Elemente de bază ale limbajului Python

Principalele diferențe față de C/C++

- Variabilele în Python nu au tip de date static: **nu se declară** tipul lor, o variabilă este "**declarată**" **când i se atribuie prima dată o valoare**. Tipul unei variabile se poate schimba pe parcursul execuției programului (vom reveni) – el se stabilește dinamic, în funcție de valoarea pe care variabila o primește la un moment dat.

```
x = 7 #variabila x este "declarata"
print(x)
x = "abc"
print(x)
```

- **Indentarea blocurilor de cod este obligatorie, fiind suficientă** pentru delimitarea acestora)**; nu sunt necesari delimitatori de blocuri** de tip {} (sau begin/end etc ca în alte limbaje).
- Nu este nevoie sa punem ; la finalul unei linii (dacă nu mai urmează alte linii de cod pe aceeași linie)
- Un comentariu pe o linie începe cu #

Exemplu: afișarea numerelor de la 1 la 10 folosind while

```
i = 1
while i<10:
     print(i)
     i = i + 1   #nu i++
print("gata afisarea")
```

Exemple – Operații fundamentale

1. Afișarea unei variabile + tipul acesteia (al valorii asignate)

Afișarea se poate face folosind funcția **`print`**, care poate primi mai mulți parametri, separați cu virgulă. **Implicit** valorile transmise vor fi afișate pe același rând separate cu un spațiu, apoi se va trece pe linie nouă. Putem modifica acest comportament folosind parametri **cu nume `sep`** și **`end`** ai funcției **`print`**.

```
print("mesaj")
x = 1
print("x=",x, type(x), id(x)) #pe acelasi rand cu spatiu, apoi linie noua
print("x=" + str(x))
x = "Sir"
print("x=", x, type(x), id(x)) #nu are acelasi id
y = 2
print(x, end=' ')#pentru a nu trece la linie noua modific parametrul end
print(y)
print(x, y, sep='*')
```

Funcția **`print`** are număr variabil de argumente; parametrul opțional **`sep`** primește ca valaore șirul separator al argumentelor afișate (implicit spațiu), iar parametrul opțional **`end`** – șirul de la sfârșitul afișării (implicit linie noua)

2. Citirea de la tastatură + funcții de conversie

Citirea de la tastatură se poate face folosind funcția input care primește ca parametru (opțional) mesajul care se va afișa pe ecran și returnează șirul de caractere introdus **până la sfârșitul de linie** (sau folosind fișierul standard de input **`sys.stdin`**, vom reveni la fișiere). Rezultatul **returnat este de tip `str`,** de aceea dacă se citește un număr întreg **este necesară conversia** de la **`str`** la **`int`**.

```
#citire-necesara conversie
x = input("x=")
print("x=", x, type(x))
x = int(input("intreg=")) #ValueError daca introducem gresit
print("x=", x, type(x))
x = float(input("real="))
print("x=", x, type(x))
x = complex(input("complex="))
print("x=", x, type(x))
```

3. Erori

Erorile sunt semnalate la rulare, când programul ajunge cu execuția la acel punct.

Exercițiu: Rulați programul următor pentru i inițializat întâi cu 1, apoi cu -1, apoi cu "ab". Adăugați linia print(y) pe ramura else si rulați din nou programul pentru aceste valori

```
i = 1  #i="ab", i=-1
print(i)
if i>0: #daca i nu este numar? TypeError
    print("ok")
else:
    print(i + " este negativ") #daca i nu este sir de caractere?
    print(y) #da eroare daca i=1?
```

Variabile

În C/C++ o variabilă se declară și are: tip (declarat), adresă, valoare.
În Python variabilele (amintim) nu au tip de date static (valorile au tip), li se pot asocia valori de tipuri diferite pe parcursul execuției programului. În limbajul Python, o variabilă nu conține o valoare, ci o referință spre un obiect care conține valoarea respectivă (un nume pentru obiect). Astfel, printr-o instrucțiune de atribuire nu se copiază valoarea respectivă, ci doar referința sa.
Variabilele sunt **referințe spre obiecte (nume date obiectelor);** orice valoare este a unui obiect.
Un obiect **ob** are asociat:

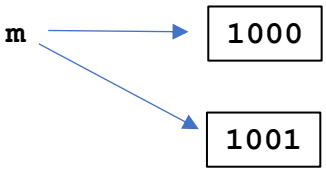
- un număr de identificare: **id(ob)**
- un tip de date: **type(ob)**
- valoare – poate fi convertită la șir de caractere **str(ob)**

Numele unei variabile este un identificator (recomandare de numire a unei variabile: **litere_mici_separate_prin_underscore)**

Exemplu – cum se alocă memorie, ce înseamnă atribuire, variabilă vs obiect
Să considerăm următoarea secvență de cod:

```
m = 1000
print(m, id(m))

m = m + 1
print(m, id(m))
```



După prima atribuire m = 1000 se creează un obiect de tip int cu valoarea 1000 și variabila m referă acest obiect (este un nume pentru acesta). La executarea celei de a doua atribuiri **`m = m + 1`**, se creează un nou obiect cu valoarea 1001 și variabila m referă acum acest obiect (deci are un alt id, fiind referință către un alt obiect), ca în figura alăturată codului.

Exercițiu: Ce se întâmplă dacă se execută **în continuarea** secvenței din exemplu instrucțiunile următoare:

```
n = m
print(n, id(n), id(m))
n = n + 1
print(n, id(n))
```

Explicații:

`n = m` **=> n și m sunt acum și nume /referinte pentru acelasi obiect, deci id(m) = id(n) (m is n)**

```
print(n, id(n), id(m))
```

`n = n + 1` **=> se creează un nou obiect cu valoarea n+1=1002 și n va arăta către acesta, deci id(n) se modifică**

```
print(n, id(n))
```

Pentru a optimiza memoria folosită (a nu crea obiecte noi cu aceeași valoare), numerele întregi din intervalul [-5, 256] sunt prealocate (în cache) – **toate obiectele care au o astfel de valoare sunt identice (au același id).**

Exemplu: În următorul tabel sunt date două exemple în care se creează obiecte noi de aceeași valoare, în prima coloană valoarea fiind mai mică decât 256, iar în a doua mai mare.

Valori mici, prealocate => toate obiectele cu acea valoare sunt identice	Valori mai mari => obiectele pot fi diferite, deși au aceeași valoare

`x = 1`
`y = 0`
`y = y + 1`
`z = x`
`print(x, y, z, x*x)`
`print(id(x),id(y),id(z),id(x*x))`
	`x = 1000`
	`y = 999`
	`y = y + 1`
	`z = x`
	`print(x, y, z, 10*x//10)`
	`print(id(x),id(y),id(z),id(10*x//10))`

La rulare se observă că toate expresiile cu valoarea 1 au același id (și x și x*x și y). În exemplul din a doua coloană, deși x, y și 10*x//10 sunt egale, nu au neapărat același id

Concluzie: Variabile cu aceeași valoare **pot avea** același id (dacă este o valoare prealocată, atunci sigur da), dar nu este neapărat (pot exista obiecte diferite cu aceeași valoare)

Pentru a șterge o variabilă din memorie se folosește instrucțiunea **del**. Mecanismul de Garbage collector va șterge obiecte către care nu mai sunt referințe:

```
m = input()
del m
print(m)#eroare
```

Tipuri de date

În Python fiecărui tip de date îi corespunde o clasă predefinită, iar constantele și variabilele sunt obiecte, adică instanțe ale clasei respective (! și literalii de tip int sunt obiecte)

- **int**
  - În Python se pot memora numere întregi (! cu semn) cu oricât de multe cifre (limita dată doar de performanța sistemului pe care se rulează). Acestea sunt memorate ca vectori de "cifre" din reprezentarea în baza 2^{30} (cu cifre de la 0 la 2^{30}-1= 1073741823)

  Exemplu: Numărul 234254646549834273498 se reprezintă astfel:

ob_size	**3**		
ob_digit	**462328538**	**197050268**	**203**

  deoarece 234254646549834273498 = 462328538 × $(2^{30})^0$ + 197050268 ×$(2^{30})^1$ + 203 × $(2^{30})^2$

  - Constantele întregi se consideră implicit în baza 10, dar există prefixe care se pot pune în fața lor pentru a fi privite în alte baze: baza 2 (prefix 0b,0B), baza 8 (prefix 0o, 0O), baza 16 (prefix 0x,0X):

    ```
    print(0b101, 0o10, 0xAb)
    ```

  - Pentru a converti o valoare la tipul **`int`** (de exemplu un șir de caractere) putem folosi constructorul (funcția) **`int(sir)`** (există și varianta **`int(sir, base=baza))`**

    ```
    print(int(9.7) + int("101", base = 2) + int("101",2))
    ```

- **float**
  - memorarea valorilor de tip real ("cu virgulă") folosind reprezentarea în virgulă mobilă cu dublă precizie din standardul IEEE-754
  - Constante: 3.5, 1e-2 (notație științifică)
  - Conversie folosind **`float([x])`**
  - Există **`float("inf"); float("infinity");`** **(pentru ∞)** și **`float("nan")`**
  - Operațiile aritmetice cu tipul de date float **nu au precizie absolută**; astfel, nu se recomandă să verificăm dacă două numere reale a și b sunt egale folosind == , ci să verificăm dacă ele au primele zecimale identice, cu o expresie de tipul
    abs(a - b) <= 1e-9:

    NU: `0.1 * 0.1 == 0.01  #print(0.1 * 0.1 == 0.01)`
    DA: `abs(0.1*0.1-0.01) < 1e-9`

- ## complex

  - de forma a + bj **(!!! nu i**, merge și J**)**

    Exemplu de utilizare:

```
z = complex(-1, 4) #creare folosind constructor
print("Numarul complex:", z)
print("Partea reala:", z.real)
print("Partea imaginara:", z.imag)
print("Conjugatul:", z.conjugate())
print("Modul:", abs(z))
```

- ## bool

  - **`True`**, **`False`**
  - **`bool()`** pentru conversie
  - în context boolean (condiție if, while; operand pentru operatori logici) **orice valorare se convertește la bool.** Se consideră **`False`** următoarele:
    - **None**, **False**
    - **0**, **0.0**, **0j, Decimal(0), Fraction(0,1)**
    - Colecții și secvențe vide (+obiecte în care **__bool__()** returneaza **False** sau **__len__()** returnează **0**)

    (restul se consideră **`True`**)

    Exercițiu: Justificați ce afișează următoarea secvență de cod:

```
print(bool(0), bool(-5))
print(bool(""), bool(" "), bool("0"))
print(bool(None), bool([]))
```

- ## NoneType
  - Constanta **None**

- Nu există tipul de date **char (doar str pentru șirul de caractere),** există însă funcții pentru determinarea codului unui caracter (memorat ca șir de lungime 1) și pentru determinarea caracterului cu un cod dat (tipul returnat fiind str) :

```
ord("a")
chr(97)
print(chr(97), type(chr(97)))
```

Următoarele tipuri de date le vom discuta în cursurile viitoare:

- **Secvențe:** pentru memorarea unor șiruri de valori, indexate de la 0
  - Mutabile (le putem modifica elementele) și imutabile
    - liste - clasa **list**: a = [3, 1, 4, 7] - mutabile
    - tupluri – clasa **tuple**: a = (3, 1, 4, 7)
    - șiruri de caractere - clasa **str**: a = "3147sir", a = '3147sir'

    (tuplurile și șirurile de caractere sunt imutabile)

- **Mulțimi:** memorarea unor valori fără duplicate (mulțimi) și efectuarea operațiilor specifice mulțimilor
  - clasa **set**: a={1, 4, 5}
  - clasa **frozenset**: fa = **frozenset**(a) – nu se poate modifica
- **Dicționare:** memorarea unor perechi de forma *cheie:valoare* (tabele asociative)
  - clasa **dict**

Operatori

Caracteristicile principale ale unui operator sunt:

- **aritate** (număr de operanzi cărora li se aplică); de exemplu, operatorul – are aritatea 1 în expresia –1 și aritatea 2 în expresia 3 – 1
- **prioritate** (precedența) – în funcție de ea se stabilește ordinea de evaluare a operatorilor dintr-o expresie; spre exemplu, operatorul * are prioritate mai mare față de +, de aceea în expresia 1 + 2 * 3 se va evalua întâi 2*3 = 6, apoi 1 + 6 = 7
- **asociativitatea** (de la stânga la dreapta sau de la dreapta la stânga) – în funcție de asociativitatea unui operator se stabilește ordinea în care se fac operațiile într-o expresie în care un operator se repetă sau, mai general, sunt operatori cu aceeași prioritate; de exemplu, în expresia 1 + 2 + 3 se va efectua întâi 1 + 2 (operatorul + are asociativitate de la stânga la dreapta, ca majoritatea operatorilor în Python:

  1+ 2+ 3 = (1+2)+3

  dar în expresia 10 ** 2 **7 (unde ** este operatorul de ridicare la putere) se va efectua întâi 2 ** 7, **** având asociativitatea de la dreapta la stânga**:

  10 ** 2 **7 = 10 ** (2 **7)

1. Operatori aritmetici

+	adunare
-	scădere
*	înmulțire
/	Împărțire exactă, **rezultat float** (nu ca în C/C++ sau Python 2)
//	împărțire cu rotunjire la cel mai apropiat întreg mai mic sau egal **decât rezultatul împărțirii exacte** **dacă un operator este float rezultatul este de tip float**
%	restul împărțirii, se calculează astfel: **x%y = x – ((x//y)*y)**
**	ridicare la putere

Operatorii se pot folosi **pentru tipurile pentru care au sens** (de exemplu și pentru numere complexe)

Tipul rezultatului se determină folosind principii similare celor din C/C++ (de exemplu: **int** + **float** este de tip **float**), principalele diferențe fiind la / și //

Exercițiu: Care este valoarea și de ce tip sunt fiecare dintre următoarele expresii:

```
3 + 2.0
2j * 3j
1 / 1
1 / 2
4 // 2
5 // 2.5
2.5 // 1.5
11 // 3
11 // -3
-11 // 3
-11 // -3
10.5 % 2
3 % 1.5
```

Câteva explicații:

- `3 + 2.0` este de tip float (5.0)
- `2j * 3j` este egal cu -6 văzut ca număr complex, adică (-6+0j)
- `11 // -3 == -4` deoarece se face rotunjire la cel mai apropiat întreg mai mic sau egal decât rezultatul împărțirii exacte
- `5//2.5` – rezultatul este de tip float (2.0)
- `11% -3 = 11 – ((-3) * (11//-3)) = 11 – (-3) * (-4) = 11 - 12 = -1`

2. Operatori relaționali

x == y	x este egal cu y
x != y	x nu este egal cu y
x > y	x mai mare decât y
x < y	x mai mic decât y
x >= y	x mai mare sau egal y
x <= y	x mai mic sau egal y

Operatorii relaționali se pot înlănțui: **1 < x < 10**

Exemplu – Se citește un număr natural x. Să se verifice dacă x are două cifre.

```
x = int(input())
if 10 <= x <= 99:
    print("da")
else:
    print("nu")
```

Spre deosebire de ==, operatorul **is** testează dacă două obiecte sunt identice (au același id). Două obiecte care au aceeași valoare nu sunt neapărat identice (amintim că doar valorile mici sunt prealocate, deci toate obiectele cu aceeași valoare în acest caz sunt identice)

Exemplu: Justificați rezultatele afișat de secvențele de cod următoare:

```
x = 1000
y = 999
y = y + 1
print(x == y)
print(x is y)
```

```
x = 1
y = 0
y = y + 1
print(x == y)
print(x is y)
```

3. Operatori de atribuire

- **=**
  Instrucțiunea de atribuire, de exemplu x = 2 este o instrucțiune, nu o expresie care se evaluează la o valoare, ca în C. Astfel, nu sunt corecte sintaxe de genul **`if x=2:`** sau **`y = (x=2) + 5*x`**
- Din versiunea 3.8 a fost introdus și operatorul de **atribuire în expresii** (operatorul **walrus**) **`:=`**
  Astfel, nu este corectă instrucțiunea **`print(x=1)`**, dar sunt corecte:

```
print(x:=1)
y = (x:=2) + 5*x # y = 5*x +(x:=2)
print(x,y)
if x := y :
    print(x,y)
while (x:=int(input()))>0: #trebuie ()
    print(x)
```

- **`+=, -=, *=, /=, **=, //=, %=,`**
- **`&=, |=, ^=, >>=, <<=`** (v. operatori pe biți)

Observație: În Python nu există operatorii ++ și --

4. Operatori logici

- **`not, and, or`**

Operatorii logici se evaluează prin **scurtcircuitare**. Astfel, evaluarea unei expresii de forma **`expr_1 and expr_2 and … and expr_n`** se face de la stânga la dreapta și se va opri la prima expresie a cărei valoare este False (dacă există), deoarece, în acest caz, valoarea întregii expresii va fi False. Similar, evaluarea unei expresii de forma expr_1 or expr_2 or … or expr_n se oprește la prima expresie a cărei valoare este True.

Exemplu: La execuția următoarei secvențe de cod se va afișa **`True`**, deși variabila y nu există; la evaluarea expresiei x or y nu se mai ajunge la evaluarea lui y deoarece x este **`True`**:

```
x = True
print(x or y) # print(x and y)
```

Dacă înlocuim **`print(x or y)`** cu **`print(x and y)`** atunci la execuție se va semnala eroarea **`NameError: name 'y' is not defined`**

În context boolean orice valoare (de orice tip) se poate evalua ca True/False. Astfel, operatorii logici nu se aplica doar pe valori de tip **bool** (ci pentru orice valori), iar rezultatul nu este neapărat de tip bool (decât în cazul operatorului not). Mai exact:

$$x\ and\ y = \begin{cases} y, & dacă\ x\ se\ evaluează\ ca\ True \\ x, & altfel \end{cases}$$

$$x\ or\ y = \begin{cases} x, & dacă\ x\ se\ evaluează\ ca\ True \\ y, & altfel \end{cases}$$

$$not\ x\ = \begin{cases} False, & dacă\ x\ se\ evaluează\ ca\ True \\ True, & altfel \end{cases}$$

De exemplu, expresia "`a" and True` are valoarea `True` (a ultimei expresii evaluate), iar expresia `"a" or True` are valoarea `"a"` (deoarece expresia "a" se evaluează ca fiind True și evaluarea se face prin scurtcircuitare).

Observații:

1. Operatorul **not** are prioritate mai mică decât operatorii de alte tipuri:
   - `not a == b` ⇔ `not (a == b)`
   - `a == not b` eroare de sintaxă, se încearcă întâi evaluarea egalității `a == not`
2. Operatorul `and` are prioritate în fața operatorului `or`. De exemplu, expresia
   `11 or "" and 3`
   se evaluează după cum arată parantezele în expresia următoare `11 or ("" and 3)` și va avea valoarea 11 (de ce?)

Exercițiu: ce afișează următoarea secvență de cod:

```
x = 0
y = 4
if x:
    print(x)
print(x and y)
print(x or y)
print(not x, not y)
print( y or (x<y))
print((x<y) and y)
print((x<y) or y)
```

Dar dacă inițializăm variabilele astfel:
x = -1
y = "0"

5. Operatori pe biți

Operatorii pe biți se folosesc pentru numere întregi și acționează asupra reprezentărilor binare ale lor (fiind rapizi). În limbajul Python toate numerele întregi sunt considerate **cu semn** și sunt reprezentate intern în complement față de 2.

Amintim (v. seminar) că pentru a reprezenta binar un număr negativ x se urmează pașii:

- se reprezintă în baza 2 modulul lui x (|x| = -x)
- se calculează complementul față de 1 a valorii obținute anterior (trecând fiecare bit din reprezentare la complement, mai exact interschimbând 0 ↔ 1)
- se adună 1 la rezultatul obținut

Exemplu: Presupunând pentru simplitate că avem doar 8 biți, reprezentările lui 11 și -11 se obțin astfel:

- x = 11: împărțim numărul succesiv la 2 și luam resturile (care vor fi cifrele reprezentării binare, de la ultima către prima)

  11 % 2 = **1**

  x = 11//2 =5

  5 % 2 = **1**

  x = 5 // 2 = 2

  2 % 2 = **0**

  x = 2 // 2 = 1

  1 % 2 = **1**

  x = 1 // 2 = 0 – stop, reprezentarea este **00001011**
- x = -11

  Reprezentăm binar 11: **00001011**

  Trecem la complement: **11110100**

  Adunăm 1: **11110101**

Operatorii pe biți în Python sunt următorii:

~x	complement față de 1 (obținut prin negarea fiecărui bit din reprezentarea lui x)
x & y	și pe biți (bitul i din reprezentarea rezultatului expresiei x&y se obține aplicând operatorul și între biții de pe poziția i din x și din y, ca în exemplele care urmează)
x | y	sau pe biți
x ^ y	xor (sau exclusiv) pe biți
x >> k	deplasare la dreapta cu k biți
x << k	deplasare la stânga cu k biți

În continuare exemplificăm fiecare dintre operatorii pe biți, amintind și modul de funcționare prin tabelele asociate.

- **Operatorul ~**

~	0	1
	1	0

x = 11 se reprezentă binar => **00001011** (presupunem pe 8 biți)
~x va avea reprezentarea binară => **11110100** – pentru a afla numărul care are această reprezentare urmăm pașii de la reprezentarea în complement față de 2 invers (începe cu 1 => ~x este număr negativ):

- scădem 1 => **11110011** (începe cu 1 => ~x este număr negativ)
- trecem la complement => **00001100** – am obținut reprezentarea binară a modulului

lui ~x, deci modul de ~x este $2^2 + 2^3 = 12$. Rezultă că valoarea lui ~x este **-12**

Observație: Pentru un bit b avem **~b = 1 – b,** iar pentru un număr x avem
~x = -(x + 1) = -x -1

- **Operatorii &, |, ^**

&	0	1
0	0	0
1	0	1

|	0	1
0	0	1
1	1	1

^	0	1
0	0	1
1	1	0

x = 11 – binar **00001011**
y = 86 – binar **01010110**

x & y – binar **00000010** = 2 în baza 10 (print(x&y) va afișa 2)
x ^ y – binar **01011101** = 93 în baza 10
x | y – exercițiu

- **Operatorul <<** (left shift)

x << k = numărul obținut deplasând la **stânga** biții din reprezentarea binară a lui x cu k poziții (astfel primii k biți din reprezentare dispar) și adăugând la sfârșitul reprezentării binare k biți nuli.

Exemplu: x = 11 = **00001011**
x << 3 = **01011000** = 88 = 11 * (2**3)

În general expresia **x = x << k** este echivalentă cu expresia **x = x * (2**k)**

- **Operatorul >>** (right shift)

Exemplu: x = 11 = **00001011**

x >> 3 = **00000001** = 1 = 11 // (2**3)

În general expresia **x = x >> k** este echivalentă cu expresia **x = x // (2**k)**

Exercițiu: Funcția bin(x) returnează reprezentarea binară a parametrului x. Ce afișează programul următor? Justificați.

```
x = 272
print(bin(x))
print(bin(x&0b10001),x&0b10001)
print(bin(17|0b10001),17|0b10001)
print(bin(~x),~x)
print(bin(x>>1),x>>1)
```

Aplicații (alte aplicații se vor discuta la seminar)

1) **Testarea parității unui număr natural**

```
x = int(input())
if x&1 == 0: #ultimul bit din reprezentarea binara este 0
   print("par")
else:
    print("impar")
```

2) **Interschimbarea conținutului a două variabile**

```
x = int(input("x="))
y = int(input("y="))
x = x^y
y = x^y # y = (x ^ y) ^ y = x ^ (y ^ y) = x ^ 0 = x
x = x^y # x = (x ^ y) ^ x = x ^ (y ^ x) = (x ^ x) ^ y = 0 ^ y = y
print(x,y)
```

6. Operatorul condițional (ternar)

expresie_1 **if** *expresie_logica* **else** *expresie_2*

Operatorul condițional este un operator ternar și furnizează valoarea expresiei *expresie_1* dacă *expresie_logica* este True, sau valoarea expresiei *expresie*_2 în caz contrar (se evaluează tot prin scurtcircuitare).

Exemple:

```
x = 5; y = 20
z = x-y if x > y else y-x
print("Modulul diferentei", z)
z = x if x > y else ""
```

7. Operatori de identitate: is, is not

Expresia `x is y` este `True` dacă și numai `x` și `y` referă același obiect, deci dacă și numai dacă `id(x) == id(y)` – vezi exemplul de la operatori relaționali

8. Operatori de apartenență : in, not in (la o colecție)

Sunt folosiți pentru a testa apartenența unui element la o colecție (număr in listă de numere, caracter în șir de caractere, șir într-un alt șir de caractere) - vom reveni la colecții

Exemplu:

```
s = "aeiou"
x = "e"
print("vocala" if x in s else "consoana")
s = "o propozitie"
s1 = "prop"
print (s1 in s, s1 in s[4:])
ls = [0, 2, 10, 5]
print(3 not in ls)
```

Precedența operatorilor

Un tabel cu precedența operatorilor se găsește la adresa:
https://docs.python.org/3/reference/expressions.html#operator-precedence

Este important să știm precedența operatorilor, altfel sunt cazuri in care nu este evidentă ordinea operațiilor și putem greși. Pentru a fi siguri puteți consulta tabelul de precedență sau folosi paranteze.

Exercițiu: Cum se evaluează următoarele expresii?

```
1 + 1 << 2
2 * 3 ** 4
2 - 3 ** 4
2 ** - 1
```
Atenție, ** are prioritate mai mare decât -, dar nu are sens expresia (2 ** -) 1

```
grupa // 10 == 14 or grupa//10 == 13 and media >= 9
```

Comentarii

Pe o linie, semnul # marchează începutul unui comentariu

Pentru a comenta mai multe linii putem pune # la începutul fiecărei lini sau putem folosi delimitatori de șiruri de caractere:

- Încadrat de ' ' ' (trei apostrofuri)
- Încadrat de " " " (trei ghilimele) => docstring – comentariu pe mai multe linii, folosit în mod special pentru documentare

Instrucțiuni

1. Instrucțiunea de atribuire

- atribuire simplă: `x = 5`
- atribuire înlănțuită / multiplă `x = y = 5`
- atribuire simultană / compusă (**tuple assignment**) `x, y, z = 5, 6, 7`

Atribuirea `x, y, z = 5, 6, 7` se numește atribuire de tupluri deoarece este echivlanetă cu atribuirea `(x, y, z) = (5, 6, 7)`, unde unui tuplu (secvență similară listei, dar

imutabilă) i se atribuie un alt tuplu, atribuirea făcându-se **element cu element**. De aceea, o atribuire de genul `x, y = y, x` are ca efect interschimbarea valorilor variabilelor x și y.

Exerciții:

1. Ce valori au variabilele x și y după fiecare atribuire din următoarea secvență de cod?

```
x = 1
x = y = 1
x, y = 1, 5
x, y = y, x #!!! Interschimbare (tupluri)
print(x,y)
x, y = x if y> x else y, y if y> x else x
print(x,y)
```

2. Ce valori au variabila i și vectorul v după executarea următoarei secvențe de cod?

```
v = [11, 12, 13, 14]
i = 2
i, v[i] = v[i], i #???mai bine v[i], i = i, v[i]?
```

2. Instrucțiunea de decizie (condițională)/alternativă if

Există mai multe variante ale acestei instrucțiuni (în funcție de cum folosim și ramurile elif și else)

Varianta 1. Instrucțiunea de decizie

```
if expresie_logică:
    instructiuni
```

Exemplu:

```
a = int(input("introduceti un numar nenegativ")
if a<0:
  print("ati introdus un numar negativ, il vom considera in modul")
  a = -a
```

Varianta 2. Instrucțiunea alternativă

```
if expresie_logică:
   instructiuni_1
else:
   instructiuni_2
```

Exemplu:

```
a = int(input("a = "))
b = int(input("b = "))
if a > b:
     minim = a
else:
     minim = b
print("minimul dintre cele doua numere este ", minim)
```

Varianta 3. Instrucțiuni alternative imbricate se pot scrie prescurtat folosind elif

Exemplu:

```
k = int(input())
print('ultima cifra a lui 3**',k, 'este',end=" ")
r = k % 4
if r == 0:
    print(1)
elif r == 1:
    print(3)
elif r == 2:
    print(9)
else:
    print(7)
```

Observații:

- else poate lipsi
- În versiunile Python anterioare versiunii 3.10 nu exista o instrucțiune de decizie multiplă cum este, de exemplu, switch în C/C++.

3. Instrucțiunea de decizie/potrivire multiplă (potrivire de tipare) match

Din versiunea 3.10 există în Python o instrucțiune care poate fi privită și ca instrucțiune de decizie multiplă, similară instrucțiunii switch în C/C++, dar care oferă mai multe facilități legate de verificarea încadrării într-un tipar https://peps.python.org/pep-0636/ .

Forma generală a instrucțiunii match este:

```
match expresie:
      case tipar1: instructiune
      case tipar2: instructiune
      case _: instructiune
```

Se execută doar instrucțiunile din prima ramură corespunzătoare tiparului în care se încadrează expresia (nu și cele de după, ca la switch). În cazul în care expresia nu se încadrează în niciun tipar se execută instrucțiunile din ramura **case _** , dacă aceasta există (similar ramurii default pentru switch). Valorile din tipar se pot înlănțui folosind operatorul |

Exemplu:

```
x = int(input())
match x:
    case 1: print("luni")
    case 2: print("marti")
    case 3: print("miercuri")
    case 4|5: print("ultimele doua zile cu ore")
    case _: print("nu avem ore") #poate lipsi
```

Expresia poate fi și de tip secvență (șir de caractere, lista etc).

```
x = [2,4]
match x:
    case [2,y]: #tipar
        print("lungime 2, contine 2 si ",y)
    case [y,4]:
        print("lungime 2, contine 4 si ",y)
    case _: print("alta lista")
```

O ramură case poate conține și if, ca în următorul exemplu:

```
x = int(input())
match x:
    case n if n<0:
        print(n, "nu este numar natural")
    case 0:
        print("introduceti numar natural strict pozitiv")
    case n if n<100:
        print("are doua cifre")
    case _: print("are mai mult de 2 cifre")
```

O ramură case nu poate conține și variabile inițializate anterior cu scopul de a se testa egalitatea cu acestea, rolul numelor de variabile din tipar este de a primi ca valoare valoarea expresiei:

```
x = 2
n = 7
match x:
    case n:
       print(x,n) #va tipari 2,2 – n primeste valoarea lui x
print(n) #va tipari 2
```

4. Instrucțiunea repetitiva cu test inițial while

Forma generală a instrucțiunii while este:

```
while expresie_logică:
      instrucțiune
```

Instrucțiunea while poate avea și clauza else (v. secțiunea dedicată clauzei else).

Exemplu:

```
#suma cifrelor unui numar
m = n = int(input())
s = 0
while n>0:
    s += n%10
    n //= 10 #!!nu /
print("suma cifrelor lui", m, "este",s)
```

În Python nu există instrucțiune repetitivă cu test final (cum este, de exemplu, do...while în C/C++).

5. Instrucțiunea repetitiva cu număr fix de iterații (for)

Instrucțiunea for din Pyhton diferă față de C/C++, fiind de fapt un “for each”, de forma

for *variabila* in *colectie_iterabila*

prin care se accesează pe rând elementele unei colecții (de exemplu secvențe de tip listă, șirur de caractere etc):

for *litera* in *sir*:
for *elem* in *lista*:

Exemplu:

```
s = [3, 1, 8, 12, 5]
suma = 0
for nr in s:
    suma += nr
print(suma)
suma = 0
for i in [0,1,2,3,4]:
    suma += s[i]
print(suma)
```

Cum parcurgem însă pe rând numerele de la 0 la 4, sau, mai general, de la 0 la n? Desigur, în primul caz putem folosi for i in [0,1,2,3,4], dar în cazul general ar fi utilă o metodă prin care să generăm "lista" numerelor de la 0 la n, pe care sa o parcurgem cu for. Pentru aceasta se folosește funcția range

Funcția **`range()`** (de fapt constructor, **`range`** fiind o clasă) se folosește pentru a genera o secvență de numere, având 3 parametri, dintre care doi opționali:

```
range([min], max, [pas])
```

Se vor genera pe rând (**nu toată lista odată**) numerele întregi cuprinse între valorile min (inclusiv) și max (**exclusiv**!!!) cu rația pas. Parametrii scriși între paranteze drepte sunt opționali. Dacă pentru parametrul min nu se specifică nicio valoare, atunci el va fi considerat în mod implicit ca fiind 0. Parametrul opțional pas se poate specifica doar dacă se specifică și parametrul opțional min și poate fi și **negativ**.
Astfel:

`range(b)` => **de la 0 la b-1**
`range(a, b)` => **de la a la b-1**
`range(a, b, p)` => **a, a+p, a+2p…**

Exemplu:

`range(10)` => **0 1 2 3 4 5 6 7 8 9**
`range(1,10)` => **1 2 3 4 5 6 7 8 9**
`range(10, 1)` => **vid**
`range(10, 10)` => **vid**
`range(1,10,2)` => **1 3 5 7 9**
`range(10,1,-2)` => **10 8 6 4 2**
`range(1,10,-2)` => **vid**

Exemplu: afișarea caracterelor dintr-un șir:

```
s = "abcde"
for i in range(len(s)):
    print(s[i])
```

6. Instrucțiunile break, continue

Instrucțiunile **break**, **continue** au aceeași semnificație ca în C; se folosesc în interiorul unei instrucțiuni repetitive astfel:

- **`continue`** - pentru a termina forțat iterația curentă (!!dar nu și instrucțiunea repetitivă)
- **`break`** - pentru a termina forțat executarea instrucțiunii repetitive

Exemple:

1) Afișarea cifrelor impare (folosind continue)

```
for i in range(1, 11):
    if i%2 == 0:
        continue
print(i, end=" ")
#Se va afișa: 1 3 5 7 9
```

2) Determinarea numărului de divizori proprii ai unui număr (folosind continue**)**

```
x = int(input())
k = 0
for d in range(2,x//2+1):
    if x%d != 0:
        continue
    k += 1
print("numarul de divizori proprii:",k)
```

3) Citirea de la tastatură până la introducerea șirului exit()

```
while True:
    comanda = input('>> ')
    if comanda == 'exit()':
        break
```

4) Se citește un șir de numere care se termină cu valoarea 0 (care nu face parte din). Să se afișeze lungimea șirului citit.

```
nr = 0
while True:
    x = int(input("x = "))
    if x == 0:
        break
    nr += 1
print(nr)
```

5) Determinarea primul divizor propriu al unui număr

```
x = int(input())
dx = None
for d in range(2,x//2+1):
    if x%d == 0:
        dx = d
        break
if dx: #if dx is not None:
    print("primul divizor propriu:",dx)
else:
    print("numar prim")
```

7. Clauza else pentru instrucțiuni repetitive

Clauza **`else`** poate fi adăugată la finalul unei instrucțiuni repetitive (**`while`**, **`for`**) și se executa dacă instrucțiunea s-a terminat fără o ieșire forțată (cu break)

Exemple

1) Determinarea primul divizor propriu al unui număr folosind clauza else.

Soluția este similară cu cea din exemplul 5 din secțiunea precedentă, dar nu vom mai memora în dx primul divizor găsit, pentru a putea testa în final dacă am găsit un divizor sau nu, ci vom face direct afișarea primului divizor găsit, iar mesajul "numar prim" se va afișa pe clauza else a for-ului (care se va executa dacă nu s-a găsit un divizor, deci nu s-a ieșit cu break din for)

```
x = int(input())
for d in range(2,x//2+1): # range(2, int(x**0.5)+1)
    if x%d == 0:
        print("primul divizor propriu:",d)
        break
else: #al for-ului, nu al if-ului
    #se executa daca din for nu s-a iesit cu break,
    #deci numarul nu are divizori proprii
    print("numar prim")
```

Observație. În loc de **`x**0.5`** puteam folosi și **`sqrt(x)`**. Funcția **`sqrt`** în Pyhon se află în modulul **`math`**. Vom discuta despre funcții predefinite în secțiunea următoare.

2) Determinarea primului număr prim din intervalul [a,b], cu a și b citite de la tastatură. Vom folosi rezolvarea exercițiului anterior, luând la rând toate valorile x de la a la b cu un for x... ; în loc de a afișa mesajul **`"numar prim"`** vom afișa x și vom opri execuția for-ului for x... cu break

```
a = int(input("a = "))
b = int(input("b = "))
for x in range(a, b+1):
    for d in range(2, x//2+1):
        if x % d == 0:
            break
    else:
  #instructiunea for d.. nu s-a terminat cu break, deci numarul este prim,
        print(x)
        break
else:
    #instructiunea for x in ... s-a terminat natural, nu cu break, deci
    # nu a fost afisat niciun număr prim cuprins între a și b
    print("Nu exista numar prim in intervalul [", a, ",", b,"]",sep="")
```

8. Instrucțiunea pass

Această instrucțiune se utilizează în cazurile în care sintactic este necesară o instrucțiune vidă:
Exemple:

1.

```
x = int(input())
if x<0:
    pass #urmeaza sa fie implementat
```

2.

```
varsta = int(input())
if varsta <= 18:
    print("Junior")
elif varsta < 65:
    #nu prelucrăm informații despre persoane cu vârsta
    #cuprinsa între 19 și 64 de ani
    pass
else:
    print("Senior")
```

Funcții predefinite

Pentru a vedea ce funcții sunt importate implicit puteți consulta documentația modulului **builtins:** https://docs.python.org/3/library/functions.html#built-in-funcs

Printre funcțiile din acest modul amintim:

- funcțiile de conversie **bin**(), **hex**(), constructori **int**(), **float**(), **str**()

- **abs**(), pentru determinarea modulului unui număr

- **min**() – cu număr variabil de parametri, care returnează minimul valorilor primite ca parametru (sau minimul unei colecții iterabile)

Pentru funcții matematice există modulul **math**. Utilizarea funcțiilor din alte module se face folosind instrucțiuni de **import**, ca în exemplul următor:

```
import math
print(math.sqrt(4))
print(math.factorial(5))
```

sau:

```
from math import sqrt
print(sqrt(5))
print(factorial(5)) #eroare, se putea: from math import sqrt,factorial
print(math.sqrt(4)) #eroare
print(math.factorial(4))#eroare
```