CURS 2
Expresii. Operatori. Instrucţiuni

O *expresie* poate fi formată din *operanzi* (constante sau variabile), *operatori* (simboluri corespunzătoarea unor operaţii) şi *paranteze rotunde* (pentru modificarea ordinii implicite de efectuare a operaţiilor). De exemplu, expresia `(x+3)*y` conţine operatorii aritmetici + şi *, operanzii sunt variabilele `x`, `y` şi constanta (literalul) `3`, iar parantezele sunt utilizate pentru a forţa efectuarea adunării înainte înmulţirii.

Operatori

Operatorii sunt simboluri corespunzătoarea anumitor operaţii. Un operator poate avea mai multe semnificaţii, în funcţie de context. De exemplu, operatorul – poate fi utilizat şi pentru a schimba semnul unei variabile şi pentru a efectua o scădere.

Un operator se caracterizează prin:

- *aritate*: numărul de operanzi asupra căruia poate acţiona operatorul respectiv (de exemplu, în expresia –3 operatorul – are aritatea 1, iar în expresia 7–3 acesta are aritatea 2);
- *prioritate*: stabileşte ordinea de evaluare a operatorilor dintr-o expresie (de exemplu, expresia `2+3*4` se evaluează în ordinea `2+3*4 = 2+12 = 14`, deoarece operatorul * are prioritate mai mare decât operatorul +);
- *asociativitate*: stabileşte ordinea de evaluare a unor operatori cu priorităţi egale dintr-o expresie (de exemplu, expresia `x+y+z` se evaluează de la stânga la dreapta, respectiv `(x+y)+z`, deoarece operatorul + are asociativitate de la stânga la dreapta).

În limbajul Python sunt definiţi mai mulţi operatori, pe care îi putem grupa în următoarele categorii:

1. *operatori aritmetici*: `+` (adunare sau semnul plus), `-` (scădere sau semnul minus), `*` (înmulţire), `/` (împărţire reală), `//` (împărţire întreagă), `%` (modulo), `**` (exponenţiere)

- Operatorul `/` efectuează întotdeauna o împărţire reală ("cu virgulă"), indiferent de tipul operanzilor (de exemplu, `7 / 2 = 3.5`).
- Expresia `a//b` furnizează cel mai mare întreg mai mic sau egal decât `a/b`, iar expresia `a%b` se calculează folosind formula `a%b = a–b*(a//b)`.

Exemple:

```
7 // 2 = 3
-7 // 2 = -4
-7.12 // 3.213 = -3.0
-7.12 % 3.213 = 2.519
11 // -3 = -4
11 % -3 = 11 – (-3) * (11//(-3)) = 11 + 3*(-4) = -1
```

- Operatorul ** are asociativitate de la dreapta la stânga.

Exemple:

```
0**0 = 1
3**4**2 = 3**(4**2) = 3**16 = 43046721
2**-3 = 0.125
-2**4 = -16
(-2)**4 = 16
31.44**0.788 = 15.136178456437747
```

2. *operatori relaționali*: `<` (strict mai mic), `<=` (mai mic sau egal), `>` (strict mai mare), `>=` (mai mare sau egal), `==` (egal), `!=` (diferit), `is` și `is not` (testarea identității), `in` și `not in` (testarea apartenenței)

- Operatorii `is/is not` testează dacă două variabile/expresii sunt identice sau nu, comparând referințele asociate valorilor lor. Practic, expresia `x is y` este `True` dacă și numai dacă `id(x) == id(y)`.

Exemple:

```
x = 3
y = 5
print(x is y)            #False
print(x+2 is y)          #True
print(y-2 is not x)      #False
```

- Operatorii `in/not in` testează apartenența unei valori la o colecție.

Exemple:

```
sir = "exemplu"
print("m" in sir)                    #True
print("emp" not in sir)              #False

lista = [11, -3, 7, 5, -10, 8]
x = 7
print(x not in lista)                #False
print(x+4 in lista)                  #True
```

- Deoarece numerele reale nu pot fi reprezentate exact în memorie, pot să apară erori în momentul comparării lor. De exemplu, expresia `1.1 + 2.2 == 3.3` va furniza valoarea `False`!

```
print(1.1 + 2.2 == 3.3)          #False
print(1.1 + 2.2, "==", 3.3)      #3.3000000000000003 == 3.3
```

Pentru a evita astfel de erori, se recomandă înlocuirea expresiei `x == y` (în care `x` și `y` sunt numere reale) cu o expresie de tipul `abs(x-y) <= 1e-9`, verificându-se astfel faptul că primele 9 zecimale ale numerelor reale `x` și `y` sunt identice.

- Spre deosebire de alte limbaje de programare, de exemplu C/C++, operatorii relaţionali pot fi înlănţuiţi!

Exemple:

```
a = 1
b = 10
x = 2
if a <= x <= b:
    print("Da")
else:
    print("Nu")
```

```
x = 1
y = 2
z = 4
if x + 2 == y + 1 > z:
    print("Da")
else:
    print("Nu")
```

În exemplul din partea stângă se va afişa mesajul "Da", iar în cel din dreapta "Nu".

3. *operatori logici*: `not` (negare), `and` (şi), `or` (sau)

- Valorile nule corespunzătoare tipurilor de date (de exemplu, valorile `0, 0.0, 0+0j, "", []` etc.) sunt considerate ca fiind echivalente cu `False`, iar orice altă valoare este considerată echivalentă cu `True`.

- În urma evaluării unor expresii care conţin operatori logici, în limbajul Python se pot obţine şi alte valori în afară de `True` sau `False`, astfel:

$$\text{not } x = \begin{cases} False, & \text{dacă } x \text{ este } True \\ True, & \text{dacă } x \text{ este } False \end{cases}$$

$$x \text{ and } y = \begin{cases} y, & \text{dacă } x \text{ este } True \\ x, & \text{dacă } x \text{ este } False \end{cases}$$

$$x \text{ or } y = \begin{cases} x, & \text{dacă } x \text{ este } True \\ y, & \text{dacă } x \text{ este } False \end{cases}$$

Exemple:

```
-100 and "test"          =>   'test'
-100 or "test"           =>   -100
not 3.14                 =>   False
-100 or "" and 3.14      =>   -100  (se evaluează mai întâi operatorul and)
"" or 10 and 3.14        =>   3.14
not (0 and 123.45)       =>   True
```

- Operatorul `not` este singurul operator logic care furnizează întotdeauna doar valorile `True` sau `False`!

- Expresiile care conţin operatori logici se evaluează prin scurtcircuitare, astfel:
  - într-o expresie de forma `expr_1 and expr_2 and ... and expr_n` evaluarea se opreşte la prima expresie a cărei valoare este `False`, deoarece, oricum, valoarea întregii expresii va fi `False`;

- într-o expresie de forma `expr_1 or expr_2 or … or expr_n` evaluarea se oprește la prima expresie a cărei valoare este `True`, deoarece, oricum, valoarea întregii expresii va fi `True`.

4. *operatori pe biți*: ~ (negare pe biți / bitwise not), & (și pe biți / bitwise and), | (sau pe biți / bitwise or), ^ (sau exclusiv / xor), << (deplasare la stânga pe biți / left shift), >> (deplasare la dreapta pe biți / right shift)

- Operatorii pe biți acționează asupra reprezentărilor binare ale valorilor de tip întreg.

- În limbajul Python toate numerele întregi sunt considerate cu semn (nu există tipuri de date asemănătoare celor de tipul `unsigned` din limbajele C/C++) și sunt reprezentate intern în complement față de 2, astfel:

Numere pozitive	Numere negative

`x = 23 = 00..010111`	`x = -24`
	`|x| = 24 = 00..011000`
	`~|x| = ~24 = 11..100111`
	`~|x|+1 = ~24 = 11..101000`
	
	`x = -24 = 11..101000`

Se observă faptul că numerele întregi pozitive se reprezintă binar direct prin scrierea lor în baza 2, în timp ce un număr întreg negativ x se reprezintă astfel:
  - se reprezintă în baza 2 valoarea absolută a lui x;
  - se calculează complementul față de 1 a valorii obținute anterior, respectiv toți biții egali cu 0 devin 1 și invers;
  - reprezentarea binară a lui x se obține adunând 1 la valoarea obținută anterior.

- Operatorul ~ (negare pe biți / bitwise not) este un operator unar care calculează numărul obținut prin negarea fiecărui bit al operandului său (complementul față de 1):

~	0	1
	1	0

`~b = 1 - b`

```
 x = 23 = 00..010111
~x = -24 = 11..101000
```

`~x = -(x + 1) = -x -1`

- Operatorii & (și pe biți / bitwise and), | (sau pe biți / bitwise or) și ^ (sau exclusiv / bitwise xor) sunt operatori binari care acționează asupra perechilor de biți aflați pe aceeași poziție în cei doi operanzi, astfel:

&	0	1
0	0	0
1	0	1

$b_1 \,\&\, b_2 = 1 \iff b_1 = b_2 = 1$

```
x = 349 = 00101011101
y = 2006 = 11111010110
x & y = 340 = 00101010100
```

|	0	1
0	0	1
1	1	1

$b_1 \mid b_2 = 0 \iff b_1 = b_2 = 0$

```
x = 349 = 00101011101
y = 2006 = 11111010110
x | y = 2015 = 11111011111
```

^	0	1
0	0	1
1	1	0

$b_1 \wedge b_2 = 1 \iff b_1 \neq b_2$

```
x = 349 = 00101011101
y = 2006 = 11111010110
x ^ y = 1675 = 11010001011
```

- Operatorul `<<` (left shift) este un operator binar care deplasează spre stânga biții unui număr întreg cu un număr dat de poziții, inserând la sfârșitul reprezentării binare a numărului respectiv un număr de biți nuli egal cu numărul de biți deplasați.

  Exemplu:
  ```
  x = 2006 = 11111010110
  x << 3 = 11111010110000 = 16048 = 2006 * (2**3)
  ```

  În general, expresia `x = x << b` este echivalentă cu expresia `x = x * (2**b)`.

- Operatorul `>>` (right shift) este un operator binar care deplasează spre dreapta biții unui număr întreg cu un număr dat de poziții, eliminându-i efectiv.

  Exemplu:
  ```
  x = 2006 = 11111010110
  x >> 3 = 11111010 = 11111010 = 250 = 2006 // (2**3)
  ```

  În general, expresia `x = x >> b` este echivalentă cu expresia `x = x // (2**b)`.

5. *operatorul condițional*: *`expr_1 if expr_logică else expr_2`*

- Operatorul condițional este un operator ternar care furnizează valoarea expresiei **`expr_1`** dacă **`expr_logică`** este **`True`** sau valoarea expresiei **`expr_2`** în caz contrar.

  Exemple:
  `max = x if x > y else y` (calculul maximului dintre două numere)
  `print("Nr. par") if x % 2 == 0 else print("Nr. impar")` (testarea parității unui număr întreg)

Prioritățile și asociativitățile operatorilor

Evaluarea unei expresii se realizează ținând cont de *priorităţile* şi *asociativităţile* operatorilor utilizaţi, aşa cum am menţionat anterior.

În limbajul Python, aproape toţi operatorii au *asociativitate de la stânga la dreapta* (mai puţin operatorul de exponenţiere care are asociativitate de la dreapta la stânga), iar priorităţile lor sunt indicate în tabelul următor, în ordine descrescătoare:

Prioritate	Operatori	Descriere
1 (maximă)	`()`	Parenteze (grupare)
2	`f(args...)`	Apel de funcţie
	`x[index_1:index_2]`	Accesare unei secvenţă (slicing)
	`x[index]`	Accesare unei element (indexare)
	`x.dată_membră`	Accesare unei date membre (obiecte)
3	`**`	Exponenţiere
4	`~x`	Negare pe biţi (bitwise NOT)
	`+x, -x`	Operatorii de semn (unari)
5	`*, /, //, %`	Înmulţire şi împărţiri
6	`+, -`	Adunare şi scădere (binari)
7	`<<, >>`	Deplasări pe biţi (bitwise shifts)
8	`&`	ŞI pe biţi (bitwise AND)
9	`^`	SAU EXCLUSIV pe biţi (bitwise XOR)
10	`|`	SAU pe biţi (bitwise OR)
11	`<, <=, >, >=, !=, ==, in, not in, is, is not`	Operatorii relaţionali
12	`not`	Negare logică (boolean NOT)
13	`and`	ŞI logic (boolean AND)
14	`or`	SAU logic (boolean OR)
15 (minimă)	`if...else`	Operatorul condiţional

În general, priorităţile operatorilor sunt "naturale" (de exemplu, operaţiile de exponenţiere, înmulţire şi împărţire au o prioritate mai mare decât cele de adunare şi scădere, operatorii logici şi relaţionali au priorităţi mici deoarece trebuie ca înaintea evaluării lor să fie evaluate restul expresiilor etc.) şi expresiile pot fi evaluate de către un

programator fără a cunoaşte în detaliu priorităţilor operatorilor. Totuşi, există şi câteva cazuri în care evaluarea corectă a unei expresii se poate efectua de către un programator doar cunoscând exact aceste priorităţi:

- valoarea expresiei `2 + 3 << 4` este `80`, deoarece operatorul `+` are prioritate mai mare decât operatorul `<<`, deci este echivalentă cu expresia `(2 + 3) << 4 = 5 << 4 = 5 * 2**4 = 5 * 16 = 80` (de multe ori, se consideră în mod eronat faptul că operatorul `<<` are prioritate mai mare decât operatorul `+`, deci expresia s-ar evalua prin `2 + (3 << 4) = 2 + 3 << 4 = 2 + 3 * 2**4 = 2 + 3 * 16 = 50`);
- expresia `x == not y` este incorectă sintactic, deoarece operatorul `==` are prioritate mai mare decât operatorul `not` şi expresia este considerată echivalentă cu `(x == not) y`, ceea ce evident nu are niciun sens! În acest caz, suntem obligaţi să utilizăm paranteze, deci expresia corectă este `x == (not y)`. Atenţie, există multe alte expresii de acest tip, de exemplu `True == not y`, `x & not y == True` etc.!
- o expresie de forma `a**-b` se evaluează corect prin a^{-b}, fiind considerată o excepţie (operatorul de exponenţiere are prioritate mai mare decât operatorul unar de semn `-`, deci expresia ar trebui să fie considerată ca fiind echivalentă cu `(a**-)b`, dar aceasta nu are niciun sens);
- secvenţa de cod de mai jos va afişa eronat mesajul `"Bursier din grupa 131 sau 132!"`, deoarece operatorul `and` are prioritate mai mare decât operatorul `or`, deci expresia va fi considerată echivalentă cu `131 == 131 or (131 == 132 and 5 >= 9)`, deci va fi evaluată prin `True or False` şi se va obţine valoarea `True`!

```
grupa = 131
media = 5
if grupa == 131 or grupa == 132 and media >= 9:
    print("Bursier din grupa 131 sau 132!")
else:
    print("Nu este bursier din grupa 131 sau 132!")
```

Evident, modificarea expresiei logice în `(grupa == 131 or grupa == 132) and media >= 9` va elimina această eroare.

Instrucţiuni

Pentru a putea controla fluxul unui program (ordinea în care se vor executa operaţiile dorite), majoritatea limbajelor de programare folosesc *instrucţiuni de control*. Aceste instrucţiuni pot fi, de exemplu, *instrucţiuni de decizie* (cu ajutorul cărora se stabileşte dacă o anumită operaţie se efectuează sau nu în funcţie de o anumită condiţie), *instrucţiuni repetitive* (cu ajutorul cărora se efectuează de mai multe ori o anumită operaţie) etc.

În limbajul Python nu există delimitatori pentru *blocurile de instrucţiuni* (cum sunt acoladele în limbajele C/C++), ci gruparea mai multor instrucţiuni se realizează prin indentarea lor în raport de instrucţiunea căreia i se subordonează.

În limbajul Python sunt definite următoarele instrucțiuni de control:

1. *instrucțiunea de atribuire*
   - Spre deosebire de limbajele C/C++, atribuirea nu este un operator, ci este o instrucțiune!
   - Instrucțiunea de atribuirea poate avea următoarele forme:
     - *atribuire simplă* (`x = 100`);
     - *atribuire multiplă* (`x = y = 100`);
     - *atribuire compusă* (`x, y, z = 100, 200, 300`).
   - Două variabile se pot interschimba prin atribuirea compusă `x, y = y, x`!
   - O atribuire de forma `x = x` *`operator`* `expresie` poate fi scrisă prescurtat sub forma `x` *`operator`*`= expresie`, unde `operator` este un operator aritmetic sau pe biți binar. De exemplu, instrucțiunea `x = x + y*10` poate fi scrisă prescurtat sub forma `x += y*10`.
   - În limbajul Python nu sunt definiți operatorii `++/--` din limbajele C/C++!

2. *instrucțiunea de decizie / alternativă* `if`
   - Instrucțiunea de decizie este utilizată pentru a executa o instrucțiune (sau un bloc de instrucțiuni) doar în cazul în care o expresie logică este adevărată:

     ```
     if expresie_logică:
          instrucțiune
     ```

     Exemplu (maximul dintre două numere):

     ```
     a = int(input("a = "))
     b = int(input("b = "))
     maxim = a
     if b > maxim:
         maxim = b
     print("Maximul dintre", a, "si", b, "este", maxim)
     ```

   - Instrucțiunea alternativă este utilizată pentru a alege executarea unei singure instrucțiuni (sau a unui bloc de instrucțiuni) dintre două posibile, în funcție de valoarea de adevăr a unei expresii logice:

     ```
     if expresie_logică:
         instrucțiune_1
     else:
         instrucțiune_2
     ```

     Exemplu (maximul dintre două numere):

     ```
     a = int(input("a = "))
     b = int(input("b = "))
     if a > b:
         maxim = a
     else:
         maxim = b
     print("Maximul dintre", a, "si", b, "este", maxim)
     ```

- Instrucţiunile alternative imbricate se pot scrie mai concis folosind instrucţiunea `elif`, aşa cum se poate observa din exemplul următor:

```
a = int(input("a = "))
if a < 0:
    print("Strict negativ")
else:
    if a == 0:
        print("Zero")
    else:
        print("Strict pozitiv")
```

```
a = int(input("a = "))
if a < 0:
    print("Strict negativ")
elif a == 0:
    print("Zero")
else:
    print("Strict pozitiv")
```

- În limbajul Python nu este definită o instrucţiune alternativă multiplă, cum este, de exemplu, instrucţiunea `switch` din limbajele C/C++!

3. *instrucţiunea repetitivă* `while`

- Instrucţiunea `while` este o instrucţiune repetitivă cu test iniţial, fiind utilizată pentru a executa o instrucţiune (sau un bloc de instrucţiuni) cât timp o expresie logică este adevărată:

while *expresie_logică*:
    instrucţiune

Exemplu (suma cifrelor unui număr natural):

```
n = int(input("n = "))
aux = n
sc = 0
while aux != 0:
    sc = sc + aux % 10
    aux = aux // 10
print("Suma cifrelor numarului", n, "este", sc)
```

- În limbajul Python nu este definită o instrucţiune repetitivă cu test final, cum este, de exemplu, instrucţiunea `do...while` din limbajele C/C++!

4. *instrucţiunea repetitivă* `for`

- Instrucţiunea `for` este utilizată pentru a accesa, pe rând, fiecare element dintr-o secvenţă (de exemplu, un şir de caractere, o listă etc.), elementele fiind considerate în ordinea în care apar în secvenţă:

for *variabilă* **in** *secvenţă*:
    instrucţiune

Exemple:

```
sir = "test"
for c in sir:
    print(c, end=" ")

#Se va afișa: t e s t
```

```
lista = [1, 2, 3]
for x in lista:
    print(x, end=" ")

#Se va afișa: 1 2 3
```

- Pentru a genera secvențe numerice de numere întregi asemănătoare unor progresii aritmetice se poate utiliza funcția `range([min], max, [pas])`, care va genera, pe rând, numerele întregi cuprinse între valorile `min` (inclusiv) și `max` (exclusiv!!!) cu rația `pas`. Parametrii scriși între paranteze drepte sunt opționali, iar parametrul opțional `pas` se poate specifica doar dacă se specifică și parametrul opțional `min`. Dacă pentru parametrul `min` nu se specifică nicio valoare, atunci el va fi considerat în mod implicit ca fiind `0`.

  Exemple:

```
range(6)          =>   0, 1, 2, 3, 4, 5
range(2, 6)       =>   2, 3, 4, 5
range(2, 11, 3)   =>   2, 5, 8
range(2, 12, 3)   =>   2, 5, 8, 11
range(7, 2)       =>   secvență vidă (deoarece 7 > 2)
range(7, 2, -1)   =>   7, 6, 5, 4, 3
```

5. *instrucțiunea* `continue`

- Instrucțiunea `continue` este utilizată în cadrul unei instrucțiuni repetitive pentru a termina forțat iterația curentă (dar nu și instrucțiunea repetitivă!), continuându-se direct cu următoarea iterație.

  Exemplu:

```
for i in range(1, 11):
    if i%2 == 0:
        continue
    print(i, end=" ")

#Se va afișa: 1 3 5 7 9
```

6. *instrucțiunea* `break`

- Instrucțiunea `break` este utilizată în cadrul unei instrucțiuni repetitive pentru a termina forțat executarea instrucțiunii respective.

  Exemplu: Se citește un șir de numere care se termină cu valoarea `0` (care se consideră că nu face parte din șir, ci este doar un marcaj al sfârșitului său). Să se afișeze suma numerelor citite.

```
s = 0
while True:
    x = int(input("x = "))
    if x == 0:
        break
    s += x
print("Suma numerelor citite: ", s)
```

Atenție, în acest program instrucțiunea `s += x` ar fi putut fi scrisă și înaintea instrucțiunii `if` fără a-i afecta corectitudinea! Totuși, în alte cazuri (de exemplu, dacă s-ar fi cerut produsul numerelor citite), acest lucru ar fi dus la afișarea unui rezultat eronat!

7. *instrucțiunea* `else`

- Instrucțiunea `else` poate fi adăugată la sfârșitul unei instrucțiuni repetitive, instrucțiunile subordonate ei fiind executate doar în cazul în care instrucțiunea repetitivă se termină natural (condiția dintr-o instrucțiune `while` devine falsă sau o instrucțiune `for` a parcurs toate elementele unei secvențe), ci nu din cauza întreruperii sale forțate (utilizând o instrucțiune `break`).

Exemple:

a) Se citește un șir format din `n` numere întregi. Să se verifice dacă toate numerele citite au fost pozitive sau nu.

```
n = int(input("n = "))
for i in range(n):
    x = int(input("x = "))
    if x < 0:
        print("A fost citit un număr negativ!")
        break
else:
    print("Toate numerele citite au fost pozitive!")
```

b) Să se afișeze cel mai mic număr prim cuprins între două numere naturale `a` și `b` sau un mesaj corespunzător în cazul în care nu există niciun număr prim cuprins între `a` și `b`.

```
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 in ... s-a terminat natural,
        #deci numărul x nu are divizori proprii,
        #ceea ce înseamnă că x este un număr prim
        print("Cel mai mic numar prim cuprins intre", a,
                "si", b, "este", x)

        #numărul x este cel mai mic număr prim cuprins între
        #a și b, deci nu are rost să mai continuăm căutarea
        #altuia și oprim forțat instrucțiunea for x in ...
        break
else:
    #instructiunea for x in ... s-a terminat natural, deci
    # nu a fost găsit niciun număr prim cuprins între a și b
    print("Nu exista niciun numar prim cuprins intre", a,
            "si", b)
```

8. *instrucțiunea* `pass`

- Instrucțiunea `pass` este o instrucțiune care nu are niciun efect în program (este similară unei instrucțiuni vide). Această instrucțiune se utilizează în cazurile în care sintactic ar fi necesară o instrucțiune vidă, deoarece în limbajul Python aceasta nu este definită.

Exemplu:

```
varsta = 10
if varsta <= 18:
    print("Junior")
elif varsta < 60:
    #nu prelucrăm informațiile despre persoanele
    #cu vârste cuprinse între 19 și 59 de ani
    pass
else:
    print("Senior")
```