Programarea calculatoarelor si limbaje de programare 1

Atribuiri, if, while, for

Universitatea Politehnica din Bucureşti

Sumar

- ***Atribuiri***
- if
- while
- for

Instructiuni de atribuire

Au forma: **<dest> = <sursa>**

unde:

- <dest> este un nume de variabila sau atribut de obiect
- <sursa> este o expresie care calculeaza un obiect.

Proprietati:

- Creeaza referinte catre obiecte, nu copii ale acestora

Instr...

- Numele sunt create prin atribuire, nu sunt declarate in avans; in expresii sunt inlocuite cu valoarea referita
- Numele trebuie asignate inaite de a fi folosite in expresii
- Atribuirea apare implicit la importarea modulelor, definirea functiilor si claselor, in variabile ale instructiunii *for* sau argumentele de apel ale functiilor

Sintaxa atribuirilor

Atribuire	Semnificatie
s = 'spam'	Clasica
s1, s2 = 'spam', 'ham'	De tuple, pozitionala: s1='spam', s2='ham'
[x, y] = ['spam', 'ham']	De liste, pozitionala: x='spam', y='ham'
a,b,c,d = 'spam'	De secvente: a='s', b='p', c='a', d='m'
a, *b = 'spam'	Cu despachetare: a='s', b=['p', 'a', 'm']
s1 = s2 = 'spam'	Cu destinatie multipla: s2='spam', s1 = s2 ('spam')
i += 33	Prescurtata, i = i + 33

- Atribuirea clasica asociaza un nume unui singur obiect

Atribuiri de secvente

- Atribuire intre tuple sau liste:

```
>>> x = 1
>>> y = 2
>>> a, b = x, y  #tuple, paranteze() omise
>>> a, b
(1, 2)
>>> [c, d] = [x, y]  #liste
>>> c, d
(1, 2)
```

- Interschimbare de valori, cu atribuiri de tuple:

```
>>> x, y = y, x  #interschimbare cu tuple!
>>> x, y
(2, 1)
```

Atribuiri...

- Generalizare, orice iterabil este posibil in dreapta simbolului **=**

```
>>> [a, b, c] = (1, 2, 3)          #tuplu
>>> a, b
(1, 2)
>>> (a, b, c) = 'ABC'              #string
>>> a, b
('A', 'B')
```

- Numarul de elemente din sursa trebuie sa fie egal cu cel din destinatie:

```
>>> a, b = 'spam'
ValueError: too many values to unpack (expected 2)
```

Atribuiri...

- Slicing-ul poate rezolva nepotrivirea:

```
>>> s = 'spam'
>>> a, b, c = s[0], s[1], s[2:]          #indexare si slicing
>>> a, b, c
('s', 'p', 'am')
>>> a, b, c = list( s[:2] ) + [s[2:]]    #slicing si concatenare
>>> a, b, c
('s', 'p', 'am')
>>> a, b = s[:2]                         #idem, mai simplu
>>> c = s[2:]
>>> a, b, c
('s', 'p', 'am')
```

Atribuiri...

- Cu secvente incluse:

```
>>> ((a, b), c) = ('sp', 'am')          #atribuire bazata pe structura si pozitie
>>> a, b, c
('s', 'p', 'am')
```

- Cu *range()*:

```
>>> a, b, c = range( 3 )                >>> list( range( 3 ) )
>>> a, b                                [0, 1, 2]
(0, 1)
```

- Tehnica: prefix, rest secventa:

```
>>> L = [1, 2, 3]
>>> while L:                            1 [2, 3]
        prefix, L = L[0], L[1:]         2 [3]
        print( prefix, L )              3 []
```

Despachetare de secvente

- Se face cu ***x** in secventa destinatie:

```
>>> seq = [1, 2, 3, 4]      #lista
>>> a, b = seq              #eroare
ValueError: too many values to unpack (expected 2)
>>> a, *b = seq             #corect
>>> b
[2, 3, 4]
```

- ***x** poate fi plasat oriunde in destinatie si **x** devine **lista** tuturor elementelor ramase neatribuite:

```
>>> *a, b, c = seq
>>> a
[1, 2]
```

- *Orice secventa:*

```
>>> a, *b, c = 'spam'
>>> a, b, c
('s', ['p', 'a'], 'm')
>>> a, *b, c = range( 4 )
>>> a, b, c
(0, [1, 2], 3)
```

Despachetare...

Reguli de despachetare:

- Mai mult de un singur ***x** – eroare!
- Prea multe, putine destinatii fara ***x** – eroare!
- ***x** nu este intr-o lista sau tuplu – eroare

  >>> ***x, = seq #corect, tuplu cu un singur element**

- **x** este intodeauna **o lista**:

  >>> **a, b, c, *x = [1, 2, 3, 4]**

  >>> **print(a, b, c, x)**

  1 2 3 **[4]**

- **x** poate primi valoarea **[]** (lista vida):

  >>> **a, b, c, *x, d = [1, 2, 3, 4]**

  >>> **print(a, b, c, x, d)**

  1 2 3 **[]** 4

Atribuiri cu destinatie multipla

```
>>> a = b = c = 'spam'
>>> a, b, c
('spam', 'spam', 'spam')
```

```
>>> c = 'spam'      #echivalent
>>> b = c
>>> a = b
```

- Sunt folosite la initializari de contoare:

```
>>> a = b = 0
>>> b = b + 1
>>> a, b  #doar b este modificat fiindca numerele nu suporta modificari in-place
(0, 1)
```

- Cazul referintelor partajate:

```
>>> a = b = []  #acelasi obiect, modificabil
>>> b.append( 33 )
>>> a, b  #aceeasi referinta
([33], [33])
```

```
>>> a, b = [], []  #obiecte diferite!
>>> b.append( 33 )
>>> a, b  #referinte diferite!
([], [33])
```

Atribuiri prescurtate

x += y	**x &= y**	**x -= y**	**x |= y**
x *= y	**x ^= y**	**x /= y**	**x >>= y**
x %= y	**x <<= y**	**x **= y**	**x //= y**

>>> **x = x + y**

>>> **x += y #prescurtat!**

Avantaje:

- Scriere prescurtata!
- **x** (poate fi o expresie complicata) este evaluat o singura data ➔viteza
- Se efectueaza schimbari in-place, daca sunt posibile (cu *.extend()*), in loc de concatenari ➔viteza

Atribuiri...

```
>>> L = [1, 2]
>>> L = L + [3]  #concatenare, mai lenta
>>> L
[1, 2, 3]
>>> L.append( 4 )  #in-place, un element
>>> L
[1, 2, 3, 4]
>>> L += [5, 6]  #in-place, rapid
>>> L
[1, 2, 3, 4, 5, 6]
>>> L.extend( [7, 8] )  #in-place, rapid
>>> L
[1, 2, 3, 4, 5, 6, 7, 8]
```

- **+=** suporta orice secventa, ca .extend()

```
>>> L = []
>>> L += 'spam'
>>> L
['s', 'p', 'a', 'm']
```

- concatenarea, nu:

```
>>> L = L + 'spam'
TypeError: can only concatenate list (not "str") to list
```

Nume de variabile in Python

Identificatorii sunt formati din:

- _ sau **litera** urmata de oricate **litere**, **cifre** sau _
- literele mici sunt diferite de majuscule
- cuvintele rezervate sunt interzise:

False	class	finally	is	return	None	continue
for	import	try	if	def	from	nonlocal
while	and	del	as	not	with	global
except	True	or	in	assert	else	lambda
pass	yield	raise	elif	break		

Conventii de denumire

- Nume care incep cu un singur _ (_x) nu sunt importate de *from nume import **
- __**x**__ sunt nume de sistem
- __**x** sunt nume locale unei clase
- _ este rezultatul ultimei expresii – interactiv
- Numele de clase incep cu o majuscula
- Numele de module – minuscule

Expresii

Se folosesc pentru:

- Apel de functii sau metode – proceduri care nu returneaza rezultate (ci ***None***)
- Afisare – print interactiv
- Instructiunile nu pot fi folosite ca expresii! (ex. atribuirea)

Expresie	Semnificatie
spam(eggs, ham)	Apel de functie
spam.ham(eggs)	Apel de metoda
spam	Afisare variabile
print(a, b, c, sep='')	print() in 3.X
yield x ** 2	Instructiune *yield*

print() in 3.X

```
>>> L = [1, 2]
>>> L.append( 3 ) ←Corect!
>>> L
[1, 2, 3]
```

Gresit→

```
>>> #L, pierdut:
>>> L = L.append( 4 )
>>> print( L )
None
```

- *print()* converteste/formateaza obiecte la reprezentarea lor textuala (cu functia predefinita *str()*) si afiseaza la *sys.stdout* (sau alt stream)
- Sintaxa (cu argumente de tip cuvinte cheie):

print([ob,...][,sep= ' '][,end= '\n'][,file=sys.stdout][,flush=False])

print...

```
>>> print()  #linie noua

>>> x = 'spam'; y = 99; z = ['eggs']
>>> print( x, y, z)  #implicit
spam 99 ['eggs']
>>>
>>> print( x, y, z, sep='')  #fara separator
spam99['eggs']
>>> print( x, y, z, sep=', ') #cu alt separator
spam, 99, ['eggs']
>>> #fara trecere la linie noua:
>>> print( x, y, z, end=''); print( x, y, z)
spam 99 ['eggs']spam 99 ['eggs']
```

```
>>> #ordinea cuvintelor cheie nu conteaza:
>>> print( x, y, z, end='!\n', sep='...')
spam...99...['eggs']!
>>> #scriere in fisier:
>>> print( x,y,z,sep='...',
          file=open('date.txt', 'wt'))
>>> print( open('date.txt', 'rt').read())
spam...99...['eggs']
>>> #Preformatare:
>>> print( '%s: %-.4f, %05d' %
          ('Rezultat', 3.14159, 33) )
Rezultat: 3.1416, 00033
```

Redirectare output

- Printare via *sys.stdout*:

```
>>> import sys
>>> sys.stdout.write('Hello, World!\n')
Hello, World!
14
```

- Redirectare manuala, cu salvare/restaurare sys.stdout:

```
>>> tmp = sys.stdout  #salvare
>>> sys.stdout = open('log.txt', 'at')
>>> print('spam'); print(1, 2, 3)
>>> sys.stdout.close()
>>> sys.stdout = tmp  #restaurare
>>> print('Hello!')
Hello!
```

```
>>> print(open('log.txt', 'rt').read())
spam
1 2 3
```

Redirectare...

- Redirectare automata, afisare fisier&ecran:

```
>>> log = open( 'log.txt', 'at' )
>>> print(1, 2, 3, file=log)
>>> print(4, 5, 6)
4 5 6
>>> log.close()
>>> print(open('log.txt', 'rt').read())
1 2 3
```

- Redirectare la *sys.stderr*:

```
>>> print( 'Eroare!' * 2, file=sys.stderr )
Eroare!Eroare!
```

- Alte redirectari:
  - Ale lui *sys.stdin*, pentru citire din fisier
  - Catre obiecte care implementeaza metodele *write()*, respectiv *read()*
  - La nivelul sistemului de operare:
    python script.py <fisieri >fisiero 2>&1

Sumar

- Atribuiri
- ***if***
- while
- for

Instructiunea *if*

- Este o instructiune compusa care alege intre alternative de executie
- Sintaxa:

```
if test1:                 #test if
    instructiuni          #bloc instr. pt. ramura if
elif test2:               #elif (oricate) este optional
    instructiuni2         #bloc instr. pt. elif
else:                     #else este optional
    instructiuni3         #bloc pt. else
```

Exemple *if*

- Numai *if* este obligatoriu:

```
>>> if True:
        print( 'Adevarat' )

Adevarat
```

```
>>> if not True:
        print( 'Adevarat' )
    else:
        print( 'Neadevarat' )

Neadevarat
```

- Alegerea multipla (nu exista switch sau case):

```
>>> aleg = 'spam'
>>> if aleg == 'spam':
        print( 1.25 )
    elif aleg == 'ham':
        print( 1.99 )
    elif aleg == 'eggs':
        print( 0.99 )
    else:
        print( 'Nu exista', aleg )

1.25
```

Exemple...

- Alegerea multipla cu *dict* si metoda *get()* pt. default:

```
>>> posibil = {'spam': 1.25,
               'ham': 1.99,
               'eggs': 0.99}
>>> print( posibil.get( aleg, 'Nu exista ' + aleg ) )
1.25
```

- Cazul implicit cu *try/except*:

```
>>> aleg = 'bacon'
>>> try:                                    Nu exista bacon
        print( posibil[ aleg ] )
    except KeyError:
        print( 'Nu exista', aleg )
```

Sintaxa Python

- Instructiunile se executa succesiv – cu exceptia celor care altereaza fluxul executiei (if, while, for, break, continue...)
- Limitele blocurilor se determina prin indentare/aliniere
- Instructiunile se termina la sfarsitul liniei
- Instructiunile compuse au antet incheiat cu ***:*** urmat de un bloc indentat

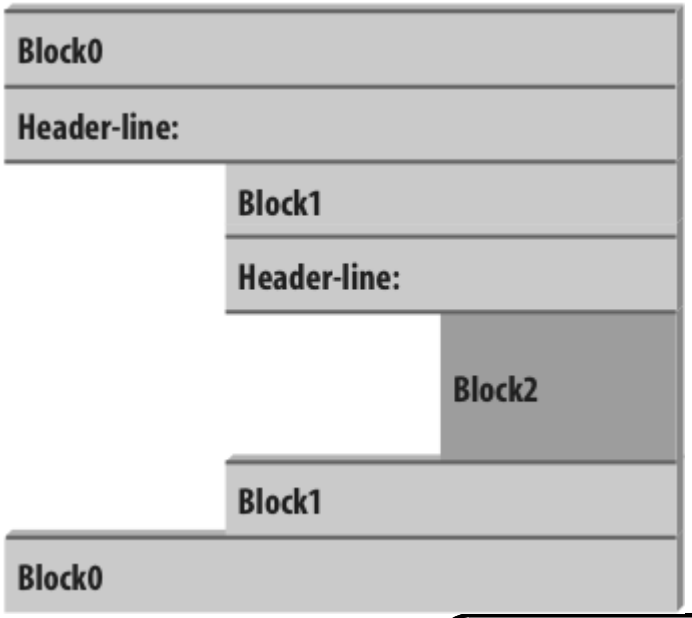
Sintaxa...

- Liniile albe, comentariile (#...) sunt ignorate
- *docstrings* sunt retinute si pot fi afisate cu PyDoc
- Indentarile neasteptate sau inegale sunt erori
- Amestecul de spatiu si tab la indentare este nerecomandabil

Blocuri incluse

- Blocurile incluse se aliniaza spre dreapta:

```
x = 1
if x:
    y = 2
    if y:
        print('Block2')
    print('Block1')
print('Block0')
```



Instructiuni multilinie

- Instructiunile se pot extinde pe mai multe linii daca sunt intre paranteze – **(), [], {}**
- Idem, daca se termina cu ****
- Stringurile cu triplu apostrof se pot scrie pe mai multe randuri
- Cu **;** se pot separa instructiuni simple pe aceeasi linie

True, *False* si testele logice

- Toate obiectele sunt fie adevarate, fie false
- Un obiect numeric nenul sau obiect nevid este adevarat
- Numerele egale cu zero, obiectele vide si *None* sunt false
- Comparatiile si testele de egalitate valorica se aplica recursiv pe structuri de date

True...

- Comparatiile si testele de egalitate returneaza *True* sau *False*
- Operatorii ***and*** si ***or*** returneaza **operanzi** – obiecte, adevarate sau false
- ***not* X** – returneaza *True* sau *False*
- Testele logice se incheie de indata ce rezultatul este deja cunoscut – pot ramane parti neevaluate!

True...

```
>>> 2 < 3, 3 < 2  #comparatii ➔ True/False
(True, False)
>>>
>>> 2 or 3, 3 or 2  #or returneaza obiecte
(2, 3)
>>>
>>> [] or 3
3
>>> [] or {}
{}
```

```
>>> 2 and 3, 3 and 2  #and, idem
(3, 2)
>>> [] and {}
[]
>>> 3 and []
[]
>>> # Evaluarea este scurtcircuitata!
```

Expresie ternara, *if*

A = Y if X else Z

- Este echivalenta cu instructiunea:

```
if X:
        A = Y
else:
        A = Z
```

- Expresia:

  A = ((X and Y) or Z)

- este echivalenta numai daca obiectul *Y* are valoarea booleana True!

```
>>> A = 't' if 'spam' else
    'f'
>>> A
't'
>>> A = 't' if '' else 'f'
>>> A
'f'
```

- Expresia:

  A = [Z, Y][bool(X)]

- este echivalenta, dar fara scurtcircuitare!

  si Z si Y sunt evaluate

filter, any, all

```
>>> L = [1, 0, 2, 0, 'spam', '', 'ham', []]  #o lista
>>> #functia predefinita filter() aplica bool() spre a
    selecta elementele adevarate din L:
>>> list( filter(bool, L) )
[1, 2, 'spam', 'ham']
>>> [x for x in L if x]  #colectie iterativa, acelasi efect!
[1, 2, 'spam', 'ham']
>>> any( L ), all( L )  #test de adevar cu any, all
(True, False)
```

Sumar

- Atribuiri
- if
- ***while***
- for

Instructiunea ***while***

- Repeta un bloc (indentat) de instructiuni cata vreme testul din antetul sau este adevarat
- Sintaxa:

```
while test:          #test initial
    instructiuni     #corp/bloc repetat
else:                #else este optional
    instructiuni     #executate daca NU s-a iesit
                         cu break
```

Exemple ***while***

- Repetitie infinita:

```
>>> while True:
        print( 'Type Ctrl-C to stop me!' )
```

- Parcurgere string:

```
>>> x = 'spam'
>>> while x:
        print(x, end=' ')
        x = x[1:]  #slicing, primul caracter este eliminat

spam pam am m
```

Exemple...

- Numarare de la ***a*** la ***b*** (exclusiv):

```
>>> a = 0; b = 10
>>> while a < b:
        print( a, end=' ' )
        a += 1

0 1 2 3 4 5 6 7 8 9
```

- Simulare *executa pana cand*, cu break:

```
>>> while True:
        ...Instructiuni...          #instructiuni de repetat
        if testDEiesire(): break    #test final
```

break, continue, pass, else

- ***break*** – incheie o instructiune repetitiva
- ***continue*** – reia executia de la pasul urmator
- ***pass*** – nu executa nimic!
- ***else: bloc*** – blocul de instructiuni este executat doar daca instructiunea repetitiva s-a incheiat (normal), fara a se iesi cu *break*.

pass

```
>>> while True: pass       #oprit cu Ctrl-C
KeyboardInterrupt
```

- Se foloseste pentru: ignorarea exceptiilor; crearea de clase “goale”, doar cu atribute (de tip record/struct); amanarea scrierii corpului functiilor
- **...** trei puncte (aka Ellipsis) au acelasi rol ca instructiunea *pass:*

```
>>> x = ... ; x
Ellipsis
```

continue

- Instructiunea reia ciclul de la testul initial:

```
>>> x = 10                          #program care afiseaza numere pare
>>> while x:
        x -= 1                      #decrementare
        if x % 2 != 0: continue     #daca restul impartirii la 2 este 1 ➔ reia ciclul
        print(x, end=' ')           #afisare numar par (restul impartirii a fost 0)

8 6 4 2 0
>>> while x:                        #versiune mai clara (fara continue)
        x -= 1
        if x % 2 == 0:
            print(x, end=' ')

8 6 4 2 0
```

break

- Produce iesirea imediata din ciclu; cu *break* se poate evita ramura *else:* a testelor:

```
>>> while True:
        nume = input( 'Numele: ' )
        if nume == 'stop': break
        ani = input( 'Varsta: ' )
        print( 'Hello', nume, '==>', int( ani ) * 2)
```

```
Numele: Ion
Varsta: 20
Hello Ion ==> 40
Numele: stop
>>>
```

Cicluri cu ***else:***

- Este **y** un numar prim?

```
>>> x = y // 2
>>> while x > 1:
        if y % x == 0:                  #restul impartirii este zero, deci divizibil
            print( y, 'se divide la', x )
            break                       #break sare dincolo de else: ...
        x -= 1                          #decrementare x
    else:                               #executie la iesirea normala, fara break,
        print( y, 'este prim' )         #cand x ajunge 1.
```

- Clauza *else:* se executa si cand corpul ciclului nu se executa niciodata

Cicluri...

- Codificare eleganta, fara variabile suplimentare, folosind *break/else:*

```
>>> while x:
        if conditie( x[0] ):
            print( 'Gasit!' )
            break
        x = x[1:]                    #avans, rest (lista) cu slicing
    else:
        print( 'Nu exista...' )
```

Sumar

- Atribuiri
- if
- while
- ***for***

Instructiunea ***for***

- Este un iterator pentru secvente (ordonate) sau alte obiecte iterabile
- Sintaxa:

```
for x in obiect:        #x parcurge elem. obiect
    instructiuni        #corp repetat, foloseste x
else:                   #else optional
    instructiuni        #executate daca NU s-a
                            iesit cu break
```

Exemple *for*

- Parcurgere lista:

```
>>> for x in ['spam', 'eggs', 'ham']:
        print(x, end=' ')

spam eggs ham
```

```
>>> x
'ham'
```

- Suma si produsul elementelor dintr-o lista:

```
>>> sum = 0; prod = 1 #elem. neutru
>>> for x in [1, 2, 3, 4]:
        sum += x        #adunare
        prod *= x       #produs
```

```
>>> sum, prod
(10, 24)
```

- Alte secvente – *str*:

```
>>> s = 'Python'
>>> for x in s:
        print( x, end=' ' )

P y t h o n
```

Tuple si *for*

```
>>> L = [(1, 2), (3, 4), (5, 6)]                #Lista de tuple
>>> for (a, b) in L:                            #atribuire de tuple
        print(a, b, end=';')
```

1 2;3 4;5 6;

```
>>> for pereche in L:
        a, b = pereche                          #atribuire “manuala” de tuple
        print(a, b, end=';')
```

1 2;3 4;5 6;

```
>>> d = {'a': 1, 'b': 2, 'c': 3}               #Dictionar
>>> for (cheie, valoare) in d.items():         #Iterare peste view-ul items() – tuple cheie/val
        print( cheie, '=>', valoare, end=';' )
```

a => 1;b => 2;c => 3;

Tuple...

```
>>> for cheie in d:                #iterare directa pe (cheile din) dictionar – keys()
        print( cheie, '=>', d[cheie], end=';' )
```

```
a => 1;b => 2;c => 3;
>>> for ((a, b), c) in [((1, 2), 3),((4, 5), 6)]:    #secvente de secvente, incluse
        print(a, b, c, end=';')
```

```
1 2 3;4 5 6;
>>> for ((a, b), c) in [((1, 2), 3),('XY', 6)]:      #orice secventa, cu atribuire de secvente
        print(a, b, c, end=';')
```

```
1 2 3;X Y 6;
```

Atribuire despachetata si *for*

```
>>> for (a, *b, c) in [(1, 2, 3, 4),(5, 6, 7, 8)]:#cu atribuire despachetata, list ➔ *b
        print(a, b, c)

1 [2, 3] 4
5 [6, 7] 8

>>> for tpl in [(1, 2, 3, 4),(5, 6, 7, 8)]:
        a, b, c = tpl[0], tpl[1:3], tpl[3]      #slicing ➔ tip specific, aici tuplu (nu list)
        print(a, b, c)

1 (2, 3) 4
5 (6, 7) 8
```

for in *for*

- Cautare in doua liste:

```
>>> car_cu_fan = ['aaa', 111, (4, 5), 2.7]
>>> ace = [(4, 5), 3.14]
>>> for a in ace:#pt. fiecare ac, cauta
        for p in car_cu_fan:
            if p == a:
                print( a, 'a fost gasit' )
                break
        else:
            print( a, 'nu a fost gasit' )
```

```
(4, 5) a fost gasit
3.14 nu a fost gasit
```

```
>>> for a in ace:  #Mult mai simplu!
        if a in car_cu_fan:
            print( a, 'a fost gasit' )
        else:
            print( a, 'nu a fost gasit' )
```

```
(4, 5) a fost gasit
3.14 nu a fost gasit
```

for...

- Generalizare, secvente:

```
>>> seq1 = 'spam'
>>> seq2 = 'scam'
>>> [x for x in seq1 if x in seq2]
['s', 'a', 'm']
```

Citire fisiere cu **for**

```
>>> f = open( 'date.bin', 'rb' )                    #fisier binar, citit pe blocuri
>>> while True:
        bl = f.read( 8 )
        if not bl: break                            #test de sfarsit de fisier
        print( bl )

>>> for linie in open( 'date.txt', 'rt' ).readlines():      #citire linie cu linie
        print( linie.rstrip() )

>>> for linie in open( 'date.txt', 'rt' ):                  #cu iteratori, cel mai bine!
        print( linie.rstrip() )

>>> #Totusi, cu reversed/readlines pentru inversarea ordinii:
>>> for linie in reversed( open( 'date.txt', 'rt' ).readlines() ): ...
```

Tehnici de iterare

- *for* este preferabil
- Cu *range()* se pot genera valori pentru indexarea unui *for*
- Cu *zip()* se obtin tuple la traversarea in paralel a secventelor multiple
- Cu *enumerate()* se acceseaza atat indexuri cat si valorile corespunzatoare dintr-un iterabil

range & for

```
>>> list( range( -5, 5 ) )      #ascendent, pas 1 implicit
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
>>> list( range( 5, -5, -1 ) ) #descendent
[5, 4, 3, 2, 1, 0, -1, -2, -3, -4]
>>> for i in range( 3 ):        #repetitie de un numar dat de ori
        print( i, 'pythons' )

0 pythons
1 pythons
2 pythons
```

Parcurgere de secvente

```
>>> s = 'spam'
>>> for x in s: print( x, end=' ' )                    #cu for, iterativ, cel mai bine!

s p a m
>>> i = 0
>>> while i < len( s ):                                #cu while, manual
        print( s[i], end=' ' )
        i += 1

s p a m
>>> for i in range( len( s ) ): print( s[i], end=' ' )  #cu for/range

s p a m
```

Amestecari/range/for

```
>>> for i in range( len( s ) ):
        s = s[1:] + s[:1]      #rest + cap
        print( s, end=' ' )

pams amsp mspa spam
```

```
>>> for i in range( len( s ) ):
        x = s[i:] + s[:i]      #rest + fata
        print( x, end=' ' )

spam pams amsp mspa
```

```
>>> L = [1, 2, 3]             #orice secvente, aceeasi tehnica
>>> for i in range( len( L ) ):
        x = L[i:] + L[:i]
        print( x, end=' ' )

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

range vs. slice

- *range()* salveaza memorie, slicing-ul face copii – conteaza doar pentru stringuri foarte mari

```
>>> s = 'abcdefghijk'
>>> for i in range( 0, len( s ), 2 ): print( s[i], end=' ' )

a c e g i k

>>> for c in s[::2]: print( c, end=' ' )    #slicing, copiere, cu pasul 2

a c e g i k
```

Modificari de liste

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

>>> L
[2, 3, 4, 5]
```

```
>>> i = 0
>>> while i < len( L ):    #cu while
        L[i] += 1
        i += 1

>>> L
[3, 4, 5, 6]
```

```
>>> #Elegant, cu colectii iterative:
>>> L = [x + 1 for x in L]      #dar nu este modificare in-place!
>>> L
[4, 5, 6, 7]
```

Traversari in paralel cu *zip*

```
>>> L1 = [1, 2, 3, 4]
>>> L2 = [5, 6, 7, 8]
>>> list( zip( L1, L2 ) )   #aducere aminte
[(1, 5), (2, 6), (3, 7), (4, 8)]
>>> for (x, y) in zip( L1, L2 ): print( x, y, '--', x + y, end='; ' )

1 5 -- 6; 2 6 -- 8; 3 7 -- 10; 4 8 -- 12;
```

- Observatie: zip trunchiaza tuplele la lungimea secventei celei mai scurte:

```
>>> s1 = 'abc'
>>> s2 = 'xyz1234567'
>>> list( zip( s1, s2 ) )
[('a', 'x'), ('b', 'y'), ('c', 'z')]
```

Dictionare cu *zip*

- Dictionare create dinamic, la executie:

```
>>> chei = ['spam', 'eggs', 'ham' ]
>>> valori = [1, 2, 3]
>>> d1 = {}
>>> for (c, v) in zip( chei, valori ):
        d1[c] = v
>>> d1
{'spam': 1, 'eggs': 2, 'ham': 3}
>>> #fara for!
>>> d2 = dict( zip( chei, valori ) )
>>> d2
{'spam': 1, 'eggs': 2, 'ham': 3}
```

- Cu colectii iterative:

```
>>> {c: v for (c, v) in zip( chei, valori )}
{'spam': 1, 'eggs': 2, 'ham': 3}
```

enumerate() si for

- Manual:

```
>>> s = 'spam'                          s este in pozitia 0
>>> poz = 0                             p este in pozitia 1
>>> for c in s:                         a este in pozitia 2
        print( c, 'este in pozitia', poz )   m este in pozitia 3
        poz += 1
```

- Cu enumerate() (care returneaza un generator):

```
>>> for (poz, c) in enumerate( s ):     s este in pozitia 0
        print( c, 'este in pozitia', poz )   p este in pozitia 1
                                             a este in pozitia 2
                                             m este in pozitia 3
```

popen() si for

- *os.popen()* (pipe open) permite accesarea stream-urilor comenzii executate:

```
>>> import os
>>> for lin in os.popen( 'systeminfo' ):      #comanda systeminfo
        parts = lin.split( ':' )
        if parts and parts[0].lower() == 'system type':
            print( parts[1].strip() )          #strip() elimina white space la
                                                 inceput si sfarsit
            break
```

x64-based PC

urlopen si for

- Cu *urlopen()* se pot citi sursele paginilor web:

```
>>> from urllib.request import urlopen
>>> for lin in urlopen( 'http://bbftppro.myftp.org' ):
        print( lin )

b'<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">\n'
b'<html>\n'
b'\n'
b'<head>\n'
...
```