Funcții

Funcții 2
Funcții predefinite 2
Definirea unei funcții în Python. Parametri și valori returnate 2
Specificarea parametrilor 3
Parametri obligatorii 4
Parametri cu valoare default (valoare implicită) 5
Număr variabil de parametri 6
Transmiterea parametrilor 8
Vizibilitatea variabilelor. Variabile locale și globale 11
Funcții recursive 13
Funcții ca parametri (callback) 14
Funcții imbricate - Suplimentar 15
Generatori - Suplimentar 16

Funcții

O funcție este un ansamblu de instrucțiuni, grupate sub un nume, care prelucrează cu un cop bine definit datele primite ca parametri în momentul în care este apelată.
Funcțiile permit împărțirea unui program în secțiuni cu scopuri bine și reutilizarea unor astfel de secțiuni în alte programe, prin gruparea lor în module care pot fi importate.

Funcții predefinite

Am amintit deja o serie de funcții predefinite în Python: funcții de conversie (**hex**, **bin** etc.), funcții matematice, funcții pentru secvențe:
https://docs.python.org/3/library/functions.html

Definirea unei funcții în Python. Parametri și valori returnate

O funcție în Python se definește astfel:

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

O funcție are de obicei un număr fix de parametri, dar poate avea și un număr variabil de parametri (vom reveni). Parametrii formali sunt cei specificați în antetul unei funcții, indicând datele cu care trebuie apelată funcția. O funcție este apelată cu parametri actuali, care sunt valori sau expresii ale căror valori se atribuie parametrilor formali în momentul apelării unei funcții.

O funcție poate returna (explicit) valori de orice tip, folosind instrucțiunea **return**. Dacă o funcție nu returnează nimic explicit, valoarea returnată de ea va fi **None** (!deci tot va returna un rezultat)

Exemple

```
def f(x,y): #x, y = parametri formali
    while x!=y:
        if x<y:
            y = y-x
        else:
            x = x-y
    return x
x = f(15, 25) #parametri actuali
print(x)
```

```
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
```

O funcție poate returna mai multe valori (împachetate implicit într-un tuplu, după cum am amintit deja la capitolul dedicat tuplurilor)

```
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))
```

Putem "despacheta" rezultatul întors de o funcție în variabile; numărul de variabile trebuie sa fie egal cu numărul de valori returnate:

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

Amintim că la atribuirea de tupluri putem prefixa o variabila 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)
```

Există mai multe aspecte legate de parametri formali ai unei funcții, pe care le vom discuta în secțiunile următoare :

- parametri cu valori default
- număr variabil de parametri
- parametri cu nume

Suplimentar: Pentru fiecare parametru se poate preciza și ce tip dorim să aibă, dar aceasta este doar o adnotare, oferind doar informații despre ce tip de date se așteaptă, fără a îl impune; funcția va putea fi apelată cu orice tip de date:

```
def f(x : int):
    print(x)
f(12)
f("abc")
f([3,5,7])
```

Specificarea parametrilor

O funcție poate avea mai multe tipuri de parametri. Unii parametri trebuie să primească obligatoriu valoare la apel, alții au valori default (implicite) și pot să nu aibă valori asociate la apelul funcției (caz în care se folosește valoarea default).

Parametri obligatorii

Parametrii obligatorii au valori asociate la apelul funcției (sunt parametri formali cărora le corespunde obligatoriu parametri actuali la apel). Valoarea unui astfel de parametru se poate specifica la apel:

- **prin poziție**: respectând la apel ordinea și numărul parametrilor din antetul funcției:

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

- **prin nume**, sub forma `nume_parametru = expresie`; dacă parametri se transmit prin nume atunci nu mai trebuie respectată ordinea, dar trebuie respectat numărul lor:

```
#prin pozitie
f(1,2,3)
f(1,*[3,4]) #despachetarea unui iterabil în elemente
```

- cele două modalități se pot combina, dar primii vor fi cei dați prin poziție:

```
#prin nume -> nu trebuie respectata ordinea
f(y=1,z=2,x=5)
#combinat
f(1,z=2,y=3)
```

Suplimentar:

- putem impune ca acești parametri sa fie dați doar prin poziție (nu și prin nume), adăugând ca parametru un / după ei (positional-only arguments):

```
def f(x,y,/,z):
    print(f"x={x},y={y},z={z}")
f(1,2,3)
#f(x=1,y=2,z=3) #NU
f(1,2,z=3)
```

- putem impune ca acești parametri sa fie dați doar prin nume adăugând ca parametru o * înaintea lor (keyword argument: v. secțiunea dedicată funcțiilor cu număr variabil de parametri):

```
def f(x,*,y,z):
    print(f"x={x},y={y},z={z}")
#f(1,2,3) #NU
#f(x=1,y=2,z=3) #NU
f(1,z=2,y=3)
f(z=2,y=3,x=1)
```

- Similar cu operatorul *, care se folosește pentru împachetarea/ despachetarea de secvențe, există operatorul ** pentru împachetarea/despachetarea de dicționare, utilă de exemplu pentru transmiterea parametrilor prin nume.

Spre exemplu:

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

```
d = {"x":4, "z":1, "y":3}
f(**d) #este similar cu apelul f(x=4,z=1,y=3)

def f(**param):
    print(param)
    print(type(param))
    for cheie, val in param.items():
        print(cheie, ':', val)

f(a=3,b=123,suma=126)
```

Parametri cu valoare default (valoare implicită)

O funcție poate avea parametri cărora li se atribuie în antet o valoare, cu care este inițializat dacă parametrul nu este specificat în apelul funcției (parametrului formal nu ii corespunde un parametru actual) .

Parametri cu valoare implicită se pun în antet după parametrii obligatorii (la final, după ei pot eventual urma parametrii variabili).

Spre exemplu:

```
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)
```

Observație: Valoarea implicită se evaluează o singură dată. Acest fapt este relevant pentru parametri mutabili, a căror valoare se modifică în interiorul funcției – la următorul apel ei vor rămâne cu valoarea implicită de la apelul anterior

```
def f(x, ls = []):
    ls.append(x) #se modifica valoarea obiectului ls
    print(ls)

f(1) #va afisa [1] si se va modifica valoarea implicita in functie
f(2) #va afisa [1, 2], nu s-a inițializat din nou ls cu []
f(3) #va afisa [1, 2, 3]
```

Atenție, la încercarea de modificare a valorii unui obiect de tip numeric de exemplu efectul este crearea unui obiect nou, de aceea în acest caz valoarea implicită a unui parametru de tip numeric (sau, mai general, imutabil) este mereu aceeași:

```
def f(x, k = 1):
    k = k + x #se creeaza un nou obiect
    print(k)

f(1) # se va afisa 2
f(2) # 3 (!valoarea implicita a lui k a ramas 1, in apelul anterior
efectul instructiunii k = k + x este crearea unui nou obiect)
f(3) # 4
```

Număr variabil de parametri

În limbajul Python se pot defini și funcții cu număr variabil de parametri. De exemplu, putem avea nevoie să calculăm în același program suma a două numere și suma a trei numere. O soluție nerecomandată ar fi să creăm funcții diferite, dar cu comportament similar (în limbajul Python nu putem defini două funcții cu același nume și număr diferit de parametri, dacă definim mai multe funcții cu același nume va fi luată în considerare doar ultima definită). O altă soluție ar fi sa creăm o funcție suma care primește ca parametru o lista de numere, dar atunci apelul nu s-ar mai putea face la fel de simplu (va trebui să scriem suma([5,6]) în loc de suma(5,6))

Și folosind parametri cu valoare implicită putem avea la apel un număr variabil de parametri, dar numărul lor este limitat de numărul total de parametri din antet.

Pentru a specifica faptul că o funcție are număr variabil de parametri, unul dintre parametri va fi prefixat de simbolul *. Atunci în parametrul respectiv se vor grupa, sub forma unui tuplu, un număr variabil de parametri efectivi cu care este apelată funcția.

De exemplu, o funcție cu număr variabil de parametri care calculează suma acestora se poate defini și apela astfel:

```
def suma(*numere):
    s = 0
    for x in numere:
        s += x
    return s
print(suma(1,5), suma(3,4,6,8))
```

Astfel, parametri prefixați în antet de * pot "împacheta" în ei mai multe valori transmise la apelul funcției, sub formă de tuplu (tuple packing). Spre exemplu**:**

```
#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)
f_var() #pot sa nu trimit niciun parametru
```

Spre exemplu, să considerăm următoarea funcție care determină numărul de valori primite ca parametru mai mici decât o valoare x dată ca prim parametru:

```
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))
```

După parametrul prefixat cu * în antet pot urma și alți parametri, dar aceștia trebuie dați **prin nume** la apel, deoarece ei nu mai pot fi identificați prin poziție, fiind precedați de

un număr necunoscut de parametrii (altfel valorile transmise devin elemente ale parametrului variabil). Astfel, dacă în exemplul anterior am pune parametrul x după ***param**, el va putea fi apelat doar prin nume:

```
def mai_mici_decat(*param,x):
    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)) #eroare
print(mai_mici_decat(1,7,4,2,0,9,x=3))
```

După parametrul prefixat cu * în antet pot urma și parametri cu valoare implicită:

```
def mai_mici_decat(*param,x=0):
    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)) #nu mai da eroare, x va fi 0
print(mai_mici_decat(1,7,4,2,0,9,x=3))

def mai_mici_decat(*param,x=0,min_value=-float("inf")):
    print(param,type(param))
    nr = 0
    for y in param:
        if min_value<y<x:
            nr = nr+1
    return nr
print(mai_mici_decat(3,1,7,4,2,0,9)) #nu mai da eroare, x va fi 0
print(mai_mici_decat(3,1,7,4,2,0,9,x=8 ))
print(mai_mici_decat(3,1,7,4,2,0,9,x=8,min_value=2))
print(mai_mici_decat(3,1,7,4,2,0,9,min_value=2,x=8))
```

Un alt exemplu este următorul:

```
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

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

#op trebuie specificat prin nume
print(operatie(1,2,3,op='*'))

#op poate sa nu primeasca valoare la apel
print(operatie(1,2,3))
```

Exercițiu: Ce va afișa următoarea secvență de cod?

```
def suma(*param, s=12):
   rez=s
   for x in param:
      rez+=x
   return rez
print(suma(1,2,3))
print(suma(1,2,s=3))
print(suma(s=3,1,2))
```

Transmiterea parametrilor

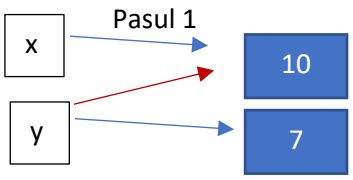
În limbajele C/C++ există două modalități de transmitere a unui parametru efectiv către o funcție:

- **prin valoare**: se transmite o **copie** a valorii parametrului efectiv; în acest caz modificările efectuate asupra parametrului respectiv în interiorul funcției se vor face asupra copiei (NU se vor reflecta în exteriorul funcției)

- **prin adresă/referință**: se transmite adresa parametrului efectiv (deci modificările efectuate asupra parametrului respectiv în interiorul funcției se reflectă și în exteriorul său)

Amintim că în Python variabilele sunt nume (referințe) pentru obiecte. O atribuire de forma `x = val` are ca efect faptul că x va referi un nou obiect (id(x) se modifică – v. secțiunea dedicată variabilelor):

x = 10
y = x
y = 7

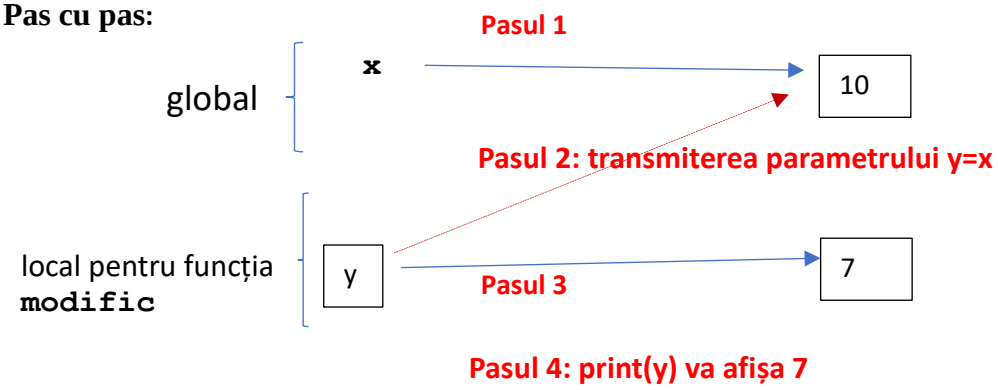


Parametri unei funcții sunt de fapt tot **nume (referințe) pentru obiecte definite în spațiul local al funcției**. Mecanismul de transmitere a parametrilor este **prin atribuire (parametru_formal=parametru_actual)**, și se întâlnește cu denumirile de **pass by object reference = pass by assignment. Astfel, se transmit prin valoare referințe de obiecte**, deci modificarea referinței respective în interiorul funcției nu se va reflecta în exteriorul său (!dar a valorii da).

Să considerăm următorul exemplu:

```
def modific(y):
    y = 7               #Pas 3
    print(y)            #Pas 4
x = 10                  #Pas 1
modific(x)              #Pas 2 – transmitere parametru
print("x = ",x)         #pas 5
```

Pas cu pas:



Înainte de pasul 5, după încheierea apelului funcției **modific**, spațiul local pentru funcție se eliberează și rămâne doar cel global

Astfel, la Pasul 5 se va afișa 10.

Chiar dacă parametrul formal (al funcției) s-ar numi tot x – principiul de funcționare este același, doar că în loc de y **apare o variabilă x și în spațiul local.**

Dacă într-o funcție există definită o variabilă locală având același nume cu o variabilă globală, atunci, implicit, se va utiliza variabila locală în interiorul funcției (vom detalia în secțiunea dedicată vizibilității variabilelor):

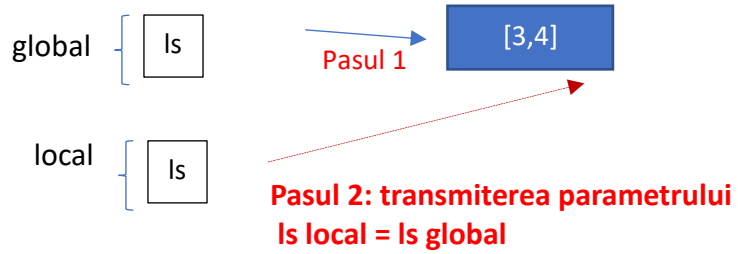
```
def modific( ):
    x = 7
    print(x)
x = 10
```

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

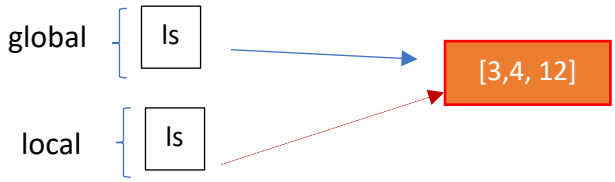
Când parametrul actual este mutabil – principiul de transmitere este același, dar, atenție, modificarea valorii parametrului (nu a referinței) se va face asupra obiectului transmis ca parametru (deci va rămâne și după terminarea execuției funcției).

Spre exemplu:

```
def modific_lista(ls):
    ls.append(12)  #Pasul 3
ls = [3,4]         #Pasul 1
modific_lista(ls)  #Pasul 2
print(ls)          #Pasul 4
```



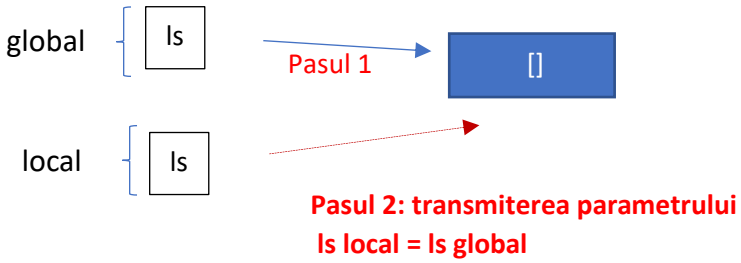
Primii doi pași sunt ilustrați în figura alăturată. La Pasul 3 se modifică valoarea obiectului referit de ls din local, care este același cu cel referit de variabila globală **`ls`**:



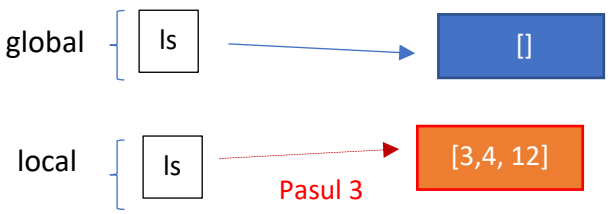
Înainte de pasul 4, după încheierea apelului funcției **`modific_lista`**, variabilele locale se eliberează și rămân doar cele globale, deci doar variabila **`ls`** din global, care referă obiectul cu valoarea [3,4,12], de aceea la Pasul 4 se va afișa lista modificată.

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]     #Pasul 3
ls = []              #Pasul 1
creez_lista(ls)      #Pasul 2
print(ls)            #Pasul 4
```



Primii doi pași sunt ilustrați în figura alăturată codului. La Pasul 3 se face o atribuire în care este implicată variabila **`ls`**. Fiind în interiorul funcției, variabila **`ls`** căreia i se atribuie o valoare este cea din local. Astfel, variabila **`ls`** din local va referi un nou obiect (iar cea din global va referi în continuare vechiul obiect):



Înainte de pasul 4, după încheierea apelului funcției **`creez_lista`**, variabilele locale se eliberează și rămân doar cele globale, deci doar variabila **`ls`** din global care referă obiectul cu valoarea [] (o listă vidă), care va fi afișată la pasul 4.

În concluzie, mecanismul de transmitere a parametrilor în Python este prin atribuire (se transmit prin valoare referințele către obiectele corespunzătoare parametrilor efectivi) și deci singurele modificări care se reflectă în exterior sunt cele făcute in funcție asupra valorii unor parametri mutabili.

Vizibilitatea variabilelor. Variabile locale și globale

Amintim că în Python o variabilă se creează când i se atribuie prima dată o valoare. Locul în care se efectuează această atribuire determină domeniul de vizibilitate a unei variabile. Dacă unei variabile i se atribuie prima data o valoare în afara oricărei funcții, ea va fi **vizibilă** în tot modul (în domeniul **global**) și o vom numi variabilă globală.

```
def f():
    print(x) #cauta x in global deoarece nu exista in local

x = 10
f()
```

Dacă unei variabile i se atribuie prima data o valoare în interiorul unei funcții, ea va fi creată în spațiul local al funcției, deci va fi o **variabilă locală** și va fi vizibilă doar în interiorul acelei funcții (în domeniul **local** în care a fost creată).

```
def f():
    x = 10
    print(x)
f()
print(x) #eroare NameError
```

La interogarea (accesarea valorii) unei variabile în interiorul unei funcții, numele variabilei este căutat întâi în spațiul local funcției, apoi, dacă nu există, în cel exterior (global dacă funcția nu este inclusă în altă funcție, sau nonlocal, al funcției care o include, altfel). Mai exact se caută variabila după regula LEGB (în ordinea local, enclosing, global and built-in).

Când unei variabile i se atribuie o valoare, aceasta este căutată doar în domeniul curent (local, daca atribuirea este în interiorul funcției, global dacă este în afara funcțiilor). Dacă nu există variabila, atunci se creează:

```
def f():
    x = 10 #creata variabila x in local
    print(x) #cauta x in local si il gaseste, nu mai cauta in global
x = 7
f()
print(x) #7
```

Dar dacă în exemplul anterior vrem însă să modificăm în interiorul unei funcții valoarea variabilei globale x? Pentru a specifica faptul că vrem ca variabila să fie căutată într-un alt domeniu, de exemplu cel global, trebuie să adăugăm instrucțiunea **global x** (!dacă avem doar accesare a valorii lui x nu și atribuire nu este necesar); este o diferență între operația de atribuire și cea de accesare a unei variabile, pentru că în urma unei atribuiri o variabila se creează dacă nu există în domeniul în care se face atribuirea (curent sau specificat) sau își modifică valoarea, dacă există

```
def f():
    global x
    x = 10 #actualizata variabila x din global
    print(x)
x = 7
f()
print(x) #10
```

Atenție, exemplu următor va da eroare:

```
def f():
    print(x) #eroare, este un x in local (deci x nu va mai fi cautat in
              global), dar primeste valoare dupa UnboundLocalError:
    x = 7
x = 10
f()
```

Putem însă specifica însă că vrem să folosim variabila globală x (și atunci prin atribuirea x=7 se va modifica variabila globală, nu se va crea o variabila în local)

```
def f():
    global x
    print(x)
    x = 7
x = 10
f()
print(x)
```

Suplimentar. Domeniile în Python sunt implementate ca dicționare cu cheile nume de variabile.

```
def f():
    globals()['x']=3
    x = 10 #se creeaza in local variabila x
    print(locals())
    print(x) #10

x=12
f()
print(x) #3
```

Funcții recursive

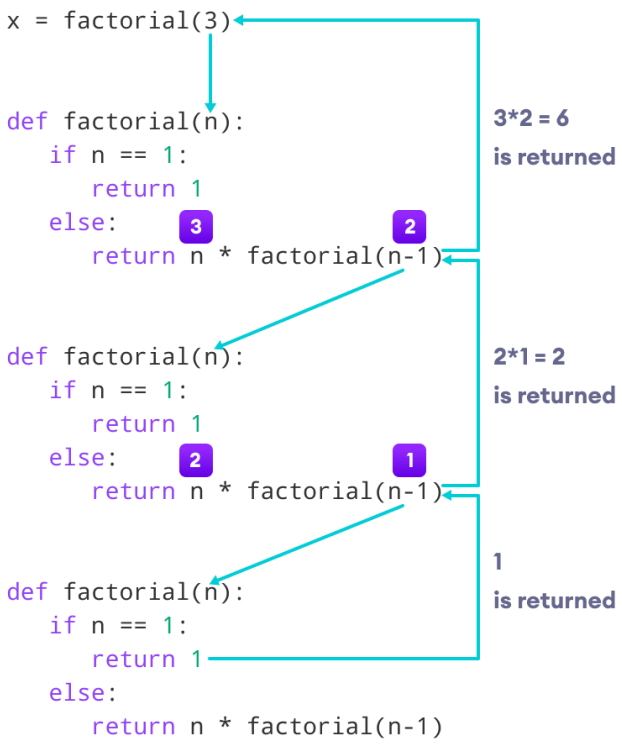
O funcție recursivă este o funcție care se autoapelează (direct sau indirect).
Un exemplu clasic de funcție recursivă îl reprezintă calculul factorialului unui număr natural n (i.e., $n!=1\cdot 2\cdot \ldots \cdot n$), folosind următoarea relație de recurență:

$$n! = \begin{cases} 1, & \text{dacă n = 1} \\ n\cdot(n-1)!, & \text{altfel} \end{cases}$$

O funcție recursivă care implementează în limbajul Python relația de recurență de mai sus este următoarea:

```
def factorial(n):
    if n == 0:
        return 1
    return n*factorial(n-1)
```

La apelarea unei funcții se salvează în stiva alocată programului contextul de apel (numele, variabilele locale, parametrii, adresa de revenire etc.). Când executarea apelului se termină, contextul de apel este eliminat din stivă.
De exemplu, pentru apelul f = factorial(3), stiva programului (reprezentată invers) va avea următoarea evoluție (sursa imaginii: https://www.programiz.com/python-programming/recursion):



În limbajul Python se pot defini funcții recursive, numărului maxim de apeluri recursive care pot fi salvate pe stivă permis implicit fiind egal cu 1000. Acest număr se poate modifica folosind funcția **setrecursionlimit** din modulul **sys**

```
import sys
sys.setrecursionlimit(10010)
print(factorial(1001)) #RecursionError fara setrecursionlimit
```

Funcții ca parametri (callback)

O funcție poate primi ca parametru o altă funcție. Acest lucru este util și întâlnit în multe situații, una dintre cele mai cunoscute fiind în programarea interfețelor grafice, când o funcție este asociată (apariții) unui eveniment, spre exemplu apăsarea unui buton.
Transmiterea unei funcții ca parametru este utilă și în funcții care realizează prelucrări generice, care nu sunt cunoscute la momentul scrieri funcției, ca de exemplu reprezentarea grafică a unei funcții, filtrarea după un criteriu, calculul unei sume cu termen generic (paradigma de programare generica/funcțională).

Exemple

1. O funcție care calculează suma $\sum_{i=1}^{n} f(i)$. Cu ajutorul ei putem calcula, de exemplu, suma primelor n numere naturale, a primelor n pătrate perfecte etc

```
import math
def inv(i):
    return 1/i

def suma(*arg, functie = int):
    s = 0
    for x in arg:
        s = s+functie(x)
    return s
print(suma(1,2,3,4))
print(suma(1,2,3,functie = inv))
print(suma(1,2,3,functie = lambda x:x*x))
print(suma(1,2,3,functie = math.sqrt))
```

2. Filtrarea unei liste după un criteriu (dat printr-o funcție care returnează True/False)

```
def pozitiv(x):
    return x>0
def filtreaza(lista,functie):
    return [x for x in lista if functie(x)]
l = filtreaza([3,-1,6,8,-3],pozitiv)
print(l)
```

```
def filtreaza(*lista,functie):
    return (x for x in lista if functie(x))
l = filtreaza(3,-1,6,8,-3,functie=pozitiv)
```

```
print("filtr",l) #generator
```

Exercițiu: Scrieți o funcție care primește ca parametru un număr variabil de numere $x_1,...,x_n$ și o funcție f și calculează media $\sum_{i=1}^{n} f(x_i)/n$

Funcții imbricate - Suplimentar

În limbajul Python putem defini o funcție în interiorul altei funcții. O astfel de funcție imbricată va putea fi apelată doar in interiorul funcției în care a fost definită:

```
def distanta_minima(p1,p2,p3):
    def distanta(a,b):
        return (a[0]-b[0])**2+(a[1]-b[1])**2
    return min(distanta(p1,p2),distanta(p1,p3),distanta(p2,p3))
print(distanta_minima((0,1),(5,7),(2,5)))
```

O variabilă dintr-o funcție imbricată este căutată după regula LEGB, deci funcția imbricată are acces la variabilele locale ale funcției în care este definită.

```
def f(x):
    def f2():
        return 2*x
    return f2()+1
print(f(3))
----------------------------
def f():
    x = 3
    def f2():
        return 2*x
    return f2()+1
print(f())
x = 3
----------------------
def f():
    def f2():
        return 2*x
    return f2()+1
print(f())
```

Pentru a modifica in interiorul funcției imbricate o variabilă x globală putem folosi **`global`** x, iar o variabila din funcția in care este definita putem folosi **`nonlocal`** x:

Exemplul 1:

```
def f():
    x = 3
    def f2():
        nonlocal x
        x = 4
    f2()
    print(x)
f()
```

Exemplul 2:

```
x = 3
def f():
    def f2():
        global x
        x = 5
    f2()
    print(x)
f()
print(x)
```

O funcție poate să returneze o funcție imbricată, așa cum se poate observa din exemplul următor:

```
def putere(baza):
    def putere_baza(exponent):
        return baza ** exponent
    return putere_baza
p2 = putere(2)
print(p2(5)) #2**5
print(p2(1))
print(putere(4)(2))
```

Utilitatea returnării unei funcții imbricate se poate vedea și la sortări, ca în exemplul următor.

Exemplu: Se dau două șiruri de caractere s1 și s2 (de la tastatură) și un fișier propozitii.in în care pe fiecare linie se află câte o propoziție. Să se scrie în fișierul prop_sort.out propozițiile din fișier în ordinea descrescătoare în raport cu numărul de litere pe care îl au în comun cu șirul s1 (câte una pe linie), apoi crescătoare în raport cu numărul de caractere pe care îl au în comun cu șirul s2.

```
def cheie_s(s):
    def nr_litere_comune(t):
        return len(set(s) & set(t))
    return nr_litere_comune
f = open("propozitii.in")
ls = f.readlines()
ls[-1] = ls[-1]+"\n" #adaug \n si la ultima linie
f.close()
s1 = input()
s2 = input()
f = open("prop_sort.out","w")
f.writelines(sorted(ls, key=cheie_s(s1), reverse=True))
f.writelines(sorted(ls, key=cheie_s(s2)))
f.close()
```

Generatori - Suplimentar

Un generator este un tip de funcție a cărei executare nu se termină în momentul în care furnizează o valoare (vom vedea ca nu o returnează cu return, ci o furnizează cu yield), ceea

ce îi permite furnizarea succesiva mai multor valori (una câte una), ca o secvență (iterabil) a cărei elemente se generează pe rând, similar cu modul de funcționare al „funcției” range. Intuitiv, un generator generează o secvență de elemente iterabile element cu element, aceasta putând fi iterată cu for sau transformată în listă cu **`list`** (dacă este finită). Un element poate fi accesat și folosind funcția **`next`** (astfel este solicitata explicit generarea unui nou element din secvență)

```
def gen_patrate_pana_la(n):
    for k in range(n):
        yield k ** 2  # producem, nu returnam
#(to yield = a produce, to return = a returna)

print(gen_patrate_pana_la(10))
for x in gen_patrate_pana_la(10):
    print(x,end=" ")
print("\n---")
t = gen_patrate_pana_la(10)
print(next(t))
print(next(t))
print(list(t))
print("---")
t = gen_patrate_pana_la(12)
for x in t:
    print(x)
print(next(t))#eroare StopIteration
```

Contextul de apel al unui generator nu este eliminat de pe stiva programului în momentul executării unei instrucțiuni yield. Astfel, după revenirea dintr-un apel, un generator îți poate continua executarea din starea în care se afla înaintea apelului respectiv.

Pentru a întrerupe forțat executarea unui generator, infinit sau nu, poate fi apelată metoda close:

```
def gen_patrate( ):
    k = 0
    while True:
        yield k ** 2
        k += 1
t = gen_patrate()
for i in range(10):
    print(next(t),end=" ")
print()
print(next(t))
t.close()
#print(next(t)) #eroare
```

Generatorii sunt utili în lucru cu secvențe pentru care nu avem nevoie de existența simultană a tuturor elementelor în memorie (de exemplu calcul de sume)