

Elemente de bază ale limbajului Python



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

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

Diferențe față de C/C++

- ▶ Nu sunt necesari delimitatori de blocuri de tip {} sau begin/end etc, este obligatorie indentarea blocurilor de cod (și suficientă pentru delimitarea acestora)
- ▶ Nu este nevoie sa punem ; la finalul unei linii (dacă nu mai urmează alte linii de cod pe aceeași linie)

```
#include <iostream>
using namespace std;
int main() {
    int i;
    i = 1;
    while (i<10) {
        cout<<i<<endl;
        i++;
    }
    cout<<"gata afisarea";
}
```



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

Elemente de bază ale limbajului

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

print

- număr variabil de argumente
- parametrul opțional **sep** – șirul separator al argumentelor afișate (implicit spațiu)
- parametrul opțional **end** – șirul de la sfârșitul afișării (implicit linie nouă)

Elemente de bază ale limbajului

2. Citirea de la tastatură + funcții de conversie

▶ funcția `input`

- parametru (opțional) mesajul care se va afișa pe ecran
- returnează **șirul de caractere** introdus până la sfârșitul de linie (de tip `str`, **este necesară conversia** dacă vrem alt tip de date)

Elemente de bază ale limbajului

3. Erori

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


Variabile

Variabile

- În **C/C++** o variabilă se declară și are: **tip, adresa, valoare**:
 1. **nume**
 2. **tip de date** (stabilește numărul de octeți pe care se va memora variabila \Leftrightarrow valoarea minimă și maximă a variabilei)
 3. **adresă de memorie** (adresa de început a zonei de memorie alocată variabilei respective)
 4. **valoarea** la un moment dat


Variabile

- În **C/C++** o variabilă se declară și are: **tip, adresa, valoare**:

```
int i;  
i = 2;  prin atribuire variabila primește valoare  
cout<<(&i)<<" "<<sizeof(i)<<endl;
```

`&i` este 0x7ffdc8826c34
(adresa octetului în hexa)

	int => sizeof(int) = 4 octeți				
...	00000000	00000000	00000000	00000010	...
	i (numele variabilei)				



Variabile

- În **C/C++** o variabilă se declară și are: **tip, adresa, valoare**:

```
int i;  
i = 2;  
cout<<(&i)<<" "<<sizeof(i)<<endl;  
i = i + 1; ————— prin atribuire variabila primește valoare  
cout<<(&i)<<endl;
```

&i este **0x7ffdc8826c34** - **ACEEAȘI**
(adresa octetului în hexa)

	int => sizeof(int) = 4 octeți				
...	00000000	00000000	00000000	000000 11	...
	i (numele variabilei)				

Variabile

- În Python **variabilele** sunt **referințe spre obiecte (nume date obiectelor)**
- orice **valoare** => un **obiect** (!inclusiv tipurile numerice)
- Un obiect **ob** are asociat:
 - un număr de identificare: `id(ob)`
 - un tip de date: `type(ob)`
 - o valoare – poate fi convertită la șir de caractere `str(ob)`
- Printr-o instrucțiune de atribuire **nu se copiază valoarea** respectivă, ci **doar referința sa**

Variabile

- Printr-o instrucțiune de atribuire **nu se copiază valoarea** respectivă, ci **doar referința sa**

`i = ob` \longrightarrow prin atribuire variabila `i` referă
(numește) obiectul `ob`

- Tipul unei variabile (a obiectului referit) se stabilește prin inițializare (prima atribuire) și se poate schimba prin atribuiri de valori de alt tip

```
i = 3
print(type(i), id(i))    #<class 'int'> 2494987632944
i = "ab"
print(type(i), id(i))    #<class 'str'> 2494997181488
```

Variabila; obiect; alocare; atribuire

C

```
int n,m;
```

m:

n:

Variabila; obiect; alocare; atribuire

C

Python

`m = 1000`

m: 1000

m → 1000

n:

Variabila; obiect; alocare; atribuire

C

Python

```
m = 1000
```

m: 1000

m → 1000

```
m = m+1
```

n:

Variabila; obiect; alocare; atribuire

C

```
m = 1000
```

m:

1001

```
m = m+1
```

n:



Python

m



1000

1001

Variabila; obiect; alocare; atribuire

C

```
m = 1000
```

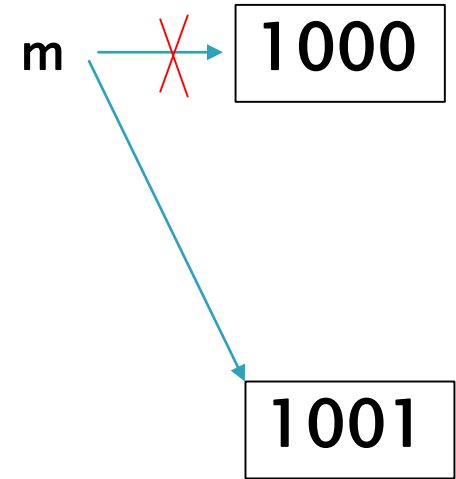
m: 1001

```
m = m+1
```

```
n = m
```

n:

Python



Variabila; obiect; alocare; atribuire

C

```
m = 1000
```


m: 1001

```
m = m+1
```

```
n = m
```

n: 1001

Python

m  1000

1001

n



Variabila; obiect; alocare; atribuire

C

```
m = 1000
```

m: 1001

```
m = m+1
```

```
n = m
```

n: 1001

```
n = n+1 ???
```

Python

m  1000

1001

n 

Variabila; obiect; alocare; atribuire

C

```
m = 1000
```

m: 1001


```
m = m+1
```

```
n = m
```

n: 1001

```
n = "sir" ???
```

Python

m  1000

1001

n 

Variabile

- **Numele unei variabile – identificatori**
 - fiecare variabilă trebuie să aibă un **nume unic**
 - sunt permise doar literele alfabetului, cifrele și _(underscore)
 - identificatorul **nu poate începe cu o cifră sau cu un caracter special** (nu putem declara variabile cu numele: 4n, %par etc).
 - **case sensitive**: literele mari sunt tratate diferit de literele mici (Maxim, maxim, maXim, MaxiM ar desemna variabile diferite)
 - numele nu pot fi cuvinte cheie sau instrucțiuni ale limbajului Python
- https://docs.python.org/3/reference/lexical_analysis.html?highlight=keywords#identifiers
- **Recomandare nume:**
`litere_mici_separate_prin_underscore`

Variabile

- **Optimizare**: numerele întregi din intervalul $[-5, 256]$ sunt prealocate (în cache)
 - alocate de la început, nu când apar în program

⇒ **toate obiectele care au o astfel de valoare sunt identice (au același id)**
- variabile cu aceeași valoare **pot avea** același id (dacă este o valoare prealocată, atunci sigur da)

Variabile

▶ Exempu

```
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))
```

Variabile

- **del x** – șterge o variabilă din memorie

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

- **Garbage collector** – șterge obiecte către care nu mai sunt referințe

Tipuri de date

Tipuri de date

- ▶ **Tip de date** – îi corespunde o **clasă** predefinită
- ▶ **constantele și variabilele** – sunt **obiecte** (instanțe ale clasei corespunzătoare tipului)

Tipuri de date

- `int` – clasa `int`
 - numere întregi (!cu semn) cu oricât de multe cifre (limita dată doar de performanța sistemului pe care se rulează)
 - Din versiunea 3.10.7 – limita (4300) pentru conversia la str (de exemplu la afișare)
 - Putem mări limita

```
import sys
```

```
sys.set_int_max_str_digits(limita_noua)
```

Tipuri de date

- **int**

- numere întregi (!cu semn) cu oricât de multe cifre (limita dată doar de performanța sistemului pe care se rulează)
- memorate ca vectori de “cifre” din reprezentarea în baza 2^{30} (cu cifre de la 0 la $2^{30}-1 = 1073741823$)

Exemplu: Reprezentarea pentru 234254646549834273498:

ob_size	3		
ob_digit	462328538	197050268	203

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

Tipuri de date

- **int**

- constante în baza 10 (implicit), dar și în bazele 2 (prefix 0b,0B), 8 (prefix 0o, 0O), 16 (prefix 0x,0X):

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

Tipuri de date

- **int**

- constante în baza 10 (implicit), dar și în bazele 2 (prefix 0b,0B), 8 (prefix 0o, 0O), 16 (prefix 0x,0X):

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

- putem folosi `int(sir)` pentru creare/conversie (există și varianta `int(sir, base=baza)`)

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

Tipuri de date

- **float**

- Constante: 3.5, 1e-2 (notație științifică)
- float([x]):

`float("inf"); float("infinity"); float("nan")`

Tipuri de date

- **float**
 - IEEE-754 double precision
 - operațiile aritmetice cu tipul de date *float* nu au precizie absolută:

NU: $0.1 * 0.1 == 0.01$

DA: $\text{abs}(0.1 * 0.1 - 0.01) < 1e-9$

Tipuri de date

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

Tipuri de date

- **complex**

```
z = complex(-1, 4)

print("Numarul complex:", z)

print("Partea reala:", z.real)

print("Partea imaginara:", z.imag)

print("Conjugatul:", z.conjugate())

print("Modul:", abs(z))
```

Tipuri de date

- **bool**
 - True, False
 - putem folosi `bool()` pentru conversie
 - În context boolean – **conversia oricărei valori la bool**

Context boolean – condiție if, while; operand pentru operatori logici – conversii

Tipuri de date

- **bool**

Se consideră **False**:

- **None, False**
- **0, 0.0, 0j, Decimal(0), Fraction(0,1)**
- **Colecții și secvențe vide** (+obiecte în care `__bool__()` returnează False sau `__len__()` returnează 0)

Tipuri de date

- **bool**

Se consideră **False**:

- **None, False**
- **0, 0.0, 0j, Decimal(0), Fraction(0,1)**
- **Colecții și secvențe vide** (+obiecte în care `__bool__()` returnează False sau `__len__()` returnează 0)

```
print(bool(0), bool(-5))
```

```
print(bool(""), bool(" "))
```

```
print(bool(None), bool([]))
```

Tipuri de date

- NoneType

- **None**

- Nu exista char

`ord("a")`

`chr(97)`

Tipuri de date

Secvențe – șiruri de valori, indexate de la 0

Mutable (le putem modifica elementele) și imutabile

- liste – clasa `list`: `a = [3, 1, 4, 7]` – mutable
- tupleuri – clasa `tuple`: `a = (3, 1, 4, 7)`
- șiruri de caractere – clasa `str`: `a = "31sir"`, `a = '31sir'`

Tipuri de date

Mulțimi – set de valori fără duplicate (mulțimi), cu 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

Comentarii

- Prefixat de # => comentariu pe o linie
- Pentru mai multe linii – # pe fiecare linie sau delimitatori de șiruri de caractere
 - Încadrat de ' ' ' => pe mai multe linii
 - Încadrat de " " " => docstring – comentariu pe mai multe linii, folosit în mod special pentru documentare