# Laboratorul 1: Introducere. Funcții în Python

## 1. Despre Python

### Generalități

Python este un limbaj de programare **interpretat**, **dinamic** și **de nivel înalt**. Acesta suportă paradigme multiple de programare cum ar fi:

- programarea procedurală;
- programarea orientată pe obiecte;
- programarea funcțională.

Un *interpretor* poate rula (evalua) fragmente de program (expresii sau instrucțiuni) pe măsură ce ele sunt introduse.
Un *compilator* transformă programul *sursă* într-un fișier *executabil* care poate fi apoi executat de sine stătător.

### Mediul de lucru

Odată ce scriem exemple de program e util să le salvăm pentru a le putea refolosi. Programele Python se scriu în fișiere text cu extensia `.py` . Ele se pot edita cu orice *editor de texte* (de exemplu Notepad++, *nu* Word care editează fișiere `.doc` cu formatare specifică). Aceste programe salvate in fișiere `.py` se rulează din linia de comandă (cmd/terminal) cu comanda `python3 fisier.py`
Pentru a simplifica procesul putem folosi un IDE (Integrated Development Environment). Un IDE este un program care de obicei conține cel puțin 3 lucruri: editor de cod, compilator si debugger. Un astfel de IDE este IDLE. Acesta se poate instala foarte usor de pe `python.org` din sectiunea Downloads. Trebuie descărcat și rulat fișierul de instalare corespunzător sistemului de operare, după care veți putea folosi python atât din linia de comandă cât și din IDLE.

### IDLE

La deschiderea programului IDLE ne întâmpină o fereastră numită `IDLE Shell`. Interpretorul rulează în această fereastră. Aici putem rula fragmente de cod. În cazul în care dorim să scriem programe într-un fișier pe care îl putem salva și ulterior rula, putem apăsa *File->New File* pentru a deschide altă fereastră. Aici putem scrie cod care poate fi salvat într-un fișier. Pentru a compila și rula un program se apasă *Run->Run Module* sau tasta F5.
Fișierele folosite recent pot fi accesate ușor de la *File->Recent Files*.
Pentru a naviga în istoricul comenzilor din interpretor putem folosi:

- Alt-p : comanda anterioară
- Alt-n : comanda următoare

# 2. Python - Noțiuni fundamentale

## Evaluarea unor expresii

La prompterul >>> al interpretorului putem în general **evalua expresii**. Cele mai simple expresii sunt calculele cu numere, scrise în notația obișnuită din matematică. De exemplu, putem introduce:

```
>>> 2+3
```

iar interpretorul răspunde:

```
5
```

În cadrul primului chenar >>> reprezintă prompterul interpretorului (deci el nu trebuie introdus), iar în cel de-al doilea chenar, fără prompter, se găsește răspunsul dat de interpretor, și anume 5.
Spațiile în cadrul expresiei nu contează, putem scrie și:

```
>>> 2 + 3
```

Pe lângă operația de adunare, există și operații de scădere (-), înmulțire (*), împărțire (/), modulo (%) și ridicare la putere(**).

## Tipuri de date

Python pune la dispozitie 4 tipuri de date primitive:

- Integer
- Float
- String
- Boolean

Tipurile de date primitive sunt imutabile, adică, odată ce acestea au fost create ele nu se mai pot modifica. Dacă o variabilă x = 3 își modifică valoarea în 4, de fapt un nou întreg este creat și atribuit variabilei x. Nu putem face:

```
>>> 2 = 5
    SyntaxError: cannot assign to literal here. Maybe you meant
'==' instead of '='?
```

### 1. Integer

Tipul de date **integer (int)** reprezintă datele numerice cu sau fără semn, fără zecimale și de lungime nelimitată (numere întregi cu valori cuprinse între -∞ și ∞).

```
 Ex: 3, 6, -234.
```

## 2. Float

Tipul de date **float** este folosit pentru reprezentarea numerelor în virgulă flotantă, cu sau fără semn.

```
Ex: 3.34, -0.123456.
```

**! Atenție** 2 e o valoare întreagă. Pentru o valoare reală (float) trebuie să scriem 2.0 (sau prescurtat 2.). În Python conversia de tip de la int la float se face automat. Astfel, rezultatul operațiilor ce conțin atât numere întregi cât și numere reale va fi un număr real (de exemplu 5 + 2.0 va da 7.0).
Putem folosi și funcția float() dacă dorim să facem o conversie în mod explicit:

```
>>> float (3 * 2)
    6.0
```

## 3. String

Un **string** (șir de caractere) reprezintă o colecție de caractere, cuvinte sau fraze. Pentru a crea un string în Python, folosim semnele `''` (apostrof) sau `""` (ghilimele).
Exemplu: `'carte'` , `"23abc"`.
Asupra unui șir de caractere putem efectua diverse operații. Una din cele mai comune operații este concaterarea șirurilor de caractere. Aceasta se face prin operatorul + :

```
>>> 'abc' + 'def'
    'abcdef'
```

Caracterele dintr-un string se pot accesa direct prin `"string"[index]`:

```
>>> 'A string'[2]
    's'
```

Rezultatul este al treilea caracter (numerotarea caracterelor începe de la 0). Python permite și accesarea de caractere folosind valori negative pentru index. Pentru exemplul de mai jos, selectarea oricărui alt întreg care se află în afara intervalului [-8; 7] va genera o excepție:

```
'A' ' ' 's' 't' 'r' 'i' 'n' 'g'
 0   1   2   3   4   5   6   7
-8  -7  -6  -5  -4  -3  -2  -1
>>> 'A string'[-8]
```

```
    'A'
```

**! Atenție** Deoarece Python este un limbaj de programare **strongly typed**, în cazul în care doriți să realizați operații matematice cu șirurile de caractere ce conțin valori numerice conversia nu se va face automat, ci va trebui să o faceți folosind una din funcțiile int() sau float().

```
>>> 5 + int('2')
    7
```

Pentru mai multe operații ce pot fi aplicate șirurilor de caractere (precum conversia unui string la lowercase/uppercase, împărțirea unui string, înlocuirea unui anumit caracter dintr-un string, etc.) puteți consulta pagina.

## 4. Boolean

Tipul de date **boolean** reprezintă valoarea de adevăr a unei expresii. Acesta poate avea valorile *True* sau *False*. De obicei, un boolean se folosește în scrierea condițiilor sau în compararea unor expresii.

```
>>> 3 == 4
    False
```

Folosind operatorii de comparație == (egal), != (diferit), > (mai mare), < (mai mic), >= (mai mare sau egal), <= (mai mic sau egal), putem face comparații între diferite expresii.
Pentru a scrie condiții mai complicate puteți folosi cuvintele cheie **and**, **or** și **not**.
Atunci când folosiți and, dacă cei doi operanzi au valoarea logică True, atunci rezultatul final va fi True. Dacă cel puțin un operand e False, întreaga expresie va fi evaluată ca fiind False.

```
>>> 3 == (2 + 1) and (3 + 1) != (2 ** 2)
    False
```

În cazul operatorului logic or, este suficient ca măcar unul din cei doi operanzi să fie adevărat pentru ca expresia rezultată să fie adevărată. Dacă cei doi operanzi sunt False, atunci și rezultatul expresiei va fi tot False.

```
>>> 3 == (2 + 1) or (3 + 1) != (2 ** 2)
    True
```

Operatorul logic not se aplică pentru un singur operand. Dacă acesta are valoarea True, atunci rezultatul final va fi false. Dacă operandul are valoarea False, atunci rezultatul final va fi True.

```
>>> not (3 + 1) != (2 ** 2)
```

```
    True
```

Pe lângă aceste tipuri de date fundamentale, Python mai oferă 4 tipuri predefinite pentru a putea reține colecțiile de date. Astfel, în Python putem lucra cu următoarele:

- Listă
- Tuplu
- Mulțime (Set)
- Dicționar

În continuare prezentăm doar modul de definire al tipurilor de date **listă** și **tuplu** în Python (mai multe detalii despre aceste tipuri, dar și despre celelalte două, vor fi prezentate în cadrul laboratoarelor viitoare). Astfel dacă dorim să definim o listă vom enumera între paranteze pătrate elementele acesteia, elementele fiind despărțite între ele prin virgulă:

```
Exemplu: even_digits_list = [0, 2, 4, 6, 8]
```

Dacă dorim să accesăm un element al listei procedăm la fel ca și în cazul șirurilor de caractere (Exemplu: dacă dorim să accesăm elementul 2 din lista *evenDigits* vom scrie *evenDigits[1]*, numerotarea făcându-se și în acest caz de la 0).

În mod asemănător listelor, putem defini un **tuplu** folosind paranteze rotunde:

```
Exemplu: even_digits_tuple = (0, 2, 4, 6, 8)
```

# Tipărirea

Până acum ne-am folosit de faptul că interpretorul afișează automat rezultatul unei evaluări. Pentru a scrie și rula programe de sine stătătoare sunt însă necesare funcții care să tipărească valori.
Python dispune de funcția print(), ce poate fi folosită pentru a tipări pe ecran diverse mesaje sau tipuri de date ce vor fi convertite în string. Caracterul de linie nouă se reprezintă ca în C: `'\n'`.
Putem trece la linie nouă apelând `print('\n')`.
Funcția print() acceptă un număr variabil de parametri. Astfel, putem tipări diferite șiruri de caractere în felul următor:

```
>>> print("Hello.", "How are you?")
    Hello. How are you?
```

De asemenea, putem specifica un separator între șirurile de caractere printate, prin atribuirea unui șir de caractere parametrului `sep` al funcției print:

```
>>> print("Hello.", "How are you?", sep="\n-----\n")
    Hello.
    -----
    How are you?
```

Putem folosi funcția() print și într-un mod asemănător funcției printf din C, în felul următor:

```
>>> print("Result: %d + %.2f = %.2f" % (2, 3.25, 2 + 3.25))
    Result: 2 + 3.25 = 5.25
```

# Structura condițională

În mod implicit, codul din Python se execută secvențial, linie cu linie, însă pot exista situații în care o instrucțiune să fie executată doar dacă o anumită condiție este îndeplinită. Pentru a realiza acest lucru putem utiliza următoarea structură:

```
if condition_1:
   instruction_block_1
elif condition_2:
   instruction_block_2
else:
   instruction_block_3
```

Observații

- pot exista oricât de multe ramuri elif;
- ramură elif poate lipsi complet din cadrul structurii;
- ramura de else este opțională (poate apărea o dată sau deloc).

Exemplu:

```
if (x < 0):
    print("x < 0")
elif (x <= 1):
    print("0 <= x <= 1")
else:
    print("x > 1")
```

În cadrul exemplului de mai sus se verifică dacă variabila x este mai mică decât 0, caz în care se tipărește "x<0". Dacă x este mai mare sau egal cu 0 (adică prima condiție nu este verificată), se va verifica condiția de pe ramura elif, și anume dacă x este mai mic sau egal cu 1. În cazul în care nici această condiție nu este verificată (adica x este mai mare decât 1) se va executa instrucțiunea corespunzătoare ramurii else.

# Particularități de sintaxă

## 1. Indentarea

În Python, indentarea este foarte importantă în scrierea codului. Spre deosebire de alte limbaje de programare ce folosesc acolade { }, în Python indentarea este folosită pentru blocurile de cod. Indentarea este întotdeauna precedată de semnul **:** (două puncte).

## 2. Comentarii

Comentariile sunt folosite pentru a explica porțiuni de cod. În Python comentariile încep cu simbolul #:

```
#This is a single line comment.
```

De asemenea comentariile pot fi plasate și la finalul unei linii:

```
print("Python") #This is another example.
```

În cazul în care comentariul se întinde pe mai multe rânduri, fiecare linie poate să înceapă cu # sau putem opta pentru utilizarea """ (3 ghilimele la începutul comentariului și 3 ghilimele la sfârșitul acestuia):

```
"""This is
 a
 multiline comment"""
 print(5 + 2)
```

# Introducerea de cod extern (Module)

La fel ca și in alte limbaje de programare și în Python se pot introduce funcții și constante din alte fișiere. Un exemplu util în acest sens este dat de includerea modulului math (puteți consulta documentația oficială pentru mai multe informații despre constantele și funcțiile pe care le pune la dispoziție).

```
>>> import math
```

Pentru a folosi acum funcții precum floor sau ceil:

```
>>> math.floor(3.7)
    3
>>> math.ceil(3.7)
    4
```

Daca dorim să redenumim modulul importat putem folosi:

```
>>> import math as m
```

Acum putem folosi funcțiile apelând:

```
>>> m.floor(3.7)
    3
```

**Observație!** Funcția `int()` poate fi folosită pentru a trunchia partea fracționară.

# 3.Programarea funcțională

În programarea funcțională, programele sunt construite prin aplicarea și compunerea funcțiilor. Spre deosebire de programarea procedurală, unde se folosesc secvențe de instrucțiuni ce modifică starea programului, programarea funcțională se bazează pe evaluarea funcțiilor matematice, evitând astfel starea și datele mutabile.
Deși Python ne permite lucrul direct cu date și modificarea lor, pentru a programa funcțional este foarte important să nu scriem funcții ce modifică starea globală a programului, în momentul apelării lor.

# 4.Funcții în Python

## Definire

În Python funcțiile se definesc cu ajutorul cuvântului cheie **def**. După cuvântul cheie, este trecut numele funcției, urmat de paranteze rotunde unde se pun parametri acesteia separați prin virgulă, iar la final este adăugat și simbolul :(două puncte). Daca funcția nu are niciun parametru, nu veți pune nimic între paranteze, însă prezența acestora este obligatorie.

**Observație!** Nu uitați să indentați întreg corpul funcției!

```
def function_name(param_1, param_2, param_3, ..., param_n):
    #some_instructions
```

Astfel, funcția $f : \mathbb{Z} \to \mathbb{Z}$, $f(x) = x + 3$ se scrie în Python:

```
def f(x):
    return x + 3
```

Putem defini funcția și cu ajutorul interpetatorului:

```
>>> def f(x):
...     return x + 3
...
```

Apasăm de doua ori enter pentru a semnala interpretorului că am terminat de scris codul. Funcția va rămâne definită în memorie până la închiderea interpretorului. Prompterul ... al interpretorului ne indică faptul că pe următoarea linie se continuă codul.

## Apelarea funcțiilor

Odată definită funcția, aceasta se apelează în felul următor:

```
>>> f(1)
    4
```

Când apelăm o funcție putem să specificăm și numele parametrului la apel:

```
>>> f(x=5)
    8
```

Putem da funcției și o expresie complexă ca parametru:

```
>>> f(2*3)
    9
```

## Funcții cu mai mulți parametri

Fie funcția din matematică: *integer_add:* $\mathbb{Z}\times\mathbb{Z} \rightarrow \mathbb{Z}$, *integer_add(x, y) = x + y*
Varianta cea mai uzuală de a transcrie această funcție într-un program Python este:

```
def integer_add(x, y):
     return x + y
```

## Funcții care returnează mai multe valori

În Python putem implementa funcții care returnează mai multe valori cu ajutorul listelor. De exemplu, dacă dorim să implementăm funcția f: $\mathbb{Z}\times\mathbb{Z} \rightarrow \mathbb{Z}\times\mathbb{Z}$, *f(x, y) = [x + y, x - y]* putem scrie astfel:

```
def f(x, y):
    return [x + y, x - y]
```

O altă modalitate prin care putem scrie astfel de funcții este prin folosirea tipului de date Tuplu:

```
def f(x, y):
    return x + y, x - y
```

Exemplu de apelare:

```
result_1, result_2 = f(2, 1)
```

# Exemplu

În programarea funcțională dorim să evaluăm expresii, nu să executăm o serie de instrucțiuni. Astfel, valoarea returnată de către o funcție se va folosi ca parametru de către altă funcție.
Totuși, în exemplul de mai jos, deși se execută print() în interiorul unei funcții, acest apel nu afectează starea sistemului.

```
def print_abs(x):
    if (x > 0):
        print("Positive: " + str(x))
        return x
    else:
        print("Negative: " + str(x))
        return -x
```