Dicționar Python C++

Acest tutorial se adresează celor care au cunoștințe de programare în C++ și acum, după ce au reușit să ruleze unele programe în Python și să citească diverse tutoriale de pe internet[1], vor să-și aprofundeze cunoștințele apelând la documentația oficială, publicată pe docs.python.org

Recomand călduros The Python Tutorial, scris și întreținut chiar de Guido van Rossum, tot aici se găsesc și normele sintactice ale limbajului, în The Python Language Reference , precum și descrierea modulelor standard distribuite odată cu interpretorul Python, The Python Standard Library.

Pentru a veni în sprijinul celor care vor să citească documentația, voi încerca să explic unii termeni folosiți în documentația Python în corelație cu cei întâlniți in C++. Intrările în dicționarul următor nu sunt ordonate alfabetic, voi respecta pe cât posibil cerința să nu folosesc un termen Python înainte de fi explicat.

program Python: o secvență de instrucțiuni. Un program C/C++ este o secvență de declarații de variabile globale, de definiții de tipuri utilizator și de funcții, dintre care una, și numai una, are numele main() și cu ea începe execuția programului.

Spre deosebire de C++, în Python definițiile de clase, de funcții și de variabile se fac prin instrucțiuni executabile, prin urmare un program Python este doar o secvență de instrucțiuni scrisă într-un fisier.py.

De exemplu, următorul program

```
def f():
    print("start")
print(locals())
```

care are rezultatul

```
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__':
<_frozen_importlib_external.SourceFileLoader object at 0x000001F61B566D00>,
'__spec__': None, '__annotations__': {}, '__builtins__': <module 'builtins'
(built-in)>, '__file__': 'D:\test1.py','f': <function f at 0x000001F61B79C310>}
```

este format din două instrucțiuni: definiția lui `f` și apelul lui `print`.

La execuția instrucțiunuii `def f()`, a fost actualizat dicționarul spațiului de nume local cu înregistrarea `'f': <function f at 0x000001F61B79C310>` prin care numele funcției este asociat de un "function object" (o structură C definită în CPython) care conține, printre altele, codul executabil (în format *bytecode*) al funcției `f`, format la compilare.

---

1 un exemplu: Real Python

Citiți documentația: A function definition is an executable statement. Its execution binds the function name in the current local namespace to a function object (a wrapper around the executable code for the function). This function object contains a reference to the current global namespace as the global namespace to be used when the function is called.

The function definition does not execute the function body; this gets executed only when the function is called.

Dacă ați înțeles tot ce scrie mai sus, nu mai aveți nevoie de acest dicționar.

Python și CPython: limbajul și implementarea sa oficială. Distincția dintre limbaj și implementare se aplică și în C++ unde, de exemplu, limbajul nu are instrucțiuni de intrare/ieșire iar pentru aceste operații trebuie utilizate clase de *stream*-uri de intrare/ieșire importate din biblioteci care, deși sunt standardizate, depind totuși de mediul de programare.

Limbajul Python are acum mai multe implementări scrise în diverse limbaje, Jython în Java, IronPython în C#, de exemplu, dar implementarea standard este cea dezvoltată de Python Software Foundation, și care poate fi descărcată de pe saitul oficial, python.org. In documentație interpretorul este numit CPython, pentru că este scris în C (nu în C++), dar această denumire nu este folosită nicăieri în cadrul distribuției: CPython-ul este lansat de python.exe.

CPython-ul este numit *interpretor* deoarece nu traduce programul Python în cod mașină, cum fac compilatoarele de C/C++, totuși, atunci când prelucrează un fișier script.py, modul de lucru despre care discutăm aici, el se comportă într-o primă fază ca un *compilator* care traduce textul sursă într-un limbaj intermediar de tip *bytecode* (micro-instrucțiuni pe doi octeți) și apoi *interpretează* aceste micro-instrucțiuni pe o mașină virtuală (adică simulând funcționarea unui microprocesor standard asociat unei stive de execuție).

Compilarea începe cu analiza sintactică a codului, când se depistează erorile de sintaxă:

```
a = 1 +* 5
      ^
SyntaxError: invalid syntax
```

O eroare de tipul

```
a = 1
b = "doi"
c = a +b
```

trece de compilare, dar la rulare obținem:

```
Traceback (most recent call last):
File "D:\test_modul.py", line 19, in <module>
      c = a +b
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Programul tradus în *bytecode* este salvat in fișiere binare cu extensia .pyc și poate fi dezasamblat cu metode din modul dis. (vezi https://docs.python.org/3.9/library/dis.html)

Pentru a transforma un script.py într-un fișier executabil, care să ruleze independent de existența interpretorului, sunt necesare programe speciale, de exemplu cel furnizat de pachetul

`pyinstaller`, indexat PyPI (vezi https://pyinstaller.org/en/stable/). Executabilul obținut este destul de voluminos, deoarece înglobează practic mașina virtuală Python, și nu rulează mai repede decât varianta rulată cu interpretorul Python.

script: modulul curent, cel aflat în prelucrare, mai simplu: fisierul .py care conține textul sursă al programui și care este prelucrat cu comanda shell `python script.py` sau cu comanda Run 'fisier.py' din PyCharm.

funcții și tipuri predefinite: conținutul bibliotecilor standard. Limbajul Python este ușor de utilizat în aplicații deoarece interpretorul său, la lansare, încorporează automat o bibliotecă vastă de funcții și de clase, așa numitele *built-in types*[2]. Acestea nu aparțin limbajului, depind de implementare, dar au un caracter standard, iar numele lor sunt accesibile din orice punct al textului sursă și la orice moment al rulării, dacă nu sunt ascunse din greșelă:

Script:

```
f = print
f("ok")
print = 10
print("zece")
```

Rezultat

```
ok
Traceback (most recent call last):
  File "D:\test1.py", line 21, in <module>
    print("zece")
TypeError: 'int' object is not callable
```

obiect: instanță a unei clase Python. Obiectele au apărut odată cu C++-ul, prin introducerea conceptelor de clasă și de instanță a unei clase. Primul compilator de C++, scris de Bjarne Stroustrup în 1979, traducea mai întâi codul C++ în cod C, după care utiliza vechiul compilator de C, implementând obiectele ca structuri dotate cu pointeri către funcțiile membru.

Obiectele din C++ au date și funcții membre, la fel și cele din Python, dar aici ele sunt numite *atribute*, în ambele limbaje pentru accesarea lor se utilizează simbolul punct:

```
obiect.atribut
```

In Python există clasa primordială `object`, moștenită implicit de orice altă clasă, prin urmare toate obiectele au atributele acestei clase:

Script:

```
a = object()
print(dir(a))
print(a.__class__)
```

---

2 vezi built-in functions și built-in types

```
a=5
print(a.__class__)
```

Rezultat:

```
['__class__', '__delattr__', '__dir__', '__doc__', ... '__str__']
<class 'object'>
<class 'int'>
```

Funcția predefinită `dir()` returnează lista atributelor argumentului, și, după cum se observă, atributul `__class__`, pe care îl are orice obiect Python, precizează tipul obiectului, clasa a cărei instanță este. Odată creat, tipul unui obiect uzual nu mai poate fi schimbat:

```
class MyClass:
    pass
a = object()
a.__class__ = MyClass
#TypeError: __class__ assignment only supported for heap types or ModuleType
subclasses
```

Să reținem acestă diferență esențială față de C++: toate obiectele Python au un atribut care le precizează tipul.

identitatea unui obiect: **adresa de memorie a obiectului**. Orice obiect primește când este creat un număr de identificare pe care îl putem obține funcția predefinită `id()`.

Script:

```
a = 10
print(id(a))
a = "zece"
print(id(a))
```

Rezultat:

```
1810733689424
1810734729968
```

Limbajul nu atribuie vreo semnificație numărului returnat de funcția `id()`, se precizează numai că obiectele distincte au *id*-uri diferite, dar pentru CPython se specifică în clar: identitatea este adresa de memorie a obiectului[3]. Ca dovadă, în scriptul următor, în care am afișat *id*-ul lui `f` în format hexazecimal, constatăm că el coincide cu adresa unde este depus codul funcției:

Script:

```
def f():
    pass

print(hex(id(f)))
print(str(f))
```

Rezultat:

---

3 vezi https://docs.python.org/3.9/reference/datamodel.html

```
0x1a59838c310
<function f at 0x000001A59838C310>
```

Programatorul poate folosi *id*-ul numai pentru depanare, eventual pentru testarea identității a două obiecte, așa cum face operatorul `is` din exemplul următor, dar nu are nici o posibilitate să acceseze un obiect folosind *id*-ul său[4] : în Python nu există pointeri.

valoarea unui obiect: informația utilă înmagazinată în acel obiect. Modul în care este utilizată valoarea obiectului depinde de tipul său. De exemplu, valoarea unui 'int' sau a unui 'float' este numărul consemnat în acel obiect. Două obiecte distincte pot avea aceeași valoare, situație care se testează cu operatorul de comparație:

```
a = 5
b = 5.0
print(a == b)
# True
print(a is b)
# False
print(id(a))
print(id(b))
# 1633011394992
# 1633018346320
```

mutabil/ imutabil (*mutable* / *immutable*): modificabil / nemodificabil. Un tip este modificabil dacă valoarea unui obiect de acel tip poate fi modificată fără să se schimbe identitatea, *id*-ul obiectului. De exemplu tipul 'list' este modificabil, iar 'int' nu:

Script:

```
a = 10
print(a, type(a), id(a))
a = 11
print(a, type(a), id(a))
a = [10, 11]
print(a, type(a), id(a))
a[0] = 12
print(a, type(a), id(a))
```

Rezultat:

```
10 <class 'int'> 2173839567440
11 <class 'int'> 2173839567472
[10, 11] <class 'list'> 2173845523136
[12, 11] <class 'list'> 2173845523136
```

4 pentru CPython poate fi folosit în acest scop modulul `ctypes`, dar acesta nu este un modul standard.

identificatori speciali : __atribut__. Identificatorii Python au același înțeles ca în C/C++, cu aceleași reguli de scriere. Totuși, prin convenție, identificatorii care încep și se termină cu două caractere *underscore* sunt utilizați la numirea atributelor claselor standard sau li se atribuie o semnificație de sine stătătoare, și nu ar trebui utilizați de programator în alte scopuri.

Un exemplu de utilizare al atributului `__name__` al modulului curent: când fisierul exemplu.py

```
print("sunt fisierul exemplu.py")
def felicitare():
    print("Hello Python")

if __name__ == '__main__':
    felicitare()
```

este executat ca script, se obține

```
sunt fisierul exemplu.py
Hello Python
```

iar dacă îl importăm în alt script:

```
print("sunt fisierul test.py")
import exemplu
```

obținem

```
sunt fisierul test.py
sunt fisierul exemplu.py
```

Dacă ținem cont că în momentul în care un modul este importat acesta este mai întâi executat, trebuie să ne întrebăm: de ce nu mai apare `Hello Python`? Răspunsul e aici: Top-level script environment.

nume : orice identificator diferit de un cuvânt rezervat al limbajului.[5] In C/C++ avem nume de variabile, nume de constante și nume de tipuri utilizator (nume de enumerări, structuri, clase). La rulare, numele unei variabile desemnează o locație de memorie cu conținut variabil, dar organizată în acord cu tipul variabilei, declarat la compilare. Numele unui tablou este un nume de constantă (adresa tabloului), la fel și numele unei funcții (adresa codului funcției).

In Python orice nume desemnează, la rulare, un obiect, inclusiv un nume de funcție sau de clasă:

```
def f():
    pass

print(isinstance(f, object))
print(hex(id(f)))
# True
# 0x219133ec310

f = "panta rhei"
```

5 vezi reserved words

```
print(isinstance(f, object))
print(hex(id(f)))
# True
# 0x21913296a50
```

Obiectul referit de un nume poate fi schimbat la rulare, prin urmare nu există nume de constante, doar nume de variabile (în sensul că se referă la ceva schimbător). In acest caz, se poate face abstracție de termenul "variabilă", și documentația oficială se străduiește din greu să folosească termenul "*name*" în loc de "*variable*", care este păstrat doar ca termen sinonim[6].

referințe: pointeri ascunși. Referințele au apărut în C++ pentru a înlocui parțial lucrul cu pointeri, utilizarea lor a fost generalizată în Java și Python, unde pointerii au dispărut cu totul, și au fost preluate parțial în parțial în C#, unde avem variabile de tip valoare și variabile de tip referință.

In programul C++ următor

```
int main() {
  int x = 5;
  int& y = x;
  y = 10;
  cout << x << endl;
  // 10
  return 0;
 }
```

este declarat `y` ca o referință a lui `x`, dar poate fi înțeles ca un pointer constant:

```
int main() {
  int x = 5;
  int * const y = &x;
  *y = 10;
  cout << x << endl;
  // 10
  return 0;
}
```

In documentația Python, expresia "*Names refer to objects*" are în limbajul C++ înțelesul "variabilele sunt pointeri care țintesc obiecte alocate dinamic".

Comportamentul de tip pointer al numelor este bine mascat în cazul tipurilor imutabile, dar vizibil pentru tipurile mutabile:

```
a = 10
b = a
b += 10
print(a) # 10
print(b) # 20
```

---

6 vezi https://docs.python.org/3/reference/executionmodel.html

```
a = [10]
b = a
b += [20]
print(a) # [10,20]
print(b) # [10,20]
```

Utilizând terminologia C#, putem spune că la atribuire obiectele imutabile au un comportament de tip valoare, iar cele mutabile au un comportament de tip referință.

definirea unui nume: consemnarea lui, la compilare, într-o tabelă de simboluri (*symbol table*). In C/C++ lucrurile sunt simple: un identificator nu poate fi folosit în textul sursă înainte de a fi declarat, declarația precizează că el este un nume și că se referă la date de un anumit tip.

In Python situația este mai nuanțată: la compilare un nume poate să apară în textul sursă înainte de a fi definit, dar în acest caz la rulare avem o eroare la execuție.

Un nume este definit explicit în următoarele situații: definiția unei clase declară numele acesteia și al atributelor sale, definitia unei funcții îi declară numele și parametrii, iar sintaxa instrucțiunii for declară numele iteratorului.

Definirea implicită a unui nume provine din declarațiile C/C++ de variabile simple cu inițializare, în care ștergem tipul declarat al variabilei și terminatorul punct și virgulă; astfel declarația din C/C++

```
int x = 10;
```

devine în Python instrucțiunea de atribuire

```
x = 10
```

care la compilare definește sau re-definește pe `x` ca nume (`'x'` este introdus de compilator într-o tabelă de simboluri), iar la rulare apelează mai întâi constructorul adecvat literalului `'10'`, `int('10')` în acest caz, și apoi *id*-ul obiectului returnat este asociat numelui `x`, despre care se poate spune doar că este un nume care se referă acum la un 'int' de valoare 10.

domeniu de vizibilitate (*scope*) : o regiune din textul sursă în care numele definite acolo nu sunt vizibile din exterior. La compilarea textului sursă sunt detectate definițiile numelor și, pe baza locului în care apar, numele sunt grupate, analog limbajului C++, în domenii de vizibilitate: un nume definit în corpul unei funcții, inclusiv în lista parametrilor, este nume de variabilă locală, un nume definit la nivelul modulului, exterior oricărui domeniu local este nume global, vizibil din tot textul sursă.

Atenție: spre deosebire de C/C++, blocul unei instrucțiuni compuse (cel identat în Python) nu constituie un domeniu de vizibilitate, un nume definit în interiorul lui este vizibil în tot domeniul în care se află instrucțiunea.

Deoarece în Python sunt permise definiri de funcții și de clase în interiorul altei funcții, pot să apară șiruri oricât de lungi de domenii imbricate, toate având la bază domeniul numelor predefinite urmat de domeniul global al modulului curent.

Script:

```
def f():
    def ff():
        b = 1
        print(a * b)
    print(b)
    ff()
a = 10
f()
```

Rezultat:

```
Traceback (most recent call last):
  File "D:\test1.py", line 7, in <module>
    f()
  File "D:\test1.py", line 5, in f
    print(b)
NameError: name 'b' is not defined
```

Observăm aici că numele global `a` poate fi utilizat în funcția `ff()`, dar `b`, care este definit local în `ff()` nu este vizibil din exteriorul funcției `ff()`.

rezoluția numelor: determinarea, la rulare, a valorii asociate unui nume. In interiorul unui domeniu de vizibilitate citirea valorii unei variabile locale este clară, problema se pune când variabila este definită într-un domeniu exterior. Rezolvarea este aceeași ca în C/C++ și are la bază regula: dacă un nume vizibil dar extern unui domeniu este re-definit în acel domeniu, noua definiție o ascunde pe prima:

Script:

```
def f():
    def ff():
        b = 1
        print(a * b)
    a = 5
    ff()
a = 10
f()
print(a)
```

Rezultat:

```
5
10
```

Observăm că atribuirea `a = 5` re-definește numele a ca variabilă locală funcției `f()`, vizibil din `ff()`și care ascunde numele global `a`. Dacă dorim să modificăm valoarea lui a din interiorul lui f, folosim instrucțiunea `global`:

Script:

```
def f():
    global a
    def ff():
        b = 1
        print(a * b)
    a = 5
    ff()
a = 10
f()
print(a)
```

Rezultat:

```
5
5
```

Pentru mai multe amănunte, inclusiv utilizarea instrucțiunii `nonlocal`, vezi documentația[7]. Rezoluția numelor se execută la rulare, atunci când trebuie citită valoarea obiectului referit de nume.

legarea numelor (*name binding*): **asocierea unei valori unui nume,** stabilirea, la rulare, a unei conexiuni între un nume și o valoare, prin intermediul unui obiect referit de acel nume. In urma execuției instrucțiunii Python

```
x = 10
```

se spune că x este legat de valoarea 10. Nu se poate spune că x are valoarea 10, sau că lui x i s-a atribuit valoarea 10, deoarece s-ar ascunde faptul că x este un nume care se referă la un obiect (valoarea lui x este de fapt *id*-ul obiectului referit)

Prin analogie, în C/C++ avem situația

```
int main() {
   int ob = 10;
   int* x = &ob;
   cout << "valoarea lui x = "<< x << endl;
   // valoarea lui x = 000000B1D6F7FA14
   cout << "valoarea tintei lui x = " << *x << endl;
   // valoarea tintei lui x = 10
   return 0;
}
```

în care vedem clar că valoarea unui pointer este adresa țintei și nu valoarea țintei. Putem spune cel mult că pointerul este legat de valoarea țintei.

Un nume Python nu este un pointer, în CPython este implementat doar ca un string, dar este utilizat ca un pointer.

---

7 vezi https://docs.python.org/3/tutorial/classes.html#python-scopes-and-namespaces

Așa cum în C/C++ o variabilă nu poate fi folosită înainte de a fi inițializată, la fel și aici, la rulare un nume nu poate fi folosit înainte de a fi legat de o valoare.

```
def f(x):
    y = a*x
    return y
a = 13
print(f(10))
# 130
```

Aici, la execuția apelului `f(10)` parametrul `x` este legat de valoarea `10` iar `y` de `130`.

spațiu de nume, *namespace*: **dicționarul numelor locale deja inițializate**. Cuvântul "*namespace*" este cuvânt cheie în C++, unde definește, practic, un nume de famile pentru un grup de identificatori. Aici `Vasilescu :: Vasile` este Vasile al familiei Vasilescu. Nici *namespace* și nici operatorul de rezoluție `::` nu mai este folosit în limbajul Python, iar termenul "rezoluție" are alt înțeles, după cum am văzut deja.

In Python, "*namespace*" este doar un termen folosit intens în documentație, în strânsă corelație cu "*scope*", diferența principală dintre aceste două noțiuni fiind următoarea: domeniile de vizibilitate sunt determinate static, la compilare, iar *namespace*-urile sunt construite dinamic, la rulare, imediat ce intră în execuție un bloc de execuție (modul, funcție, clasă) și sunt actualizate pe tot timpul execuției ori de câte ori un nume este legat de o valoare.

La lansare, interpretorul parcurge o etapă de configurare în care, printre altele, crează *namespace*-ul tipurilor predefinite, pe care îl putem consulta sub forma unei liste de nume, ele toate fiind deja legate de valorile lor prestabilite:

Script:

```
print(dir(__builtins__))
```

Rezultat:

```
['ArithmeticError', 'AssertionError', ..., 'ZeroDivisionError',
'__build_class__', '__debug__', '__doc__', '__import__', '__loader__',
'__name__', '__package__', '__spec__', 'abs', 'all', 'any', 'ascii', 'bin',
'bool', 'breakpoint', 'bytearray', 'bytes', 'callable', 'chr', 'classmethod',
'compile', 'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod',
'enumerate', 'eval', 'exec', 'exit', 'filter', 'float', 'format', 'frozenset',
'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int',
'isinstance', 'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'map',
'max', 'memoryview', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow',
'print', 'property', 'quit', 'range', 'repr', 'reversed', 'round', 'set',
'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple',
'type', 'vars', 'zip']
```

După ce este compilat scripul, execuția începe cu formarea și actualizarea pas cu pas a *name-space*-ului global și apoi pentru fiecare bloc intrat în execuție se crează un *namespace* local, la fiecare intrare în execuție a blocului. In cazul apelurilor recursive ale unei funcții, apare o stivă de *namespace*-uri.

Programatorul poate consulta aceste *numespace*-urile sub forma unor dicționare nume: valoare, returnate de funcțiile predefinite `globals()` și `locals()`.

Script:

```
def f(x):
    if x >= xmax:
        y = 13
        print("aici:", locals())
        return
    print("inaine:", locals())
    f(x + 1)
    print("dupa:", locals())
xmax = 3
f(0)
print("global:", globals())
```

Rezultat:

```
inaine: {'x': 0}
inaine: {'x': 1}
inaine: {'x': 2}
aici: {'x': 3, 'y': 13}
dupa: {'x': 2}
dupa: {'x': 1}
dupa: {'x': 0}
global: {'__name__': '__main__',  '__doc__': None, '__package__': None,
... ... ,  'f': <function f at 0x000001F2E118C3A0>, 'xmax': 3}
```

Intrebare: de ce legătura `'y': 13` apare numai o singură dată?

Incă un exemplu: programul

```
a, b = 10, 20
def f():
    print("a=",a)
    print("b=",b)
f()
```

are rezultaul asteptat

```
a= 10
b= 20
```

dar dacă mai adăugăm o instrucțiune,

```
a, b = 10, 20
def f():
    print("a=",a)
```

```
        print("b=",b)
        b = 13
    f()
```

obținem

```
a= 10
Traceback (most recent call last):
  File "D:\exemplu.py", line 7, in <module>
    f()
  File "D:\exemplu.py", line 4, in f
    print("b=",b)
UnboundLocalError: local variable 'b' referenced before assignment
```

De ce? Răspuns: deoarece acum la compilare atribuirea `b = 13` re-definește pe `b` ca variabilă locală funcției `f()`, prin urmare la execuția liniei `print("b=",b)` ea este acum căutată în *name-space*-ul local, care este gol la momentul consultării.

`UnboundLocalError` apare la încercarea de a folosi o variabilă ne-inițializată.

***

In final, un test: programul

```
N = 3
A = [None] * N
linie = [None] * N
print("Matricea A:")
for i in range(N):
    for j in range(N):
        linie[j] = i + j
    A[i] = linie
    print(A[i])
tr = sum([A[i][i] for i in range(N)])
print("are urma Tr(A)=", tr)
```

scrie pe monitor:

```
Matricea A:
[0, 1, 2]
[1, 2, 3]
[2, 3, 4]
are urma Tr(A)= 9
```

Afirmația este falsă, unde este eroarea?