Računarstvo i praktikum / Uvod u python

 [1]

Maro Cvitan

22. prosinca 2015.

Sadržaj

Uvod

Ovaj dokument služi kao pomoć pri učenju osnova pythona u sklopu kolegija Računarstvo i praktikum na prvoj godini istraživačkog smjera studija fizike na Fizičkom 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 je napravljen tako da olakša (=ubrza) pisanje programa. Na primjer, nema deklariranja varijabli unaprijed, a dopušteno je miješanje tipova što znači dopušteno je `a=1` i nakon toga `a="xy"`.
- Programe napisane u C-u je potrebno prvo prevesti u strojni jezik, a zatim izvršiti. Python umjesto toga koristi "simultano prevođenje" tj. program se prevodi u strojni jezik malo po malo. 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`, upisujemo/izvršavamo naredbu po naredbu, iz ljuske izađemo pomoću CTRL-d — ovakav način nam omogućuje brzo isprobavanje dijelova programa;
  - samostalni programi: upišemo cijeli program pomoću tekst-editora i snimimo ga npr. pod imenom `prg.py` zatim pokrenemo pomoću `python prg.py`. (Ako postavimo da je datoteka `prg.py` izvršna, pomoću `chmod u+x prg.py`, i ako je prva linija 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.
  - python2 (npr. python 2.6, python 2.7)
  - python3 (npr. python 3.4) - **nije** *kompatibilan unazad* s pythonom 2 (to znači da programe za python2 treba prepraviti ako ih se želi izvršiti pomoću pythona 3).
- Kako je python2 raširenija, koristit ćemo njega.
- Gore spomenuti python interpreteri (oni s https://www.python.org) predstavljaju jednu (standardnu) implementaciju jezika python, napisanu u C-u koja se naziva CPython. 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). Također postoje i RPython (podskup pythona) i Cython (http://cython.org, prevoditelj iz pythona u C) kojima je glavni cilj ubrzavanje izvršavanja. U nastavku ćemo se baviti samo standardnom, CPython, implementacijom.

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:

- http://cs.stanford.edu/people/nick/python-in-one-easy-lesson/
- https://developers.google.com/edu/python/
- https://docs.python.org/2/tutorial/

- http://www.tutorialspoint.com/python/python_variable_types.htm

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).

3. Ako ništa drugo ne upali, možete probati tražiti po službenim uputstvima za python:

- https://docs.python.org/2/
- https://docs.python.org/2/library/index.html

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

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

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

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

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

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

7. Za određene primjene dobro je znati da postoje tzv. *regular expressions*. Ovo izlazi, naravno, van okvira kolegija, iako ste najjednostavniji oblik *regular expression*a već vidjeli prilikom korištenja `scanf()` npr. `scanf("%[^\n]",s)` učitava u `s` sve do znaka za novi red.

- https://developers.google.com/edu/python/regular-expressions
- http://www.tutorialspoint.com/python/python_reg_expressions.htm
- https://docs.python.org/2/howto/regex.html

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.

Usporedba osnovnih elemenata C-a i python-a

2.1 Osnovni tipovi podataka

Python	Komentar	C

`boolean`
`>>> a = True`
`>>> print a, type(a)`
`True <type '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`.	Nema, u C-u se `int` koristi umjesto `boolean` (to 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 igrati se s pojedinačnim bitovima kao ovdje ili ovdje.

`int`
`>>> a = -1000`
`>>> print a, type(a)`
`-1000 <type 'int'>`
	Pythonov `int` je implementiran pomoću C-ovog `long`-a.	`long`

`long`
`>>> a = 123456789012345678901234567890`
`>>> print a, type(a)`
`123456789012345678901234567890 <type 'long'>`
	Pythonov `long` drži cijele brojeve proizvoljnog broja znamenaka.	C nema ugrađen takav tip.

`float`
`>>> a = 1.0e-300`
`>>> print a, type(a)`
`1e-300 <type '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) <type '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 <type '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 <type '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/2/library/stdtypes.html#numeric-types-int-float-long-complex
- http://www.diveintopython3.net/native-datatypes.html
- http://www.tutorialspoint.com/python/python_variable_types.htm
- http://www.python-course.eu/sequential_data_types.php
- http://stackoverflow.com/questions/47981/how-do-you-set-clear-and-toggle-a-single-bit-in-c-c
- http://graphics.stanford.edu/~seander/bithacks.html

2.2 Osnovni elementi

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

Python	C
`and`	`&&`
`or`	`||`
`a**b`	`pow(a,b)`
`not`	`!`
`id(x)`	`&x` (adresa od x)
`break`	`break;`
`continue`	`continue;`

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

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

U sljedećoj tablici su dane daljnje sličnosti i razlike.

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 ne radi ništa.	
svaka naredba u svom redu ili `;`	`;`
`'xx'` ili `"xx"` ili `"""xx"""` (trostruki navodnici mogu uključivati više redaka teksta)	`"xx"`
nema tip podataka znak (char). umjesto toga koristi string duljine 1: `"x"` ili `'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

2.3 `for` petlja

Python

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

funkcija `range()` u pythonu proizvodi listu sa zadanim granicama.

```
print range(1,26,5)
```

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

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

Savjet: ako se na ovakav način radi, python2, u memoriji drži odjednom sve vrijednosti koje se pojavljuju u petlji. to, naravno, nije zgodno u slučaju `range(0,1000000000)` pa se u tom slučaju koristi `xrange(0,1000000000)`. `xrange()` ne generira cijelu listu unaprijed pa ne zauzima nepotrebno memoriju. (u pythonu 3 range se ponaša kao xrange u pythonu 2. u pythonu 3 xrange ne postoji.) http://stackoverflow.com/questions/135041/should-you-always-favor-xrange-over-range

C

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

2.4 Formatiranje (formatirani ispis)

formatiranje stringova i brojeva.

Savjet: korisna stvar u pythonu 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);
printf("%s\n",b);
sprintf(d,"x=%f,y=%f", 1.0, 2.2);
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/2/library/string.html#format-examples
- https://docs.python.org/2/library/string.html#template-strings

2.5 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
```

2.5.1 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 onda takve strukture nadodavati s jedne liste na drugu...

Napomena: python omogućuje i razne druge operacije nad listama - pogledati https://docs.python.org/2/tutorial/datastructures.html

2.6 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

```
# -*- coding: utf-8 -*-
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/2/library/stdtypes.html#string-methods

2.7 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 se ne kopira sadržaj. U sljedećem primjeru `a` i `b` pokazuju na isti objekt što znači kad se promijeni `a[0]`, promijeni se i `b`.

```
>>> 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

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)`. (U C-u: `&a`.)

immutable	mutable	mutable

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

`1234` `123`	`[1, 2, 3, 4]` `[1, 2, 3, 4]`	`[1, 2, 3, 4]` `[1, 2, 3, 4]`
Treba primijetiti da se `b` nije promijenio jer se stringovi ne mogu mijenjati, samo može nastati novi string.	Treba primijetiti da se i `b` promijenio zato što je pokazivao na listu koja se u međuvremenu promijenila.	Isto kao primjer s `list.append()` (ostale moguće operacije u tablici *Mutable Sequence Types*).

Literatura

- 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/2/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*

Datoteke

4.1 Tekstualne datoteke

Primjer koji upisuje neka slova i brojeve u datoteku.

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

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 pa onda 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.

4.2 Binarne datoteke

C interno u memoriji drži samo gole podatke koje je stoga lako direktno prepisati iz memorije u datoteku. Pyhton 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 repr(struct.pack('10s',"tekst"))
'tekst\x00\x00\x00\x00\x00'
>>> print repr(struct.pack('20s',"tekst"))
'tekst\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
>>> print repr(struct.pack('i',10))
'\n\x00\x00\x00'
>>> print repr(struct.pack('d',0.1))
'\x9a\x99\x99\x99\x99\x99\xb9?'
```

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

```python
import struct
L=["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/2/library/struct.html#byte-order-size-and-alignment i na https://docs.python.org/2/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
```

odnosno `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
```

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

```
('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
```

Detalji

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

5.1 Uobičajeno zaglavlje

Za kodiranje slova s kvačicama preporuča se koristiti utf-8 kodiranje. U pythonu2, potrebno je dodati zaglavlje `# -*- coding: utf-8 -*-`. U pythonu3, utf-8 je default.

Da bi se program u pythonu mogao pokretati kao samostalan program (tj. direktno, kao što se pokreće `./a.out`) potrebno je 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
# -*- coding: utf-8 -*-

print "test kvačice"
```

```
$ chmod +x zag.py
$ ./zag.py
test kvačice
```

Umjesto zaglavlja `#!/usr/bin/env python` koji nalaže da se koristi defaultni sistemski python interpreter, može se u zaglavlje staviti putanja na konkretni python interpreter, npr: `#!/usr/bin/python2.7`. Ako se radi na ovaj način moguće je poslati parametre pythonu: `#!/usr/bin/python2.7 -tt`. Opcija `-tt` uzrokuje prekid izvršavanja ako interpreter primijeti u programu miješanje razmaka i tabova.

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

5.2 Parametri programa

Slično kao u C-u postoji polje u kojem je naziv programa nakon kojeg slijede parametri main-a.

```
#include <stdio.h>

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

```
$ python op1_arg1.py 1 2 3
0 ./a.out
1 1
2 2
3 3
```

```
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>

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

```
$ 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
```

5.3 Automatsko oslobađanje memorije

Python shvati kad se objekt više ne koristi i onda ga uništi. 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
```

5.4 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 neki načini normalizacije mogu biti ireverzibilni.

Više na:

- http://en.wikipedia.org/wiki/UTF-8

- https://docs.python.org/2/howto/unicode.html
- http://www.joelonsoftware.com/articles/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 programima dodati liniju koja označava da je kodiranje koje se koristi UTF-8

U nekim slučajevima je korisno da python interno vodi neki string kao unicode. To se može postići prefixom `u` ispred navodnika.

```
# -*- coding: utf-8 -*-

a = "č"
print a
print repr(a)

a = u"č"
print a
print repr(a)
```

```
č
'\xc4\x8d'
č
u'\u010d'
```

Ako imamo string koji nije interno prikazan kao unicode možemo ga dekodirati iz njegovog kôda u unicode pomoću `string.decode()`.

```
# -*- coding: utf-8 -*-

a = "č"
print repr(a.decode("utf8"))
```

```
u'\u010d'
```

Ako imamo string koji jest interno prikazan kao unicode, možemo ga kodirati u neki drugi kôd pomoću `string.encode()`.

```
# -*- coding: utf-8 -*-
b = u"č"
print repr(b.encode('utf-8'))
print repr(b.encode('ISO-8859-2'))
print repr(b.encode('ISO-8859-16'))
print repr(b.encode('cp852'))
print repr(b.encode('cp1250'))
print repr(b.encode('UTF-32'))
print repr(b.encode('UTF-16'))
```

```
'\xc4\x8d'
'\xe8'
'\xb9'
'\x9f'
'\xe8'
'\xff\xfe\x00\x00\r\x01\x00\x00'
'\xff\xfe\r\x01'
```

Popis kodiranja koje python podržava: https://docs.python.org/2/library/codecs.html#standard-encodings

Na ovaj način, koristeći prvo decode, a zatim encode, možemo promijeniti kodiranje neke tekstualne datoteke. Za tu svrhu postoje i gotovi programi (npr. `recode`) (http://stackoverflow.com/questions/64860/best-way-to-convert-text-files-between-character-sets).

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

```
# -*- coding: utf-8 -*-
import unicodedata

a = u"čćžšđ"

for c in a:
  print '%04x' % ord(c), "%10s" % repr(c), unicodedata.category(c), unicodedata.name(c)
```

```
010d  u'\u010d' Ll LATIN SMALL LETTER C WITH CARON
0107  u'\u0107' Ll LATIN SMALL LETTER C WITH ACUTE
017e  u'\u017e' Ll LATIN SMALL LETTER Z WITH CARON
0161  u'\u0161' Ll LATIN SMALL LETTER S WITH CARON
0111  u'\u0111' Ll LATIN SMALL LETTER D WITH STROKE
```

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

```
# -*- coding: utf-8 -*-
import unicodedata

a = u"č"
b = unicodedata.normalize("NFD",a)

print repr(a)
for c in a:
  print '%04x' % ord(c), "%10s" % repr(c), unicodedata.category(c), unicodedata.name(c)

print repr(b)
for c in b:
  print '%04x' % ord(c), "%10s" % repr(c), unicodedata.category(c), unicodedata.name(c)
```

```
u'\u010d'
010d  u'\u010d' Ll LATIN SMALL LETTER C WITH CARON
u'c\u030c'
0063       u'c' Ll LATIN SMALL LETTER C
030c  u'\u030c' Mn COMBINING CARON
```

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

```
# -*- coding: utf-8 -*-
import unicodedata

a = u"№"
b = unicodedata.normalize("NFKC",a)

print repr(a)
for c in a:
  print '%04x' % ord(c), "%10s" % repr(c), unicodedata.category(c), unicodedata.name(c)

print repr(b)
for c in b:
  print '%04x' % ord(c), "%10s" % repr(c), unicodedata.category(c), unicodedata.name(c)
```

```
u'\u2116'
2116  u'\u2116' So NUMERO SIGN
u'No'
004e       u'N' Lu LATIN CAPITAL LETTER N
006f       u'o' Ll LATIN SMALL LETTER O
```

Rječnik (Dictionary)

Dictionary (rječnik) je tip podataka sličan polju, samo što index nije cjeli broj nego bilo koji *immutable* tip podataka kao što su *string*ovi ili brojevi.

6.1 Kreiranje

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

```
{'sije\xc4\x8danj': 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
```

6.2 Očitavanje i mijenjanje

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

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

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

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

```
True
False
```

6.3 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
limun 1
jabuka 2
```

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
limun 1
jabuka 2
```

Dodatni primjeri na https://docs.python.org/2/library/collections.html#defaultdict-examples.

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

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

for i,s in enumerate(L):
  print "%d %s" % (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 "%d %s" % (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/2/howto/sorting.html#sortinghowto
- https://docs.python.org/2/library/functions.html#sorted

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

8.1 Zadaća 1 (map(), lambda, math.sin(), sum)

8.1.1 Zadatak 3b

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

Pokrenemo program i unesemo npr. 10, rezultat je:

```
6.31375151468
```

U interaktivnom modu možemo pratiti izvršavanje:

```
>>> import math
>>> N = 10
>>> L = range(0,N+1)
>>> L
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> L = map( lambda x: x*math.pi/10., L )
>>> 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 = map( math.sin, L )
>>> 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)
6.31375151468
```

8.2 Zadaća 2 ([... for ... in ...], string.join())

8.2.1 Zadatak 1b

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

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

```
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
```

Naredba `print` zajedno sa zarezom omogućuje da ostanemo u istom retku ali ostavlja neželjeni razmak. Rješenje se može naći tražeći `python print without newline space` na google-u, i to je koristiti `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
```

```
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
```

Za rješavanje ovog zadatka može poslužiti funkcija `string.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)
```

```
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
```

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

8.3.1 Zadatak 4

Promotrimo ovaj primjer:

```
>>> 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 = input()
a = []
b = []
for i in range(0,N):
  a.append( input() )

for i in range(0,N):
  b.append( 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 0.707106781187
```

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 (`string.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 python
# -*- coding: utf-8 -*-

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 python
# -*- coding: utf-8 -*-

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 +x rle.py
$ chmod +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 python
# -*- coding: utf-8 -*-

import sys
import struct
import os

def isprint(c):
```

```
  return 0x20<=ord(c)<=0x7e

def konverzija(c):
  if isprint(c):
    return c
  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 >>sys.stderr, "s pozicije %d procitati %d procitano %d. " % (P,duljina,len(byteovi)), repr(byteovi)
  assert len(byteovi) == duljina

vrijednosti = struct.unpack( fmt, byteovi )
if T == "c":
  vrijednosti = map( konverzija, vrijednosti )
  s = "".join(vrijednosti)
else:
  vrijednosti = map( str, vrijednosti )
  s = " ".join(vrijednosti)

print s
```

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 +x rr.py
./rr.py bin1.bin 0 10 c
./rr.py bin1.bin 12 1 d
./rr.py bin1.bin 16 1 lf
```

```
tekst[00][00][00][00][00]
10
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 jednostavno koje je malo nepotpuno, a zatim potpuno rješenje.

Za obradu grešaka je u pythonu 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 python
# -*- coding: utf-8 -*-

import sys
```

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

try:
  p = readfile( sys.argv[1] )
  print "datoteka je uspjesno procitana, i sadrzi %d byteova" % len(p)
  errcode = 0
except Exception as e:
  print "greska ", e
  errcode = 1

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 python
# -*- coding: utf-8 -*-

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, "r" )
    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 "datoteka je uspjesno procitana, i sadrzi %d byteova" % len(p)
  errcode = 0
except GrOtvaranje as e:
  print "greska prilikom otvaranja datoteke", e
  errcode = 1
except GrRezMem as e:
  print "greska prilikom rezerviranja memorije", e
  errcode = 2
except GrCitanje as e:
  print "greska prilikom citanja datoteke", e
  errcode = 3
except GrNepoznata as e:
  print "nepoznata greska u readfile", e
  errcode = 4
except:
  print "neocekivana greska", 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
./readfile2.py bin1.bin
```

```
datoteka je uspjesno procitana, i sadrzi 24 byteova
datoteka je uspjesno procitana, i sadrzi 24 byteova
```

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

```
./readfile.py nepostojeca.datoteka
echo $?
./readfile2.py nepostojeca.datoteka
echo $?
```

```
greska  [Errno 2] No such file or directory: 'nepostojeca.datoteka'
1
greska 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 $?
./readfile2.py bin1.bin
echo $?
chmod u+r bin1.bin
```

```
greska  [Errno 13] Permission denied: 'bin1.bin'
1
greska prilikom otvaranja datoteke [Errno 13] Permission denied: 'bin1.bin'
1
```