Programarea calculatoarelor si limbaje de programare 1

Str, List, Dict

Universitatea Politehnica din Bucureşti

Sumar

- ***str***
- list
- dict

str – tipul string

- Definitie: sunt secvente fixe, imuabile de caractere – "puncte" Unicode (aka numere de identificare)
- Reprezinta:
  - text – cuvinte, continut fisiere sursa Python…
  - *bytes* – pentru fisiere media, transfer in retea
  - text Unicode, non-ASCII – internationalizare
- Nu exista tipul caracter, ci doar stringuri de un caracter lungime.

str, expresii

Expresie *str*	Semnificatie
s = **''**	String vid
s = **"**king's**"**	Delimitatori – apostrofi (simpli sau dubli)
s = 's\np\ta\x00m'	Secvente escape – cu ****
s = **'''**…linii multiple…**'''**	String multilinie – delimitat de trei apostrofi
s = **r**'c:\temp\spam'	*raw* – escape neinterpretat
b = **b**'sp\xc4m'	String – *byte*
u = **u**'sp\u00cam'	String – Unicode
s1 **+** s2	Concatenare
s ***** 3	Repetitie
s**[**i**]**	Indexare
s**[**i**:**j**:**k**]**	Slice (felie)
len(s)	Lungimea stringului

str…

Expresie *str*	Semnificatie
'un ***%s*** de program' ***%*** model	Expresie de formatare, cu **%**
'un ***{0}*** de program'.***format***(model)	Metoda format()
s.***find***('pa')	Metode ale *str*
s.***rstrip***()	Inlaturare spatiu alb (white space)
s.***replace***('pa', 'xx')	Inlocuire
s.***split***(',')	Impartire in substringuri – cu delimitator
s.***isdigit***()	Testare continut
s.***lower***()	Conversie continut – la litere mici
s.***endswith***('spam')	Testare suffix
'spam'.***join***(strlist)	Alipire – cu 'spam' intre parti
s.***encode***('latin-1')	Codificare Unicode
b.***decode***('utf-8')	Decodificare Unicode

str…

Expresie *str*	Semnificatie
for x ***in*** s: print(x)	Iteratie
'spam' ***in*** s	Test – s contine 'spam' ?
[c * 2 ***for*** c ***in*** s**]**	Colectie iterativa, de tip list
map(*ord*, s)	Mapare
re.match('sp***(.*)***am', linie)	Expresii regulate

- Stringurile se reprezinta atat cu apostrofi simpli cat si dubli
- Procesarea se face cu expresii sau/si metode specifice stringurilor
- Procesarea avansta – recunoastere sabloane, se poate face cu modulul ***re*** – expresii regulate

Apostrofi simpli sau dubli

- Apostroful simplu, **'**, si cel dublu, **"**, au rol de delimitatori ai unui string – de preferat cel simplu (folosit de Python)

```
>>> 'spam', "spam"
('spam', 'spam')
```

- Permit includerea apostrofului fara a-l preceda cu **** (escape).

```
>>> 'All the King"s Men', "All the King's Men"
('All the King"s Men', "All the King's Men")
```

- Fara virgula rezulta concatenare implicita de stringuri:

```
>>> titlu = "Alice "    'in'      " Wonderland"
>>> titlu
'Alice in Wonderland'
```

Secvente escape

- **** (backslash) precede secvente escape, folosite pentru caractere greu tiparibile.

```
>>> s = 'a\nb\tc'     #fiecare secventa escape reprezinta un singur caracter
>>> s                 #afisare
'a\nb\tc'
>>> print( s )        #interpretare
a
b      c
>>> len( s )          #lungime – 5: a, line feed, b, tab, c.
5
```

- Un caracter nu este un octet!

Secvente…

Secventa	Semnificatie	Secventa	Semnificatie
\<Enter>	Continuare linie	\v	Tab vertical
\\	Backslash	\xhh	Valoare hexa *hh*
\'	Apostrof simplu	\ooo	Valoare octala *ooo*
\"	Apostrof dublu	\0	Caracter zero
\a	Bell	\N{ id }	ID de baza de date Unicode
\f	Formfeed	\uhhhh	Unicode pe 16 biti
\n	Linie noua (linefeed)	\Uhhhhhhhh	Unicode pe 32 biti
\r	Carriage return	\<altceva>	**** si caracterul **<altceva>**, secventa nerecunoscuta
\t	Tab orizontal		

- Includere de valori binare in string:

```
>>> s = 'a\0bc' #0 nu incheie un string!
>>> s  #totdeauna afisare in hexa: \x00
'a\x00bc'
>>> len( s )
4
```

Stringurile "raw"

- Plasand caracterul ***r*** in fata stringului, mecanismul de escape este oprit:

>>> **myfile = open(*r*'C:\new\text.dat', 'w') #fara prefixul *r* s-ar fi interpretat \n ca new line si \t ca tab…**

>>> myfile = open('C:\\new\\text.dat', 'w') #cu \ dublat, de asemenea corect

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

>>> **path** **#si Python dubleaza \ la afisare**

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

- Un string raw NU se poate termina intr-un singur backslash!

Stringuri multilinie '''…'''

- Sunt delimitate de trei apostrofi, fie simpli, fie dubli si pot contine linii multiple de text

```
>>> proverb = '''Ce
tie nu-ti place
altuia nu face'''
>>> proverb
'Ce\ntie nu-ti place\naltuia nu face'
```

- Pe parcurs, apostrofii simpli sau dubli nu trebuie sa fie precedati de \ (dar pot fi)
- Comentariile cu # trebuie sa preceada sau sa urmeze stringul multilinie, altfel ar face parte din string!

Stringuri…

```
>>> menu = (        #parantezele permit scrierea pe mai multe randuri
    'spam\n'         #alaturarea inseamna concatenare
    "eggs\n"         #\n trebuie introdus manual
    )
>>> menu
'spam\neggs\n'
```

- Apostrofii tripli se pot folosi la comentarea (temporara) a unor portiuni din codul Python (ineficient – se genereaza un string, dar rapid)

Operatii pe stringuri

- Se efectueaza cu expresii, metode sau formatari
- Operatii elementare:

```
>>> len( 'abc' )          #lungime string
3
>>> 'abc' + 'def'         #concatenare
'abcdef'
>>> 'abc' + 9             #eroare cu numere!
TypeError: can only concatenate str (not "int") to str
>>> 'Hi!' * 3             #repetitie
'Hi!Hi!Hi!'
>>> print( '-' * 80 )     #utila, trage o linie
--------------------------------------------------------------------------------
```

Operatii…

```
>>> #iteratii
>>> myjob = 'operator'
>>> for c in myjob: print( c, end=' ' )

o p e r a t o r
>>> 'p' in myjob                    #testarea prezentei unui substring, cu in
True
>>> 'x' in myjob
False
>>> 'spam' in 'abcspamdef'   #pozitia nu este returnata de operatorul in
True
```

Indexare si slicing

- Indexarea se face cu paranteze patrate; incepe de la zero; se returneaza stringul de un caracter aflat in pozitia respectiva.

```
>>> s = 'spam'
>>> s[0], s[3], s[len( s ) - 1]  #ultima pozitie este len(s) - 1
('s', 'm', 'm')
>>>                              #index negativ, se numara de la dreapta spre stanga
>>> s[0], s[-2]                 #de fapt se aduna cu len( s ): 4-2=2
('s', 'a')
>>>                              #slicing, forma generalizata de indexare
>>> s[1:3], s[1:], s[:-1]       #returneaza zona dintre primul indice - inclusiv si cel
                                  de-al doilea – exclusiv.
('pa', 'pam', 'spa')
```

Secvente – indexarea

Indexarea, s[i]

- primul element este in pozitia zero
- indexul negativ se socoteste de la dreapta (modulo lungimea stringului)
- s[0] este primul element
- s[-2] este al doilea element socotit de la urma

Secvente – slicing

Slicing, s[i:j]

- limita dreapta este exclusiva
- limitele omise sunt zero sau respectiv lungimea secventei
- s[1:3] – intre 1 inclusiv si 3 exclusiv
- s[1:] – intre 1 si sfarsit
- s[:3] – intre inceput si 3 exclusiv
- s[:-1] – tot fara ultimul element
- s[:] – tot, copie a lui s

Secvente – slicing extins

Slicing extins s[i:j:k]

- foloseste pasul *k*, implicit egal cu +1
- se pot sari elemente:

```
>>> s = 'abcdefghijklmnop'
>>> s[1:10:2]                    #din 2 in 2 de la 1 la 10-1.
'bdfhj'
>>> s[::2]                       #din 2 in 2, de la 0 la sfarsit
'acegikmo'
```

- se pot face parcurgeri in sens opus

```
>>> s = 'hello'                  #inversare, k negativ
>>> s[::-1]                      #tot, de la dreapta la stanga
'olleh'
```

Slicing extins

```
>>> s = 'abcdefg'                    #cu pas negativ, primul index > al doilea index
>>> s[5:1:-1]
'fedc'
```

- slicing cu un obiect de tip *slice*:

```
>>> 'spam'[1:3]
'pa'
>>> 'spam'[slice( 1, 3 )]            #cu obiect slice
'pa'
>>>
>>> 'spam'[::-1]
'maps'
>>> 'spam'[slice( None, None, -1 )]     #idem
'maps'
```

Slicing, exemplu practic

- Script test.py:

```
import sys                  #importare a modulului sys
print( sys.argv )           #inclusiv numele programului
print( sys.argv[1:] )       #doar lista de argumente ale comenzii, spre prelucrare
```

- Executie

```
C:\Users\Dan>python test.py -1 -3 -c
['test.py', '-1', '-3', '-c']
['-1', '-3', '-c']
```

Conversii de stringuri

- Sunt necesare deoarece Python nu face conversii implicite – ex. *str* ←→ *int:*

```
>>> "33" + 1
TypeError: can only concatenate str (not "int") to str
>>> int( '33' ), str( 33 )              #conversii explicite
(33, '33')
>>> repr( 33 )                          #reprezentare string "as-code", cu repr()
'33'
>>> str( 'spam' ), repr( 'spam' )       #incadrare cu apostrofi dubli pentru repr()
('spam', "'spam'")
>>> s = '33'; i = 1
>>> s + i
TypeError: can only concatenate str (not "int") to str
>>> int( s ) + i                        #corect, adunare
```

Conversii...

```
>>> s + str( i )                    #corect, concatenare
'331'
```

- Sau conversii *float* ←→ *str:*

```
>>> str( 3.14 ), float( '1.5' )
('3.14', 1.5)
>>> numar = '123.45e-10'
>>> float( numar )
1.2345e-08
```

- Sau conversii cu *eval()* – converteste orice string la orice obiect.

```
>>> eval( numar )
1.2345e-08
```

Conversii cu *ord* si *chr*

- *ord()* calculeaza pozitia in tabela ASCII a stringului de un caracter primit ca argument
- *chr()* face operatia inversa – insa stringul rezultat poate ocupa mai mult de un octet in memorie (Unicode)

```
>>> ord( 's' )
115
>>> chr( 115 )
's'
>>> #calculul unui intreg reprezentat in baza 2:
>>> b = '1101'    #str
>>> i = 0         #initializare la zero
>>> while b:
        i = i * 2 + ord( b[0] ) - ord( '0' )  #cu ord()
        b = b[1:]    #slicing, deplasare in str
>>> i
13
>>> #elementar, cu int(), functie predefinita:
>>> int( '1101', 2 )  #str la int
13
>>> bin( 13 )      #int la str binar
'0b1101'
>>>
```

Modificari de text in Python

- Stringurile sunt fixe, dar se pot crea stringuri noi prin concatenare, slicing, aplicare de metode specifice stringurilor, formatare – dar si cu tipul *bytearray* care este modificabil (secvente de octeti – pe 8 biti).

```
>>> s = 'spam'
>>> s = s + 'SPAM'        #concatenare
>>> s
'spamSPAM'
>>> s = s[:4] + 'ham'     #slicing
>>> s
'spamham'
>>> #rezultatele se retin prin atribuire la (aceeasi) variabila.
```

Metode

- Metodele sunt specifice tipului caruia ii apartin
- Exista metode cu acelasi nume in diverse tipuri, ex. *count(), copy()*
- Sintaxa apelului de metode:
  - *obiect.atribut* – valoarea atributului din obiect
  - *functie(argumente)* – apel de functie
  - *obiect.metoda(argumente)* – apel de metoda care proceseaza obiect, cu argumente

```
>>> s = 'spam'
>>> rezultat = s.find( 'pa' )    #apel metoda, 'pa' in poz. 1
>>> rezultat
1
```

Metode ale *str*

s.capitalize()	s.isalpha()	s.rindex(sub[,start[,end]])
s.casefold()	s.isdecimal()	s.rjust(width[,fill])
s.center(width[,fill])	s.isdigit()	s.rpartition(sep)
s.count(sub[,start[,end]])	s.isidentifier()	s.rsplit(sep[,max])
s.encode([enc[,err]])	s.ljust(width[,fill])	s.rstrip([chars])
s.endswith(suffix[,start[,end]])	s.lower()	s.split(sep[,max])
s.expandtabs([size])	s.lstrip([chars])	s.splitlines([keepends])
s.find(sub[,start[,end]])	s.maketrans(x[,y[,z]])	s.startswith(prf[,st[,end]])
s.format(*args,**kwargs)	s.partition(sep)	s.islower()
s.index(sub[,start[,end]])	s.replace(old,new[,cnt])	s.isnumeric()
s.isalnum()	r.rfind(sub[,start[,end]])	s.isprintable()

Metode...

s.isspace()	s.strip([chars])	s.upper()
s.istitle()	s.swapcase()	s.zfill(width)
s.isupper()	s.title()	
s.join(iterabil)	s.translate(map)	

- **Modificari de stringuri cu metode:**

```
>>> s = 'spamm'
>>> s = s.replace( 'mm', 'xx' )          #inlocuire, cu replace(), toate 'mm' cu 'xx'
>>> s
'spaxx'
>>> 'aa$bb$cc$dd'.replace( '$', 'SPAM' )
'aaSPAMbbSPAMccSPAMdd'
```

Modificari

```
>>> #inlocuire doar prima aparitie, cu find() si slicing:
>>> s = 'aaaaaSPAMaaaaaaaaSPAMaaaaaaa'
>>> poz = s.find( 'SPAM' )
>>> poz
5
>>> s = s[:poz] + 'EGGS' + s[(poz+4):]
>>> s
'aaaaaEGGSaaaaaaaaSPAMaaaaaaa'
>>> #mai simplu, cu replace() cu trei argumente:
>>> s = 'aaaaaSPAMaaaaaaaaSPAMaaaaaaa'
>>> s.replace( 'SPAM', 'EGGS', 1 )
'aaaaaEGGSaaaaaaaaSPAMaaaaaaa'
```

Optimizari

- Pentru a se evita generarea de stringuri noi la fiecare concatenare sau *replace()*, se poate face o conversie initiala la *list* – tip ce suporta modificari *in-place*:

```
>>> s = 'spamm'
>>> L = list( s )      #conversie la list
>>> L
['s', 'p', 'a', 'm', 'm']
>>> L[3] = 'x'         #modificari in-place
>>> L[4] = 'x'
>>> L
['s', 'p', 'a', 'x', 'x']
>>> s = ''.join( L ) #refacere string prin
                     join() cu delimitator
                     vid
>>> s
'spaxx'
```

- **Join(), in general:**

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

Scanari de text

- **Cu slicing pozitional (pe coloane):**

>>> **linie = 'aaa bbb ccc'**

>>> **camp1 = linie[:3]**

>>> **camp2 = linie[4:7]**

>>> **camp3 = linie[8:]**

>>> **camp1, camp3**

('aaa', 'ccc')

- **Cu split() – fara args, white space**

>>> **campuri = linie.split()**

>>> **campuri**

['aaa', 'bbb', 'ccc']

- **Cu delimitator de un caracter:**

>>> **linie = 'ion,student,18' #formatul CSV (Comma – Separated Values)**

>>> **linie.split(',') #rezultat *list***

['ion', 'student', '18']

- **Cu delimitator multicaracter:**

>>> linie = **"I'm theSPAMoperatorSPAMof my pocketSPAMcalculator!"**

>>> **linie.split('SPAM')**

["I'm the", 'operator', 'of my pocket', 'calculator!']

Alte utilizari

```
>>> linie = 'Framework pentru compunerea serviciilor!\n'
>>> linie.rstrip()                    #elimina white space in dreapta (' ', \t, \n)
'Framework pentru compunerea serviciilor!'
>>> linie.upper()                     #conversie la litere mari
'FRAMEWORK PENTRU COMPUNEREA SERVICIILOR!\n'
>>> linie.isalpha()  #este numai alfanumerica?
False
>>> linie.endswith( 'lor!' )          #test sfarsit de linie
False
>>> linie.startswith( 'Fram' )        #test inceput de linie
True
>>> sufix = 'lor!\n'
>>> linie[-len( sufix ):] == sufix    #test sfarsit cu slicing!
True
```

Formatarea *str*

- Permite multiple substitutii cu o singura trecere printr-un string
- Clasificare:
  - Expresii de formatare: **'....%s....' % (valori)**

    Bazate pe modelul *printf* din C
  - Metoda format(): **'....{}....'.format(valori)**

    Bazata pe *format*-ul din C#/.NET
- Ambele sisteme sunt echivalente

Expresii de formatare cu **%**

- Operatorul **%** este *overloaded*! (inseamna restul impartirii pentru *int*)

Sintaxa:

- string care contine marcaje **%<cod>**, ex. %d
- operatorul **%**
- obiect sau tuplu de obiecte ce vor inlocui marcajele din primul string (pozitional).

Exemple %

```
>>> '%d %s %g' % (33, 'spam', 3.14 )          #substitutii specifice, %d, %s, %g
'33 spam 3.14'
>>> '%s ... %s ... %s' % (33, 3.14, [1, 2, 3])    #orice obiect poate folosi %s
'33 ... 3.14 ... [1, 2, 3]'
```

cod	Semnificatie	cod	Semnificatie
s	String, cu str()	X	Ca x, cu litere mari
r	String, cu repr()	e	Float cu exponent
c	Caracter (int sau str)	E	Ca e, cu E mare
d	Numar in baza 10	f	Float
i	Intreg	F	Ca f, cu litere mari
u	Ca d	g	Float, e sau f
o	Intreg in baza 8	G	Ca g, cu litere mari
x	Intreg in baza 16	%	Chiar %, dublat

Sintaxa marcajului %

- Sintaxa marcajului %:

  **%*[*(cheie)*][*flags*][*latime*][*.precizie*]*cod

  unde:

  - (cheie) – pentru indexarea dictionarului din dreapta **%**
  - flags – ex: **–** aliniere stanga, **+** semn numere (e – la negative), **0** completare cu zerouri
  - latime – latime minima a campului
  - precizie – numar de zecimale dupa punctul zecimal

  Observatie: latime si precizie pot fi inlocuite cu ***** si substituite la executie cu urmatoarea valoare din dreapta **%**

Exemple avansate, %

```
>>> x = 1234
>>> 'ints: ...%d...%-6d...%06d' % (x, x, x)
'ints: ...1234...1234  ...001234'
>>> 'floats: %e | %f | %g' % (x, x, x)
'floats: 1.234568e+00 | 1.234568 | 1.23457'
>>> '%E' % x
'1.234568E+00'
>>> '%-6.2f | %06.2f | %+06.1f' % (x, x, x)
'1.23   | 001.23 | +001.2'
```

Dictionare cu %

- Dictionar cu doua chei, qty si fel, se vor substitui valorile asociate respectivelor chei, 3 si 'spam':

>>> **'De adaugat de %(qty)d ori %(fel)s' % { 'qty': 3, 'fel': 'spam' }**

'De adaugat de 3 ori spam'

- Cu functia predefinita *vars()* care returneaza un dictionar al tuturor variabilelor existente la momentul apelului sau:

>>> **fel = 'spam'**

>>> **qty = 3**

>>> **vars()**

{ ..... 'fel': 'spam', 'qty': 3}

>>> **'De adaugat de %(qty)d ori %(fel)s' % *vars()***

'De adaugat de 3 ori spam'

Formatare cu *.format()*

Metoda a ***str***:

- String cu rol de sablon, ce cuprinde
- Marcaje de substituire intre acolade **{}**
- Argumente: pozitionale, **{1}** sau bazate pe cuvinte cheie, **{fel}** ori cu ordine relativa.

```
>>> sablon = '{0}, {1} si {2}'
>>> sablon.format( 'spam', 'ham', 'eggs' )                    #pozitional
'spam, ham si eggs'
>>> sablon = '{logo}, {mezel} si {fel}'
>>> sablon.format( logo='spam', mezel='ham', fel='eggs' )     #cu cuvinte cheie
'spam, ham si eggs'
```

Exemple *.format()*

```
>>> sablon = '{logo}, {0} si {fel}'                    #combinat:
>>> sablon.format( 'ham', logo='spam', fel='eggs' )    #pozitional si chei
'spam, ham si eggs'
>>> sablon = '{}, {} si {}'
>>> sablon.format( 'spam', 'ham', 'eggs' )             #cu ordine relativa
'spam, ham si eggs'
```

- *format()* creeaza obiecte noi de tip *str* – deoarece *str* este fix (imuabil):

```
>>> x = '{logo}, {0} si {fel}'.format( 33, logo=3.14, fel=[1, 2] )   #x, un str nou
>>> x
'3.14, 33 si [1, 2]'
>>> x.split( ' si ' )
['3.14, 33', '[1, 2]']
```

Exemple...

```
>>> #Cu dictionar indexat pe cheie, [tip] si selectie de atribut, sys.platform:
>>> import sys
>>> '{1[tip]}-ul meu ruleaza {0.platform}'.format( sys, {'tip':'laptop'} ) #pozitional
'laptop-ul meu ruleaza win32'
>>> #La fel, dar cu cuvintele cheie s si m:
>>> '{m[tip]}-ul meu ruleaza {s.platform}'.format( s=sys, m={'tip':'laptop'} )
'laptop-ul meu ruleaza win32'
>>> #Pozitiile sunt numere nenegative, deci slicing se face in afara sablonului:
>>> L = list( 'spam' )
>>> L
['s', 'p', 'a', 'm']
>>> 'prima={0}, ultima litera={1}'.format( L[0], L[-1] )
'prima=s, ultima litera=m'
```

Sintaxa marcajului {}

```
>>> #Cu tuplul t, expandat la lista de argumente, *t:
>>> t = L[0], L[-1], L[1:3]                                  #t, tuple
>>> 'prima={0}, ultima litera={1}, mijlocul={2}'.format( *t )  #*t, expandare
"prima=s, ultima litera=m, mijlocul=['p', 'a']"
```

- Sintaxa marcajului {}:

  {camp component !flag :format}

  unde:

  - camp – numar sau cheie ce identifica un argument (poate lipsi)
  - component – selectie sau indexare, .*nume*, [*index*]
  - flag – incepe cu **!**, urmat de *r, s, a* pentru apel de *repr*, *str*, *ascii*
  - format – incepe cu **:**, are sintaxa:

    [[**fill**]**aliniere**][**semn**][#][**0**][**latime**][,][.**precizie**][**cod**]

Exemple avansate, *.format()*

```
>>> '{0:10} = {1:10}'.format('spam', 3.14)          #latime, 10
'spam       =       3.14'
>>> '{0:>10} = {1:<10}'.format('spam', 3.14)       #si aliniere, stanga <, dreapta >
'      spam = 3.14      '
>>> #selectie atribut, indexare dictionar construit cu dict
>>> '{0.platform:>10} = {1[tip]:<10}'.format( sys, dict(tip='laptop') )
'     win32 = laptop    '
>>> '{0:e}...{1:3e}...{2:2g}'.format( 3.14159, 3.14159, 3.14159 )       #float
'3.141590e+00...3.141590e+00...3.14159'
>>> '{0:f}...{1:.2f}...{2:06.2f}'.format( 3.14159, 3.14159, 3.14159 )
'3.141590...3.14...003.14'
>>> '{0:X}...{1:o}...{2:b}'.format( 255, 255, 255 )       #hex, oct, bin
'FF...377...11111111'
```

Exemple...

```
>>> #argumente dinamice, la executie, cu 4 zecimale:
>>> '{0:.{1}f}'.format( 1 / 3, 4 )
'0.3333'
>>> '%.*f' % (4, 1 / 3 )          #cu % si *
'0.3333'
```

- Cu functia predefinita *format()*:

```
>>> format( 3.14159, '.2f' )
'3.14'
>>>
>>> '{:.2f}'.format( 3.14159 )      #cu metoda .format()
'3.14'
>>> '%.2f' % 3.14159                #cu expresii cu operatorul %
'3.14'
```

Clasificari

Obiecte care suporta operatii asemanatoare:

- **Numere** – int, float, decimal, etc.
  - operatii: adunare, inmultire, etc.
- **Secvente** – str, list, tuple
  - operatii: indexare, slicing, concatenare
- **Asocieri** (mapari) – dict
  - operatii: indexare cu chei, etc.

Obiecte fixe/modificabile:

- **Imuabile** – numere, str, tuple, frozenset
- **Modificabile** – list, dict, set, bytearray

Sumar

- str
- ***list***
- dict

list – tipul lista

- Definitie: obiect modificabil, o colectie ordonata (secventa) de obiecte eterogene
- Proprietati:
  - colectie ordonata de obiecte arbitrare – sunt secvente (ordonate)
  - acces pozitional – indexare, slicing, concatenare
  - lungime variabila, eterogene – cuprind orice obiecte, suporta incluziunea

list...

- secventa modificabila – list poate fi modificat in-place.
- vector de referinte catre obiectele componente (analog cu C, array de pointers)

Listele se reprezinta cu paranteze patrate **[]**

-/-

list, expresii

Operatie	Semnificatie	Operatie	Semnificatie
L = []	Lista vida	L * 4	Repetitie
L = [1, 'a', 1.5, {}]	Lista cu 4 elem.	for x in L: print(x)	Iteratie
L = ['a', [1, 2]]	Sublista	3 in L	Apartenenta
L = list('spam')	Lista din iterabile	L.append(4)	Adaugare element
L = list(range(-3,3))	Idem	L.extend([4,5])	Adaugare lista
L[i]	Indexare	L.insert(i, x)	Inserare in poz. i
L[i][j]	Index de index	L.index(x)	Poz. lui x
L[i:j]	Slice(felie)	L.count(x)	Numar de aparitii
len(L)	Lungime	L.sort()	Sortare
L1 + L2	Concatenare	L.reverse()	Inversare a ordinii

list, operatii de baza

Operatie	Semnificatie	Operatie	Semnificatie
L.copy()	Copiere lista	L[i:j]=[]	Stergere slice
L.clear()	Stergere lista	L[i]=4	Asignare, index
L.pop(i)	Eliminare index i	L[i:j]=[4,5,6]	Asignare de slice
L.remove(x)	Eliminare obiect x	L=[x*2 for x in range(4)]	Colectie iterativa
del L[i]	Elim. index i	list(map(ord, 'spam'))	Mapare
del L[i:j]	Eliminare slice		

```
>>> len( [1, 2, 3] )          #Nr. de obiecte din lista
3
>>> [1, 2, 3] + [4, 5, 6]     #Concatenarea a doua liste
[1, 2, 3, 4, 5, 6]
>>> ['Hi!'] * 4               #Repetitie
['Hi!', 'Hi!', 'Hi!', 'Hi!']
```

Iteratii, colectii iterative

```
>>> str( [1, 2] ) + '34'     #str + str, nu mixati operanzii concatenarii!
'[1, 2]34'
>>> [1, 2] + list( '34' )    list + list
[1, 2, '3', '4']
```

- ***Iteratii:***

```
>>> for x in [1, 2, 3]: print( x, end=' ' )

1 2 3
```

- ***Colectii iterative:***

```
>>> rez = [c * 3 for c in 'spam']
>>> rez
['sss', 'ppp', 'aaa', 'mmm']
```

```
>>> #cu map:
>>> list( map( abs, range( -2, 3 ) ) )
[2, 1, 0, 1, 2]
```

Indexare, slicing, matrici

- Rezultatul indexarii este obiectul aflat in respectiva pozitie; *slicing* produce o lista:

```
>>> L = ['spam', 'Spam', 'SPAM']
>>> L[2]        #indexare
'SPAM'
>>> L[-2]
'Spam'
>>> L[1:]       #slicing, rezultat lista
['Spam', 'SPAM']
>>> matrice = [[1,2,3], [4,5,6], [7,8,9]]
>>> matrice[1]          #linia a 2 a
[4, 5, 6]
>>> matrice[1][1]   #linia a 2a, coloana a 2a
5
>>> matrice = [[1, 2, 3], #pe mai multe linii,
              [4, 5, 6],  #se poate, datorita
              [7, 8, 9]] #parantezelor []
>>>
```

Modificari, *list*

Asignari – in place:

- per index:

```
>>> L = ['spam', 'Spam', 'SPAM']
>>> L[1] = 'eggs'                #index
>>> L
['spam', 'eggs', 'SPAM']
```

- per slice = stergere + inserare:

```
>>> L[0:2] = ['ham', 'milk']    #slice
>>> L
['ham', 'milk', 'SPAM']
```

Slicing

Portiunile inlocuite/inlocuitoare pot fi de dimensiuni diferite, rezultand combinatiile:

- Stergere+insertie

```
>>> L = [1, 2, 3]
>>> L[1:2] = [4, 5]
>>> L
[1, 4, 5, 3]
```

- Stergere(fara insertie)

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

- Insertie(fara inlocuire)

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

Slicing...

Slicing – folosit pentru adaugare:

- La inceput

```
>>> L = [1]
>>> L[:0] = [2, 3, 4]          #slice vid la inceput :0

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

- La sfarsit

```
>>> L[len( L ):] = [5, 6, 7]   #slice vid la sfarsit len(L):

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

- La sfarsit, cu metoda *extend()*

```
>>> L.extend( [8, 9, 10] )
>>> L
[2, 3, 4, 1, 5, 6, 7, 8, 9, 10]
```

Metode, *list*

- Metode specifice listelor

```
>>> L = ['spam', 'ham']
>>> L.append( 'eggs' )        #adauga un obiect la sfarsit, in-place
>>> L
['spam', 'ham', 'eggs']
```

- Sortarea listelor

```
>>> L = ['ham', 'HAM', 'hAm']
>>> L.sort()    #majusculele sunt inaintea minusculelor in tabela ASCII
>>> L
['HAM', 'hAm', 'ham']
```

Sortare, *list*

```
>>> L = ['ham', 'HAM', 'hAm']
>>> L.sort( key=str.lower )          #cu argumentul key = str.lower(), aplicat on-the-fly
>>> L
['ham', 'HAM', 'hAm']
>>> L = ['ham', 'HAM', 'hAm']
>>> L.sort( key=str.lower, reverse=True )  #si cu argumentul  reverse = True
>>> L
['ham', 'HAM', 'hAm']
```

- Atentie, functia *sorted()* este aplicata unei liste deja transformate:

```
>>> L = ['ham', 'HAM', 'hAt']
>>> sorted( [x.lower() for x in L], reverse=True )
['hat', 'ham', 'ham']
```

Alte metode, *list*

```
>>> L = [1, 2, 3, 4, 5]
>>> L.pop()    #ultimul element, sters
5
>>> L
[1, 2, 3, 4]
>>> L.reverse()      #ordine, inversata
>>> L
[4, 3, 2, 1]
>>> list( reversed( L ) ) #cu reversed(),
                          functie predefinita
[1, 2, 3, 4]
```

- **Emularea unei stive cu append si pop**

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

Alte...

```
>>> L = ['spam', 'eggs', 'ham' ]
>>> L.index( 'eggs' ) #gaseste pozitia
1
>>> L.insert( 1, 'toast' ) #insereaza in
    pozitie
>>> L
['spam', 'toast', 'eggs', 'ham']
>>> L.remove( 'eggs' ) #sterge obiect
>>> L
['spam', 'toast', 'ham']
>>> L.pop( 1 ) #pop cu index specificat
'toast'
>>> L
['spam', 'ham']
>>> L.count( 'spam' ) #nr. aparitii obiect
1
```

- ***Operatorul del:***

```
>>> L = ['spam', 'eggs', 'ham' ]
>>> del L[0]   #sterge un obiect
>>> L
['eggs', 'ham']
>>> del L[1:]  #sterge un slice
>>> L
['eggs']
```

Sumar

- str
- list
- ***dict***

dict – tipul dictionar

- Definitie: obiect modificabil, o colectie neordonata de obiecte, accesibile prin chei (similar cu record, struct)
- Proprietati:
  - Obiectele componente se acceseaza prin chei – nu prin pozitie
  - Obiectele nu sunt ordonate – cheile fac o localizare simbolica, nu fizica

dict...

- *dict* contine obiecte de orice tip, incluse pe oricate nivele – o cheie are asociat un singur obiect, iar acelasi obiect poate apare sub chei diferite
- Maparea cheie – obiect este dinamica, iar operatii de tip concatenare sau slicing **nu** au sens pentru *dict*
- *dict* este implementat cu o tabela *hash*
  - cheile trebuie sa fie obiecte imuabile, hash-abile (ex. *str*, *int*, *tuple*, clase hash-abile)

dict, expresii

Operatie	Semnificatie
d = {}	Dictionar vid
d = {'nume': 'Ion', 'varsta': 44}	Dictionar cu 2 obiecte
d = { 'ceo': {'nume': 'Ion', 'varsta': 44}}	Incluziune
d = dict(nume='Ion', varsta=44)	Constructor, cuvinte cheie
d = dict([('nume', 'Ion'),('varsta',44)])	Constructor, perechi cheie/valoare
d = dict(zip(lista_chei, lista_valori))	Constructor, zip()
d = dict.fromkeys(['nume', 'varsta'])	Constructie cu lista de chei – valori *None*
d['nume']	Indexare cu cheie
d['ceo']['nume']	Indexare
'nume' in d	Apartenenta de cheie in dictionar

dict...

Operatie	Semnificatie
d.keys()	Metoda, toate cheile
d.values()	Toate valorile
d.items()	Toate tuplele cheie,valoare
d.copy()	Copie dictionar
d.clear()	Stergere dictionare
d.update(d2)	Actualizare, prin chei
d.get(cheie, default)	Returnare obiect, cu default sau None
d.pop(cheie, default)	Stergere obiect asociat cu cheie
d.setdefault(cheie, default)	Inserare, daca cheie absenta
d.popitem()	Returneaza & Sterge o pereche cheie/valoare

Operatii de baza, *dict*

Operatie	Semnificatie
len(d)	Numar de intrari
d[cheie] = 44	Adaugare/schimbare valori
del d[cheie]	Sterge intrarea cu cheia cheie
list(d.keys())	View de dictionar
d1.keys() & d2.keys()	View
d = {x: x*2 for x in range(7)}	Colectie iterabila, de tip dictionar

```
>>> d = {'spam': 2, 'ham':1, 'eggs': 3}      #dict cu trei intrari
>>> d['spam']            #obiect asociat cheii 'spam'
2
>>> len( d )             #numar de intrari in dictionar
3
>>> 'spam' in d          #este cheia 'spam' in dictionar?
True
```

Modificari in-place, *dict*

```
>>> list( d.keys() )        #lista cheilor din dict
['spam', 'ham', 'eggs']
```

- ***dict este modificabil, in-place:***

```
>>> d['ham'] = ['grill', 'bake', 'fry']      #schimbare ob. asoc. cu cheia 'ham'
>>> d
{'spam': 2, 'ham': ['grill', 'bake', 'fry'], 'eggs': 3}
>>> del d['eggs']                            #stergere intrare cu cheia 'eggs'
>>> d
{'spam': 2, 'ham': ['grill', 'bake', 'fry']}
>>> d['brunch'] = 'bacon'                    #adaugare intrare, cu cheia noua 'brunch'
>>> d
{'spam': 2, 'ham': ['grill', 'bake', 'fry'], 'brunch': 'bacon'}
```

Metode, *dict*

```
>>> d = {'spam': 2, 'ham':1, 'eggs': 3}
>>> list( d.values() )          #lista valorilor din dict
[2, 1, 3]
>>> list( d.items() )           #lista tuplelor cheie/valoare din dict
[('spam', 2), ('ham', 1), ('eggs', 3)]
>>> d.get( 'spam' )             #cheie existenta
2
>>> print( d.get( 'toast' ) )   #cheie lipsa
None
>>> d.get( 'toast', 99 )        #lipsa, dar cu valoare de default, 99
99
>>> d2 = { 'toast': 4, 'cake': 5}
>>> d.update( d2 )              #actualizare cu intrarile din d2
>>> d
{'spam': 2, 'ham': 1, 'eggs': 3, 'toast': 4, 'cake': 5}
```

Metode...

```
>>> d.pop( 'cake' )          #sterge intrare, returneaza valoarea stearsa
5
>>> d.pop( 'toast' )
4
>>> d
{'spam': 2, 'ham': 1, 'eggs': 3}
>>> d.pop( 'sugar', 88 )     #cheie inexistenta, returneaza valoare implicita, 88
88
```

Mapari valori / cheie

```
>>> d = { 'The Terminator': 1984,        #mai multe linii, se poate, cu acolade {}
          'Terminator 2: Judgment Day': 1991,
          'Terminator 3: Rise of the Machines': 2003 }
>>> k = 'The Terminator'
>>> d[k]        #mapare cheie ➔ valoare, normala
1984
>>> v = 1991
>>> [key for (key, value) in d.items() if value == v]    #mapare valoare ➔ cheie
['Terminator 2: Judgment Day']
>>> [key for key in d.keys() if d[key] == v]             #idem
['Terminator 2: Judgment Day']
```

- Unei valori ii pot corespunde mai multe chei.
- Unei chei, o singura valoare – eventual un obiect colectie (adica mai multe valori)

Liste, matrici cu *dict*

- ***Liste flexibile***

```
>>> d = {}
>>> d[99] = 'spam'        #cheie de tip int
>>> d[99]
'spam'
>>> d                     #pare o lista cu 100 de valori
{99: 'spam'}
```

- ***Matrici sparse ("goale")***

```
>>> m = {}
>>> m[(4, 5, 6)] = 88     #cheie de tip tuple
>>> m[(7, 8, 9)] = 99
>>> x=4; y=5; z=6
>>> m[(x, y, z)]
88
>>> m      #pare un vector tridimensional
{(4, 5, 6): 88, (7, 8, 9): 99}
```

Chei lipsa in *dict*

```
>>> if (1, 2, 3) in m:              #se evita cu teste, if
        print( m[(1, 2, 3)] )
    else:
        print( 0 )


0
>>> try:                            #se evita cu clauze try/except
        print( m[(1, 2, 3)] )
    except KeyError:
        print( 0 )


0
>>> m.get( (1, 2, 3), 0 )           #cu get() cu valoare implicita
0
```

Incluziune, *dict*

- *dict* cu rol de inregistrare (*record*):

```
>>> rec = {}
>>> rec['nume'] = 'Ion'
>>> rec['varsta'] = 44
>>> rec['job'] = 'inginer/director'
>>> print( rec['nume'] )
Ion
>>>
>>> rec = {'nume': 'Ion',
      'jobs': ['inginer', 'director'],
      'web': 'www.ion.org',
      'adresa': {'sector': 1, 'cod': 12345} }
```

```
>>> rec['nume']
'Ion'
>>> rec['jobs'][-1]  #indexari in adancime
'director'
>>> rec['adresa']['cod']
12345
```

Alte constructii de *dict*

- Cu constante de tip *dict:*

  { 'nume': 'Ion', 'varsta': 44 }

- Asignare de chei la rulare:

  d = {}; d['nume'] = 'Ion'; d['varsta'] = 44

- Cu chei numai de tip string:

  dict(nume='Ion', varsta=44)

- Cu secvente (liste) de tuple la rulare:

  dict([('nume', 'Ion'), ('varsta', 44)])

Alte...

- Cu functia predefinita *zip()*:

  dict(zip(keylist, valuelist))

- Cu metoda *fromkeys(iterabil,val=None)*:

  >>> ***dict.fromkeys(['a', 'b', 'c'], 0)***

  {'a': 0, 'b': 0, 'c': 0}

Colectii iterative, *dict*

```
>>> d = {k: v for (k, v) in zip( ['a', 'b', 'c'], [1, 2, 3] )}   #cu zip()
>>> d
{'a': 1, 'b': 2, 'c': 3}
>>> d = {x: x ** 2 for x in range(1, 5)}                        #cu range()
>>> d
{1: 1, 2: 4, 3: 9, 4: 16}
>>> d = {c: c * 4 for c in 'honey'}                             #cu parcurgere de string
>>> d
{'h': 'hhhh', 'o': 'oooo', 'n': 'nnnn', 'e': 'eeee', 'y': 'yyyy'}
>>> d = {c.lower(): c + '!' for c in ['spam', 'ham']}           #chei si valori din expresii
>>> d
{'spam': 'spam!', 'ham': 'ham!'}
>>> {x: 0 for x in ['a', 'b', 'c']}                             #pentru initializari de dict
{'a': 0, 'b': 0, 'c': 0}
```

View, *dict*

- keys(), values() si items() returneaza obiecte de tip **view** – care sunt iterabile:

```
>>> d = dict( a=1, b=2, c=3 )
>>> d
{'a': 1, 'b': 2, 'c': 3}
>>> i = d.items()    #obiect de tip view
>>> i
dict_items([('a', 1), ('b', 2), ('c', 3)])
>>> list( i )
[('a', 1), ('b', 2), ('c', 3)]
>>> for k in d: print( k ) #iterare directa,dict
a
b
c
```

- Modificari ulterioare ale *dict* se reflecta in toate view-urile existente:

```
>>> del d['b']      #intrarea cheii 'b' este stearsa din dict
>>> list( i )       #view-ul i este actualizat automat
[('a', 1), ('c', 3)]
```

dict view si *set*

- View-ul *keys* (si *values*, daca valorile sunt si ele imuabile) sunt asemanatoare cu tipul *set*

```
>>> k = d.keys()     #view
>>> k | {'x': 4 }    #reuniune cu dict, rezultat set
{'c', 'x', 'a'}
```

- *view*, *set*, *dict* sunt interschimbabile

```
>>> d.keys() & d.keys() #intersectie view/view
{'c', 'a'}
>>> d.keys() & {'a'} #intersectie view/set
{'a'}
>>> d.keys() & {'a': 1} #intersectie view/dict
{'a'}
>>> d.keys() | {'b', 'c', 'd' } #reuniune view/set
{'c', 'b', 'a', 'd'}
```

Sortari de chei, *dict*

```
>>> d = dict( a=1, b=2, c=3 )
>>> k = d.keys()     #view
>>> k.sort()         #eroare
AttributeError: 'dict_keys' object has no attribute 'sort'
>>> k = list( k )    #transformare in lista
>>> k.sort()         #acum da!
>>> k = d.keys()
>>> for x in sorted( k ): print( x, d[x], end= ';' )        #sorted() accepta orice iterabil

a 1;b 2;c 3;
>>> for x in sorted( d ): print( x, d[x], end= ';' )        #chiar d este iterabil!

a 1;b 2;c 3;
```