



Uvod u programski jezik Python



Programski jezik Python

- autor: Guido van Rossum, 1991.
- Monty Python's Flying Circus
- Python 2.6.2, travanj 2009.
- Python 3.0.1, veljača 2009.
 - značajne izmjene jezika – nekompatibilnost starijeg kôda



Literatura i Internet resursi

- *Python homepage* (<http://www.python.org/>)
- *službeni tutorial* (Guido van Rossum)
(<http://docs.python.org/tut/tut.html>)
- Mark Pilgrim, *Dive Into Python* (on-line book
<http://diveintopython.org/>)
- **Mark Lutz, *Learning Python*, 3rd Edition, O'Reilly, 2007**
- Mark Lutz, *Programming Python*, 3rd Edition, O'Reilly, 2006



Izvođenje Python programa

- instalacija Python interpretera
 - cygwin
 - Linux distribucije
 - Windows
 - IDLE – razvojna okolina (IDE)
 - `python.org`
- izvođenje naredbi/programa
 - interaktivno
 - kao *modul*
 - skripta iz komandne linije



Interaktivno izvođenje naredbi

- pokrenemo python interpreter i ukucavamo naredbu po naredbu
- rezultat se odmah ispisuje

```
$ python
>>> print 'Hello world!'
Hello world!
>>> print 2 ** 8
256
```

- napuštanje interpretera
ctrl-D (ili ctrl-Z)
- vrlo prikladno za eksperimentiranje s konstruktima jezika
- testiranje modula od kojih se grade složeniji programi



Programske datoteke

- programe želimo zapamtiti – pohraniti u (tekst) datoteke
→ *moduli*
- primjer: program `prvi.py`
`print 2 ** 8 # 2^8`
`tekst = 'the bright side '`
`print tekst + 'of life' # + je ulancavanje`
- pokretanje
`$ python prvi.py`
256
the bright side of life
- ekstenzija imena (`.py`) nije nužna za “prvu razinu” programa
- ako modul namjeravamo “uvoziti” (*import*), nastavak imena je potreban



Skripte

- ništa novo :-)

- primjer drugi

```
#!/usr/bin/python  
print 'leteci cirkus'
```

```
$ chmod +x drugi
```

```
$ ./drugi
```

```
leteci cirkus
```

- primjena naredbe `env` za postizanje neovisnosti o smještaju Python interpretera

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



Učitavanje modula

- svaka datoteka s Python naredbama, čije ime sadrži nastavak `.py` je *modul*
- program može učitati (*import*) neki modul, čime dobiva pristup njegovom sadržaju
- sadržaj modula se stavlja na raspolaganje okolini preko svojih *atributa*
- osnovna ideja programske arhitekture u Pythonu
 - veći programi se obično grade kao skup modula, koji učitavaju alate definirane u drugim modulima



Učitavanje modula (2)

- operacija učitavanja modula, između ostalog, pokreće izvršavanje naredbi tog modula
→ jedan od načina pokretanja programa

```
$ python  
>>> import prvi  
the bright side of life
```

- izvođenje naredbi modula obavlja se *samo pri prvom učitavanju* !
 - naknadno učitavanje neće učiniti ništa, čak i ako se kôd modula u međuvremenu promijenio !

```
>>> import prvi  
>>> import prvi
```

- učitavanje modula je “skupa” operacija
 - pronaći datoteku
 - prevesti u *byte-code* – (.pyc)
 - izvršiti naredbe



Ponovno učitavanje modula

- ako (doista) želimo ponovno učitati i izvršiti (eventualno izmijenjeni) modul, možemo to učiniti ugrađenom funkcijom `reload`

```
>>> reload(prvi)
promijenili smo tekst :-)
<module 'prvi' from 'prvi.py'>
```

- `reload` je funkcija, dok je `import` naredba
- samo se uspješno učitani modul može ponovo-učitati



Moduli i atributi

- namjena modula je organizacija biblioteka i alata
- općenito – modul predstavlja prostor imena (*namespace*)
 - imena unutar modula nazivaju se *atributi*
 - tipično – program koji učitava modul dobiva pristup svim imenima pridijeljenima u najvišoj razini modula
 - ta imena su najčešće pridijeljena *uslugama* koje modul nudi (*eksportira*): funkcije, klase, varijable, namijenjene korištenju iz drugih programa
 - pristup tim imenima postiže se naredbama `import` i `from`, ili pozivom funkcije `reload`



Moduli i atributi (2)

- primjer: modul `myfile.py`
`title = "The Meaning of Life"`
- dva načina pristupa atributima modula
- učitavanje cijelog modula, ime se *kvalificira* imenom modula

```
$ python  
>>> import myfile  
>>> print myfile.title # kvalificiranje  
The Meaning of Life
```
- sintaksa `objekt.atribut` omogućuje pristup atributima bilo kojeg *objekta* i uobičajena je u Pythonu



Moduli i atributi (3)

- drugi način pristupa imenima u modulu (u ovom slučaju radi se o kopiranju!) – korištenjem naredbe `from`

```
$ python
```

```
>>> from myfile import title # ime se kopira
```

```
>>> print title # ne treba kvalifikacija
```

```
The Meaning of Life
```

- `from` kopira attribute iz modula, tako da oni postaju varijable programa koji ih je učitao
- uočiti: imena modula se navode bez nastavka `.py` (Python automatski dodaje nastavak i traži datoteku)
- pri učitavanju izvršavaju se naredbe modula, a komponenta koja je modul učitala dobiva pristup imenima najviše razine modula (objekti poput funkcija i klasa)
→ ponovo iskoristive programske komponente



Moduli i atributi (4)

- obično modul definira više imena, raspoloživih za učitavanje u druge komponente – npr. `tri.py`

```
a = 'dead' # definira 3 atributa
b = 'parrot' # eksportiraju se
c = 'sketch'
print a, b, c # koriste se i u samom modulu
```

```
$ python tri.py
dead parrot sketch
```

- učitavanje modula

```
$ python
>>> import tri # cijeli modul
dead parrot sketch
>>>
>>> tri.b, tri.c # tuple
('parrot', 'sketch')
```



Moduli i atributi (5)

- učitavanje modula naredbom `from`

```
>>>  
>>> from tri import a, b, c # kopira  
>>> b, c  
( 'parrot', 'sketch' )
```

- ugrađena funkcija `dir` omogućuje dobivanje liste svih imena raspoloživih unutar modula

```
>>>  
>>> dir(tri)  
['__builtins__', '__doc__', '__file__',  
['__name__', 'a', 'b', 'c']
```

- kada se `dir` pozove s imenom učitanoog modula, vraća listu svih atributa unutar tog modula
- neka od imena su ugrađena (počinju i završavaju dvostrukom podvlakom) – Python ih unaprijed definira i imaju posebna značenja za interpreter



Struktura Python programa

- Python programi mogu se razložiti na module, izraze i objekte
 - programi se grade od modula
 - moduli sadrže naredbe
 - naredbe sadrže izraze
 - izrazi stvaraju i obrađuju objekte



Ugrađeni tipovi podataka

- Python nudi moćne tipove objekata kao sastavni dio jezika
 - ugrađeni tipovi omogućavaju lako pisanje jednostavnih programa
 - liste
 - rječnici
- obično su ugrađeni tipovi učinkovitiji od nadograđenih struktura podataka
 - koriste se optimirani algoritmi za baratanje strukturama podataka, koji su zbog brzine pisani u C-u



Ugrađeni tipovi podataka (2)

● tipovi podataka u Pythonu

Tip objekta	Primjer literala/stvaranja
Broj	<code>3.1415, 1234, 999L, 3+4j</code>
Znakovni niz	<code>'spam', "guido's"</code>
Lista	<code>[1, [2, 'three'], 4]</code>
Rječnik	<code>{'food': 'spam', 'taste': 'yum'}</code>
n-torka	<code>(1, 'spam', 4, 'U')</code>
datoteka	<code>text = open('eggs', 'r').read()</code>



Brojevi

● primjeri literala

Literal	Interpretacija
1234, -24, 0	Normal integers (C longs)
999999999999999999999999L	Long integers (unlimited size)
1.23, 3.14e-10, 4E210	Floating-point (C doubles)
0177, 0x9ff, 0xFF	Octal and hex literals
3+4j, 3.0+4.0j, 3J	Complex number literals

- cjelobrojni literal koji završava s `L` ili `l` je Python `long integer` – proizvoljne veličine
- od verzije 2.2 cijeli brojevi koji prekorače opseg automatski postaju *dugi* – oznaka `L` nije neophodna
- kompleksni brojevi – imaginarni dio završava s `J` ili `j`
- interno su to parovi `floating point` brojeva



Operatori

- tablica (prednost raste prema dnu tablice)

Operatori	Opis
<code>lambda args: expression</code>	neimenovana funkcija
<code>x or y</code>	y se evaluira ako je x false
<code>x and y</code>	y se evaluira ako je x true
<code>not x</code>	logička negacija
<code>x < y, x <= y, x > y, x >= y,</code> <code>x == y, x <> y, x != y,</code> <code>x is y, x is not y,</code> <code>x in y, x not in y</code>	usporedbe provjera identiteta pripadnost sekvenci



Operatori (2)

● tablica (nastavak)

Operatori	Opis
<code>x y</code>	Bitwise or
<code>x ^ y</code>	Bitwise exclusive or
<code>x & y</code>	Bitwise and
<code>x << y, x >> y</code>	Shift x left or right by y bits
<code>-x + y, x - y</code>	Addition/concatenation, subtraction
<code>x * y, x % y, x / y, x // y</code>	Multiplication/repetition, remainder/format, division
<code>-x, +x, ~x, x ** y</code>	..., bitwise complement; potencija
<code>x[i], x[i:j], x.attr, x(...)</code>	Indexing, slicing, qualification, function calls
<code>(...), [...], {...}, '...'</code>	Tuple, list, dictionary, conversion to string

● redoslijed primjene operatora

● grupiranje izraza zagradama



Miješanje tipova

- u izrazima u kojima se pojavljuju različiti tipovi podataka, Python se ponaša kao i drugi programski jezici – operandi se pretvaraju u najsloženiji tip

`40 + 3.14`

- složenost tipova u Pythonu

- integer
- long integer
- floating-point
- complex number

- od verzije 2.2 `integer` se pretvara u `long integer` ako vrijednost postane prevelika

- pretvaranja se obavljaju između *numeričkih* tipova

- općenito, Python ne obavlja pretvorbe između ostalih tipova
- npr. pribrajanje znakovnog niza cijelom broju generira pogrešku, osim ako “ručno” obavimo pretvorbu



Variable

- varijable se stvaraju kada im se prvi puta dodjeljuje vrijednost

```
$ python
```

```
>>> a = 3           # Name created
```

```
>>> b = 4
```

- u izrazima se varijable zamjenjuju njihovim vrijednostima
- da bismo varijablu mogli koristiti u izrazu, mora joj biti dodijeljena vrijednost
- varijable se odnose na konkretne objekte i nikada se ne deklariraju unaprijed



Numerički izrazi

● primjeri

```
>>> a = 3
```

```
>>> b = 4
```

```
>>> b * 3, b / 2 # (4*3), (4/2)
(12, 2)
```

```
>>> a % 2, b ** 2 # modulo, potencija
(1, 16)
```

```
>>> 2 + 4.0, 2.0 ** b # pretvorba tipova
(6.0, 16.0)
```

- u interaktivnom načinu rada rezultat izraza se automatski ispisuje
- dva izraza odvojena zarezom -> rezultat je uređeni par (općenito n-torka, *tuple*)



Numerički izrazi (2)

- ako se u izrazu koristi varijabla kojoj nije dodijeljena vrijednost, Python prijavljuje grešku, a ne koristi neku pretpostavljenu vrijednost

```
>>> c * 2
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in ?
```

```
NameError: name 'c' is not defined
```

- redoslijed primjene operatora i grupiranje zagradama

```
>>> b / 2 + a          # kao ((4 / 2) + 3)
```

```
5
```

```
>>> print b / (2.0 + a) # kao (4 / (2.0 + 3))
```

```
0.8
```



Prikaz brojeva

- upotreba naredbe `print` i automatski ispis

```
>>> b / (2.0 + a)      # Auto echo output: more digits
0.800000000000000004
```

```
>>> print b / (2.0 + a) # print rounds off digits
0.8
```

- automatski ispis prikazuje sve znamenke rezultata, u konkretnom primjeru razlog je pogreška u FP prikazu/aritmetici

```
>>> 1 / 2.0 # laki brojevi :-)
0.5
```

- naredba `print` zaokružuje brojeve koje ispisuje



Prikaz brojeva (2)

- upravljanje prikazom – upravljanje formatom znakovnih nizova

```
>>> num = 1 / 3.0
```

```
>>> num # Echoes
```

```
0.3333333333333333331
```

```
>>> print num # Print rounds
```

```
0.33333333333333
```

```
>>> "%e" % num # String formatting
```

```
'3.333333e-001'
```

```
>>> "%4.2f" % num # String formatting
```

```
'0.33'
```



Dijeljenje

- klasično dijeljenje – kad se dijele cijeli brojevi, necijeli dio rezultata se odbacuje; pri dijeljenju brojeva s pomičnim zarezom, nema odbacivanja necijelog dijela

X / Y

```
>>> (5 / 2), (5 / 2.0), (5 / -2.0), (5 / -2)
(2, 2.5, -2.5, -3)
```

- dijeljenje s “odsijecanjem” – necijeli dio rezultata se odbacuje, neovisno o tipu operandada

$X // Y$

```
>>> (5 // 2), (5 // 2.0), (5 // -2.0), (5 // -2)
(2, 2.0, -3.0, -3)
```

- razlog dodavanja – ovisnost rezultata o tipu operandada je nezgodan kod jezika s dinamičkim tipovima podataka !



Operacije nad bitovima

- *bitwise* operacije – posmak, I, ILI

```
>>> x = 1 # 0001
```

```
>>> x << 2 # Shift left 2 bits: 0100  
4
```

```
>>> x | 2 # bitwise OR: 0011
```

```
3 >>> x & 1 # bitwise AND: 0001  
1
```

- operacije nad bitovima nisu baš uobičajene kao u jezicima niže razine poput C-a

- "... if you find yourself wanting to flip bits in Python, you should think about which language you're really coding"

Dugi zapis cijelih brojeva

- u Python-u cijeli brojevi u dugom zapisu (oznaka \mathbb{L}) mogu biti *proizvoljne* duljine – ograničenje je jedino količina memorije

```
>>> 999999999999999999999999999999999999999999999L + 1  
1000000000000000000000000000000000000000000000L
```

- od verzije 2.2 oznaka \mathbb{L} nije nužna – automatska pretvorba
- ranije verzije – preljev

- mogućnost računanja s vrlo velikim brojevima, naravno aritmetika s dugim brojevima je puno sporija od obične

```
>>> 2 ** 200
1606938044258990275541962092341162602522202993782792835301376L
```

Kompleksni brojevi

- kompleksni brojevi su ugrađeni tip podataka u Pythonu
 - izvedeni kao parovi brojeva s pomičnim zarezom, pri čemu se imaginarni dio označava dodavanjem oznake \mathcal{J} ili j , realni i imaginarni dio povezani su oznakom $+$
 - npr. kompleksni broj čiji je realni dio 2 i imaginarni dio -3 zapisuje se $2 + -3j$

- primjeri

```
>>> 1j * 1J  
(-1+0j)
```

```
>>> 2 + 1j * 3  
(2+3j)
```

```
>>> (2+1j) * 3  
(6+3j)
```

- nad kompleksnim brojevima podržani su uobičajeni matematički izrazi, te dodatni alati sadržani u standardnom modulu `cmath`

Heksadekadska i oktalna notacija

- kao u C-u: oktalni literali započinu s 0, heksadekadski s 0x ili 0X

```
>>> 01, 010, 0x10, 0xFF
```

```
(1, 8, 16, 255)
```

- ispis je uobičajeno dekadski, ali postoje ugrađene funkcije za pretvorbu u hekso/oktalne *znakovne nizove*

```
>>> oct(64), hex(64), hex(255)
```

```
('0100', '0x40', '0xff')
```

- obrnuta pretvorba – iz znakovnog niza koji sadrži broj u broj – ugrađena funkcija `int`, drugi argument određuje brojevnu bazu

```
>>> int('0100'), int('0100', 8), int('0x40', 16)
```

```
(100, 64, 64)
```

- izrazi za formatiranje znakovnih nizova

```
>>> "%o %x %X" % (64, 63, 255)
```

```
'100 3f FF'
```




Ostali numerički alati

- Python sadrži ugrađene funkcije i ugrađene *module* za matematiku
- par primjera

```
>>> import math
```

```
>>> math.pi, math.e
```

```
(3.1415926535897931, 2.7182818284590451)
```

```
>>> math.sin(2 * math.pi / 180)
```

```
0.034899496702500969
```

```
>>> abs(-42), 2**4, pow(2, 4) # ugradjene f-je  
(42, 16, 16)
```

```
>>> int(2.567), round(2.567), round(2.567, 2)  
(2, 3.0, 2.5699999999999998)
```



Dinamičko upravljanje tipovima

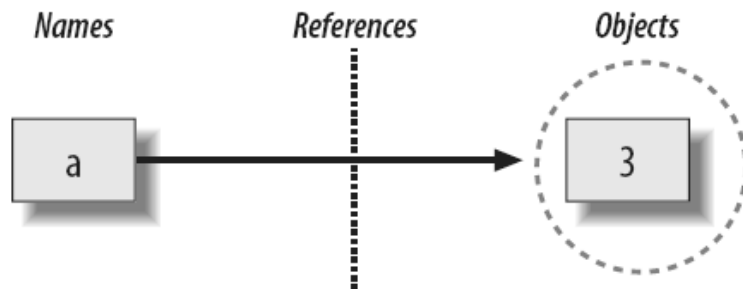
- upoznali smo Perl i ljusku, pa nam to više nije novo :-)
- tipovi podataka određuju se automatski tijekom izvođenja programa, a ne na temelju deklaracija u programskom kodu
 - varijabla se stvara kada joj se u programu prvi put dodijeli vrijednost
 - kasnije dodjele vrijednosti mijenjaju vrijednost pridruženu varijabli
 - varijable (*imena*) ne sadrže informaciju o tipu podatka – tip je pridružen *objektu* a ne imenu
→ varijabla naprosto pokazuje na neki objekt
 - kada se varijabla pojavljuje u izrazu, zamjenjuje se objektom na koji pokazuje
 - da bi mogla biti upotrijebljena u izrazu, varijabli *mora biti dodijeljena vrijednost* – inače greška



Variable

- varijable su *reference* na objekte

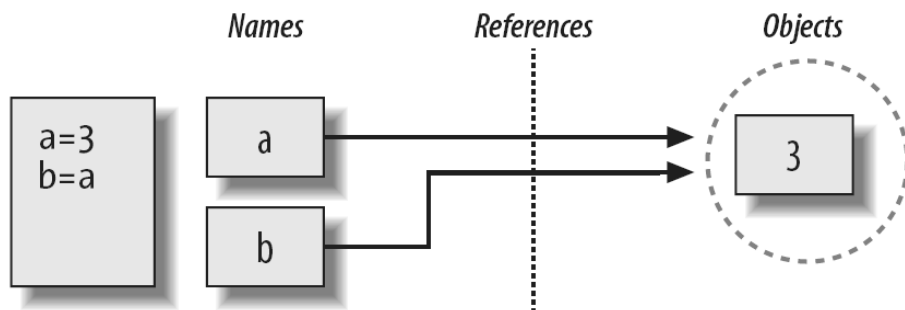
```
>>> a = 3
```



- što ako varijabli pridružimo vrijednost druge varijable ?
→ *dijeljene reference*

```
>>> a = 3
```

```
>>> b = a
```





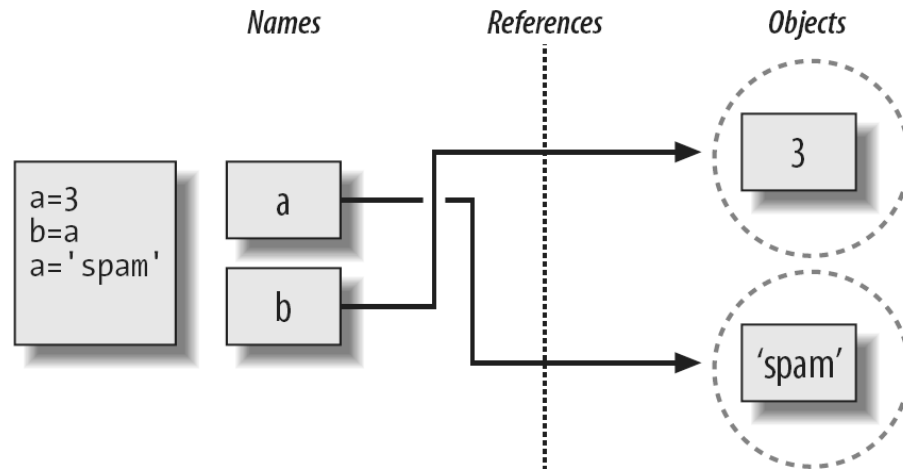
Variable (2)

- što ako nakon toga varijabli *a* dodijelimo novu vrijednost ?

```
>>> a = 3
```

```
>>> b = a
```

```
>>> a = 'spam'
```



- stvoren je novi objekt (znakovni niz), s kojim se povezuje varijabla *a*
- ne utječe se na varijablu *b* – ona i dalje pokazuje na isti objekt
- varijabla *a* pokazuje na objekt drugačijeg tipa
→ tip je svojstvo objekta, a ne imena (varijable)



Reference i izmjenjivi objekti

- slično kao u gornjem primjeru (samo bez različitog tipa objekta):

```
>>> a = 3
```

```
>>> b = a
```

```
>>> a = 5
```

- drugo dodjeljivanje vrijednosti varijabli *a* ne utječe na objekt na koji ona pokazuje, a time ni na vrijednost varijable *b*
- cijeli brojevi (kao ni znakovni nizovi) su objekti koji se ne mogu izmijeniti – *nepromjenljivost (immutability)*
- postoje vrste objekata i operacije nad njima koje uzrokuju izmjenu objekata “na licu mjesta”
 - npr. operacije nad listama uzrokuju promjene elemenata, a ne stvaranje nove liste
 - kod takvih objekata treba paziti na dijeljene reference, jer promjena korištenjem jednog imena utječe i na vrijednost na koju pokazuje drugo ime !



Reference i izmjenjivi objekti (2)

● primjer

```
>>> L1 = [2, 3, 4]
```

```
>>> L2 = L1
```

- varijable L1 i L2 pokazuju na istu listu
- ako varijabli L1 dodijelimo kao novu vrijednost cijeli broj, reference se razdvajaju, a lista ostaje netaknuta

```
>>> L1 = 24
```

- no, ako preko L1 dodijelimo novu vrijednost jednom elementu liste, lista se mijenja (i za L2)

```
>>> L1[0] = 24
```

```
>>> L2
```

```
[24, 3, 4]
```



Reference i prikupljanje otpada

- kada se varijabla poveže s novim objektom, Python provjerava referencira li “napušteni” objekt neka druga varijabla (ili neki drugi objekt) i preuzima objekte bez reference
→ *garbage collection*

- primjer

```
>>> x = 42
>>> x = 'shrubbery' # Reclaim 42 now (?)
>>> x = 3.1415      # Reclaim 'shrubbery' now (?)
>>> x = [1,2,3]     # Reclaim 3.1415 now (?)
```



Znakovni nizovi

- *string* – uređena kolekcija znakova
- Python ima dobru podršku za obradu znakovnih nizova
- Python nema posebnog tipa za pojedinačni znak
- znakovni nizovi svrstavaju se u nepromjenjive sekvence (*immutable sequences*)
 - znakovni niz ne može se mijenjati – možemo jedino stvoriti novi s promijenjenim sadržajem
 - sekvence su posebna kategorija podatkovnih tipova, operacije nad svim tipovima te kategorije su slične
- regularni izrazi (standardni modul)



Operacije nad znakovnim nizovima

Operacija	Opis
<code>s1 = ''</code>	prazni niz
<code>s2 = "spam's"</code>	dvostruki navodnici
<code>block = """..."""</code>	<i>blok</i> – može se protezati kroz više redaka
<code>s3 = r'\temp\spam'</code>	“sirovi” znakovni niz
<code>s4 = u'spam'</code>	Unicode Strings
<code>s1 + s2 ; s2 * 3</code>	ulančavanje, ponavljanje
<code>s2[i] ; s2[i:j] ; len(s2)</code>	indeksiranje, izrezivanje, duljina
<code>"a %s parrot" % 'dead'</code>	formatiranje
<code>s2.find('pa')</code> <code>s2.replace('pa', 'xx')</code> <code>s1.split()</code>	pozivanje metoda
<code>for x in s2</code>	iteriranje
<code>'m' in s2</code>	pripadnost



Znakovni nizovi – literali

- razni načini zapisivanja
 - jednostruki navodnici: `'spam'`
 - dvostruki navodnici: `"spam"`
 - trostruki navodnici:
`'''... spam ...''', """... spam ..."""`
 - *escape* sekvence: `"s\tp\na\0m"`
 - sirovi string: `r"C:\new\test.spm"`
 - Unicode string: `u'eggs\u0020spam'`
- jednostruki i dvostruki navodnici – ekvivalentni
→ oba oblika djeluju jednako i vraćaju isti tip objekta

```
>>> 'student', "student"  
( 'student', 'student')
```



Navodnici

- primjena različitih navodnika omogućuje gniježđenje navodnika (druge vrste) unutar znakovnog niza:

```
>>> 'knight"s', "knight's"  
( 'knight"s', "knight's")
```

- Python automatski nadovezuje susjedne znakovne nizove literale i bez eksplicitnog korištenja operatora nadovezivanja (+)

```
>>> title = "Meaning " 'of' " Life"  
>>> title  
'Meaning of Life'
```

- uključivanje navodnika u znakovni niz moguće je i predznačivanjem s \

```
>>> 'knight\'s', "knight\"s"  
("knight's", 'knight"s')
```



Escape sekvence

- predznačavanje s \ omogućuje umetanje specijalnih znakova u nizove (*escape sekvence*)

```
>>> s = 'a\nb\tc'
```

- interaktivni automatski ispis prikazuje specijalne znakove kao escape sekvence

```
>>> s
'a\nb\tc'
```

- ispis naredbom `print` interpretira specijalne znakove

```
>>> print s
a
b      c
```

- duljinu znakovnog niza možemo doznati primjenom funkcije `len`

```
>>> len(s)
5
```



Escape sekvence (2)

● tablica escape sekvenci

Escape	Značenje
\<novi_red>	ignorira se – nastavljanje retka
\\	Backslash
\'	jednostruki navodnik
\"	dvostruki navodnik
\a	Bell
\b	Backspace
\f	Formfeed
\n	Newline (linefeed)
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab



Escape sekvence (3)

● tablica escape sekvenci (nastavak)

Escape	Značenje
<code>\N{id}</code>	Unicode dbase id
<code>\uhhhh</code>	Unicode 16-bit hex
<code>\Uhhhh...</code>	Unicode 32-bit hex
<code>\xhh</code>	Hex digits value hh
<code>\ooo</code>	Octal digits value
<code>\0</code>	Null (doesn't end string)
<code>\other</code>	Not an escape (kept)



Binarni znakovi u nizovima

- u Pythonu *null-znak* ne predstavlja kraj niza (kao ni u Perlu) – Python vodi računa o duljini stringa

- znakovni nizovi mogu sadržavati proizvoljne binarne znakove

```
>>> s = '\001\002\x03'
```

```
>>> s
```

```
'\x01\x02\x03'
```

```
>>> len(s)
```

```
3
```

- mogućnost učitavanja i obrade binarnih podataka u znakovnim nizovima



Sirovi znakovni nizovi

- u “sirovim” znakovnim nizovima ne interpretiraju se escape sekvence

```
ne_valja = open('C:\new\text.dat', 'w')
```

- sirovi niz označava se slovom `r` ili `R`

```
bolje = open(r'C:\new\text.dat', 'w')
```

- naravno, može i običnim stringom

```
moze = open('C:\\new\\text.dat', 'w')
```

- Python koristi ovakav način ispisa `"\"` unutar stringova

```
>>> path = r'C:\new\text.dat'
```

```
>>> path # Show as Python code.
```

```
'C:\\new\\text.dat'
```

```
>>> print path # "User-friendly" format
```

```
C:\new\text.dat
```

```
>>> len(path)
```

```
15
```




Trostruki navodnici

- Python podržava zapisivanje znakovnih nizova koji se protežu kroz više redaka – navode se unutar trostrukih navodnika
- sintaksno – blok započinje s tri navodnika (jednostruka ili dvostruka), slijedi proizvoljan broj redaka teksta, a završetak bloka označava pojava tri navodnika (jednaka početnim)
- unutar bloka mogu se pojaviti i navodnici i ne moraju biti predznačeni s "\ "

```
>>> mantra = """Always look  
... on the bright  
... side of life."""
```

```
>>>
```

```
>>> mantra  
'Always look\n on the bright\nside of life.'
```

- Python prikuplja sve retke ovako definiranog bloka teksta i pakira ga u jedan niz, pri čemu se prelasci u novi red kodiraju kao "\n"



Operacije sa znakovnim nizovima

- duljina niza

```
$ python  
>>> len('abc')  
3
```

- ulančavanje

```
>>> 'abc' + 'def'  
'abcdef'
```

- ponavljanje

```
>>> 'bla ' * 4  
'bla bla bla bla '
```

```
>>> print '-'*80 # primjer korisne uporabe :-)
```

- ulančavanjem dva znakovna niza *stvara se novi* niz

- za razliku od C-a, nije potrebno alocirati memoriju i voditi računa o veličini polja u koje se niz pohranjuje



Operacije sa znakovnim nizovima (2)

- primjena "+" i "*" na znakovnim nizovima primjer je *preopterećenja operatora*

- paziti na miješanje tipova !

'abc'+9

izaziva pogrešku – broj se ne pretvara automatski u niz

- kroz nizove se može prolaziti (*iterirati*) u petlji, uz provjeru pripadnosti operatorom `in`

```
>>> myjob = "hacker"
```

```
>>> for c in myjob: print c, # Step through items
```

```
...
```

```
h a c k e r
```

```
>>> "k" in myjob
```

```
True
```

```
>>> "ack" in myjob # moze i podniz
```

```
True
```

```
>>> "z" in myjob # False = (nije nadjen)
```

```
False
```



Indeksiranje i izrezivanje

- znakovni niz u Pythonu je definiran kao *uređena kolekcije* znakova, svakom se znaku može pristupiti na temelju njegovog indeksa – rezultat operacije je *niz* duljine 1 znak
 - indeksi kreću od 0
 - negativni indeksi pribrajaju se *duljini niza*, što odgovara brojanju elemenata od kraja niza

```
>>> S = 'spam'
>>> S[0], S[-2] # Indexing from front or end
('s', 'a')
```

- izrezivanje (*slicing*) stvara novi znakovni niz koji sadrži podniz izvornog niza

```
>>> S[1:3], S[1:], S[: -1] # Slicing: extract section
('pa', 'pam', 'spa')

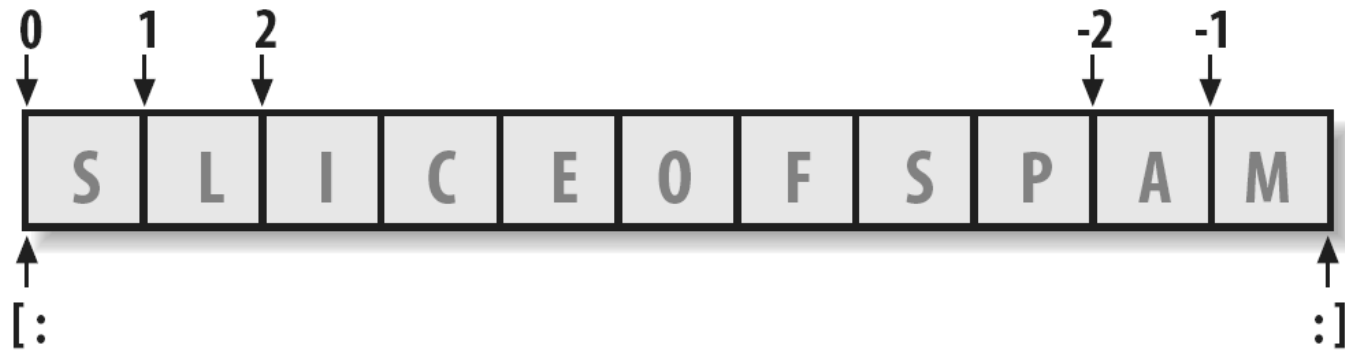
>>> kopija = S[:] # jednostavan način kopiranja
>>> kopija
'spam'
```

Izrezivanje

● ilustracija indeksa pri izrezivanju

[start:end]

Indexes refer to places the knife "cuts."



Defaults are beginning of sequence and end of sequence.

- pri operaciji izrezivanja navode se dva indeksa odvojena dvotočkom – prvi indeks označava prvi element koji se izrezuje, a drugi pokazuje *iza* zadnjeg elementa koji se izrezuje
 - ako se izostavi prvi indeks, podrazumijeva se 0
 - podrazumijevana vrijednost drugog indeksa je duljina niza
 - od verzije 2.3 podržava se i treći “indeks”, koji predstavlja *korak izrezivanja*



Pretvorbe znakovnih nizova

- Python ne dozvoljava zbrajanje broja sa znakovnim nizom, čak ni ako niz slični na broj
 - budući da operator "+" može označavati zbrajanje ili ulančavanje, izbor pretvorbe bio bi *dvosmislen*
- Python ima alate za pretvorbu znakovnih nizova u cijele brojeve i obrnuto

```
>>> int("42"), str(42) # Convert from/to string
(42, '42')
```

```
>>> int("42") + 1 # Force addition
43
```

```
>>> "spam" + str(42) # Force concatenation
'spam42'
```



Pretvorbe znakovnih nizova (2)

- slično – ugrađene funkcije za pretvorbu brojeva s pomičnom točkom

```
>>> str(3.1415), float("1.5")  
( '3.1415', 1.5)
```

```
>>> text = "1.234E-10"  
>>> float(text)  
1.2340000000000000001e-010
```



Izmjena znakovnih nizova

- znakovni nizovi su *nepromjenljive* (*immutable*) sekvence
→ ne mogu se mijenjati (npr. dodjeljivanjem vrijednosti nekom elementu preko indeksa)

```
>>> S = 'spam'
```

```
>>> S[0] = "x"
```

```
TypeError: object does not support item assignment
```

- kako bismo izmijenili znakovni niz, moramo kreirati novi niz s izmijenjenim sadržajem, uz eventualno pridjeljivanje imenu izvornog stringa

```
>>> S = S + 'SPAM!' # kreira se novi string!
```

```
>>> S
```

```
'spamSPAM!'
```

```
>>> S = S[:4] + 'Burger' + S[-1]
```

```
>>> S
```

```
'spamBurger!'
```




Formatiranje znakovnih nizova

- Python preopterećuje binarni operator `"%"` – kada se primjeni na znakovne nizove, obavlja ulogu poput funkcije `sprintf` u C-u
 - jednostavno formatiranje vrijednosti u znakovne nizove
 - lijevo od operatora `"%"` navodi se formatni niz u kojem se definiraju pretvorbe podataka
 - na desnoj strani se navodi objekt, ili više objekata u zagradama (*n-torka*), koje želimo umetnuti u konačni znakovni niz
 - tehnički gledano, isti rezultat možemo ostvariti ulančavanjima i pretvorbama, no formatiranje omogućava obavljanje većeg broja koraka u jednu operaciju

```
>>> ime = "Fiona"
>>> "The knight loves %s." % ime

>>> "%d %s %d you" % (1, 'spam', 4)
'1 spam 4 you'
```



Formatiranje znakovnih nizova (2)

- svaki tip objekta može se pretvoriti u string

```
>>> "%s -- %s -- %s" % (42, 3.14159, [1, 2, 3])  
'42 -- 3.14159 -- [1, 2, 3]'
```

- za formatiranje raznih tipova podataka na raspolaganju je cijeli niz formatskih kodova – podržani su svi formati koji se koriste u C funkciji `printf`
- tablica formatskih kôdova

Kôd	Opis
<code>%s</code>	znakovni niz, ili bilo koji objekt
<code>%r</code>	kao <code>%s</code> , ali koristi <code>repr()</code> , a ne <code>str()</code>
<code>%c</code>	znak
<code>%d</code>	dekadski cijeli broj



Formatiranje znakovnih nizova (3)

● tablica formatskih kôdova (nastavak)

Kôd	Opis
%i	cijeli broj
%u	cijeli broj bez predznaka
%o	oktalni cijeli broj
%x	heksadekadski cijeli broj
%X	heksadekadski cijeli broj, zapis velikim slovima
%e	eksponencijalni zapis broja s pomičnom točkom
%E	kao %e ali veliki E
%f	floating-point decimal
%g	floating-point %e ili %f
%G	floating-point %E ili %f
%%	doslovno "%"

Formatiranje znakovnih nizova (4)

- kodovi za pretvorbu podržavaju detaljnije specificiranje formata
- opća struktura je:

`%[(name)][flags][width][.precision]code`

- između znaka `%` i koda pretvorbe mogu se navesti: ključ rječnika, lista zastavica (npr. `-` označava lijevo poravnavanje, `+` označava predznak, `0` popunjavanje vodećim nulama), ukupna širina polja, te broj znamenaka iza decimalne točke

```
>>> x = 1234
>>> res = "integers: ...%d...%-6d...%06d" % (x, x, x)
>>> res
'integers: ...1234...1234 ...001234'
```

```
>>> x = 1.23456789
>>> x
1.2345678899999999
```



Formatiranje znakovnih nizova (5)

- brojevi s pomičnom točkom mogu se formatirati na različite načine

```
>>> x = 1.23456789
```

```
>>> '%e | %f | %g' % (x, x, x)
```

```
'1.234568e+000 | 1.234568 | 1.23457'
```

```
>>> '%-6.2f | %05.2f | %+06.1f' % (x, x, x)
```

```
'1.23 | 01.23 | +001.2'
```

```
>>> "%s" % x, str(x)
```

```
('1.23456789', '1.23456789')
```

Formatiranje znakovnih nizova (6)

- formatiranje nizova omogućuje i referenciranje elemenata rječnika navedenog na desnoj strani, putem ključa

```
>>> "%(n)d %(x)s" % {"n":1, "x":"spam"}  
'1 spam'
```

- ovaj trik se često koristi zajedno s ugrađenom funkcijom `vars`, koja vraća rječnik koji sadrži sve varijable koje *postoje* na mjestu na kojem je funkcija pozvana

```
>>> food = 'spam'  
>>> age = 40  
>>> vars( )  
{ 'food': 'spam', 'age': 40, ...i ostale... }
```

- ovako se iz formatnog stringa može referencirati varijable imenom (ključ rječnika)

```
>>> "%(age)d %(food)s" % vars( )  
'40 spam'
```



Metode znakovnih nizova

- znakovni niz kao objekt ima skup metoda (funkcija koje su atributi objekta), koje implementiraju razne zadatke obrade teksta
 - fukcije su *paketi* kôda, a poziv metode objedinjuje dvije operacije: dohvat atributa i poziv funkcije
 - izraz oblika `objekt.atribut` dohvaća vrijednost atributa u objektu
 - izraz oblika `funkcija(argumenti)` poziva kod funkcije, pri čemu se prosljeđuje nula ili više objekata-argumenata odvojenih zarezima, i vraća se povratna vrijednost funkcije
 - izraz oblika `objekt.funkcija(argumenti)` poziva metodu



Metode znakovnih nizova (2)

- znakovni nizovi su nepromjenljivi – nužno je stvoriti novi string

```
>>> S = 'spammy'
>>> S = S[:3] + 'xx' + S[5:]
>>> S
'spaxxy'
```

- zamjena podniza u znakovnom nizu – metoda `replace`

```
>>> S = 'spammy'
>>> S = S.replace('mm', 'xx')
>>> S
'spaxxy'
```

- metoda `replace` obavlja *globalnu* zamjenu

```
>>> 'aa$bb$cc$dd'.replace('$', 'SPAM')
'aaSPAMbbSPAMccSPAMdd'
```


Metode znakovnih nizova (2)

- zamjena prve pojave podniza

```
>>> S = 'xxxxSPAMxxxxSPAMxxxx'
>>> where = S.find('SPAM') # Search for position
>>> where # Occurs at offset 4
4
>>> S = S[:where] + 'EGGS' + S[(where+4):]
>>> S
'xxxxEGGSxxxxSPAMxxxx'
```

- metoda `find` vraća indeks na kojem započinje traženi podniz, ili `-1` ako podniz nije pronađen

- jednostruka zamjena može se postići metodom `replace`, uz dodavanje trećeg argumenta (broj zamjena)

```
>>> S.replace('SPAM', 'EGGS', 1) # Replace one
'xxxxEGGSxxxxSPAMxxxx'
```

- još jednom: *stvara se novi string!*

Metode znakovnih nizova (3)

- ako se primjenjuje veći broj uzastopnih operacija nad dugačkim znakovnim nizovima, nezgodno je što svaka operacija generira *novi* niz
 - ponekad se isplati pretvoriti string u izmjenljivi objekt – *listu*, kako bi se operacije obavljale na licu mjesta
 - ugrađena funkcija `list` (zapravo konstruktor liste) – formira novu listu na temelju objekta koji joj se predaje kao argument

```
>>> S = 'spammy'
```

```
>>> L = list(S)
```

```
>>> L
```

```
['s', 'p', 'a', 'm', 'm', 'y']
```

```
>>> L[3] = 'x' # radi s listama, ne sa stringovima
```

```
>>> L[4] = 'x'
```

```
>>> L
```

```
['s', 'p', 'a', 'x', 'x', 'y']
```



Metode znakovnih nizova (4)

- nakon obavljenih izmjena, listu možemo “spakirati” nazad u string primjenom metode `join`

```
>>> S = ''.join(L) # "uvlaci" listu u string
>>> S
'spaxxy'
```

- `join` je metoda znakovnih nizova, a poziva se preko željenog graničnika

- znakovni nizovi iz liste povezuju se u jedan niz, pri čemu se između njih umeće niz koji je naveden kao graničnik

```
>>> 'SPAM'.join(['eggs', 'sausage', 'ham', 'toast'])
'eggsSPAMsausageSPAMhamSPAMtoast'
```



Metode znakovnih nizova (5)

- ako su podaci u znakovnom nizu poznate širine polja, lako ih je izrezati

```
>>> line = 'aaa bbb ccc'
>>> podatak1 = line[0:3]
>>> podatak3 = line[8:]
```

- ako su polja odvojena graničnikom, primjenjuje se metoda `split` – rezultat je lista podnizova

- ako se ne proslijedi argument, dijeljenje se obavlja ne prazninama (jedan ili više praznih znakova, tabova, prelezaka u novi red)

```
>>> line = 'aaa bbb ccc'
>>> cols = line.split( )
>>> cols
['aaa', 'bbb', 'ccc']
```



Metode znakovnih nizova (6)

- metoda `split`

- ako se kao argument navede znakovni niz, on se koristi kao graničnik

```
>>> line = 'bob,hacker,40'  
>>> line.split(',')  
['bob', 'hacker', '40']
```

```
>>> line = "i'mSPAMaSPAMlumberjack"  
>>> line.split("SPAM")  
["i'm", 'a', 'lumberjack']
```

- izrezivanje i `split` rade vrlo brzo, obavljaju osnovnu ekstrakciju teksta
- mnoštvo drugih metoda nad stringovima
- metode *ne prihvataju* regularne izraze
 - koristi se modul `re` iz standardne biblioteke



Liste

- liste su fleksibilne uređene kolekcije
- za razliku od stringova, liste mogu sadržavati bilo koju vrstu objekata, pa i druge liste
 - definirano je uređenje, pristup elementu liste putem indeksa
 - podržano je izrezivanje i nadovezivanje
 - duljina se može mijenjati (umetanje i brisanje objekata)
 - elementi su proizvoljni objekti – heterogenost
 - proizvoljno gniježđenje
 - liste u Pythonu sadrže 0 ili više *referenci* na objekte
 - slično polju pokazivača u C-u
 - dohvat elementa Python liste je brzo gotovo kao indeksiranje polja u C-u
 - zapravo su liste i ostvarene kao C polja unutar Python interpretera
 - kada se umeće objekt u listu, pohranjuje se referenca, a ne kopija objekta



Operacije nad listama

- tablica – liste: literali i (neke) operacije

Operacija	Opis
<code>L1 = []</code>	prazna lista
<code>L2 = [0, 1, 2, 3]</code>	četiri elementa, indeksi 0..3
<code>L3 = ['abc', ['def', 'ghi']]</code>	ugniježdena podlista
<code>L2[i] ; L3[i][j]</code>	indeksiranje
<code>L2[i:j] ; len(L2)</code>	izrezivanje, određivanje duljine
<code>L1 + L2 ; L2 * 3</code>	ulančavanje, ponavljanje
<code>for x in L2 ; 3 in L2</code>	iteriranje, pripadnost
<code>L2.append(4) ; L2.extend([5,6,7])</code>	metode: dodavanje elemenata
<code>L2.sort() ; L2.index(1) ; L2.reverse()</code>	sortiranje, pretraživanje,...
<code>del L2[k] ; del L2[i:j] ; L2.pop() ; L2[i:j] = []</code>	brisanje
<code>L2[i] = 1 ; L2[i:j] = [4,5,6]</code>	pridruživanje vrijednosti



Operacije nad listama (2)

- na liste se primjenjuju operacije nadovezivanja (" + ") i ponavljanja (" * "), slično kao kod znakovnih nizova (općenito – operacije nad sekvencama)

- rezultat je nova lista

```
>>> len([1, 2, 3])  
3
```

```
>>> [1, 2, 3] + [4, 5, 6]  
[1, 2, 3, 4, 5, 6]
```

```
>>> ['Ni!'] * 4  
['Ni!', 'Ni!', 'Ni!', 'Ni!']
```

```
>>> 3 in [1, 2, 3] # pripadnost (True)  
True
```

```
>>> for x in [1, 2, 3]: print x, # Iteration  
...  
1 2 3
```




Operacije nad listama (3)

- iako ulančavanja djeluje na liste i na stringove, očekuje se isti tip sekvence na obje strane operatora – inače greška
 - ne možemo ulančavati listu sa znakovnim nizom, osim ako najprije pretvorimo listu u znakovni niz (pomoću obrnutih navodnika, funkcije `str` ili `%` formatiranja); ili znakovni niz u listu

```
>>> L=[1, 2]
```

```
>>> 'L' + "34" # kao "[1, 2]" + "34"  
'[1, 2]34'
```

```
>>> [1, 2] + list("34") # kao [1, 2] + ["3", "4"]  
[1, 2, '3', '4']
```



Indeksiranje i izrezivanje

- indeksiranje i izrezivanje radi na listama kao i na znakovnim nizovima
 - indeksiranje vraća (kopiju) objekta koji se nalazi u listi na mjestu na koje indeks pokazuje
 - izrezivanje vraća novu listu

```
>>> L = ['spam', 'Spam', 'SPAM!']
```

```
>>> L[2] # indeksi krecu od 0  
'SPAM!'
```

```
>>> L[-2] # negativni indeks - odbrojavanje od kraja  
'Spam'
```

```
>>> L[1:]  
['Spam', 'SPAM!']
```



Matrice

- jedan od načina za prikaz matrica (višedimenzionalnih polja) u Pythonu je gniježđenjem listi – npr. matrica 3×3

```
>>> matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

- primjenom jednog indeksa dobivamo cijeli redak matrice (ugniježdenu listu), s dva indeksa dohvaćamo jedan element

```
>>> matrix[1]
```

```
[4, 5, 6]
```

```
>>> matrix[1][1]
```

```
5
```

```
>>> matrix[2][0]
```

```
7
```

```
>>> matrix = [[1, 2, 3],
```

```
...           [4, 5, 6],
```

```
...           [7, 8, 9]]
```

```
>>> matrix[1][1]
```

```
5
```



Izmjena liste

- liste su izmjenljivi objekti – mogu se mijenjati na licu mjesta, bez stvaranja kopije
- kako Python radi s referencama, izmjena objekta utječe i na druge reference na taj objekt !
- sadržaj liste može se izmijeniti pridjeljivanjem vrijednosti pojedinom elementu, ili cijelom isječku (*slice*)

```
>>> L = ['spam', 'Spam', 'SPAM!']
>>> L[1] = 'eggs' # Index assignment
>>> L
['spam', 'eggs', 'SPAM!']
```

```
>>> L[0:2] = ['eat', 'more'] # Slice assignment
>>> L
['eat', 'more', 'SPAM!']
```

Izmjena liste (2)

- pridjeljivanje vrijednosti isječku ponaša se *kao* brisanje i umetanje (broj elemenata koji se umeće ne mora odgovarati broju elemenata koji se brišu)

```
>>> L = [1, 2, 3]
```

```
>>> L[1:2]=[4,5]
```

```
>>> L
```

```
[1, 4, 5, 3]
```

```
>>> L[1:3]=[ ] # brisanje elemenata
```

```
>>> L
```

```
[1, 3]
```

- funkcioniра i u slučaju preklapanja isječka s vrijednostima i isječka kojem se dodjeljuju vrijednosti

```
>>> L = [1, 2, 3, 4, 5, 6, 7, 8]
```

```
>>> L[2:5]=L[3:6]
```

```
>>> L
```

```
[1, 2, 4, 5, 6, 6, 7, 8]
```

Metode listi

- liste imaju specifične *metode*

- primjeri

```
>>> L.append('please') # dodavanje jednog elementa
>>> L
['eat', 'more', 'SPAM!', 'please']
>>> L.sort( ) # sortiranje ('S' < 'e').
>>> L
['SPAM!', 'eat', 'more', 'please']
```

- metoda `append` prima samo jedan objekt, koji se nadodaje na kraj liste
- efekt izraza `L.append(X)` je sličan kao `L+[X]`, ali drugi izraz stvara *novu* listu – `append` je obično brže
- obje metode mijenjaju listu na licu mjesta, ali *ne vraćaju listu kao rezultat*
`L=L.append(X)` će *obrisati* referencu na listu !



Metode listi (2)

● još primjera

```
>>> L = [1, 2]
>>> L.extend([3,4,5]) # dodavanje liste elemenata
>>> L
[1, 2, 3, 4, 5]

>>> L.pop( ) # brise i vraca zadnji element
5
>>> L
[1, 2, 3, 4]

>>> L.reverse( ) # promjena redoslijeda
>>> L
[4, 3, 2, 1]
```



Metode listi (3)

- korištenjem metoda `pop` i `append` lako je implementirati brzu stogovnu strukturu (kraj liste je vrh stoga)

```
>>> L = [ ]
>>> L.append(1) # Push
>>> L.append(2)
>>> L
[1, 2]
>>> L.pop( ) # Pop
2
>>> L
[1]
```

- `pop` se može koristiti i s indeksom – briše se proizvoljni element

```
>>> L = [1, 2, 3, 4, 5]
>>> L.pop(3)
4
>>> L
[1, 2, 3, 5]
```


Metode listi (4)

- za brisanje elementa ili dijela liste može se primijeniti naredba `del`

```
>>> L
['SPAM!', 'eat', 'more', 'please']
>>> del L[0] # brisanje jednog elementa
>>> L
['eat', 'more', 'please']

>>> del L[1:] # brisanje isječka
>>> L
['eat']
```

- dijelovi liste mogu se obrisati i operacijom pridruživanja isječku

```
>>> L = ['Already', 'got', 'one']
>>> L[1:] = [ ]
>>> L
['Already']
```

- pridjeljivanje prazne liste indeksiranjem pohranjuje u indeksirani element referencu na praznu listu !



Rječnici

- uz liste, najfleksibilniji ugrađeni tip u Pythonu
- neuređena kolekcija objekata
- elementi se dohvaćaju i pohranjuju na temelju *ključa*
 - indeksiranje u rječniku je vrlo brza operacija pretraživanja
 - asocijativno polje (*hash*)
 - elementi nisu složeni u određenom redosljedu, ključevi određuju simboličko mjesto elemenata u rječniku
 - Python koristi optimirane *hashing* algoritme, pa su dohvati vrlo brzi
- duljina rječnika je varijabilna, elementi su proizvoljnih tipova
- mogu sadržavati liste i rječnike (gniježđenje proizvoljne dubine)
- poput lista – izmjenljivi objekti
 - kako nisu uređeni, ne podupiru operacije nad sekvencama
 - rječnici sadrže *reference* na objekte, kao i liste



Operacije nad rječnicima

- tablica – rječnici: literali i (neke) operacije

Operacija	Opis
<code>D1 = { }</code>	prazni rječnik
<code>D2 = {'spam': 2, 'eggs': 3}</code>	rječnik s dva elementa
<code>D3 = {'food': {'ham': 1, 'egg': 2}}</code>	gniježđenje
<code>D2['eggs'] ; d3['food']['ham']</code>	indeksiranje ključem
<code>D2.has_key('eggs'); 'eggs' in D2</code> <code>D2.keys() ; D2.values()</code> <code>D2.copy() ; D2.get(key, default)</code>	metode: pripadnost ključevi, vrijednosti kopiranje ...
<code>len(d1)</code>	duljina (ukupni broj elemenata)
<code>D2[key] = 42 ; del d2[key]</code>	dodavanje/izmjena; brisanje



Operacije nad rječnicima (2)

- stvaranje rječnika i pristup elementima (vrijednostima)

```
>>> d2 = {'spam': 2, 'ham': 1, 'eggs': 3}
>>> d2['spam'] # dohvat vrijednosti na temelju ključa
2
>>> d2 # redoslijed je nepredvidljiv
{'eggs': 3, 'ham': 1, 'spam': 2}
```

- redoslijed elemenata je različit od onog kojim su elementi uneseni – zbog implementacije brzog dohvata (*hashing*)

- operacije nad sekvencama (izrezivanje, nadovezivanje) ne mogu se primjenjivati na rječnicima

- ugrađena funkcija `len()` primjenjiva je na rječnike – vraća broj pohranjenih elemenata

```
>>> len(d2)
3
```



Metode rječnika

- metoda `has_key()` omogućuje ispitivanje postojanja zadanog ključa

```
>>> d2.has_key('ham') # postojanje ključa
True
```

```
>>> 'ham' in d2 # drugi način - pripadnost
True
```

- metoda `keys()` vraća listu svih ključeva rječnika

```
>>> d2.keys()
['eggs', 'ham', 'spam']
```



Izmjene rječnika

- rječnici su izmjenljivi (*mutable*) objekti
- mogu se mijenjati, proširivati ili smanjivati bez stvaranja kopije

```
>>> d2['ham'] = ['grill', 'bake', 'fry'] # izmjena elementa
```

```
>>> d2
```

```
{'eggs': 3, 'spam': 2, 'ham': ['grill', 'bake', 'fry']}
```

```
>>> del d2['eggs'] # brisanje elementa
```

```
>>> d2
```

```
{'spam': 2, 'ham': ['grill', 'bake', 'fry']}
```

```
>>> d2['brunch'] = 'Bacon' # dodavanje elementa
```

```
>>> d2
```

```
{'brunch': 'Bacon', 'spam': 2, 'ham': ['grill', 'bake', 'fry']}
```



Izmjene rječnika (2)

- kao i kod listi, dodjela vrijednosti preko postojećeg ključa mijenja vrijednost elementa rječnika
- ako dodjeljujemo vrijednost za nepostojeći ključ, stvara se novi element rječnika
 - *kod liste pokušaj dodjele vrijednosti preko nepostojećeg indeksa uzrokuje grešku*
 - dodavanje novog elementa u listu moguće je pomoću metode `append()` ili dodjelom vrijednosti isječku



Još neke metode

- rječnici imaju različite korisne metode
- metoda `values()` vraća listu vrijednosti elemenata rječnika

```
>>> d2.values()
[3, 1, 2]
```
- metoda `items()` vraća listu parova (ključ, vrijednost)

```
>>> d2.items()
[('eggs', 3), ('ham', 1), ('spam', 2)]
```
- pokušaj dohvata preko nepostojećeg ključa rezultira greškom, no može se koristiti metoda `get()` koja u tom slučaju vraća pretpostavljenu vrijednost (`None` ili pretpostavljenu vrijednost koja se proslijeđuje kao argument)

```
>>> d2.get('ham'), d2.get('salt'), d2.get('salt', 8)
(1, None, 8)
```


Još neke metode (2)

- metoda `update()` nudi funkcionalnost sličnu ulančavanju – ključevi i vrijednosti jednog rječnika spajaju se s drugim rječnikom
 - vrijednosti s istim ključem se pritom *prepisuju*

```
>>> d2 = {'eggs':3, 'ham':1, 'spam':2}
>>> d3 = {'toast':4, 'muffin':5, 'eggs':7}
>>> d2.update(d3)
>>> d2
{'toast': 4, 'ham': 1, 'spam': 2, 'muffin': 5,
'eggs': 7}
```

- metoda `pop()` briše element iz rječnika i vraća njegovu vrijednost

```
>>> d2 = {'toast':4, 'eggs':3, 'ham':1, 'spam':2}
>>> d2.pop('toast')
4
>>> d2
{'eggs': 3, 'ham': 1, 'spam': 2}
```



Primjer

● primjer – podaci o autorima skriptnih jezika

```
>>> table = {'Python': 'Guido van Rossum',  
...         'Perl': 'Larry Wall',  
...         'Tcl': 'John Ousterhout' }  
>>> language = 'Python'  
>>> creator = table[language]  
>>> creator  
'Guido van Rossum'  
  
>>> for lang in table.keys( ):  
...     print lang, '\t', table[lang]  
...  
Tcl      John Ousterhout  
Python   Guido van Rossum  
Perl     Larry Wall
```



Rječnik kao fleksibilna "lista"

- ključ ne mora biti string – može biti bilo koji nepromjenljivi tip, npr. cijeli broj
- pridruživanje vrijednosti nepostojećem elementu liste nije dozvoljeno

```
>>> L = [ ]
```

```
>>> L[99] = 'spam'
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in ?
```

```
IndexError: list assignment index out of range
```

- mogli bismo alocirati veliku listu primjenom operatora ponavljanja: npr. `L = [0]*100`, ali nešto slično možemo postići rječnikom



Rječnik kao fleksibilna "lista" (2)

- korištenjem cjelobrojnog ključa možemo oponašati listu koja raste prema potrebi

```
>>> D = { }  
>>> D[99] = 'spam'  
>>> D[99]  
'spam'  
>>> D  
{99: 'spam'}
```

- izgleda kao lista od 100 elemenata, a zapravo je rječnik s jednim elementom



Pohrana rijetkih podatkovnih struktura

- rječnik je prikladan za pohranu rijetkih matrica – samo mali broj elemenata ima vrijednost

```
>>> Matrix = { }
```

```
>>> Matrix[(2,3,4)] = 88
```

```
>>> Matrix[(7,8,9)] = 99
```

```
>>> X = 2; Y = 3; Z = 4 # ; odvajanje naredbe
```

```
>>> Matrix[(X,Y,Z)]
```

```
88
```

```
>>> Matrix
```

```
{(2, 3, 4): 88, (7, 8, 9): 99}
```

- koristimo rječnik za prikaz trodimenzionalnog polja
- ključevi su trojke (*tuples*), čiji su elementi koordinate elementa
- samo su dva elementa popunjena, nije nam potrebna velika 3D matrica
- pristup nedefiniranim "elementima" uzrokuje pogrešku (nepostojeći ključ)



Pohrana rijetkih podatkovnih struktura (2)

- pristup nedefiniranim "elementima" uzrokuje pogrešku (nepostojeći ključ) – više rješenja

- provjera postojanja ključa

```
if Matrix.has_key((2,3,6)): # provjera ključa
    print Matrix[(2,3,6)]
else:
    print 0
```

- korištenjem naredbe `try` (hvatanje iznimke)

```
try:
    print Matrix[(2,3,6)] # Try to index
except KeyError:         # Catch and recover
    print 0
```

- najjednostavnije – metoda `get()`

```
>>> Matrix.get((2,3,4), 0) # postoji
88
>>> Matrix.get((2,3,6), 0) # ne postoji
0
```



n-torke

- *n*-torka (*tuple*) je uređena kolekcija (kao i lista) proizvoljnih objekata, koja *nije izmjenljiva*
 - ponaša se jednako kao i lista, jedino se ne može mijenjati na licu mjesta
 - ne podržava pozive metoda
- zašto uopće i liste i *n*-torke ?
 - nepromjenljivost daje određenu dozu *integriteta* – možemo biti sigurni da *n*-torka neće biti promijenjena preko neke druge reference u programu
 - *n*-torke se mogu koristiti u nekim situacijama u kojima liste ne mogu – npr. kao ključevi rječnika



Uobičajene operacije

- tablica – n-torke: literali i uobičajene operacije

Operacija	Opis
<code>()</code>	prazna n-torka
<code>t1 = (0,)</code>	jednočlana n-torka (!)
<code>t2 = (0, 'Ni', 1.2, 3)</code>	četveročlana n-torka
<code>t2 = 0, 'Ni', 1.2, 3</code>	može i ovako
<code>t3 = ('abc', ('def', 'ghi'))</code>	ugniježdene n-torke
<code>t1[i] ; t3[i][j]</code>	indeksiranje
<code>t1[i:j] ; len(t1)</code>	izrezivanje, duljina
<code>t1 + t2 ; t2 * 3</code>	ulančavanje, ponavljanje
<code>for x in t2 ; 3 in t2</code>	iteracija, pripadnost



Operacije s n-torkama

● standardne operacije

```
>>> (1, 2) + (3, 4) # nadovezivanje  
(1, 2, 3, 4)
```

```
>>> (1, 2) * 4 # ponavljanje  
(1, 2, 1, 2, 1, 2, 1, 2)
```

```
>>> T = (1, 2, 3, 4)  
>>> T[0], T[1:3] # indexing, slicing  
(1, (2, 3))
```

● n-torku s jednim elementom ne možemo napisati kao $t = (1)$ jer će to Python shvatiti kao izraz, tj. kao $t = 1$

● zato jednočlanu n-torku zapisujemo s dodanim zarezom

```
>>> y = (40, )  
>>> y  
(40, )
```

Operacije s n-torkama (2)

- n-torke ne podržavaju metode !
- npr. ako hoćemo sortirati n-torku, obično ćemo je najprije pretvoriti u listu

```
>>> T = ('cc', 'aa', 'dd', 'bb')
```

```
>>> tmp = list(T)
```

```
>>> tmp.sort( )
```

```
>>> tmp
```

```
['aa', 'bb', 'cc', 'dd']
```

```
>>> T = tuple(tmp)
```

```
>>> T
```

```
('aa', 'bb', 'cc', 'dd')
```

- ugrađene funkcije za pretvorbu `list()` i `tuple()`
- obje funkcije vraćaju nove objekte



Operacije s n-torkama (3)

- pravilo o nepromjenljivosti n-torki odnosi se samo na najvišu razinu same n-torke
 - n-torka može sadržavati druge, izmjenljive objekte, npr. liste

```
>>> T = (1, [2, 3], 4)
>>> T[1][0] = 'spam' # Works
>>> T
(1, ['spam', 3], 4)
>>> T[1] = 'spam'      # Fails
TypeError: object doesn't support item assignment
```



Datoteke

- datoteke – imenovani spremnici na računalu, kojima upravlja operacijski sustav
- datoteke (*files*) kao ugrađeni tip podataka u Pythonu omogućuju pristup datotekama
 - ugrađena funkcija `open()` stvara Python objekt tipa `file` koji služi kao veza prema fizičkoj datoteci
 - čitanje i pisanje u datoteku ostvaruje se pozivanjem metoda datotečnog objekta
 - ugrađeno ime `file` je sinonim za `open`



Operacije s datotekama

- tablica – uobičajene operacije s datotekama

Operacija	Opis
<code>out = open('/tmp/spam', 'w')</code>	otvara datoteku za pisanje
<code>in = open('data', 'r')</code>	otvara datoteku za čitanje
<code>S = in.read()</code>	učitava cijelu datoteku u string
<code>S = in.read(N)</code>	čita N bajtova
<code>S = in.readline()</code>	učitava sljedeći redak
<code>L = in.readlines()</code>	cijelu datoteku u listu stringova L
<code>out.write(S)</code>	zapisuje string
<code>out.writelines(L)</code>	zapisuje stringove iz liste L
<code>out.close()</code>	zatvaranje datoteke



Operacije s datotekama (2)

- otvaranje datoteke funkcijom `open()`
 - navodi se ime datoteke i način pristupa (čitanje, pisanje, nadodavanje)
 - argumenti su stringovi
 - ime datoteke može sadržavati apsolutni ili relativni put
 - ako se put ne navede, datoteka se traži u tekućem kazalu (gdje je skripta pokrenuta)
- učitavanje i zapisivanje podataka – u obliku znakovnih nizova
- zatvaranje datoteke – metodom `close()`
 - zahvaljujući mehanizmu prikupljanja otpada, Python automatski zatvara datoteku kada se oslobađa nereferencirani objekt tipa `file`
 - nije nužno “ručno” zatvarati datoteke, osobito u kratkim skriptama
 - ipak je zatvaranje datoteka dobra navika u većim projektima



Operacije s datotekama (3)

● primjer

```
>>> myfile = open('myfile', 'w') # otvara za pisanje
>>> myfile.write('hello text file\n')
>>> myfile.close( )
```

```
>>> myfile = open('myfile', 'r') # otvara za citanje
>>> myfile.readline( )
'hello text file\n'
>>> myfile.readline( ) # prazni string: EOF
''
```

- kraj datoteke signalizira prazni string – čitanje praznog retka daje `'\n'`
- prilikom otvaranja datoteke, uz način pristupa može se dodati oznaka `'b'` – binarni pristup (bez pretvorbi oznake kraja retka)
- dodavanje oznake `'+'` omogućuje otvaranje datoteke za čitanje i pisanje



Operacije s datotekama (4)

- objekte koje želimo zapisati nužno je pretvoriti u znakovne nizove – metode za učitavanje i zapisivanje ne obavljaju automatski pretvorbu

```
>>> X, Y, Z = 43, 44, 45 # razni tipovi objekata
>>> S = 'Spam'           # moraju se pretvoriti
>>> D = {'a': 1, 'b': 2}
>>> L = [1, 2, 3]

>>> F = open('datafile.txt', 'w')
>>> F.write(S + '\n') # kraj retka \n
>>> F.write('%s,%s,%s\n' % (X, Y, Z)) # pretvorba
>>> F.write(str(L) + '$' + str(D) + '\n')
>>> F.close( )

>>> bytes = open('datafile.txt').read( )
>>> bytes
"Spam\n43,44,45\n[1, 2, 3]${'a': 1, 'b': 2}\n"
```




Operacije s datotekama (5)

- nakon učitavanja, znakovne nizove je potrebno pretvoriti nazad u objekte

```
>>> F = open('datafile.txt')
>>> line = F.readline( )
>>> line
'Spam\n'
>>> line.rstrip( ) # Remove end-of-line
'Spam'

>>> line = F.readline( ) # sljedeći redak
>>> line
'43,44,45\n'
>>> parts = line.split(',') # dijelimo na zarezi
>>> parts
['43', '44', '45\n']

>>> numbers = [int(P) for P in parts] # int liste
>>> numbers # int zanemaruje \n
[43, 44, 45]
```



Operacije s datotekama (6)

- nastavak – lista i rječnik
- za pretvorbu ćemo iskoristiti ugrađenu funkciju `eval`, koja koristi znakovni niz kao Python izraz (izvršivi kôd)

```
>>> line = F.readline( )
>>> line
"[1, 2, 3]${'a': 1, 'b': 2}\n"
>>> parts = line.split('$') # dijeli na $
>>> parts
['[1, 2, 3]', '{"a': 1, 'b': 2}\n"]
>>> eval(parts[0])
[1, 2, 3]
>>> objects = [eval(P) for P in parts] # lista
>>> objects
[[1, 2, 3], {'a': 1, 'b': 2}]
```



Pohranjivanje Python objekata

- ugrađena funkcija `eval` je moćna, može se čak reći “previše moćna” !
- izvršit će *svaki* Python kôd učitani iz datoteke (pa i onaj *štetni*)
- za pohranu i učitavanje Python objekata vrlo je prikladan standardni modul `pickle`

```
>>> F = open('datafile.txt', 'w')
>>> import pickle
>>> pickle.dump(D, F) # D je dictionary
>>> F.close( )

>>> F = open('datafile.txt')
>>> E = pickle.load(F) # Load any object
>>> E
{'a': 1, 'b': 2}
```



Usporedbe

- objekti u Pythonu mogu se uspoređivati
 - uspoređuju se svi dijelovi složenih objekata, do trenutka kada se može odlučiti o odnosu uspoređivanih objekata
 - ugniježdeni objekti uspoređuju se rekurzivno, do potrebne dubine – prva pronađena razlika određuje rezultat usporedbe

```
>>> L1 = [1, ('a', 3)] # ista vrijednost
>>> L2 = [1, ('a', 3)] # različiti objekti
>>> L1 == L2, L1 is L2 # jednaki? isti?
(True, False)
```

- operator `==` ispituje jednakost objekata, a operator `is` njihov identitet (nalaze li se objekti na istoj adresi u memoriji)



Usporedbe (2)

- zanimljivo je pogledati što se događa npr. s *kratkim* stringovima (slično je i s brojevima):

```
>>> S1 = 'spam'
>>> S2 = 'spam'
>>> S1 == S2, S1 is S2
(True, True)
```

- očekivali bismo dva različita objekta iste vrijednosti, no oba imena referenciraju *isti* objekt
- razlog je interna optimizacija koju Python primjenjuje

- za dulje stringove to se ne događa

```
>>> S1 = 'malo dulji string'
>>> S2 = 'malo dulji string'
>>> S1 == S2, S1 is S2
(True, False)
```

- zbog nepromjenljivosti stringova i brojeva – nema utjecaja na ponašanje programa

Usporedbe (3)

- usporedba veličine objekata obavlja se rekurzivno nad ugniježdenim strukturama podataka

```
>>> L1 = [1, ('a', 3)]
>>> L2 = [1, ('a', 2)]
>>> L1 < L2, L1 == L2, L1 > L2
(False, False, True)
```

- općenito, usporedbe u Pythonu se obavljaju:

- brojevi se uspoređuju po veličini
- znakovni nizovi se uspoređuju leksikografski ("abc" < "ac")
- liste se uspoređuju komponentu po komponentu, s lijeva udesno
- rječnici se uspoređuju kao sortirane liste parova (ključ, vrijednost)

- vrijednosti istinitosti u Pythonu

- brojevi su `true` ako su različiti od 0
- ostali objekti su `true` ako su *neprazni*



Naredbe u Pythonu

- do sada smo susretali jednostavne *izraze*, te naredbe pridruživanja vrijednosti (*assignment*)
- složene naredbe (*compound statements*) u Pythonu imaju specifičnu strukturu
 - primjer – naredba `if`
 - oblik u C-u (C++, Java, JavaScript, Perl)

```
if (x > y) {  
    x = 1;  
    y = 2;  
}
```

- oblik u Pythonu:

```
if x > y:  
    x = 1  
    y = 2
```



Naredbe u Pythonu (2)

● oblik naredbe u Pythonu:

```
if x > y: # header line
    x = 1  # nested block
    y = 2
```

- nema oznaka početka i kraja bloka
 - blok naredbi određuje *uvlačenje* kôda
 - sve naredbe bloka moraju biti jednako uvučene
- dvotočka na kraju zaglavlja je *obavezna* – jedna od najčešćih pogrešaka kod početnika :-)
- zagrade oko uvjeta koji se ispituje su *neobavezne* – mogu se pisati, ali nije Pythonski stil
- kraj retka je kraj naredbe – nema potrebe za " ; "
- u jedan redak možemo “ugurati” više *jednostavnih* naredbi

```
a = 1; b = 2; print a + b
```




Naredbe u Pythonu (3)

- pojedinu naredbu možemo “razvući” na više redaka
 - ako dio naredbe uključuje par zagrada (oblih, uglatih ili vitičastih) – naredba ne može završiti dok se ne dođe do zatvarajuće zagrade

```
mlist = [111,  
          222,  
          333]
```

- kako se bilo koji izraz može zatvoriti u zagrade, na taj način je lako ostvariti “dozvolu” prelaska naredbe u novi red

```
X = (A + B +  
     C + D)
```

- funkcioniра i za složene naredbe

```
if (A == 1 and  
    B == 2 and  
    C == 3) :  
    print 'spam' * 3
```



Naredbe u Pythonu (4)

- pisanje naredbe kroz više redaka moguće je i označavanjem nastavljajna znakom "`\`"

```
X = A + B + \
    C + D
```

- zastarjelo, nepopularno u "Pythonlandu"
- teško je uočiti "`\`", ne smije biti praznina nakon oznake
- još jedan izuzetak – složene naredbe u jednom retku

```
if x > y: print x
```

 - dozvoljeno pisanje naredbi `if`, ili kratkih petlji u jednom retku
 - najčešća primjena – prijevremeni izlazak iz petlje (`break`)



Naredba **if**

- naredba `if` – tipična za većinu proceduralnih jezika

- općeniti oblik:

```
if <test1>:  
    <statements1> #blok naredbi  
elif <test2>:    # opcionalni elif  
    <statements2>  
else:            # opcionalni else  
    <statements3>
```

- Python izvršava blok naredbi pridružen prvom uvjetu koji je ispunjen (`true`), ili blok naredbi pridružen grani `else` ukoliko niti jedan od uvjeta nije ispunjen
- dijelovi `elif` i `else` mogu se izostaviti
- vezanje dijelova iste naredbe: `if`, `elif` i `else` određuje se njihovim vertikalnim poravnavanjem !



Naredba `if` (2)

● primjer

```
>>> x = 'killer rabbit'
>>> if x == 'roger':
...     print "how's jessica?"
... elif x == 'bugs':
...     print "what's up doc?"
... else:
...     print 'Run away! Run away!'
...
Run away! Run away!
```



Višestruko grananje

- Python nema naredbu oblika `switch` ili `case`, već se izvodi nizom `if-elif` ispitivanja ili indeksiranjem rječnika
 - kako se rječnici mogu graditi tijekom izvođenja programa, taj način upravljanja programskim slijedom može biti fleksibilniji od unaprijed kodiranih grananja

```
>>> choice = 'ham'
>>> branch = {'spam': 1.25,
...           'ham': 1.99,
...           'eggs': 0.99}
>>> print branch.get(choice, 'Bad choice') # default
1.99
>>> print branch.get('bacon', 'Bad choice')
```

- rječnici su prikladni za ovakvu vrstu akcija, no mogu se primijeniti i za složenije situacije – rječnici mogu sadržavati i funkcije



Programska petlja `while`

- najopćenitija naredba programske petlje
 - izvršava se blok naredbi dok god je uvjet ispunjen
 - ako uvjet već na početku nije ispunjen, tijelo petlje se ne izvrši niti jednom
- općeniti oblik naredbe:

```
while <test>:      # uvjet
    <statements1>  # tijelo
else:              # opcionalno
    <statements2>  # ako nije završena s break
```
- dio `else` je opcionalan, a izvršava se ukoliko petlja nije prekinuta naredbom `break`
 - specifičnost Pythona



Programska petlja `while` (2)

● primjeri

```
>>> while 1: # beskonacna petlja
...     print 'Type Ctrl-C to stop me'!
```

```
>>> x = 'spam'
>>> while x: # dok je niz neprazan
...     print x,
...     x = x[1:] # odrezuje prvi znak
...
spam pam am m
```

```
>>> a=0; b=10
>>> while a < b: # petlja brojastica
...     print a,
...     a += 1 # isto kao a = a+1
...
0 1 2 3 4 5 6 7 8 9
```



Naredbe `break` i `continue`

- naredbe `break` i `continue` imaju smisla samo unutar programskih petlji
 - `break` uzrokuje izlazak iz (najbliže) petlje unutar koje se nalazi
 - `continue` uzrokuje prelazak na sljedeću iteraciju petlje u kojoj se nalazi, odnosno premještanje izvođenja na zaglavlje petlje
 - naredba `pass` je *prazna* naredba – ne radi ništa; koristi se na mjestima gdje sintaksa zahtijeva naredbu, a nemamo ništa “pametno za raditi”
 - blok `else` u petlji izvršava se u slučaju *normalnog* završetka petlje (bez nailaska na naredbu `break`)

- oblik petlje uključujući `break` i `continue`

```
while <test1>:
    <statements1>
    if <test2>: break #izlaz
    if <test3>: continue #na pocetak
else:
    <statements2> #ako nije bio break
```




Naredbe break i continue (2)

● primjeri

```
while 1: pass # Type Ctrl-C to stop me!
```

```
x = 10
```

```
while x:
```

```
    x = x-1 # ili x -= 1
```

```
    if x % 2 != 0: continue # preskoci neparan
```

```
    print x,
```

```
>>> while 1:
```

```
...     name = raw_input('Enter name: ')
```

```
...     if name == 'stop': break
```

```
...     age = raw_input('Enter age: ')
```

```
...     print 'Hello', name, '=>', int(age) ** 2
```

```
...
```

```
Enter name: mel
```

```
Enter age: 40
```

```
Hello mel => 1600
```

```
Enter name: stop
```



Programska petlja `for`

- petlja `for` je namijenjena *iteriranju* kroz sekvence – funkcioniра za znakovne nizove, liste i n-torke, ali i za odgovarajuće oblikovane korisničke objekte

- oblik naredbe:

```
for <target> in <object>: #elementi objekta  
    <statements> #tijelo petlje
```

else:

```
    <statements> #ako nije bio break
```

- varijabli petlje pridjeljuje se vrijednost jednog po jednog elementa sekvence, i za svaki se izvršava blok naredbi
- i petlja `for` podržava opcionalnu granu `else`, koja se izvršava pri normalnom (neprekinutom) završetku petlje
- mogu se koristiti i naredbe `break` i `continue`



Programska petlja for (2)

● primjeri

```
>>> for x in ["spam", "eggs", "ham"]:  
...     print x,  
...  
spam eggs ham
```

```
>>> sum = 0  
>>> for x in [1, 2, 3, 4]:  
...     sum = sum + x  
...  
>>> sum  
10
```

```
>>> prod = 1  
>>> for item in [1, 2, 3, 4]: prod *= item  
...  
>>> prod  
24
```



Programska petlja for (3)

- petlja for može se primijeniti i sa stringovima i n-torkama

```
>>> S, T = "lumberjack", ("and", "I'm", "okay")
>>> for x in S: print x,
...
l u m b e r j a c k
>>> for x in T: print x,
...
and I'm okay
```

- ako se prolazi kroz sekvencu n-torki, varijabla petlje može biti n-torka varijabli

```
>>> T = [(1, 2), (3, 4), (5, 6)]
>>> for (a, b) in T: # tuple assignment
...     print a, b
...
1 2
3 4
5 6
```



Programska petlja for (4)

● primjer ugniježdenih petlji

```
>>> items = ["aaa", 111, (4, 5), 2.01] # skup objekata
>>> tests = [(4, 5), 3.14] # elementi koje trazimo
>>>
>>> for key in tests: # za svaki kljuc
...     for item in items: # za svaki objekt
...         if item == key: # provjera podudaranja
...             print key, "was found"
...             break
...     else:
...         print key, "not found"
...
(4, 5) was found
3.14 not found
```



Programska petlja for (4)

- isti primjer, elegantnije rješenje

```
>>> for key in tests: # For all keys
...     if key in items: # Let Python check for a match
...         print key, "was found"
...     else:
...         print key, "not found"
...
(4, 5) was found
3.14 not found
```

- u ovom primjeru koristi se operator `in` kako bi ispitao pripadnost – on implicitno prolazi kroz listu tražeći podudaranje i tako zamjenjuje unutrašnju petlju
- općenito – dobro je prepustiti Pythonu da obavi čim veći dio posla, kako zbog jezgrovitosti kôda, tako i zbog performanse



Funkcija range

- ugrađena funkcija `range` generira listu cijelih brojeva

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

- s jednim argumentom `range` generira listu cijelih brojeva počevši od 0 i *ne uključujući* argument

- s dva argumenta – prvi argument određuje prvi broj liste

- ako se navede i treći argument, on određuje korak

- raspon može uključivati i negativne brojeve, a može biti i silazan (negativan korak!)

```
>>> range(-5, 5)
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
```

```
>>> range(5, -5, -1)
[5, 4, 3, 2, 1, 0, -1, -2, -3, -4]
```



Funkcija range (2)

- generirani raspon (lista) može se koristiti kao sekvenca kroz koju iteira petlja `for`

```
>>> for i in range(3):  
...     print i, 'Pythons'  
...  
0 Pythons  
1 Pythons  
2 Pythons
```

- indeksiranje elemenata liste

```
>>> L = [1, 2, 3, 4, 5]  
>>>  
>>> for i in range(len(L)):  
...     L[i] += 1  
...  
>>> L  
[2, 3, 4, 5, 6]
```




Funkcija `zip`

- ugrađena funkcija `zip` omogućava paralelno iteriranje kroz više sekvenci
- funkcija kao argumente prima jednu ili više sekvenci, a vraća listu n-torki koje uparuju/združuju elemente tih sekvenci

```
>>> L1 = [1, 2, 3]
>>> L2 = [5, 6, 7]

>>> zip(L1, L2)
[(1, 5), (2, 6), (3, 7)]
```

- rezultat može poslužiti za istovremeni prolazak kroz obje sekvence

```
>>> for (x, y) in zip(L1, L2):
...     print x, y, '--', x+y
...
1 5 -- 6
2 6 -- 8
3 7 -- 10
```



Funkcija `zip` (2)

- funkcija `zip` može se iskoristiti i za inicijalizaciju rječnika
 - recimo da imamo izgrađene liste ključeva i vrijednosti koje želimo upariti u rječniku

```
>>> keys = ['spam', 'eggs', 'toast']
>>> vals = [1, 3, 5]

>>> D2 = { }
>>> for (k, v) in zip(keys, vals): D2[k] = v
...
>>> D2
{'toast': 5, 'eggs': 3, 'spam': 1}
```

- može i bolje – konstruktoru rječnika može se predati lista kreirana funkcijom `zip`

```
>>> D3 = dict(zip(keys, vals))
>>> D3
{'toast': 5, 'eggs': 3, 'spam': 1}
```



Funkcije

- funkcije u Pythonu ponašaju se drugačije nego funkcije u prevođenim jezicima
- funkcija se definira *naredbom* `def`
 - `def` je naredba koja se *izvršava*
 - funkcija *ne postoji* dok se ne izvrši `def`
 - moguće je gnijezditi naredbu `def` unutar petlji, `if` konstrukcija, pa i unutar drugih naredbi `def`
 - tipično se naredba `def` koristi u *modulima*, gdje generiraju funkcije prilikom prvog učitavanja modula
 - `def` stvara *objekt* funkciju i pridružuje ga imenu – ime je *referenca* na funkciju (objekt)



Funkcije (2)

● oblik naredbe `def`

```
def <name>(arg1, arg2, ... argN) :  
    <statements>  
    ...  
    return <value> #nije obavezno
```

- tipična složena naredba s retkom zaglavlja
- blok naredbi postaje tijelo funkcije – kôd koji se izvršava svaki put kada se funkcija pozove
- imena argumenata navedena u zaglavlju definicije funkcije pridružuju se objektima koji se pri pozivu proslijeđuju funkciji
- naredba `return` može se pojaviti bilo gdje u tijelu funkcije
 - završava se izvršavanje funkcije i vraća rezultat pozivateljuž
 - ako je nema, funkcija vraća objekt `None`



Funkcija je objekt

- `def` je naredba koja se *izvršava* – objekt-funkcija nastaje tijekom izvođenja programa

```
if test:
    def func( ): # jedna definicija
        ...
else:
    def func( ): # alternativna definicija
        ...

...
func( ) # poziv odabrane verzije
```

- funkcija je objekt, a njeno ime je referenca

```
othername = func # objekt se pridružuje drugom imenu
othername( ) # poziv iste funkcije drugim imenom
```



Definicija i poziv funkcije

● primjer funkcije

```
>>> def times(x, y):  
...     return x * y # tijelo se izvodi pri pozivu  
...
```

● poziv funkcije

```
>>> times(2, 4) # argumenti u zagradama  
8
```

● rezultat se može pohraniti

```
>>> x = times(3.14, 4)  
>>> x  
12.56
```

● tip argumenata nije zadan

```
>>> times('Ni', 4)  
'NiNiNiNi'
```

● inherentan *polimorfizam*



Primjer

- funkcija koja određuje presjek dviju sekvenci

```
def intersect(seq1, seq2):  
    res = [ ]          # lokalna varijabla, prazna lista  
    for x in seq1:      # scan seq1  
        if x in seq2:   # common item?  
            res.append(x) # add to end  
    return res
```

- poziv

```
>>> s1 = "SPAM"  
>>> s2 = "SCAM"  
  
>>> intersect(s1, s2) # strings  
['S', 'A', 'M']
```

- polimorfizam – prosljeđeni objekti moraju podržavati operacije u funkciji (ovdje: prvi – iteriranje s `for`, drugi – operator `in`)

```
>>> intersect([1, 2, 3], (1, 4)) # mixed types  
[1]
```



Doseg varijabli

- imena nastaju kad im se prvi put dodijeli vrijednost
- na temelju mjesta u programu gdje se dodjeljivanje vrijednosti obavlja, Python povezuje ime s odgovarajućim *prostorom imena*, odnosno određuje *doseg* (*scope*) u kojem je ime vidljivo
- imena definirana unutar funkcije su *lokalna* – vidljiva samo unutar funkcije
 - ne može doći do sukoba imena s imenima van funkcije
 - svaki poziv funkcije stvara *novi* lokalni doseg
 - ime u funkciji moguće je deklarirati *globalnim*
- svaki modul predstavlja *globalni* doseg
 - globalne varijable su atributi modula
 - globalni doseg se proteže najviše kroz jednu datoteku (modul)



Razrješavanje imena

- razrješavanje imena može se sažeti u tri jednostavna pravila:
 - referenca imena traži se u najviše četiri dosega (LEGB), i to sljedećim redom:
 - lokalni – L
 - doseg obuhvaćajuće (*enclosing*) funkcije – E
 - globalni – G
 - ugrađeni (*built-in*) – B
 - dodjela vrijednosti stvara ili mijenja *lokalno* ime *by default*
 - globalne deklaracije mapiraju imena u doseg obuhvaćajućeg modula
- funkcije mogu *koristiti* imena iz obuhvaćajućeg(E) ili globalnog (G) dosega, ali ih *ne mogu mijenjati* ako ih ne deklariraju globalnima
- ovaj način razrješavanja primjenjuje se za *jednostavna* imena varijabli, za kvalificirana imena (`objekt.atribut`) primjenjuju se drugačija pravila



Razrješavanje imena (2)

● primjer

```
# Global scope  
X = 99 # global
```

```
def func(Y):  
    # local scope  
    Z = X + Y # X is a global  
    return Z
```

```
func(1) # func in module: result=100
```

- globalna imena: X i func, X se može koristiti unutar funkcije bez posebnog deklariranja
- lokalna imena funkcije func: Y i Z
 - postoje samo tijekom izvršavanja funkcije, vrijednost im se dodjeljuje u definiciji funkcije
 - argumenti (Y) se uvijek proslijeđuju *pridruživanjem*



Razrješavanje imena (3)

- budući da se imena traže zadanim redoslijedom dosega (LEGB), uvođenje istog imena u lokalni doseg *zakriva* globalno ime

```
X = 88          # Global X
```

```
def func( ) :  
    X = 99      # Local X: hides global
```

```
func( )  
print X        # Prints 88: unchanged
```

- dodjela vrijednosti varijabli `x` unutar funkcije stvara *lokalnu* varijablu `x`
- to je varijabla koja nema nikakve veze s globalnom varijablom `x` u modulu izvan funkcije
- da bi funkcija mogla mijenjati globalnu varijablu, mora je proglasiti globalnom



Naredba `global`

- naredba `global` je jedina “stvar” u Pythonu koja *podsjeca* na deklaraciju :-)
- iza ključne riječi `global` navode se imena varijabli odvojena zarezima
- navedena imena se mapiraju u doseg obuhvaćajućeg modula

```
X = 88          # Global X
```

```
def func( ):  
    global X  
    X = 99 # mijenja globalnu varijablu
```

```
func( )  
print X          # Prints 99
```



Prenošenje argumenata u funkcije

- argumenti se prenose *dodjeljivanjem*
 - automatsko dodjeljivanje objekata lokalnim imenima
→ argumenti funkcije su reference na potencijalno dijeljene objekte koje referencira pozivatelj
 - dodjela *imenu* argumenta unutar funkcije *ne utječe* na pozivatelja – *doseg* imena je unutar funkcije
 - ako je argument izmjenljivi objekt, njegova izmjena u funkciji može utjecati na pozivatelja
- iako se zapravo proslijeđuju reference, ponašanje argumenata ovisi o njihovoj vrsti (izmjenljivosti)
 - nepromjenljivi argumenti se ponašaju *kao da su* proslijeđeni kao vrijednosti (iako se ne stvara kopija)
 - izmjenljivi argumenti ponašaju se kao da su proslijeđeni kao reference



Argumenti i dijeljene reference

● primjer

```
>>> def changer(x, y):  
...     x = 2          # mijenja samo lokalnu vrijednost  
...     y[0] = 'spam' # mijenja dijeljeni objekt  
...  
>>> X = 1  
>>> L = [1, 2] # pozivatelj  
>>> changer(X, L) # immutable and mutable  
>>> X, L          # X nepromijenjen, L razlicit  
(1, ['spam', 2])
```

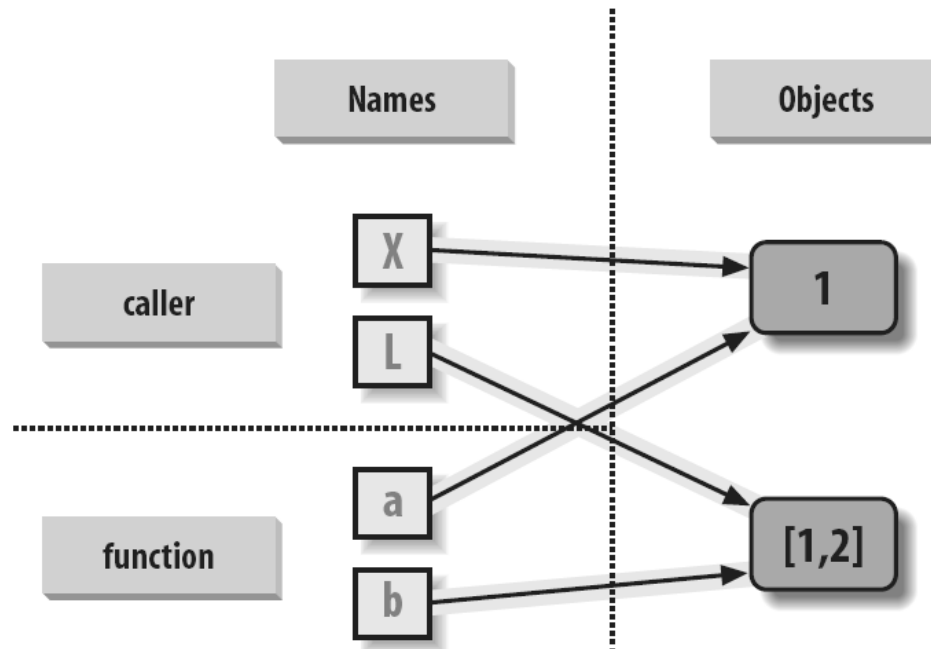
● `x` je lokalno ime, dodjeljuje mu se novi objekt, veza globalne varijable `X` i objekta `1` nije narušena

● dodjela `y[0] = 'spam'` mijenja element liste `L`



Argumenti i dijeljene reference (2)

- radi se naprosto o *dijeljenim referencama*





Izbjegavanje izmjene argumenata

- ako želimo spriječiti izmjenu izmjenljivih objekata proslijeđenih u funkciju, možemo proslijediti eksplicitnu *kopiju*

```
L = [1, 2]
```

```
changer(X, L[:]) # proslijedjuje se kopija
```

- kopiju može napraviti i sama funkcija

```
def changer(x, y):
```

```
    y = y[:]      # kopija !
```

```
    x = 2
```

```
    y[0] = 'spam' # izmjena na kopiji
```

- možemo i objekt pretvoriti u nepromjenljivi

```
L = [1, 2]
```

```
changer(X, tuple(L)) # nepromjenljiv objekt
```

- *upozorenje*: izvorna lista može sadržavati i promjenljive objekte – niti kopija niti pretvorba u n-torku ne može spriječiti mijenjanje tih ugniježđenih objekata !



Oponašanje izlaznih argumenata

- Python nema mehanizam poziva s referencom (*call-by-reference*)
- možemo ga oponašati tako da se izlazni parametri vrate kao n-torka, koja se dodjeljuje izvornim argumentima u pozivatelju

```
>>> def multiple(x, y):  
...     x = 2 # mijenjaju se lokalna imena  
...     y = [3, 4]  
...     return x, y # povratna n-torka  
...  
>>> X = 1  
>>> L = [1, 2]  
>>> X, L = multiple(X, L) # povratna dodjela rezultata  
>>> X, L  
(2, [3, 4])
```

- konačan rezultat je izmjena izvornih argumenata



Posebni načini imenovanja argumenata

- uobičajeni način dodjele imena argumentima funkcije je *pozicijski* – pri pozivu se argumenti navode istim redoslijedom kao u definiciji funkcije, mora ih biti jednak broj
- Python nudi i druge mogućnosti, koje omogućuju dodatnu fleksibilnost
 - podudaranje argumenata imenom (*keywords*) pri pozivu funkcije – argumenti se mogu navesti proizvoljnim redoslijedom
 - pretpostavljene vrijednosti za argumente koji se na proslijede (*default*) – omogućava pozivanje funkcije s “manjkom” argumenata
 - preuzimanje promjenjivog broja argumenata (*varargs*) – omogućava pozivanje funkcije s “viškom” argumenata



Dodjela imena argumentima

- pozicijsko i imenovano povezivanje argumenata

```
>>> def f(a, b, c): print a, b, c
```

```
...
```

```
>>> f(1, 2, 3) # pozicijsko povezivanje
```

```
1 2 3
```

```
>>> f(c=1, b=3, a=2) # povezivanje imenom
```

```
2 3 1
```

```
>>> f(3, c=2, b=1) # miješano
```

```
3 1 2
```

- broj argumenata u pozivu mora odgovarati broju argumenata u definiciji funkcije
- u slučaju miješanog povezivanja imena, najprije se dodjeljuju pozicijski argumenti, s lijeva udesno

```
>>> f(3, c=2, a=1) # ne valja!
```



Pretpostavljene vrijednosti argumenata

- pretpostavljene vrijednosti argumenata omogućavaju da neki argumenti pri pozivu budu opcionalni – ako se argument ne proslijedi, dodjeljuje mu se pretpostavljena vrijednost

```
>>> def f(a, b=2, c=3): print a, b, c
```

```
...
```

```
>>> f(1) # moramo proslijediti vrijednost za a
1 2 3
```

```
>>> f(a=1) # moze i po imenu
1 2 3
```

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

```
>>> f(1, 4, 5)
1 4 5
```

```
>>> f(1, c=6) # imenovanje pomaze
1 2 6
```



Proizvoljna lista argumenata

- Python nudi mogućnost prosljeđivanja proizvoljnog broja argumenata
- prvi oblik primjene prikuplja prekobrojne argumente u n-torku

```
>>> def f(*args): print args
```

```
...
```

```
>>> f( )
```

```
( )
```

```
>>> f(1)
```

```
(1, )
```

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

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



Proizvoljna lista argumenata (2)

- drugi oblik radi samo za pridruživanje argumenata imenom, a argumenti se prikupljaju u rječnik

```
>>> def f(**args): print args
```

```
...
```

```
>>> f( )
```

```
{ }
```

```
>>> f(a=1, b=2)
```

```
{ 'a': 1, 'b': 2 }
```

- moguće je i kombiniranje

```
>>> def f(a, *pargs, **kargs): print a, pargs, kargs
```

```
...
```

```
>>> f(1, 2, 3, x=1, y=2)
```

```
1 (2, 3) { 'y': 2, 'x': 1 }
```

```
>>> f(1, 2, 3, x=1, y=2, 4, 5) # ovo ipak ne ide
```

```
SyntaxError: non-keyword arg after keyword arg
```



Bezimene funkcije (lambda izrazi)

- *lambda izrazi* omogućavaju generiranje funkcijskih objekata izrazom – bez definicije i bez imena (anonimna funkcija)
 - *izraz* a ne naredba – može se pojaviti na mjestima gdje sintaksa ne dozvoljava `def`, npr. unutar liste ili u pozivu funkcije
 - kao izraz, lambda vraća vrijednost (novu funkciju), kojoj se može dodijeliti ime (ali i ne mora)

● opći oblik:

lambda argument1, argument2, ... argumentN : izraz

- nakon ključne riječi `lambda` navode se argumenti, koji se koriste u izrazu
- generirana funkcija radi jednako kao i funkcija stvorena naredbom `def`
- tijelo lambda funkcije je *jedan izraz*
→ ograničena funkcionalnost



Bezimene funkcije (2)

● primjer

```
>>> f = lambda x, y, z: x + y + z
>>> f(2, 3, 4) # normalan poziv
9
```

● pretpostavljene vrijednosti argumenata rade i za lambda funkcije

```
>>> x = (lambda a="fee", b="fie", c="foe": a + b + c)
>>> x("wee")
'weefiefoe'
```

● pravila za razrješavanje imena (dosega) vrijede

```
>>> def knights( ):
...     title = 'Sir'
...     action = (lambda x: title + ' ' + x) # title
...     return action                        # vraca funkciju
...
>>> act = knights( )
>>> act('robin')
'Sir robin'
```




Bezimene funkcije (3)

- čemu uopće lambda izrazi ?
 - način da ugradimo definiciju funkcije unutar kôda koji ju koristi – npr. povratne funkcije (*callback handlers*) kodirane kao lambda izrazi u samom pozivu za registriranje
 - tablice grananja pohranjene u listi ili rječniku

```
L = [(lambda x: x**2), (lambda x: x**3),  
      (lambda x: x**4)]
```

```
for f in L:  
    print f(2) # ispisuje 4, 8, 16  
print L[0](3) # ispisuje 9
```

- ostvarenje višestrukog grananja rječnikom:

```
>>> key = 'got'  
>>> {'already': (lambda: 2 + 2),  
...  'got': (lambda: 2 * 4),  
...  'one': (lambda: 2 ** 6)  
...  }[key] ( )
```

8



Još malo o modulima

- koncept modula smo upoznali (slajdovi 8–15)
 - datoteka s nastavkom imena `".py"` je modul
 - modul se može učitati naredbama `import` ili `from`
 - prilikom *prvog* učitavanja izvršavaju se naredbe modula i stvara objekt modul
 - globalna imena modula (atributi) postaju dostupna
 - ako je modul učitán s `import modul`, atributima se pristupa *kvalifikacijom*: `modul.atribut`
 - ako je modul učitán s `from modul import atribut`, imena atributa *kopiraju se* u prostor imena učitavatelja
 - naredbom `from modul import *`, kopiraju se imena svih atributa učitánog modula
 - ukoliko je potrebno ponovo učitati modul (npr. promijenili smo ga), koristi se ugrađena funkcija `reload`



Moduli i prostori imena

- kao i `def`, `import` i `from` su *naredbe* – modul i njegova imena nisu raspoloživi dok se naredba učitavanja ne izvrši
- kao i `def`, `import` i `from` su implicitna dodjeljivanja
 - `import` cijeli modul kao objekt dodjeljuje jednom imenu
 - `from` kopira jedno ili više imena – ta imena postaju reference na dijeljene objekte
 - dodjela novog objekta imenu ne utječe na objekte u modulu
 - izmjena (izmjenljivog) objekta preko imena u učitavatelju utječe na objekt u učitanom modulu
- primjer

```
# modul small.py  
x = 1  
y = [1, 2]
```



Moduli i prostori imena (2)

● primjer (nastavak)

```
$ python
>>> from small import x, y # kopiramo imena
>>> x = 42 # mijenja se lokalna kopija
>>> y[0] = 42 # mijenjamo izvorni objekt

>>> import small # nedostaje nam referenca objekta
>>> small.x # to je drugi x
1
>>> small.y # ali listu y dijelimo
[42, 2]

>>> reload(small) # nanovo stvaramo objekt modul
>>> small.y # obnovljen
[1, 2]
>>> y # ovaj pokazuje na stari objekt
[42, 2]
```



Moduli i prostori imena (3)

- za učitani modul globalni prostor imena je datoteka u kojoj je definiran – ne vidi prostor imena modula koji ga je učitao

```
# modul moda.py
X = 88 # globalna varijabla za ovaj modul

def f( ):
    global X # mijenja globalnu var OVOG MODULA
    X = 99 # ne vidi imena u drugim modulima
```

```
# modul modb.py
X = 11 # i on ima svoj globalni X
import moda
moda.f( ) # postavlja moda.X, a ne X iz ovog modula
print X, moda.X
```

```
$ python modb.py
11 99
```



Moduli i prostori imena (4)

gniježdenje modula

```
# modul mod3.py  
X = 3
```

```
# modul mod2.py  
X = 2  
import mod3  
print X, mod3.X
```

```
# modul mod1.py  
X = 1  
import mod2  
print X, mod2.X,  
print mod2.mod3.X # nested mod3's X
```

```
$ python mod1.py  
2 3  
1 2 3
```



Skrivanje podataka u modulu

- modul u Pythonu eksportira sva imena dodjeljena u najvišj razini datoteke u kojoj se nalazi
 - nema načina za sprječavanje klijenta da promijeni ime u modulu ako to baš želi
 - u Pythonu je skrivanje podataka u modulu *konvencija* a ne sintaktičko ograničenje
- naredba `from *` zapravo *ne kopira sva* imena učitano modula
 - imena koja započinju znakom "_" (podvlaka) ne kopiraju se naredbom `from *`
 - namjera je smanjivanje "onečišćenja" prostora imena (ne kopiraju se imena koja nisu predviđena za korištenje od strane klijenta), a ne *zabrana* izmjene
 - ako to želimo, izmjena je i za takva imena moguća nakon učitavanja modula naredbom `import`



Skrivanje podataka u modulu (2)

- sličan efekt može se postići dodjelom liste stringova s imenima varijabli koje želimo eksportirati varijabli `__all__` u najvišoj razini modula

```
__all__ = ["Error", "encode", "decode"]  
          # Export these only
```

- i ovaj oblik skrivanja odnosi se *samo* na `from *` oblik učitavanja modula
- Python najprije traži `__all__` listu unutar modula
- ako takva lista nije definirana, kopiraju se sva imena bez vodeće podvlake



Klase i OOP

- i do sada smo koristili pojam *objekt*, a vidjeli smo i objekte koji imaju *metode*
- Python podržava i oblikovanje novih tipova objekata – *klasa*, te *nasljeđivanje*
- stvaranje novog tipa objekta ostvaruje se naredbom **class**
- novi tipovi objekata mogu se oblikovati tako da se ponašaju slično ugrađenim tipovima objekata
- nasljeđivanje je mehanizam koji podupire prilagođavanje i višestruko korištenje koda
- korištenje OOP u Pythonu je neobavezno, ali kako ga koriste mnogi alati, dobro je steći barem grubu sliku o ovim konceptima



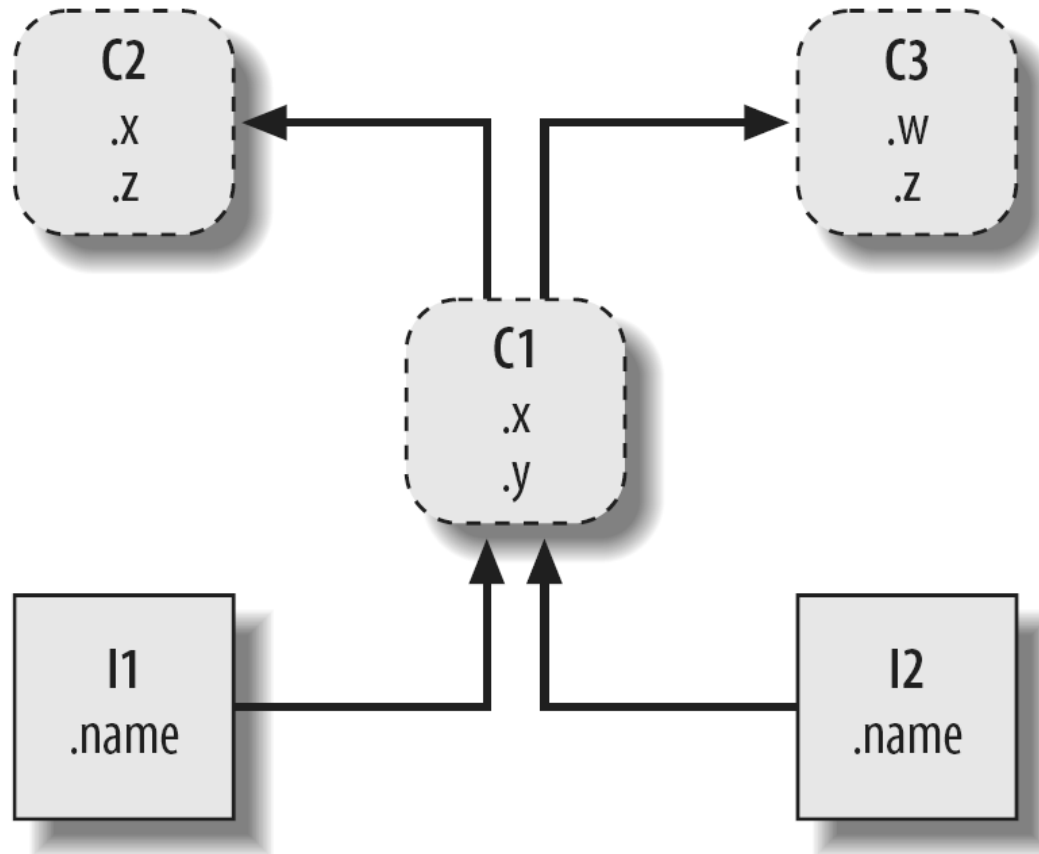
Nasljeđivanje kao pretraživanje

- objekt ima *attribute*, a pristup atributu započinje izrazom poput `objekt.attribute`
 - kad se ovakav izraz primijeni na objekt izveden na temelju naredbe `class`, pokreće se operacija *pretraživanja* stabla povezanih objekata – traži se *prvo pojavljivanje* atributa
 - stablo se pretražuje od dna prema vrhu, krenuvši od samog objekta, pa zatim kroz klase iz kojih je izveden → *nasljeđivanje*
 - objekti niže u stablu *nasljeđuju* attribute objekata smještenih više u stablu
- u Pythonu je ovaj koncept doista i izveden kao pretraživanje – u kôdu gradimo stablastu strukturu povezanih objekata, a Python tijekom izvođenja programa pokreće pretraživanje svaki put kada naiđe na izraz tipa `objekt.attribute`



Nasljeđivanje kao pretraživanje (2)

- primjer – stablo pretraživanja





Nasljeđivanje kao pretraživanje (3)

- *razred (klasa)* – objekt koji svojim atributima (podaci i funkcije) definira ponašanje koje nasljeđuju svi *primjerci (instance)* koji se iz te klase generiraju
- *primjerak* predstavlja konkretni objekt u domeni programa, njegovi atributi sadrže podatke koji variraju od primjerka do primjerka
- primjerak nasljeđuje attribute svojeg razreda, a razred attribute svojih *nad-razreda*
 - odgovara stablu pretraživanja
 - kako pretraživanje napreduje od dna prema vrhu, podklasa može izmijeniti (*override*) ponašanje definirano u nadklasi



Nasljeđivanje kao pretraživanje (4)

- primjer sa slike, izraz `I2.w`
 - primjer nasljeđivanja
 - započinje pretraživanje stabla prikazanog na slici (str. 163)
 - Python traži atribut `w` počevši od `I2`, prema vrhu stabla, pretraživanjem povezanih objekata `I2`, `C1`, `C2`, `C3`
 - staje prvim nailaskom na atribut odgovarajućeg imena – u ovom slučaju `C3.w` i^m drugim riječima, `I2` nasljeđuje atribut `w` od `C3`
- referenciranje drugih atributa završava pretraživanjem drugih grana stabla
 - `I1.x` i `I2.x` pronalaze `x` u `C1` jer je niže u stablu od `C2` – *overriding*
 - `I1.y` i `I2.y` pronalaze `C1.y`
 - `I1.z` i `I2.z` pronalaze `C2.y` – pretraživanje je *s lijeva u desno*
 - `I2.name` pronalazi `name` u `I2` bez penjanja uz stablo



Razredi

- naredba **class** generira novi *objekt razred*

```
class C2: ... # novi razred
class C3: ...
class C1(C2, C3): ... # nasljedjuje C2 i C3
```

- razredi se povezuju sa svojim nad-razredima njihovim navođenjem u zagradama; redoslijed navođenja određuje redoslijed pretraživanja u stablu
- primjer višestrukog nasljeđivanja

- *poziv razreda* generira novi *primjerak* razreda

```
I1 = C1( ) # novi primjerci razreda C1
I2 = C1( )
```

- primjerci se automatski povezuju s razredima iz kojih su generirani



Razredi (2)

- zbog načina na koji se izvodi pretraživanje/nasljeđivanje, izbor objekta za koji se veže atribut je ključan – on određuje doseg imena
 - atributi vezani za primjerke, odnose se samo na konkretan primjerak
 - attribute vezane za razred *dijele* svi pod-razredi i svi primjerci izvedeni iz tog razreda
 - atributi koji su vezani uz razred obično se stvaraju dodjelom u naredbama unutar definicije razreda
 - atributi vezani za primjerke obično se stvaraju dodjelama posebnom argumentu `self` koji se prosljeđuje funkcijama unutar razreda

```
class C1(C2, C3):  
    def setname(self, who): # C1.setname  
        self.name = who # self je I1 ili I2
```



Razredi (3)

- stvaranje primjerka i dodjela vrijednosti atributu

```
I1 = C1( ) # stvaranje primjeraka  
I2 = C1( )  
I1.setname('bob') # postavlja I1.name u 'bob'  
I2.setname('mel')  
print I1.name # ispisuje 'bob'
```

- kada se `def` nađe unutar definicije razreda, smatra se *metodom* i automatski dobiva poseban prvi argument
 - po *konvenciji* se naziva `self`
 - osigurava referencu na primjerak iz kojeg se metoda poziva
 - kao `this` u C++ ili Java, ali u Pythonu je `self` uvijek eksplicitan



Konstruktor

- prilikom stvaranja primjerka (instanciranja) ponekad želimo inicijalizirati neke njegove elemente

```
class C1(C2, C3):  
    def __init__(self, who):  
        # postavi name pri konstrukciji  
        self.name = who # self je I1 ili I2
```

```
I1 = C1('bob') # postavlja I1.name u 'bob'  
I2 = C1('mel')
```

- metoda imena `__init__` poziva se automatski kad god se generira novi primjerak razreda – *konstruktor*
- prvi argument `self` automatski se proslijeđuje zajedno s listom ostalih, navedenih argumenata
- šira skupina metoda – za preopterećivanje (*overloading*) operatora: npr. metoda `__and__` preopterećuje operator `&`, a `__sub__` operator `-`



Primjer

- primjer: baza podataka o zaposlenicima
 - bazni razred definira pretpostavljeno ponašanje zajedničko svim vrstama zaposlenika u poduzeću:

```
class Employee: # temeljna nadklasa
    def computeSalary(self): ... # zajednicko
    def giveRaise(self): ...
    def promote(self): ...
    def retire(self): ...
```
 - specijalizacija – prilagodba ponašanja za neku vrstu zaposlenika → izvedeni podrazred koji zamjenjuje (*override*) metodu za izračun plaće

```
class Engineer(Employee): # podklasa
    def computeSalary(self): ... # prilagodba
```
 - primjerci mogu biti generirani iz osnovnog ili iz izvedenog razreda

```
bob = Employee( ) # osnovno ponasanje
mel = Engineer( ) # izmijenjeno ponasanje
```