```
#####################################################################
#               ..// Osnove Python jezika //..                       #
#####################################################################
#                                                                   #
# => izvuchene najbitnije stvari iz tutorijala koji                 #
#    dolazi uz python...podrazumjeva se da vec znate                #
#    osnove programiranja...                                        #
#                                                                   #
# => by Mladen Kostresevic - [zevs][at][sendmail.ru]                #
#                                                                   #
# => ljeta gospodnjeg 2004.                                         #
#    Belgrade, SD "4.April", Vozdovac                               #
#####################################################################
```

Neke osnovne informacije:

- Interpreterski jezik
- Pisan u C-u i moguce ga je prosirivati u istom jeziku
- Zastupljen na velikom broju platformi
- Komentari pocinju sa #
- Nema blokova BEGIN..END ili {...} vec se blokovi naredbi
  grupisu "uvlacenjem" koda...
- Nema ; na kraju naredbe
- Source Code Encoding je moguce mjenjati..npr:

```
 # -*- coding: iso-8859-1 -*-
```

- Za dodjelu vrijednosti koristi se =
- Stringovi mogu biti ograniceni sa 'moj_string' ili "moj_string"
- Viselinijski stringovi mogu biti napravljeni na dva nacina:

```
  print "Prva linija\n\
        Druga linija"

  ili """Prva linija
        Druga linija"""
```

- Moguce je indeksirati stringove, npr:

```
  Rijec = "Masina"

  Rijec[0]      # je M
  Rijec[0:2]    # je Ma
  Rijec[2:4]    # je si

  Rijec[:2]    # prva dva karaktera
  Rijec[2:]    # sve osim prva dva karaktera
```

- moguce je koristiti negativne brojeve da bi smo indeksirali string
  s desne strane:

Rijec[-1] # poslednji karakter
Rijec[-2:] # podlednja dva karaktera

- Funkcija len vraca duzinu stringa

Hint: nije dozvoljena dodjela vrijednosti nekom indeksu..npr:
    Rijec[0]='M'

##
LISTE
##

- sadrze elemente odvojene zarezom
- elementi nemoraju biti istog tipa

a = ['elemen1', 'e2', 111, 222]

svakom elementu se moze pristupiti preko indeksa

a[0] #'element1'
a[1] #'e2'
a[-2] #111

- funkcija len() moze da se koristi i na listama
- moguce je kreirati liste ciji clanovi su takodje liste:

a=[1,2,3]
b=[a,'test']

u tom slucaju pristup elementima se obavlja na sledeci nacin:
b[0][0] #1
b[0][1] #2

==========

Operatori poredjenja: < > <= >= == != kao u C-u
Naredba Print:

b=1
while b<4
  print b
  b=b+1

... bice isprintano:
1
2
3

...ako zelimo da se sve printa u jednom redu potrebno je staviti zarez
iza print naredbe: print b,

```
#####################################################################
# PETLJE
#####################################################################
```

-> IF

```
... x = int(raw_input("Please enter an integer: "))
... if x < 0:
...      x = 0
...      print 'Negative changed to zero'
... elif x == 0:
...      print 'Zero'
... elif x == 1:
...      print 'Single'
... else:
...      print 'More'
```

HINT: primjetiti mogucnost navodjenja vise elif komandi cime se
zamjenjuje case ili switch naredba u drugim jezicima...

-> FOR

Za razliku od Pascala gdje se iteracija obavlja pomocu brojeva, u
Pythonu se iteracija obavlja preko elemenata nekog niza(liste, stringa):

```
a = ['cat', 'window', 'defenestrate']
for x in a:
   print x, len(x)
```

-> RANGE() Funkcija

ako je potrebno da se iteracija obavlja pomocu brojeva, koristi se
funkcija range() koja kreira listu ciju su elementi brojevi...

1.primjer:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

2.primjer:

```
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(-10, -100, -30)
[-10, -40, -70]
```

3.primjer:
```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print i, a[i]
...
```

-> BREAK i CONTINUE
==========

- break se koristi za "iskakanje" iz najblize for ili while petlje...
- continue se koristi da bi se skocilo na izvrsavanje sledece iteracije u okviru nekog ciklusa...

PASS
==========

ova naredba ne radi nishta..:) koristi se onda kada je radi ispravnosti sintakse potrebno navesti neku naredbu, a program ne treba da radi nista

```
#####################################################################
# FUNKCIJE
#####################################################################
```

Kljucna rijec def oznacava pocetak definisanja funkcije...

```
def activate_bomb(x):
    naredba1
    naredba2
```

Ispod zaglavlja funkcije moguce je postaviti neku vrstu komentara, tzv. docstring...postoje alati koji na osnovu ovih stringova prave html dokumentaciju za svaku funkciju...ovaj docstring je ustvari samo atribut objekta(posto su sve funkcije ustvari objekti) i njegovo ime je __doc__:

```
def activate_bomb(x):
    """ Ovo je opis funkcije """
    naredba1
    naredba2


print ime_funkcije.__doc__
```

- ako je potrebno da funkcija vraca neku vrijednost koristi se kljucna rijec return

- moguce je definisati funkciju sa promjenljivim brojem argumenata... postoji nekoliko nacina za to...najcesci je kada kreiramo funkciju i nekim njenim argumentima dodjelimo default vrijednosti...tako definisana

f-ja moze biti pozvana sa manje argumenata nego sto je definisana...

```
#################################################################
# LISTE - NASTAVAK
#################################################################
```

Spisak metoda koje mozete koristiti sa listama:

- append(x) - dodaje novi element na kraj liste

- extend(L) - prosiruje listu sa elementima liste L

- insert(i,x) - ubacuje element x u listu..parametar i odredjuje indeks
  elementa ispred koga ce se ubaciti x...

- remove(x) - uklanja prvi element iz liste koji ima vrednost x

- del lista[i] - iklanja element iz liste ciji je index i

- pop([i]) - vraca elemenat sa pozicije i + brise ga iz liste...
  zagrade [] nije potrebno navoditi, one oznacavaju da
  je parametar i opcionalan...znaci mozemo ga izostaviti
  lista.pop() ce da vrati i izbrise zadnji elemenat liste

- index(x) - vraca indeks elementa koji ima vrednost x

- count(x) - vraca broj pojavljivanja elementa x u listi

- sort() - sortira listu

- reverse() - "obrce" listu

- filter(funkcija, lista) - vraca listu ciji su elementi oni clanovi
  liste za koje je vrijednost f-je true

```
>>> def f(x): return x % 2 != 0 and x % 3 != 0
...
>>> filter(f, range(2, 25))
[5, 7, 11, 13, 17, 19, 23]
```

- map(funkcije, lista) - poziva se funkcija za svaki od elemenata liste
  i vraca se lista ciji su elementi vracene
  vrijednosti funkcije

```
>>> def cube(x): return x*x*x
...
>>> map(cube, range(1, 11))
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

- reduce(binarna funkcija, lista) - vraca vrijednost binarne funkcije
  za prva dva elementa liste, zatim ponovo poziva funkciju prosledjujuci
  joj tu vrijednost i sledeci element...tako sve do kraja liste...

```
>>> def sum(seq):
...     def add(x,y): return x+y
...     return reduce(add, seq, 0)
...
>>> sum(range(1, 11))
55
```

```
#####################################################################
# Tuples, nesto kao skupovi u Pascalu
#####################################################################
```

Pored lista i stringova u pythonu postoji jos jedan standardni
sekvencioni tip podataka: Tuples... primjer:

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

- u sustini slicno listama...jedna od razlika je u tome sto nije
moguce dodjeliti neku vrijednost odredjenom elementu Tuple-a...ali to
je moguce uraditi pomocu par fazona..:) takodje moguce je u okviru
nekog Tuple-a ubaciti listu kao jedan njegov element...

```
#####################################################################
# RIJECNICI
#####################################################################
```

Jos jedan vrlo koristan tip podataka koji je ugradjen u python jeste
rijecnik...:) u nekim drugim jezicima rijecnici se nazivaju
asocijativnim nizovima..Za razliku od sekvencionalnih tipova podataka
koji su indeksirani brojevima,rijecnici su indeksirani pomocu "kljuceva".
Kljucevi mogu biti npr. brojevi ili stringovi...Rijecnik predstavlja
skup parova kljuc:vrijednost...s tim da jedan rijecnik ne smije imati
vise istih kljuceva...

Rijecnik se kreira na sledeci nacin:

```
Rec = {'Mladen':'064 555 666', 'Milan':'011 555 777'}
```

kreirali smo rijecnik koji ima dva elementa...

Rec.keys() - vraca listu definisanih kljuceva rijecnika

del Rec['Mladen'] - brise elemenat rijecnika ciji je kljuc 'Mladen'

Rec.has_key('Mladen') - vraca true ili false

```
#####################################################################
# MODULI
#####################################################################
```

Kada definisete neki modul(snimite npr neku funkciju u fajl.py), taj
modul mozete koristiti u nekom drugom modulu ili u glavnom programu...
Potrebno je napisati import <ime fajla>, bez ekstenzije...
Tada se stvara novi namespace, u kome se nalazi ono sto je definisano
u modulu...pa cete tim stvarima pristupati npr. ovako:

ime_modula.f-ja...

Postoji mogucnost da importujete modul na sledeci nacin:

from ime_modula import *

...cime ce vam sve iz modula biti dostupno(sem simbola koji pocinju
sa _)...

Svaki modul ima svoju tabelu simbola, koja se koristi kao globalna
tabela simbola od strane f-ja definisanih u tom modulu...

SearchPath za module:
1. trenutni direktorij
2. environment variable PYTHONPATH
3. instalacioni dir

Python dolazi sa bibliotekom standardnih modula..detalje obavezno
pogledajte u "Python Library Reference"..tamo se nalazi dosta funkcija
koje ce vam sigurno biti potrebne...

Npr. mozete koristiti dir(ime_modula) funkciju da bi ste izlistali
sva dostupna imena iz tog modula(znaci imena f-ja, promjenljivih)...

Paketi :
==========
su nacin organizovanja modula za python i organizovanja
tzv. namespace-ova... npr. ako je ima modula A.B, to bi moglo da
znaci da se submodul B nalazi u okviru paketa A... Kao sto
organizovanje programa u module, sprijecava konflikte globalnih
promjenljivih, tako i organizovanje modula u pakete sprijecava

pojavljivanje istih modula istog imena od strane razlicitih autora...

Npr...ako razvijamo neke biblioteke za rad sa audio podacima, struktura naseg paketa bi mogla izgledati ovako:

```
Sound/                  Top-level package
    __init__.py         Initialize the sound package
    Formats/            Subpackage for file format conversions
        __init__.py
        wavread.py
        wavwrite.py
        aiffread.py
        aiffwrite.py
        auread.py
        auwrite.py
        ...
    Effects/            Subpackage for sound effects
        __init__.py
        echo.py
        surround.py
        reverse.py
        ...
    Filters/            Subpackage for filters
        __init__.py
        equalizer.py
        vocoder.py
        karaoke.py
        ...
```

Prilikom importovanja nekog paketa, python pretrazuje direktorije definisane u sys.path u potrazi za package direktorijem.

__init__.py fajl je potreban da bi python tretirao navedene direktorije kao pakete... on moze biti potpuno prazan, ali takodje moze sadrzati inicijalizacioni kod za paket...

Korisnici paketa mogu importovati pojedinacne module npr:

import Sound.Effects.echo

Ista stvar je mogla biti uradjena i ovako:

from Sound.Effects import echo

.....

Postoji jos jedna caka oko importovanja modula/paketa....

moze se pisati: from Sound.Effects import *

to bi trebalo da importuje sve module iz paketa Effects...medjutim
ovo moze biti problem na nekim Mac i Windows sistemima...problem
je u tome sto python ne zna da li da importuje neku funkciju kao
'Saberi' ili 'SABERI' ili 'SaBeRI'..:)

Za takve stvari postoji mogucnost da se u okviru __init__.py
definise lista __all__ koja ce da sadrzi nazive modula koji ce
biti importovani kada se paket importuje pomocu *...npr:

```
__all__ = ["echo", "surround", "reverse"]
```

U sustini ne preporucuje se importovanje pomocu * jer to moze
da dovede to teze citljivog koda...:)

```
##################################################################
# INPUT & OUTPUT
##################################################################
```

funkcija str() - pretvara nesto u string...:)

modul string...sadrzi neke zgodne funkcije za manipulaciju stringovima

print string.rjust(str(x),5) - stampa broj x, u polju duzine 5
                znakova, s desnim poravnavanjem

postoji i string.ljust() - lijevo poravnavanje
kao i string.center() - centralno

postoji jos jedna zgodna funkcija koja stampa numericke vrijednosti sa
nulama ispred nje, u zavisnosti koju duzinu polja navedemo...

```
>>> import string
>>> string.zfill('12', 5)
'00012'
>>> string.zfill('-3.14', 7)
'-003.14'
```

HINT: ako funkciji print prosledimo vise parametara ona sma stampa
    blanko izmedju njih...

Kao i u C-u, tako i u pythonu postoji mogucnost da se print koristi na
sledeci nacin:

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
...     print '%-10s ==> %10d' % (name, phone)
...
Jack       ==>       4098
Dcab       ==>       7678
Sjoerd     ==>       4127
```

.....postoji jos dosta fora oko formatiranja izlaza...ali moracate
ih traziti po dokumentaciji..:)

```
#####################################################################
# FAJLOVI
#####################################################################
```

open() - vraca fajl objekat, i najcesce se koristi ovako:

open(filename, mode)

modovi su: 'r' - read
        'w' - write(ako fajl vec postoji bice prvo unisten)
        'a' - append
        'r+' - read/write
u slucaju da se ne navede mod, bice podrazumjevano 'r'

Kod Windows i Macintosh sistema...postoji i dodatak na osnovni mod
otvaranja fajla 'b', koji oznacava da se fajl otvara u binarnom modu:
'rb', 'wb', i 'r+b'.

read(size) - cita zadati niz bajtova iz fajla...ako je size izostavljeno
cita se cijeli fajl...oprez...moze vam eksplodirati hard disk ako je
velicina fajla duplo veca od vase raspolozive memorije...:)

```
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
```

HINT: primjetite da kada read() stigne na kraj fajla, vraca prazan string

readline() - cita jednu liniju iz fajla

```
>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
''
```

readlines() - vraca listu koja sadzi linije fajla

write(string) - zapisuje sadrzaj stringa u fajl

tell() - vraca integer koji predstavlja trenutnu lokaciju "fajl kursora"
        ...to je broj bajtova, racunajuci od pocetka fajla

seek(offset, from_what) - sluzi za pomjeranje "fajl kursora"

from_what =0 - poceetak fajla
=1 - trenutna pozicija
=2 - kraj fajla

```
>>> f=open('/tmp/workfile', 'r+')
>>> f.write('0123456789abcdef')
>>> f.seek(5)     # Go to the 6th byte in the file
>>> f.read(1)
'5'
>>> f.seek(-3, 2) # Go to the 3rd byte before the end
>>> f.read(1)
'd'
```

.....

f.close() - zatvara fajl

fajl objekti imaju jos metoda, kao npr. truncate()..detalje pogledati u Library Reference...

Pickle Modul
==========

Stringovi se lako mogu citati/pisati u fajl...ali kada je potrebno neki slozeniji tip podataka zapisati/procitati iz faja ona nastaje frka...u pomoc stize modul pickle..:) on moze bilo koji python objekat prebaciti u njegovu string reprezentaciju i obratno...ovaj postupak se naziva pickling/unpickling..

Kada neki objekat pickling-ujemo...mozemo ga upisati u fajl ili sta god..

Ako imamo objekat x, i fajl f otvoren za pisanje...pickling x:

pickle.dump(x,f)

unpickling: x=pickle.load(f)

.....

postoji jos fazona oko pickling-ovanja...pogledaj Reference Manual...

```
#####################################################################
# GRESKE i IZUZECI
#####################################################################
```

Najcesce greske su sintaksne greske i izuzeci...Sintaksne greske su veoma ceste i njih nema potrebe objasnjavati...

Izuzeci nastaju u toku izvrsavanja programa...njih je moguce "presresti" u toku izvrsavanja programa i odrediti sta da se radi u tom slucaju...

Postoji vise definisanih tipova izuzetaka i na osnovu njih mi lakse

mozemo da "vidimo" sta se desilo i kako reagovati...neki od njih su:
ZeroDivisionError, NameError i TypeError...

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined
```

Sledeci primjer pokazuje kako se moze obraditi jedan od izuzetaka:

```
>>> while True:
...     try:
...         x = int(raw_input("Please enter a number: "))
...         break
...     except ValueError:
...         print "Oops! That was no valid number.  Try again..."
```

try klauzula radi ovako:

- prvo se izvrse naredbe izmedju try i except
- ako se ne desi izuzetak, klauzula except se preskace i to je kraj try klauzule
- ako dodje do izuzetka prilikom izvrsavanja try klauzule, ostatak klauzule se preskace, provjerava se tip izuzetka(posle except) i ako on odgovara izvrsava se except klauzula...
- ako dodje do izuzetka koji nije naveden posle except, izuzetak se prosledjuje spoljnoj(outer) try klauzuli...ako ni tamo ne bude obradjen doci ce do prekida izvrsavanja programa(unhandled exception)

.....

try klauzula moze imati vise except klauzula, za razlicite tipove izuzetaka...takodje, moguce je u okviru jedne except klauzule obraditi vise tipova izuzetaka, npr:

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

U slucaju vise except klauzula, poslednja od njih moze izostaviti tip izuzetka pa se moze koristiti kao dzoker(wildcard)...ovo treba koristiti vrlo oprezno...

```
import string, sys

try:
```

```
    f = open('myfile.txt')
    s = f.readline()
    i = int(string.strip(s))
except IOError, (errno, strerror):
    print "I/O error(%s): %s" % (errno, strerror)
except ValueError:
    print "Could not convert data to an integer."
except:
    print "Unexpected error:", sys.exc_info()[0]
    raise
```

.....

try-except klauzula moze da ima i else dio...on se koristi kada je
potrebno da se uradi nesto onda kada se ne desi izuzetak i mora
da stoji nakon svih except klauzula:

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print 'cannot open', arg
    else:
        print arg, 'has', len(f.readlines()), 'lines'
        f.close()
```

.....

Naredba raise omogucava programeru da "namjerno" pozove neki izuzetak:

```
>>> raise NameError, 'HiThere'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: HiThere
```

prvi argument je tip izuzetka, a drugi je opcioni i on predstavlja
argument izuzetka...

Korisnicki definisani izuzeci
==========

Moguce je kreirati vlastiti tip izuzetka...to se radi definisanjem
klase izuzetka, koja nasledjuje osnovnu klasu Exception:

```
>>> class MyError(Exception):
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return repr(self.value)
...
```

```
>>> try:
...     raise MyError(2*2)
... except MyError, e:
...     print 'My exception occurred, value:', e.value
...
My exception occurred, value: 4
>>> raise MyError, 'oops!'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.MyError: 'oops!'
```

.....

try klauzula ima jos jednu opcionu klauzulu, koja se koristi onda kada je potrebno obaviti neke operacije bez obzira da li je doslo do izuzetka ili ne...npr:

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print 'Goodbye, world!'
...
Goodbye, world!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
KeyboardInterrupt
```

U slucaju kada se desi izuzetak, on se "poziva" nakon izvrsenja finally klauzule....

Finally se obicno koristi kada treba osloboditi neki objekat, bez obzira da li je njegova upotreba bila uspjesna ili ne...

```
#####################################################################
# KLASE I OBJEKTI
#####################################################################
```

Klasa - korisnicki definisan tip podataka...
Objekat - instanca(jedno pojavljivanje) neke klase...

Da bi se sto bolje razumjelo objektno programiranje u Pythonu, neophodno je objasniti sta je namespace a sta python scope...

Namespace predstavlja skup imena objekata neke cjeline...npr. skup ugradjenih imena funkcija(abs() i sl...), zatim globalne promjenljive u okviru nekog modula...ili lokalne promjenljive u okviru funkcije...
Bitno je znati da ne postoje relacija izmedju imena u razlucitim namespace-ovima...tako je moguce da u dva razlicita modula imamo dvije funkcije istog imena bez ikakve greske...kada zelimo da

koristimo neku od njih, navedemo prvo ime modula a onda njeno ime.

Scope(vidljivost,doseg) predstavlja dio python koda u kome se imenima nekog namespace-a moze pristupiti direktno, znaci bez reference...

Definicija klase:

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

Gdje statement-x obicno predstavlja funkciju ili atribut klase a zajedno se nazivaju clanice klase...
Kada se definise neka klasa, tada dolazi do svaranja novog namespace-a koji se odnosi na tu klasu i preko koga se pristupa njenim clanicama(metodama i atributima).
Posto je python potpuno objektno orjentisan jezik, svaka definicija klase u stvari presdtsvlja jedan objekat...tako da je moguce pristupati njegovim atributima, kao sto su __doc__ ili ako smo definisali:

```
class MyClass:
    "A simple example class"
    i = 12345
    def f(self):
        return 'hello world'
```

tada je moguce pristupiti MyClass.i ili
MyClass.f

Naravno, svrha definisanja klasa jeste mogucnost njihovog instanciranja, tj. pravljenja novih promjenljivih(objekata)...npr:

x = MyClass() cime smo napravili novu instancu(objekat x) klase MyClass

Primjetimo da mehanizam instanciranja funkcionise kao mehanizam pozivanja funkcija...

Ako je potrebno da novonastali objekat(instanca klase) ima neke "default" vrijednosti svojih atributa potrebno je u okviru klase definisati metodu __init__()

Ova metoda se poziva svaki put kada se instancira novi objekat...moze sadrzati i listu parametara ako je potrebna veca fleksibilnost pri stvaranju novih instanci(objekata)...npr:

```
>>> class Complex:
```

```
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

Kada instanciramo neki objekat, mozemo pristupati njegovim atributima i metodama... Ono sto je interesantno jeste da je moguce i definisati nove atribute u okviru postojeceg objekta, iako oni nisu definisani u okviru klase, bas kao sto se kreira neka globalna promjenljiva...znaci bez prethodne deklaracije...npr:

```
class MojaKlasa:            # definicija klase
    i = 3


x = MojaKlasa()             # definicija objekta te klase


x.brojac = 0                # kreiranje novog atributa

del x.brojac                # brisanje atributa
```

Jos neke sitnice koje su bitne kod klasa:

- nazivi atributa "override"-ju nazive metoda, tako da treba biti oprezan kod davanja naziva, te se drzati neke konvencije kako bi se moguce greske izbjegle.

- za razliku od nekih drugih jezika u Pythonu ne postoji "data hiding"...znaci nije moguce odvojiti koje su clanice klase private a koje public kao u Pascalu npr. Doduse to je moguce uraditi, ali pisanjem C koda, cime se inace python moze prosirivati.

- prilikom pozivanja metoda nekog objekta, kao prvi parametar toj metodi prosledjuje se ime objekta..iz tog razloga obicno se metode definisu tako sto im je prvi parametar promjenljiva self u koju se posle smjesta naziv objekta koji zove metodu...inace, rijec self nema nekog specijalnog znacenja za Python, to je cista konvencija...

- ako imamo definisanu neku funkciju...tu f-ju mozemo ukljuciti u definiciju nase nove klase vrlo jednostavno:

```
# Function defined outside the class
def f1(self, x, y):
      return min(x, x+y)

class C:
      f = f1
      def g(self):
```

```
        return 'hello world'
```
.....
naravno, ovo se ne koristi cesto...i moze dovesti do otezanog razumijevnja koda od strane vasih kolega..:)

- metode neke klase mogu "pozivati" druge metode preko self parametra, npr:

```
class Bag:
    def __init__(self):
        self.data = []
    def add(self, x):
        self.data.append(x)
    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

.....

Nasledjivanje
==========

Posto je Python objektno orjentisan jezik, normalno je da "podrzava" nasledjivanje klasa.
Bazna klasa - klasa koja se nasledjuje
Izvedena klasa(derivat) - nova klasa koja nasledjuje prethodnu

Sintaksa je sledeca:

```
class IzvedenaKlada(BaznaKlasa1, BaznaKlasa2, BaznaKlasaX):
    <statement-1>
    .
    .
    .
    <statement-N>
```
.....

umjesto bazne klase moze biti i neki izraz, sto je zgodno ako zelite naslijediti neku klasu iz npr. drugog modula: class IzvedenaKlasa(Modul.OsnovnaKlasa)

Izvedene klase mogu da "override"-ju metode bazne klase...znaci ako smo naslijedili klasu koja ima metodu stampaj()...u izvedenoj klasi ova metoda moze ponovo da se implementira na neki drugi nacin...(za C++ programere: sve metode python posmatra kao virtual)

Ako je u nekim situacijama potrebno napraviti neku strukturu kao sto je Record u Pascalu ili struct u C-u...za to mozete vrlo jednostavno iskoristiti obicnu klasu:

```
class Radnik:
    pass
```

```
TrenutniRadnik = Radnik() # Kreira prazan record

# Popunimo record nekim podacima
TrenutniRadnik.ime = 'Jontra'
TrenutniRadnik.plata = 1000


######################################################################
# KRAJ                                                               #
#                                                                    #
# Puno srece u daljem radu...                                        #
######################################################################
```