

CURS 01

Introducere în limbajul Python

Python este un limbaj de programare creat de către Guido van Rossum în 1991 ([https://en.wikipedia.org/wiki/Python_\(programming_language\)](https://en.wikipedia.org/wiki/Python_(programming_language))). Ultima sa versiune stabilă este 3.9, iar kitul de instalare poate fi descărcat de pe site-ul oficial al limbajului, <https://www.python.org/>. Un mediu integrat de dezvoltare foarte popular pentru Python este PyCharm și poate fi descărcat de pe site-ul companiei producătoare JetBrains: <https://www.jetbrains.com/pycharm/>.

Spre deosebire de limbajele C/C++, care sunt *limbaje compilate*, limbajul Python este un *limbaj interpretat*. Acest lucru înseamnă faptul că programele Python nu sunt transformate în *cod mașină/executabil* neportabil, care poate fi executat doar de un anumit sistem de operare (așa cum se întâmplă în cazul limbajelor compilate), ci este transformat într-un *cod intermediar/bytecode* portabil, care poate fi executat de orice sistem de operare în care a fost instalată o mașină virtuală Python (Fig. 1 – sursa: <https://www.c-sharpcorner.com/article/why-learn-python-an-introduction-to-python/>).

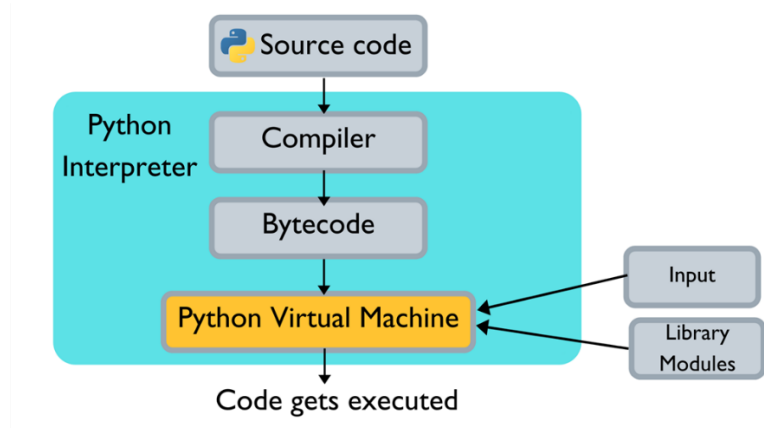


Fig. 1: Etapele executării unui program Python

Limbajul Python este un limbaj complex, care permite utilizarea mai multe paradigme de programare: *programare procedurală*, *programare orientată pe obiecte* și *programare funcțională*. De asemenea, limbajul Python conține multe librării standard dedicate unor domenii diverse (matematică, sisteme de operare, programare concurentă, programare distribuită etc.): <https://docs.python.org/3/library/>.

Tipuri de date

În orice limbaj de programare, un tip de date reprezintă un set de valori având o reprezentare binară internă unitară pentru care s-au definit anumite operații. De exemplu, în limbajele C/C++ valorile de tip `int` sunt reprezentate intern prin complement față de 2 pe 4 octeți și asupra lor se pot efectua diverse operații aritmetice cum ar fi adunarea, scăderea, înmulțirea și împărțirea.

În limbajul Python, fiecărui tip de date îi corespunde o anumită clasă predefinită, iar constantele și variabilele de tipul respectiv sunt instanțe ale clasei respective sau, altfel spus, obiecte. Practic, datele membre ale clasei modelează în mod unitar valorile de tipul respectiv, iar metodele sale implementează operațiile care pot fi efectuate cu valorile respective.

Principalele tipuri de date predefinite în limbajul Python sunt:

- *tipul NoneType* (clasa `NoneType`) conține o singură valoare, `None`, utilizată, de obicei, pentru a indica faptul că o funcție nu a întors nicio valoare sau pentru a inițializa un parametru al unei funcții cu o valoare implicită.
- *tipuri numerice* care permit memorarea valorilor numerice întregi, reale sau complexe:
 - *tipul întreg* (clasa `int`) permite memorarea valorilor întregi cu semn (în limbajul Python nu există tipuri de date întregi fără semn!). Literalii de tip întreg pot fi scriși în baza 10, în baza 2 folosind prefixele `0b` și `0B`, în baza 16 folosind prefixele `0x` și `0X` sau în baza 8 în folosind prefixele `0o` și `0O`.
 - *tipul real* (clasa `float`) permite memorarea valorilor de tip real ("cu virgulă") folosind reprezentarea în virgulă mobilă cu dublă precizie din standardul IEEE-754 (https://en.wikipedia.org/wiki/IEEE_754), fiind echivalent tipului de date `double` din limbajele C/C++.
 - *tipul complex* (clasa `complex`) permite memorarea numerelor complexe sub forma `a+bj`, unde `a` și `b` sunt două numere reale reprezentând partea reală și partea imaginară.
- *tipul boolean* (clasa `bool`) conține două valori, `True` și `False`. Valorile având alt tip de date sunt asimilate cu `False` dacă sunt nule (i.e., valorile numerice `0`, `0.0` și `0+0j`, precum și listele, tuplurile, mulțimile, dicționarele sau șirurile de caractere vide), respectiv cu `True` în caz contrar.
- *tipuri secvențiale* care permit memorarea unor șiruri de valori, indexate de la 0:
 - *șiruri de caractere* (clasa `str`) care permit memorarea secvențelor de caractere Unicode. Literalii de tip șir de caractere pot fi scriși folosind apostrofuri (e.g., `'Limbajul Python'`), ghilimele (e.g., `"Limbajul Python"`) sau triplu apostrof (e.g., `'''Limbajul Python'''`).
 - *liste* (clasa `list`) permite memorarea unei secvențe mutabile de valori (i.e., elementele listei pot fi modificate după ce lista a fost creată) care pot avea tipuri de date diferite (e.g., `[12, -3.14, 'Python', -20]`).
 - *tupluri* (clasa `tuple`) permite memorarea unei secvențe imutabile de valori (i.e., elementele tuplului nu mai pot fi modificate după ce tuplul a fost creat) care pot avea tipuri de date diferite (e.g., `(12, -3.14, 'Python', -20)`).
- *tipuri mulțime* (clasa mutabilă `set` și clasa imutabilă `frozenset`) care permit memorarea unor valori fără duplicate (*mulțimi*) și efectuarea operațiilor specifice mulțimilor (e.g., reuniune, intersecție etc.). Mulțimile nu sunt indexate!
- *tablouri asociative* (clasa `dict`) care permit memorarea unor perechi de forma *cheie:valoare*.

Variabile

În limbajul Python o variabilă nu se declară explicit, deci nu are asociat un tip de date static (i.e., care nu mai poate fi modificat ulterior). Practic, tipul de date al unei variabile se stabilește dinamic, în funcție de valoarea pe care aceasta o primește la un moment dat. Din acest motiv, orice variabilă trebuie inițializată înainte de a fi utilizată într-un program (altfel, va fi generată o eroare)!

Pentru a determina tipul de date asociat unei variabile la un moment dat se folosește funcția `type(variabilă)`:

```

x = 100
print("x =", x)
print("Tipul de date al variabilei x:", type(x), "\n")

x = x / 3
print("x =", x)
print("Tipul de date al variabilei x:", type(x), "\n")

x = "Ana are mere!"
print("x =", x)
print("Tipul de date al variabilei x:", type(x))
  
```

Run: Test_curs

```

C:\Users\Ours\Desktop\Test_Python\venv\Scripts\python.exe
x = 100
Tipul de date al variabilei x: <class 'int'>

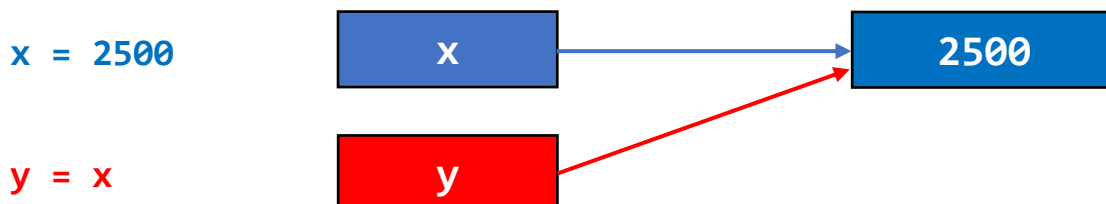
x = 33.333333333333336
Tipul de date al variabilei x: <class 'float'>

x = Ana are mere!
Tipul de date al variabilei x: <class 'str'>

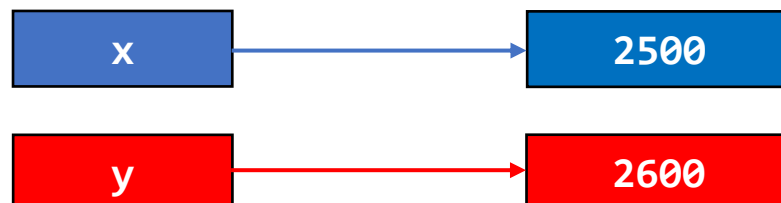
Process finished with exit code 0
  
```

În limbajul Python, o variabilă nu conține o valoare, ci o *referință* spre un obiect care conține valoarea respectivă. Astfel, printr-o instrucțiune de atribuire nu se copiază valoarea respectivă, ci doar referința sa!

Exemplu:



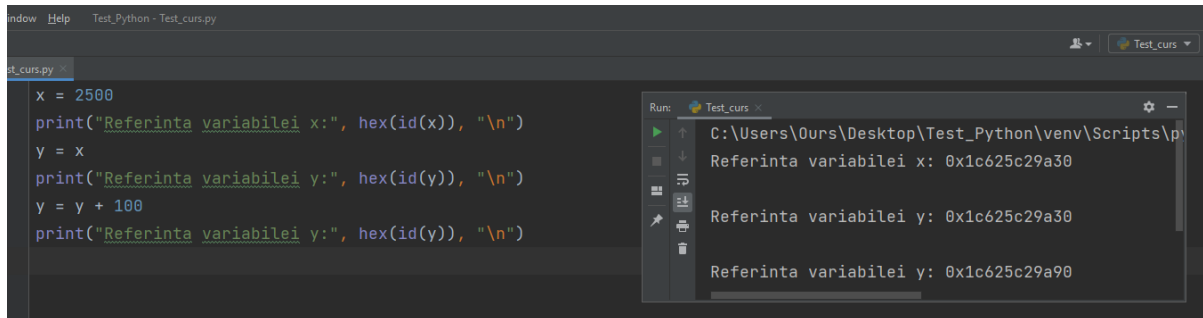
După efectuarea instrucțiunii `y = y + 100`, variabilele `x` și `y` vor conține următoarele referințe:



Mașina virtuală Python realizează automat managementul memoriei, respectiv un obiect care nu mai este utilizat (i.e., referința sa nu mai este stocată în nicio variabilă) va

fi șters de *Garbage Collector*. În exemplul anterior, dacă s-ar efectua instrucțiunea de atribuire $x = y$, atunci referința obiectului cu valoarea 2500 nu ar mai fi păstrată în nicio variabilă și acesta va fi șters, la un moment dat, de Garbage Collector.

Intern, referința unui obiect este chiar adresa sa de memorie și poate fi aflată folosind funcția `id(variabilă)`:



```

x = 2500
print("Referința variabilei x:", hex(id(x)), "\n")
y = x
print("Referința variabilei y:", hex(id(y)), "\n")
y = y + 100
print("Referința variabilei y:", hex(id(y)), "\n")

```

```

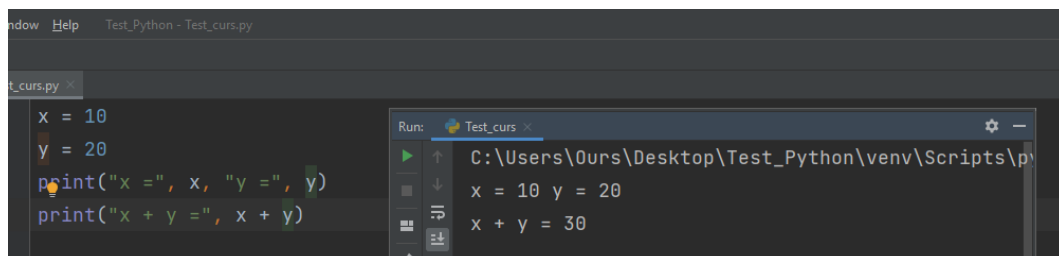
C:\Users\Ours\Desktop\Test_Python\venv\Scripts\python.exe
Referința variabilei x: 0x1c625c29a30
Referința variabilei y: 0x1c625c29a30
Referința variabilei y: 0x1c625c29a90

```

Identificatorul unui obiect este unic pe parcursul ciclului său de viață, dar pot exista obiecte cu același identificator dacă ele au cicluri de viață disjuncte.

Afișarea datelor pe monitor

Pentru afișarea datele pe monitor se utilizează funcția `print(argumente)`. Această funcție are un număr variabil de argumente care pot fi constante, variabile sau expresii. În mod implicit, argumentele sunt afișate cu un spațiu între ele, iar la sfârșit se va afișa o linie nouă, vidă:



```

x = 10
y = 20
print("x =", x, "y =", y)
print("x + y =", x + y)

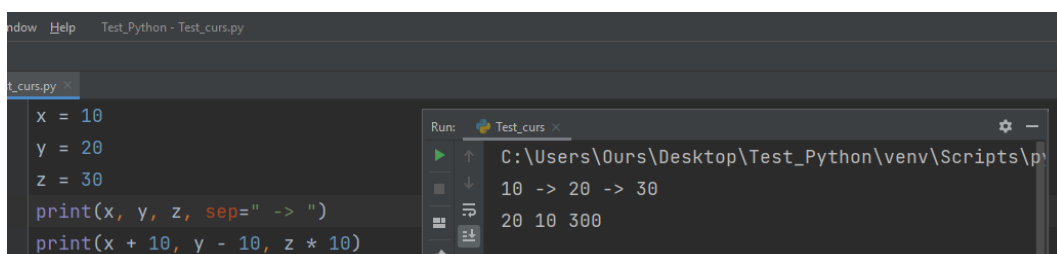
```

```

C:\Users\Ours\Desktop\Test_Python\venv\Scripts\python.exe
x = 10 y = 20
x + y = 30

```

Funcția `print` are parametrul opțional `sep`, de tip șir de caractere, prin intermediul căruia se poate stabili un alt separator pentru valorile afișate, însă doar în cadrul apelului respectiv:



```

x = 10
y = 20
z = 30
print(x, y, z, sep=" -> ")
print(x + 10, y - 10, z * 10)

```

```

C:\Users\Ours\Desktop\Test_Python\venv\Scripts\python.exe
10 -> 20 -> 30
20 10 300

```

De asemenea, funcția `print` are parametrul opțional `end`, de tip șir de caractere, prin intermediul căruia se poate modifica linia nouă afișată la sfârșitul apelului respectiv:

```

x = 10
y = 20
z = 30
print(x, end=" : ")
print(y)
print(z)

```

Run: Test_curs

```

C:\Users\Ours\Desktop\Test_Python\venv\Scripts\python.exe
10 : 20
30

```

O altă posibilitate de afișare a datelor o constituie utilizarea *șirurilor de caractere formate (f-strings)*. Un astfel de șir se indică folosind litera `f` sau `F` înaintea sa, iar în interiorul său se precizează, între acolade, valorile expresiilor care trebuie afișate:

```

x = 10
y = 20
print(f"Suma dintre numerele {x} si {y} este {x + y}.")

```

Run: Test_curs

```

C:\Users\Ours\Desktop\Test_Python\venv\Scripts\python.exe
Suma dintre numerele 10 si 20 este 30.

```

Citirea datelor de la tastatură

Pentru citirea datelor de la tastatură se utilizează funcția `input(mesaj)`. Parametrul `mesaj` este opțional, de tip șir de caractere, iar în cazul în care este utilizat se va afișa mesajul respectiv pe monitor, înainte de citirea datelor. Funcția `input` furnizează întotdeauna valoarea citită de la tastatură sub forma unui șir de caractere:

```

x = input("x = ")
y = input("y = ")
print(f"{x} + {y} = {x+y}")
print("Tipul lui x:", type(x))
print("Tipul lui y:", type(y))

```

Run: Test_curs

```

C:\Users\Ours\Desktop\Test_Python\venv\Scripts\python.exe
x = 100
y = 200
100 + 200 = 100200
Tipul lui x: <class 'str'>
Tipul lui y: <class 'str'>

```

Pentru a transforma șirurile de caractere citite în valori de alte tipuri primitive se folosesc funcțiile de conversie `int(șir)`, `float(șir)`, `complex(șir)` sau `bool(șir)`:

```

x = int(input("x = "))
y = float(input("y = "))
print(f"{x} + {y} = {x+y}")
print("Tipul lui x:", type(x))
print("Tipul lui y:", type(y))

```

Run: Test_curs

```

C:\Users\Ours\Desktop\Test_Python\venv\Scripts\python.exe
x = 100
y = 3.14
100 + 3.14 = 103.14
Tipul lui x: <class 'int'>
Tipul lui y: <class 'float'>

```

Expresii. Operatori. Instrucțiuni

O *expresie* poate fi formată din *operanzi* (constante sau variabile), *operatori* (simboluri corespunzătoare unor operații) și *paranteze rotunde* (pentru modificarea ordinii implicite de efectuare a operațiilor). De exemplu, expresia $(x+3)*y$ conține operatorii aritmetici $+$ și $*$, operanzii sunt variabilele x , y și constanta (literalul) 3 , iar parantezele sunt utilizate pentru a forța efectuarea adunării înainte înmulțirii.

Operatori

Operatorii sunt simboluri corespunzătoare anumitor operații. Un operator poate avea mai multe semnificații, în funcție de context. De exemplu, operatorul $-$ poate fi utilizat și pentru a schimba semnul unei variabile și pentru a efectua o scădere.

Un operator se caracterizează prin:

- *aritate*: numărul de operanzi asupra căruia poate acționa operatorul respectiv (de exemplu, în expresia -3 operatorul $-$ are aritatea 1, iar în expresia $7-3$ acesta are aritatea 2);
- *prioritate*: stabilește 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*: stabilește ordinea de evaluare a unor operatori cu priorități 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 definiți mai mulți operatori, pe care îi putem grupa în următoarele categorii:

1. *operatori aritmetici*: $+$ (adunare sau semnul plus), $-$ (scădere sau semnul minus), $*$ (înmulțire), $/$ (împărțire reală), $//$ (împărțire întreagă), $\%$ (modulo), $**$ (exponențiere)
- Operatorul $/$ efectuează întotdeauna o împărțire 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 relaționali pot fi înlanțuiți!

Example:

```
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 afișa mesajul "Da", iar în cel din dreapta "Nu".

3. operatori logici: **not** (negare), **and** (și), **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 conțin operatori logici, în limbajul Python se pot obține și alte valori în afară de True sau False, astfel:

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

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

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

Example:

```
-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 conțin operatori logici se evaluează prin scurtcircuitare, astfel:
 - într-o expresie de forma `expr_1 and expr_2 and ... and expr_n` evaluarea se oprește 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$ $\sim x = \sim 24 = 11..100111$ $\sim x + 1 = \sim 24 = 11..101000$ $x = -24 = 11..101000$

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):

\sim	0	1
	1	0

$\sim b = 1 - b$

$$\begin{aligned}
 x &= 23 = 00..010111 \\
 \sim x &= -24 = 11..101000 \\
 \sim x &= -(x + 1) = -x - 1
 \end{aligned}$$

- 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

$b_1 \& b_2 = 1 \Leftrightarrow b_1 = b_2 = 1$

$$\begin{aligned}
 x &= 349 = 00101011101 \\
 y &= 2006 = 11111010110 \\
 x \& y &= 340 = 00101010100
 \end{aligned}$$

	0	1
0	0	1
1	1	1

$$b_1 \mid b_2 = 0 \Leftrightarrow b_1 = b_2 = 0$$

^	0	1
0	0	1
1	1	0

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

$$\begin{aligned} x &= 349 = 00101011101 \\ y &= 2006 = 11111010110 \\ x \mid y &= 2015 = 11111011111 \end{aligned}$$

$$\begin{aligned} x &= 349 = 00101011101 \\ y &= 2006 = 11111010110 \\ x \wedge y &= 1675 = 11010001011 \end{aligned}$$

- Operatorul \ll (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:

$$x = 2006 = 11111010110$$

$$x \ll 3 = 11111010110000 = 16048 = 2006 * (2^{**}3)$$

În general, expresia $x = x \ll b$ este echivalentă cu expresia $x = x * (2^{**}b)$.

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

Exemplu:

$$x = 2006 = 11111010110$$

$$x \gg 3 = 11111010110 = 11111010 = 250 = 2006 // (2^{**}3)$$

În general, expresia $x = x \gg b$ 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 *prioritățile* și *asociativitățile* operatorilor utilizați, așa cum am menționat anterior.

În limbajul Python, aproape toți operatorii au *asociativitate de la stânga la dreapta* (mai puțin operatorul de exponențiere care are asociativitate de la dreapta la stânga), iar prioritățile lor sunt indicate în tabelul următor, în ordine descrescătoare:

Prioritate	Operatori	Descriere
1 (maximă)	()	Parenteze (grupare)
2	f(args...)	Apel de funcție
	x[index_1:index_2]	Accesare unei secvență (slicing)
	x[index]	Accesare unei element (indexare)
	x.dată_membră	Accesare unei date membre (obiecte)
3	**	Exponențiere
4	~x	Negare pe biți (bitwise NOT)
	+x, -x	Operatorii de semn (unari)
5	*, /, //, %	Înmulțire și împărțiri
6	+, -	Adunare și scădere (binari)
7	<<, >>	Deplasări pe biți (bitwise shifts)
8	&	ȘI pe biți (bitwise AND)
9	^	SAU EXCLUSIV pe biți (bitwise XOR)
10		SAU pe biți (bitwise OR)
11	<, <=, >, >=, !=, ==, in, not in, is, is not	Operatorii relaționali
12	not	Negare logică (boolean NOT)
13	and	ȘI logic (boolean AND)
14	or	SAU logic (boolean OR)
15 (minimă)	if...else	Operatorul condițional

În general, prioritățile operatorilor sunt "naturale" (de exemplu, operațiile de exponențiere, înmulțire și împărțire au o prioritate mai mare decât cele de adunare și scădere, operatorii logici și relaționali au priorități mici deoarece trebuie ca înaintea evaluării lor să fie evaluate restul expresiilor etc.) și expresiile pot fi evaluate de către un

programator fără a cunoaște în detaliu priorităților operatorilor. Totuși, există și câteva cazuri în care evaluarea corectă a unei expresii se poate efectua de către un programator doar cunoscând exact aceste priorități:

- 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 $x == \text{not } y$ este incorectă sintactic, deoarece operatorul $==$ are prioritate mai mare decât operatorul not și expresia este considerată echivalentă cu $(x == \text{not}) y$, ceea ce evident nu are niciun sens! În acest caz, suntem obligați să utilizăm paranteze, deci expresia corectă este $x == (\text{not } y)$. Atenție, există multe alte expresii de acest tip, de exemplu $\text{True} == \text{not } y$, $x \& \text{not } y == \text{True}$ etc.!
- o expresie de forma $a^{**-}b$ se evaluează corect prin a^{-b} , fiind considerată o excepție (operatorul de exponențiere 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);
- secvența de cod de mai jos va afișa 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 $131 == 131 \text{ or } (131 == 132 \text{ and } 5 \geq 9)$, deci va fi evaluată prin $\text{True} \text{ or } \text{False}$ și se va obține 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 $(grupa == 131 \text{ or } grupa == 132) \text{ and } media \geq 9$ va elimina această eroare.