []
b = []
for i in range(0,N):
  a.append( float(input()) )

for i in range(0,N):
  b.append( float(input()) )

ab = (i*j for i,j in zip(a,b))
a2 = (i*i for i in a)
b2 = (i*i for i in b)

ab = sum(ab)
a2 = math.sqrt(sum(a2))
b2 = math.sqrt(sum(b2))

print("cos(theta) je", ab/a2/b2)
```

Unesemo `2 1 0 1 1` kao input. Rezultat je:

```
cos(theta) je 0.7071067811865475
```

8.4 Zadaća 4

Problemi opisani u zadaći 4 ne javljaju se u pythonu jer u pythonu "nema" pointera, a i provjerava se da indeks polja bude unutar trenutno dopuštenog opsega. [U CPythonu možemo doznati adresu pomoću funkcije `id()`, ali na adrese ne možemo upisivati direktno.]

8.5 Zadaća 5 (`str.replace()`)

8.5.1 Zadatak 3

Trivijalno jer python podržava rad s proizvoljno dugim cijelim brojevima.

8.5.2 Zadatak 4

Trivijalno jer već postoji takva funkcija u pythonu.

```
print("1456".replace("1","123"))
```

```
123456
```

8.6 Zadaća 6 (`itertools.groupby()`)

8.6.1 Zadaci 2a i 2b

Za rješavanje ovog zadatka dobro može poslužiti funkcija **groupby** iz modula **itertools**. Ta funkcija razdvaja listu ili string na dijelove:

```
>>> import itertools
>>> [k for k,v in itertools.groupby("aaabbcd")]
['a', 'b', 'c', 'd']
>>> [list(v) for k,v in itertools.groupby("aaabbcd")]
[['a', 'a', 'a'], ['b', 'b'], ['c'], ['d']]
>>> [len(list(v)) for k,v in itertools.groupby("aaabbcd")]
[3, 2, 1, 1]
>>> [k*len(list(v)) for k,v in itertools.groupby("aaabbcd")]
['aaa', 'bb', 'c', 'd']
```

Koristeći to, lako je napraviti `rle.py` i `unrle.py`.

```
#!/usr/bin/env python3

import sys
import itertools

def rle(src,dest):
  str = src.read()
  str = "".join( ["%d%s" % (len(list(v)),k) for k,v in itertools.groupby(str)] )
  dest.write(str)

with open(sys.argv[1],"r") as src:
  with open(sys.argv[2],"w") as dest:
    rle(src,dest)
```

```
#!/usr/bin/env python3

import sys

def unrle(src,dest):
  str = src.read()
  assert len(str) % 2 == 0
  ponavljanja = str[0::2] #svaki drugi parni
  ponavljanja = map(int,ponavljanja)
  znakovi = str[1::2] #svaki drugi neparni
  str = [z*p for z,p in zip(znakovi,ponavljanja)]
  str = "".join(str)
  dest.write(str)

with open(sys.argv[1],"r") as src:
  with open(sys.argv[2],"w") as dest:
    unrle(src,dest)
```

Da bi se ti programi mogli pokretati kao samostalni, potrebno je dodati pravo izvršavanja.

```
$ chmod u+x rle.py
$ chmod u+x unrle.py
```

Kad datoteku `z6src.py`

```
aaaabbbbaaaaccccdef
```

komprimiramo

```
$ ./rle.py z6src.txt z6out.txt
```

dobijemo

```
4a4b4a4c1d1e1f
```

Kad taj rezultat dekomprimiramo

```
$ ./unrle.py z6out.txt z6out2.txt
```

dobijemo nazad početni sadržaj

```
aaaabbbbaaaaccccdef
```

Naravno, ovdje se pretpostavljalo, kako je rečeno u zadatku, da se znakovi neće ponavljati više od 9 puta.

8.7 Zadaća 7

8.7.1 Zadatak 2

Rješenje je:

```
#!/usr/bin/env python3

import sys
import struct
import os

def isprint(c):
  return 0x20<=ord(c)<=0x7e

def konverzija(c):
  if isprint(c):
    return c.decode('ascii')
  else:
    return "[%02x]" % ord(c)

D, P, N, T = sys.argv[1:5]
P = int(P)
N = int(N)
rjecnik = {}
rjecnik["c"] = "c"
rjecnik["d"] = "i"
rjecnik["ld"] = "l"
rjecnik["f"] = "f"
rjecnik["lf"] = "d"

with open(D,"rb") as f:
  fmt = '%d%s' % (N,rjecnik[T])
  duljina = struct.calcsize(fmt)
  f.seek( P, os.SEEK_SET )
  byteovi = f.read( duljina )
  #print(f"s pozicije {P} procitati {duljina} procitano {len(byteovi)}:", repr(byteovi), file=sys.stderr)
  assert len(byteovi) == duljina

vrijednosti = struct.unpack( fmt, byteovi )
if T == "c":
  print( *map( konverzija, vrijednosti ), sep='' )
else:
  print( *vrijednosti )
```

Tu smo koristili pythonov dictionary i time smo izbjegli potrebu za velikim brojem if-ova. Funkcija `isprint` ne postoji u pythonu, ali možemo pogledati kako je napravljena u C-u pa taj kôd prevesti u python. Gornju datoteku nazovemo `rr.py`.

Krećemo od datoteke iz jednog od prethodnih primjera:

```
$ hexdump -C bin1.bin
00000000  74 65 6b 73 74 00 00 00  00 00 00 00 0a 00 00 00  |tekst...........|
00000010  9a 99 99 99 99 99 b9 3f                           |.......?|
00000018
```

Kad postavimo flag izvršavanja datoteci `rr.py` možemo radi testiranja pokrenuti program nad podacima u `bin1.bin`.

```
$ chmod u+x rr.py
$ ./rr.py bin1.bin 0 10 c
tekst[00][00][00][00][00]
$ ./rr.py bin1.bin 12 1 d
10
$ ./rr.py bin1.bin 16 1 lf
0.1
```

Zaključujemo da postoji prazan prostor (zbog alignmenta) u byteovima redni broj 10 i 11.

8.8 Zadaća 8 (`try` ... `except` ... `else`, `raise`)

8.8.1 Zadatak 2

U pythonu postoji funkcija `read` slična traženoj. Python se brine oko dealokacije. Duljina `N` nam nije potrebna jer uvijek možemo koristiti `len`. Prikazat ćemo dva rješenja: prvo nepotpuno ali iznimno jednostavno, a zatim, drugo, potpuno rješenje.

Za obradu grešaka u pythonu je prirodno koristiti tzv. `try..except` konstrukciju. Izvršavanje krene u `try` blok i izvršava se red po red ali ako dođe do greške odmah se **nepovratno** napušta `try` blok i prelazi u prvi kompatibilan `except` blok. (U `except` blok se ulazi samo ako se dogodi greška u `try` bloku koji mu prethodi. U `else` blok se ulazi samo ako se čitav prethodni `try` blok izvršio bez greške.)

```
#!/usr/bin/env python3

import sys

def readfile( fn ):
  with open( fn, "rb" ) as f:
    s = f.read()
  return s

try:
  p = readfile( sys.argv[1] )
except Exception as e:
  print("greška ", e)
  errcode = 1
else:
  print(f"datoteka je uspješno pročitana, i sadrži {len(p)} byteova")
  errcode = 0

sys.exit(errcode)
```

Ako bismo baš htjeli da povratne vrijednosti budu diferencirane kao što se traži u zadatku potrebno je dodati još provjera.

```
#!/usr/bin/env python3

import sys

class Gr(Exception): pass
class GrOtvaranje(Gr): pass
class GrRezMem(Gr): pass
class GrCitanje(Gr): pass
class GrNepoznata(Gr): pass

def readfile( fn ):
  try:
    try:
      f = open( fn, "rb" )
    except IOError as e:
      raise GrOtvaranje(e)
    else:
      try:
        with f:
          s = f.read()
      except IOError as e:
        raise GrCitanje(e)
      except MemoryError as e:
        raise GrRezMem(e)
  except Gr as e:
    raise
  except Exception as e:
    raise GrNepoznata(e)

  return s

try:
  p = readfile( sys.argv[1] )
  print(f"datoteka je uspješno pročitana, i sadrži {len(p)} byteova")
  errcode = 0
except GrOtvaranje as e:
```

```
    print("greška prilikom otvaranja datoteke", e)
    errcode = 1
  except GrRezMem as e:
    print("greška prilikom rezerviranja memorije", e)
    errcode = 2
  except GrCitanje as e:
    print("greška prilikom citanja datoteke", e)
    errcode = 3
  except GrNepoznata as e:
    print("nepoznata greška u readfile", e)
    errcode = 4
  except:
    print("neočekivana greška", sys.exc_info())
    errcode = 5

sys.exit(errcode)
```

Isprobajmo sad oba primjera. Pogledajmo output sljedećih naredbi upisanih u terminal:

```
$ chmod +x readfile.py
$ chmod +x readfile2.py
$ ./readfile.py bin1.bin; echo $?
datoteka je uspješno pročitana, i sadrži 24 byteova
0
$ ./readfile2.py bin1.bin; echo $?
datoteka je uspješno pročitana, i sadrži 24 byteova
0
```

Isprobajmo kako rade kad im se zada pogrešno ime datoteke odnosno ime datoteke koja ne postoji:

```
$ ./readfile.py nepostojeca.datoteka; echo $?
greška  [Errno 2] No such file or directory: 'nepostojeca.datoteka'
1
$ ./readfile2.py nepostojeca.datoteka; echo $?
greška prilikom otvaranja datoteke [Errno 2] No such file or directory: 'nepostojeca.datoteka'
1
```

Isprobajmo što se događa u slučaju kad im se zada ime postojeće datoteke za koju je ukinuto pravo čitanja:

```
$ chmod u-r bin1.bin
$ ./readfile.py bin1.bin; echo $?
greška  [Errno 13] Permission denied: 'bin1.bin'
1
$ ./readfile2.py bin1.bin; echo $?
greška prilikom otvaranja datoteke [Errno 13] Permission denied: 'bin1.bin'
1
$ chmod u+r bin1.bin
```

Vidimo da oba programa ispravno obrađuju greške. Poanta je da koristeći `try/except` lako možemo postići osnovnu obradu grešaka kao u prvom primjeru, te po potrebi nadograđivati kao u drugom primjeru.