

Complexitate operații secvențe



Complexitate operații

- ▶ **list - intern – vector**
- ▶ **set - tabel dispersie**
- ▶ **dict - tabel dispersie**

<https://wiki.python.org/moin/TimeComplexity>

Complexitate operații

► list

Operation	Average Case	
Copy	$O(n)$	$O(n)$
Append	$O(1)$	$O(1)$
Pop last	$O(1)$	$O(1)$
Pop(i)	$O(n)$	$O(n)$
Insert	$O(n)$	$O(n)$
Get Item	$O(1)$	$O(1)$
Set Item	$O(1)$	$O(1)$
Delete Item	$O(n)$	$O(n)$
Iteration	$O(n)$	$O(n)$
Get Slice	$O(k)$	$O(k)$
Del Slice	$O(n)$	$O(n)$
Set Slice	$O(k+n)$	$O(k+n)$
Extend	$O(k)$	$O(k)$
Sort	$O(n \log n)$	$O(n \log n)$
Multiply	$O(nk)$	$O(nk)$
x in s	$O(n)$	
min(s), max(s)	$O(n)$	
Get Length	$O(1)$	$O(1)$

Complexitate operații

► dict

Operation	Average Case	
k in d	$O(1)$	$O(n)$
Copy	$O(n)$	$O(n)$
Get Item	$O(1)$	$O(n)$
Set Item	$O(1)$	$O(n)$
Delete Item	$O(1)$	$O(n)$
Iteration	$O(n)$	$O(n)$

Complexitate operații

► set

Operation	Average case	
$x \in s$	$O(1)$	$O(n)$
Union $s \cup t$	$O(\text{len}(s) + \text{len}(t))$	
Intersection $s \cap t$	$O(\min(\text{len}(s), \text{len}(t)))$	$O(\text{len}(s) * \text{len}(t))$
Difference $s - t$	$O(\text{len}(s))$	
Symmetric Difference	$O(\text{len}(s))$	$O(\text{len}(s) * \text{len}(t))$

Structuri de date – implementări Python



Stivă

- ▶ list

Coadă

- ▶ list – **lent pop NU**

- ▶ `collections.deque`

<https://docs.python.org/3.9/library/collections.html?highlight=deque#collections.deque>

- ▶ `queue.Queue` – **sincronizata, mai lenta**

<https://docs.python.org/3/library/queue.html>



Coada de prioritati

- ▶ Operația fundamentală– extragere minim/maxim
- ▶ `heapq`: heap (binar)

<https://docs.python.org/3/library/heapq.html>

- ▶ `queue.PriorityQueue`: – **sincronizata, mai lenta**

```
import queue
#from queue import PriorityQueue,Queue
q=queue.Queue()
q.put(10)
q.put(2)
q.put(7)
while q.qsize()>0:
    print(q.get())
q=queue.PriorityQueue()
q.put(10)
q.put(2)
q.put(7)
while q.qsize()>0:
    print(q.get()).
```

Funcții

Funcții

```
def nume_funcție(parametrii)    #Antet  
    corp_instrucțiuni
```

- ▶ Parametri formali vs actuali
- ▶ Număr fix de parametri / număr variabil de parametri

Funcții

```
#parametri formali
```

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

```
    while x!=y:
```

```
        if x<y:
```

```
            y = y-x
```

```
        else:
```

```
            x = x-y
```

```
    return x
```

```
x = f(15,25) #parametri actuali
```

```
print(x)
```



Funcții

- ▶ Valoare returnată – cu **return**
 - returnează None altfel
- ▶ Se pot returna mai multe valori (împachetate implicit într-un tuplu) de orice tip

Funcții

- ▶ Dacă funcția nu are return – întoarce None

```
def f():  
    print("nu returnez nimic, de fapt None")
```

```
x = f()  
print(x)
```

```
ls = [4,2,1]
```

```
#ls.sort()= modifica ls, nu il returneaza => returneaza  
None
```

```
print(ls.sort()) #None
```

```
print(sorted(ls)) #returneaza lista sortata
```

Funcții

- ▶ Putem returna valori de orice tip
- ▶ O funcție poate returna mai multe valori (care vor fi returnate “împachetat” într-un tuplu)

```
def operatii(x,y,z):  
    return x+y+z,x+y,x+z,y+z
```

```
print(type(operatii(4,5,6))) #returneaza un tuplu
```

```
t = operatii(4,5,6) #t va fi tuple  
print(t,type(t))
```


Funcții

- ▶ Putem returna valori de orice tip
- ▶ O funcție poate returna mai multe valori (care vor fi returnate “împachetat” într-un tuplu)

```
def operatii(x,y,z):  
    return x+y+z,x+y,x+z,y+z
```

- ▶ Putem “despacheta” rezultatul în variabile; **numărul de variabile trebuie sa fie egal cu numărul de valori returnate** (= câte elemente are tuplul = atribuire de tuple; elementele tuplului vor fi “despachetate” în variabile)

```
a,b,c,d = operatii(4,5,6)  
print(a,b,c,d)
```

Funcții

- ▶ La atribuirea de tupluri putem prefixa o variabila cu * pentru a “împacheta” în ea mai multe valori sub formă de listă

Exemplu (suplimentar)

```
(a,*b)=(4,5,6,7)
```

```
print(a,b)
```

```
a,*b = 4,5,6,7
```

```
print(a,b)
```

```
(*b,a)=(4,5,6,7)
```

```
print(a,b)
```

```
*b,a = 4,5,6,7
```

```
print(a,b)
```

```
(*b,) = (1,2,3,4)
```

```
print(b)
```

```
*b, = (1,2,3,4)
```

```
print(b)
```

Funcții

- ▶ La atribuirea de tupleuri putem prefixa o variabilă cu * pentru a “împacheta” în ea mai multe valori sub formă de listă

```
def operatii(x,y,z):  
    return x+y+z,x+y,x+z,y+z
```

```
a,*b = operatii(4,5,6)  
print(a,b,type(b)) #b este lista
```

```
a,*b,d = operatii(4,5,6)  
print(a,b,type(b),d)
```

Funcții

► Specificarea parametrilor

Parametri obligatorii – se specifică:

- prin poziție – respectând ordinea și numărul parametrilor din antetul funcției:
- prin nume – atunci nu mai trebuie respectată ordinea, dar trebuie respectat numărul
- se pot combina, dar primii vor fi cei dați prin poziție

Funcții

```
def f(x,y,z):  
    print(f"x={x},y={y},z={z}")
```

```
#prin pozitie
```

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

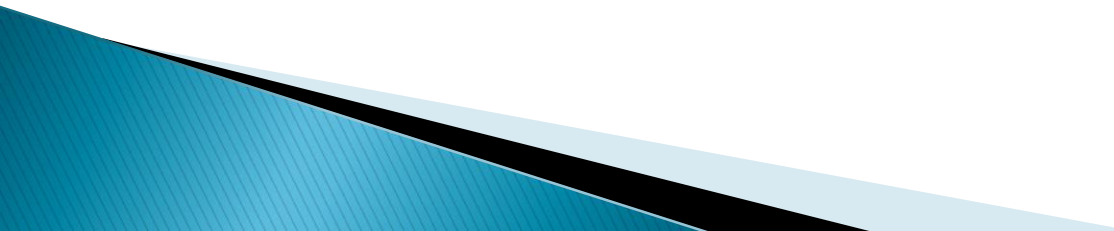
```
f(1,*[3,4]) #despachetarea unui iterabil în elemente
```

```
#prin nume -> nu trebuie respectata ordinea
```

```
f(y=1,z=2,x=5)
```

```
#combinat
```

```
f(1,z=2,y=3)
```



Funcții

- ▶ **Specificarea parametrilor**

Parametri cu valori default = implicită, care se folosește dacă parametrul nu este specificat

- se pun în antet după parametrii obligatorii (la final, după pot eventual urma parametrii variabili)

Funcții

```
def f_default(x,y,z,d=0): #ultimii  
    print(f"x={x},y={y},z={z},d={d}")
```

```
f_default(1,2,3) #d are valoarea default
```

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

Funcții

- ▶ **Specificarea parametrilor**

Număr variabil de parametri

- ▶ parametri prefixați de * => “împachetează” în ei mai multe valori sub formă de tuplu (tuple packing)

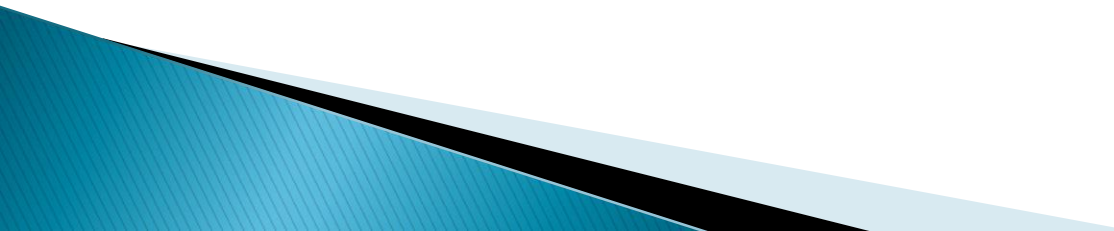
Funcții

```
#functie cu numar variabil de parametri
```

```
def f_var(*p): #tuple packing  
    print(p,type(p))
```

```
f_var(1,2)
```

```
f_var(1,2,3,4,5)
```



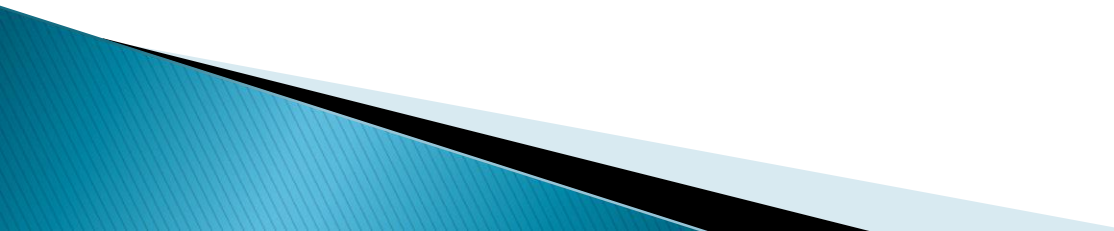
Funcții

```
def mai_mici_decat(x,*param):  
    print(param,type(param))  
    nr = 0  
    for y in param:  
        if y<x:  
            nr = nr+1  
    return nr  
  
print(mai_mici_decat(3,1,7,4,2,0,9))  
print(mai_mici_decat(3,8,9))
```

Funcții

- ▶ **Specificarea parametrilor**

Număr variabil de parametri

- ▶ parametru prefixat de * => “împachetează” în el mai multe valori sub formă de tuplu (tuple packing)
 - ▶ parametri care urmează după cel prefixat cu * în antet trebuie dați prin nume
- 

Funcții

```
def operatie(x, *param, op):  
    print(param)  
    s = x  
    if op == '+':  
        for y in param:  
            s=s+y  
        return s  
    if op == '*':  
        for y in param:  
            s=s*y  
        return s  
    return x
```

```
print(operatie(1,2,3,'*')) #EROARE TypeError: operatie()  
missing 1 required keyword-only argument: 'op', '*' devine  
element in param, nu este valoarea lui op
```

```
print(operatie(1,2,3, op = '*')) #CORECT
```

Funcții

```
def operatie(x, *param, op='+') :  
    print(param)  
    s = x  
    if op == '+':  
        for y in param:  
            s=s+y  
        return s  
    if op == '*':  
        for y in param:  
            s=s*y  
        return s  
    return x
```

`print(operatie(1,2,3,'*'))` #EROARE '*' devine element in
param, nu este valoarea lui op, op va folosi valoarea
default '+' => eroare

#op trebuie specificat prin nume

```
print(operatie(1,2,3,op='*'))
```

Funcții

Transmiterea parametrilor

- ▶ Amintim variabilele sunt nume = referințe pentru obiecte
- ▶ Prin atribuire = prin referință la obiect = se transmit prin valoare referințe de obiecte; variabilele sunt nume=referințe pentru obiecte

Funcții

Transmiterea parametrilor

- ▶ **Amintim** variabilele sunt nume = referințe pentru obiecte

x = 10



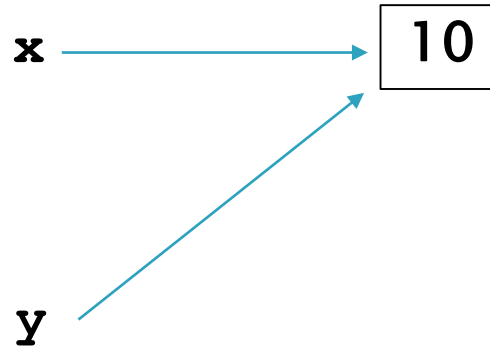
Funcții

Transmiterea parametrilor

- ▶ **Amintim** variabilele sunt nume = referințe pentru obiecte

x = 10

y = **x**



Funcții

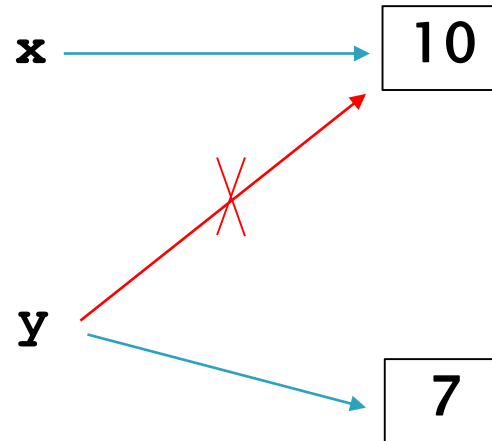
Transmiterea parametrilor

- ▶ **Amintim** variabilele sunt nume = referințe pentru obiecte

x = 10

y = **x**

y = 7



Funcții

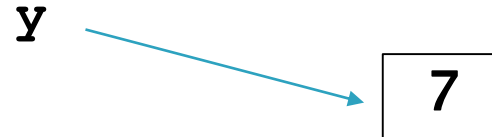
Transmiterea parametrilor

- ▶ Amintim variabilele sunt nume = referințe pentru obiecte

x = 10



y = **x**



y = 7

Funcții

Transmiterea parametrilor

- ▶ Variabilele/parametrii sunt nume pentru obiecte – în spațiul local al funcției
- ▶ Mecanism de transmitere = prin atribuire (prin referință la obiect) (= **pass by object reference** = **pass by assignment**) – se transmit prin valoare referințe de obiecte

Funcții

```
def modific(y):  
    y = 7  
    print(y)
```

```
x = 10
```

```
modific(x)
```

```
print("x = ", x)
```

Pas cu pas:



Funcții

```
def modific(y):  
    y = 7  
    print(y)
```

```
x = 10
```

global

x



10

Funcții

```
def modific(y):  
    y = 7  
    print(y)
```

`x = 10`

`modific(x)`



global

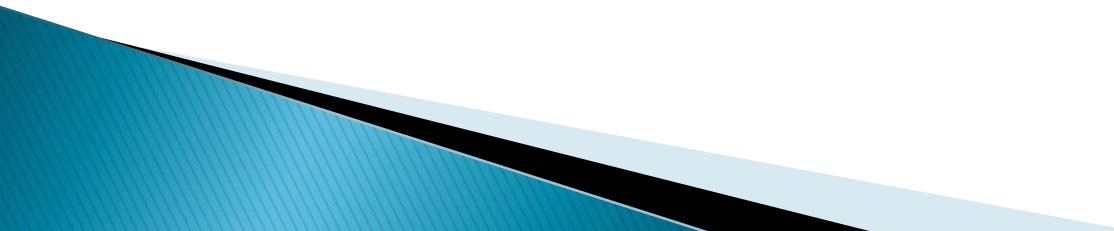
local
pentru
funcția
modific

x

10

y

transmiterea parametrului $y=x$



Funcții

```
def modific(y):  
    y = 7  
    print(y)
```

`x = 10`

`modific(x)`



global

x

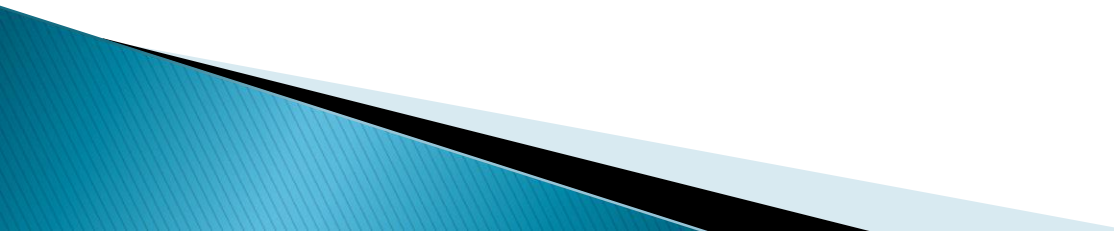
10

local
pentru
funcția
modific

y

atribuire **y=7**

7



Funcții

```
def modific(y):  
    y = 7  
    print(y)
```

`x = 10`

`modific(x)`



global

x

10

local
pentru
funcția
modific

y

7

print(y) => 7

Funcții

```
def modific(y):  
    y = 7  
    print(y)
```

```
x = 10
```

global

x

10

```
modific(x)
```

```
print("x = ", x) ➡ afiseaza 10
```

Funcții

- ▶ Chiar dacă parametrul formal (al funcției) se numește tot x – principiul de funcționare este același

```
def modific(x):
```

```
    x = 7
```

```
    print(x)
```

```
x = 10
```

```
modific(x)
```

```
print("x = ", x)
```

Pas cu pas:

Funcții

```
def modific(x):  
    x = 7  
    print(x)
```

```
x = 10
```

global

x



10

Funcții

```
def modific(x):  
    x = 7  
    print(x)
```

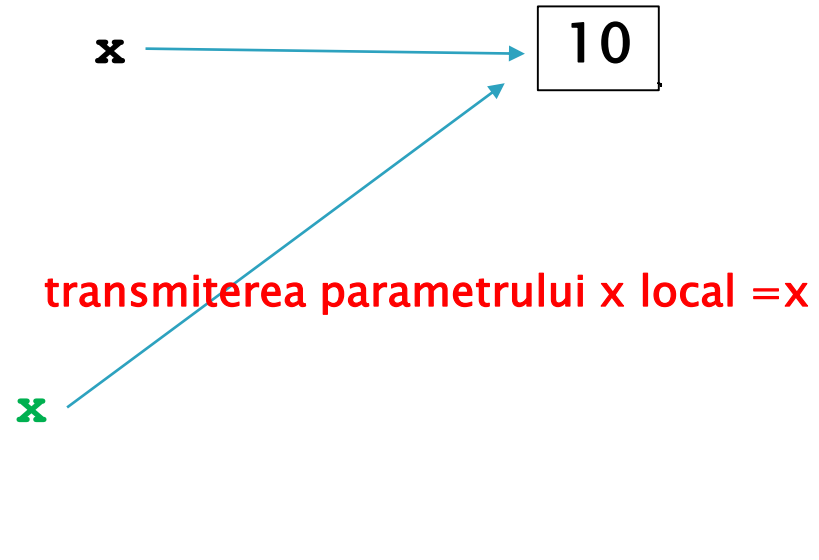
`x = 10`

`modific(x)`



global

local
pentru
funcția
modific



Funcții

```
def modific(x):  
    x = 7  
    print(x)
```

`x = 10`

`modific(x)`



global

x

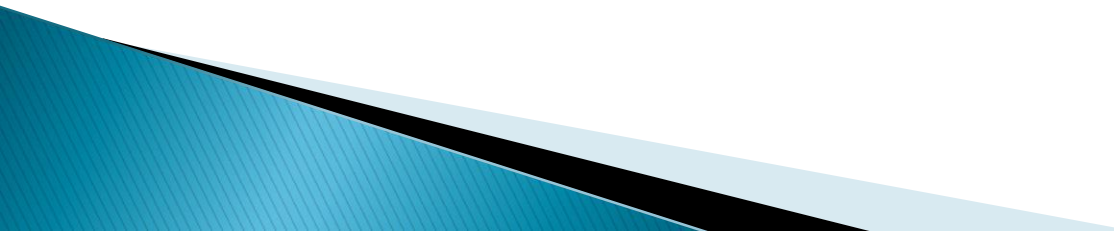
10

local
pentru
funcția
modific

x

atribuire x=7

7



Funcții

```
def modific(x):  
    x = 7  
    print(x)
```

`x = 10`

`modific(x)`



global

x

10

local
pentru
funcția
modific

x

7

print(x) => 7

Funcții

```
def modific(x):  
    x = 7  
    print(x)
```

```
x = 10
```

global

x

10

```
modific(x)
```

```
print("x = ",x) ➡ afiseaza x = 10
```

Funcții

- ▶ Când parametrul actual este **mutabil** – principiul de transmitere este același: !!! modificarea valorii parametrului (nu a referinței) se va face asupra obiectului transmis ca parametru

```
def modific_lista(ls):  
    ls.append(12)
```

```
ls = [3,4]  
modific_lista(ls)  
print(ls)
```

Pas cu pas:



Funcții

```
def modific_lista(ls):  
    ls.append(5)
```

```
ls = [3,4]
```

global

ls → [3,4]

Funcții

```
def modific_lista(ls):  
    ls.append(5)
```

```
ls = [3,4]
```

global

```
modific_lista(ls)
```



local pentru
funcția
modific_lista

ls



[3,4]

transmiterea parametrului
ls local = ls

ls



Funcții

```
def modific_lista(ls):  
    ls.append(5)
```

```
ls = [3,4]
```

global

```
modific_lista(ls)
```

local pentru
funcția
modific_lista

ls → [3,4,5]

ls.append(5)

ls

!! ls local, se putea
numi si altfel
parametrul

Funcții

```
def modific_lista(ls):  
    ls.append(5)
```

```
ls = [3,4]
```

global

```
modific_lista(ls)
```

```
print(ls) ➡ [3, 4, 5]
```

ls ➡ [3,4,5]

Funcții

- ▶ Chiar dacă parametrul actual este **mutabil** în modificarea referinței (prin atribuire) este implicată variabila locală – ca în primul exemplu

```
def creez_lista(ls):  
    ls = [1,2,3]
```

```
ls = []  
creez_lista(ls)  
print(ls)
```

Pas cu pas:



Funcții

```
def creez_lista(ls):  
    ls = [1,2,3]
```

```
ls = []
```

global

ls → []

Funcții

```
def creez_lista(ls):  
    ls = [1,2,3]
```

`ls = []`

`creez_lista(ls)`

local pentru
funcția
`creez_lista(ls)`

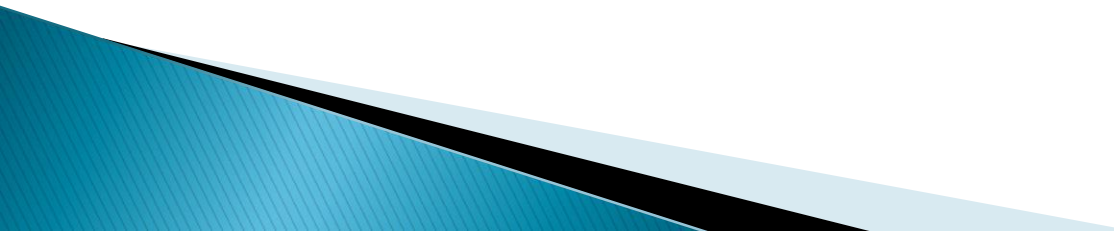
global

`ls`

`[]`

transmiterea parametrului

`ls`



Funcții

```
def creez_lista(ls):  
    ls = [1,2,3]
```

`ls = []`

global

`ls`

`[]`

`creez_lista(ls)`

local pentru
funcția
`creez_lista(ls)`

atribuire `ls = [1,2,3]`

`ls`

`[1,2,3]`

Funcții

```
def creez_lista(ls):  
    ls = [1,2,3]
```

```
ls = []
```

```
creez_lista(ls)
```

```
print(ls)
```

global

ls



[]

Funcții

Vizibilitatea variabilelor – pe scurt

- ▶ La interogarea unei variabile în interiorul unei funcții, numele variabilei este căutat întâi în spațiul local funcției, apoi în cel “exterior” (global dacă funcția este declarată în “main”)
- ▶ Când se atribuie pentru prima dată o valoare unei variabile `x` în interiorul funcției, variabila `x` este creată în spațiul local funcției; pentru a modifica o variabilă globală `x` trebuie precizat explicit că `x` va fi căutat în spațiul global: `global x`

Funcții

```
def accesez():  
    print(x)
```

```
def atribuire():  
    x = 10 #atribuire => crearea variabilei x in "local"
```

```
def modific_valoare():  
    global x  
    x = x+1
```

```
x = 100  
accesez()  
atribuire()  
print(x)
```

```
modific_valoare()  
print(x)
```