

## NOTIȚE CURS 7 :

**FUNCȚII** - modularizarea codului din program

↳ predefinite: de conversie (int, str, bool)  
matematice (min, max, sum)  
colecții (list, dict, set)  
+ <https://docs.python.org/3/library/functions.html>

### 1. definirea

def nume - funcție (parametri formali):  
instr.

### 2. parametri formali

**parametri simpli**:

def suma(x, y): → suma(3, 7)  
return x + y      suma(y=7, x=3)

putem da pt. x și y val. pt. care se poate utiliza "+",  
astfel putem da și două șiruri de caractere

**parametri cu valori implicite**:

def suma(x=0, y=0): → suma() = 0 (x=0, y=0)  
return x + y      suma(7) = 7 (x=7, y=0)  
                         suma(y=19) = 19 (x=0, y=19)

! controlează ordinea

suma(x=0, y=1) pt. suma(7) y nu are val.

suma(x, y=0) pt. suma(7) x=7 și y=0 și fct. merge  
↳ sub formă unui tuplu

**parametri variabili**:

def suma(\*numere): → s=suma() = 0  
sum=0      s=suma(1, 2) = 3  
for x in numere:      s=suma(1, ...) = 1  
sum+=x

return sum

def suma(\*numere, minim=1): → trebuie precizat când  
și care e minimul  
sum=0      s=suma(5, 7, minim=6)  
for x in numere:      = 7  
if x >= minim:  
sum+=x  
return y

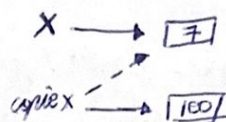
! putem returna mai multe valori

```
def suma - mod (x, y):  
    return x+y, x*y
```

! transmitere prin referință la obiect

```
def functie (t):  
    t=100
```

```
x=7  
functie(x)  
print(x) → 7
```

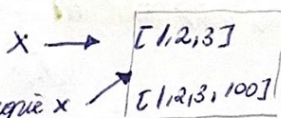


care se duce în funcție, dar x-ul "propriu" nu se modifică

```
def functie (t):  
    t.append(100)
```

```
x=[1,2,3]  
functie(x)  
print(x) → [1,2,3,100]
```

cum lista este mutabilă, val-er se va schimba și în afara funcției



! dar dacă era `t=[100]` atunci nu se modifică, de ce? pt. că programul ar fi creat o listă nouă cu o referință nouă unde era memorată `[1,2,3,100]` și x nu s-ar fi modificat

Cumva : imutabil → nu se va modifica în afara funcției  
mutabil → da

### 3. variabile :

- locale
- globale

```
def afisare():  
    print(x) → 100
```

```
x=100  
afisare()
```

```
def afisare():  
    x=200  
    print(x) → 200
```

```
x=100  
afisare()
```

```
def afisare():  
    global x  
    print(x) → 100  
    x=200 → 200
```

```
x=100  
afisare() → modifica val. lui x  
print(x)
```



#### 4. funcții cu parametrii

import math

def suma\_generica (n, f):  
↓ putem transmite orice funcție predefinită sau  
definită de noi

s = 0

for i in range(1, n+1):

s = s + f(i)

return s

def f\_k\_1(i):

return 1/i

def f\_k\_2(i):

return i\*\*2

↓ sau + altă fct.

n = 10

s = suma\_generica(n, f\_k\_1)

#### 5. funcții anonime lambda:

suma a două numere: lambda x, y: x+y

testarea parității: lambda x: x%2 == 0

testarea divizibilității: lambda x, y: False if y == 0 else x%y == 0

suma cifrelor unui nr: lambda x: sum([int(c) for c in str(x)])

nr. voc. dintr-un șir: lambda sir: len([c for c in sir if c in "aeiouy"])

facem o listă cu voc. și după le nr. cu  
len

! le putem apela în momentul definiției

print((lambda x, y: x%y == 0)(24, 6)) → True

! le putem pune în antetul unei funcții ca și cum o apelați pe ex.  
f\_k-ul de la 4)

#### 6. funcții imbricate

→ funcții în interiorul altor funcții care nu pot fi apelate în cadrul  
programului ci doar în cadrul funcției

#### monolocal

→ caută cea mai qrp. var.

## GENERATORI

→ un tip de funcție a cărei execuție nu se termină în mem.  
în care returnează o val., ceea ce permite returnarea mai multor val.  
între-o manieră secvențială

ex: `for`, `range`

### `yield`:

```
def generator_numere_pare(n):
```

```
    x = 0
```

```
    while x <= n:
```

```
        yield x
```

```
        x += 2
```

+ val. furnizate de generator pot fi accesate  
printr-un `for`

### `next`:

```
nr_pare = generator_numere_pare(10)
```

```
x = next(nr_pare, -1)
```

```
while x != -1:
```

```
    print(x)
```

```
    x = next(nr_pare, -1)
```

### `return`

→ poate fi utilizat pt. a întrerupe un generator

### `g.close`

→ închide generatorul (infinit sau nu)