

INTRODUCERE ÎN
PYTHON



FOLOSIND

GOOGLE COLAB

Adriana STAN

Adriana STAN

Introducere în Python folosind Google Colab



Introducere în Python folosind Google Colab, Adriana STAN
Copyright © 2022

UTPRESS

ISBN

CLUJ-NAPOCA, ROMANIA

The Legrand Orange Book, LaTeX Template, Version 2.0 (9/2/15)
Sursa: <http://www.LaTeXTemplates.com>.

Prima ediție, 2022

Prefață

Introducere în Python folosind Google Colab se dorește a fi o introducere practică în sintaxa și conceptele asociate limbajului de programare Python. Pentru a facilita asimilarea rapidă a acestui limbaj, exemplele de cod sunt prezentate prin intermediul mediului Google Colab. Acesta permite rularea interactivă a codului dintr-un browser web, fără a fi necesară instalarea vreunei aplicații pe mașina locală. Astfel încât, volumul are asociată o pagină web în cadrul căreia se regăsesc tutorialele în format electronic alături de resursele necesare rulării acestora pentru a putea fi accesate și rulate mult mai ușor de către utilizatori:

www.github.com/adrianastan/python-intro/

Volumul este structurat în șapte tutoriale asociate marilor capitole ale unui limbaj de programare. Resurse bibliografice suplimentare și exerciții sunt introduse la finalul fiecărui tutorial. Trebuie menționat faptul că aceste tutoriale nu sunt orientate către partea teoretică a programării, astfel că nu sunt introduse definiții extinse sau exemple teoretice de utilizare a conceptelor programatice.

Redactarea acestui volum nu ar fi fost posibilă fără sprijinul, discuțiile și ideile valoroase oferite de către Gabriel ERDEI și susținerea Pentalog, Cluj-Napoca.

Cluj-Napoca, 2022

Cuprins

T1	Mediul de lucru și primul cod	
T2	Generalități ale limbajului	
T2.1	Limbajul Python. Generalități.	12
	Cum rulează un program Python?	15
	Introducere în tipuri de date	16
	Introducere în operatori	21
	Introducere în instrucțiuni	21
T2.2	Organizarea mediului de lucru Python	23
	Pachete și module	23
	Biblioteca standard Python	26
	Python virtual environment (venv)	29
	Fișierul de dependențe	30
	Documentația codului	31
T3	Tipuri de date și operatori	
T4	Instrucțiuni	
T5	Funcții. Module	
T6	Programare obiectuală	

Mediul de lucru și primul cod

T2 Generalități ale limbajului

T2.1 Limbajul Python. Generalități. 12

- T2.1.1 Cum rulează un program Python?
- T2.1.2 Introducere în tipuri de date
- T2.1.3 Introducere în operatori
- T2.1.4 Introducere în instrucțiuni

T2.2 Organizarea mediului de lucru Python ... 23

- T2.2.1 Pachete și module
- T2.2.2 Biblioteca standard Python
- T2.2.3 Python virtual environment (venv)
- T2.2.4 Fișierul de dependențe
- T2.2.5 Documentația codului

T2.1. Limbajul Python. Generalități.

Istoric.

Limbajul [Python](#) a fost dezvoltat de către Guido van Rossum în cadrul Centrum Wiskunde & Informatica (CWI), Olanda. Python a apărut ca un succesor la limbajului [ABC](#), iar prima sa versiune a fost lansată în 20 februarie 1991.

Între timp au fost lansate alte două [versiuni majore](#):

- Python 2.0 - 16 Octombrie 2000
- Python 3.0 - 3 Decembrie 2008

Începând cu ianuarie 2020, versiunea 2.0 nu mai are suport din partea echipei de dezvoltatori. Versiunea curentă este 3.11, iar cea mai utilizată implementare este [CPython](#).

Putem afișa versiunea utilizată de interpretor astfel:

```
[1]: # Afișăm versiunea de Python utilizată
import sys
print(sys.version)
```

```
[1]: 3.7.13 (default, Apr 24 2022, 01:04:09)
      [GCC 7.5.0]
```

Rezultatul afișării ne informează privind versiunea utilizată (3.7.13), data la care a fost compilată (24 aprilie 2022) și versiunea de compilator C pe care se bazează (7.5.0).

De ce Python?

Limbajul de programare Python a venit ca urmare a necesității integrării mai multor paradigme de programare, precum și ca o simplificare a sintaxei complexe utilizate în limbajul C/C++.

Cea mai bună descriere a limbajului poate fi dată de cele 19 principii ale lui Tim Peters ce au ghidat dezvoltarea limbajului și care sunt adunate sub

denumirea *The Zen of Python*. Acestea poate fi vizualizate prin intermediul instrucțiunii `import this`.

```
[2]: import this
```

[2]: The Zen of Python, by Tim Peters

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way_
    ↳to do it.
Although that way may not be obvious at first unless you're_
    ↳Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good_
    ↳idea.
Namespaces are one honking great idea -- let's do more of_
    ↳those!
```

De ce NU Python?

Deși are numeroase avantaje, limbajul Python nu se pretează oricărei aplicații. Acest lucru se datorează și specializării limbajului către un sub-set de aplicații, cele mai importante fiind calculul numeric și învățarea automată (en. *machine learning*). Printre [dezavantajele](#) Python se numără:

- Python este puțin mai lent decât alte limbaje de programare datorită caracterului de limbaj interpretor;
- Momentan nu există suport extins pentru realizarea de aplicații mobile;

- Poate avea un consum de memorie mai ridicat decât alte limbaje;
- Accesul la baze de date se realizează greoi;
- Apariția erorilor la rulare (en. *runtime errors*) datorită caracterului de interpretor;
- Dificultatea integrării altor limbaje în cod;
- Simplitatea limbajului face ca în anumite cazuri calitatea codului să aibă de suferit.

Încadrarea limbajului și domenii de utilizare

Limbajul Python prezintă următoarele caracteristici:

- Open source - întreg limbajul este disponibil în format open source, ceea ce înseamnă că poate fi distribuit și utilizat chiar și în medii comerciale fără a fi nevoie să se achiziționeze o licență de dezvoltator;
- Multi-paradigmă - permite utilizarea mai multor paradigme de programare, precum: programare obiectuală, programare procedurală, programare funcțională, programare structurată și programare reflexivă;
- De nivel înalt (high-level) - include o serie largă de abstractizări ale utilizării resurselor mașinii de calcul, ceea ce îl face mai apropiat de limbajul natural uman;
- De tip interpretor - codul Python nu este pre-compilat, ci fiecare linie de cod este executată la momentul în care apare în cod;
- Utilizează tipizarea dinamică - nu este nevoie să se specifice în clar tipul unui obiect, acesta fiind dedus automat din expresia de inițializare;
- Utilizează rezoluția dinamică a numelor (en. *late binding*) - ceea ce înseamnă că numele funcțiilor sau a obiectelor sunt atașate unei funcționalități sau date doar la rulare și nu în partea de compilare. Acest lucru permite re-utilizarea denumirilor pentru a referi diferite elemente ale codului;
- Foarte ușor extensibil - Python deține una dintre cele mai largi biblioteci de module și pachete create de programatori terți.

Dintre cele mai importante **domenii de aplicare** ale limbajului Python, putem enumera:

- aplicații web folosind framework-urile **Django** sau **Flask**;
- aplicații științifice sau numerice folosind modulele **SciPy** și **NumPy**;

- aplicații de învățare automată folosind modulele [PyTorch](#) sau [TensorFlow](#).

Iar o listă de aplicații de succes ce utilizează limbajul Python poate fi găsită pe [site-ul oficial](#).

T2.1.1 Cum rulează un program Python?

Interpretorul Python

Codul scris în limbajul Python **NU** este compilat. Fiecare linie de cod este executată atunci când apare în cod, inclusiv partea de includere de module externe și crearea/apelarea claselor/funcțiilor/metodelor.

Există însă o formă intermediară, denumită *byte code* care rezidă în fișiere cu extensia *.pyc* și care începând cu Python 3.0 sunt stocate în directoare denumite `__pycache__`.

Pentru a obține aceste reprezentări intermediare se poate utiliza comanda de mai jos asupra fișierelor Python pe care dorim să le precompilăm:

```
python -m compileall file_1.py ... file_n.py
```

Structurarea codului Python

Convenția de notare a extensiei fișierelor ce conțin cod Python este *.py*. Din punct de vedere al ierarhiei unei aplicații Python avem următoarele componente:

- programele sunt compuse din module;
- modulele conțin instrucțiuni;
- instrucțiunile conțin expresii;
- expresiile crează și prelucrează obiecte;
- mai multe module pot fi grupate într-un pachet.

Spre deosebire de alte limbaje de programare des utilizate, Python nu folosește un simbol pentru marcarea sfârșitului unei instrucțiuni (de ex. `;`) sau simboluri speciale pentru marcarea începutului și finalului instrucțiunilor compuse (de ex. `{}`).

Modul în care Python structurează instrucțiunile se bazează pe utilizarea spațiilor albe sau a indentării codului. Acest lucru înseamnă că instrucțiunile de același nivel vor fi plasate la același nivel de indentare. Corpul instrucțiunilor compuse va fi demarcat de un nivel de indentare suplimentar, iar începutul instrucțiunii compuse se va marca prin utilizarea

simbolului `:`. Finalul acesteia este determinat de revenirea la nivelul de indentare anterior. De exemplu:

```
if a > b:
    if a > c:
        print(a)
    else:
        print(c)
else:
    print(b)
```

Este foarte important ca utilizarea spațiilor albe să fie consecventă, fie spații albe ' ', fie taburi '\t'. [Se recomandă](#) pentru simplitate utilizarea de spații albe, de obicei 2 sau 4 pentru indentarea codului.

În secțiunile următoare vor fi indexate pe scurt principalele tipuri de date, operatori și instrucțiuni specifice Python și care vor fi reluate pe larg în tutorialele următoare. Spre finalul acestui tutorial vor fi prezentate și o serie de noțiuni legate de crearea unui mediu de lucru virtual, salvarea listei de module dependente din aplicații și documentarea codului.

T2.1.2 Introducere în tipuri de date

În limbajul Python toate datele sunt OBIECTE!!!

Acest lucru înseamnă că nu vom avea tipuri de date primitive, așa cum există în C/C++ sau Java.

O altă caracteristică a limbajului ce simplifică scrierea aplicațiilor se referă la utilizarea tipizării dinamice (en. *dynamic typing*). Prin acest mecanism, tipul obiectului sau a datei utilizate nu trebuie menționat la instanțierea variabilelor. Tipul variabilei va fi determinat automat pe baza valorii cu care este inițializată:

```
[3]: # date întregi
a = 314
# date reale
b = 3.14
# șiruri de caractere (string)
c = "Python"
```

Totodată, deși nu se definesc în clar tipurile de date, limbajul Python este

tipizat puternic (en. *strongly typed*), ceea ce înseamnă că se pot realiza doar operații specifice acelui tip de date. Utilizarea unei operații nepermise este marcată de interpretor ca fiind o eroare.

Codul de mai jos va genera o eroare de tip `TypeError` deoarece interpretorul nu știe cum să adune valoarea întreagă 2 la șirul de caractere "Ana".

```
[4]: a = "Ana"
      "Ana" + 2
```

```
[4]: -----
      TypeError
      Traceback (most recent call last)
      <ipython-input-4-68f083afd972> in <module>
          1 a = "Ana"
      ----> 2 "Ana" + 2
      TypeError: can only concatenate str (not "int") to
      ↪str
```

Tipuri de date Python fundamentale

La fel ca în orice limbaj de programare, Python include un set de tipuri de date fundamentale, listate mai jos și care pot fi extinse prin definirea de obiecte de către programator.

Tipul obiectului	Exemplu
Număr	1234, 3.1415, 3+4j, 0b111, Decimal(), Fraction()
String	'Ana', "Maria", b'a\x01c', u'An\xc4'
Listă	[1, [2, 'trei'], 4.5], list(range(10))
Dicționar	{'cheie': 'valoare', 'key': 'value'}, dict(cheie=3.14)
Tuplu	(1, 'Ana', 'c', 3.14), tuple('Ana'), namedtuple
Set	set('abc'), {'a', 'b', 'c'}
Fișier	open('fisier.txt'), open(r'C:\fisier.bin', 'wb')
Alte tipuri de bază	Boolean, bytes, bytearray, None

Un aspect important legat de date în Python se referă la caracterul mutabil al acestora. **Mutabilitatea** reprezintă posibilitatea modificării conținutului unui obiect:

- **obiecte mutabile** - valorile lor pot fi modificate (ex. liste, dicționare și seturi, toate obiectele definite în clase utilizator);
- **obiecte imutabile** - valorile lor nu pot fi modificate (ex. int, float,

complex, string, tuple, frozen set, bytes).

OBS

Înainte de a intra în mai multe detalii legate de mutabilitate, este important să menționăm faptul că în Python variabilele sunt de fapt doar referințe (pointeri) la locații de memorie ce conțin datele în sine. Cu alte cuvinte, spațiul de memorie alocat unei variabile se rezumă la dimensiunea unei adrese de memorie, iar datele (valorile) vor fi stocate în alte zone de memorie. Vom reveni asupra acestui aspect în tutorialul următor.

În cazul obiectelor imutabile, putem atribui o nouă valoare variabilei utilizate, însă acest lucru duce la crearea unui nou obiect și referențierea sa prin intermediul variabilei. Putem verifica acest lucru folosind funcția `id(Object)` ce ne va returna un identificator unic pentru fiecare obiect din cod:

```
[5]: a = 3
print ("Adresa obiectului 3: ", hex(id(3)))
print("Adresa referită de a: ", hex(id(a)))
a = 4
print ("Adresa obiectului 4: ", hex(id(4)))
print ("Adresa referită de a : ", hex(id(a)))
```

```
[5]: Adresa obiectului 3: 0xabc140
Adresa referită de a: 0xabc140
Adresa obiectului 4: 0xabc160
Adresa referită de a : 0xabc160
```

În schimb, pentru obiecte mutabile, adresa referită de variabilă se păstrează la modificările conținutului obiectului:

```
[6]: # definim o listă de obiecte
lista = ['a', 1, 3.14]
print ("Lista inițială:", lista)
print("Adresa inițială:", hex(id(a)))
# modificăm primul element din listă
lista[0] = 'b'
print ("\nNoua listă:", lista)
print("Adresa după modificare:", hex(id(a)))
```

```
[6]: Lista inițială: ['a', 1, 3.14]
Adresa inițială: 0xabc160
```

Noua listă: ['b', 1, 3.14]
 Adresa după modificare: 0xabc160

Metode implicite asociate obiectelor

Tipurile de date fundamentale au o serie de *metode implicite* asociate. Pentru a afla metodele asociate unui obiect putem utiliza funcția: `dir(Object)`

```
[7]: S = "abc"
# Pentru a eficientiza spațiul, lista metodelor a fost
    ↳ concatenată
# prin spații albe. Se poate utiliza și dir(S) direct.
' '.join(dir(S))

[7]: '__add__ __class__ __contains__ __delattr__ __dir__ __doc__
    ↳ __eq__ __format__ __ge__ __getattr__ __getitem__
    ↳ __getnewargs__ __gt__ __hash__ __init__ __init_subclass__
    ↳ __iter__ __le__ __len__ __lt__ __mod__ __mul__ __ne__
    ↳ __new__ __reduce__ __reduce_ex__ __repr__ __rmod__
    ↳ __rmul__ __setattr__ __sizeof__ __str__ __subclasshook__
    ↳ capitalize casefold center count encode endswith
    ↳ expandtabs find format format_map index isalnum isalpha
    ↳ isascii isdecimal isdigit isidentifier islower isnumeric
    ↳ isprintable isspace istitle isupper join ljust lower
    ↳ lstrip maketrans partition replace rfind rindex rjust
    ↳ rpartition rsplit rstrip split splitlines startswith strip
    ↳ swapcase title translate upper zfill'
```

Metodele implicite, precum și cele create de utilizator au în mod normal asociate documentații de utilizare. Această documentație poate fi accesată prin intermediul funcției `help(Object.method)`.

```
[8]: help(S.replace)
```

Help on built-in function replace:

`replace(old, new, count=-1, /)` method of `builtins.str` instance

Return a copy with all occurrences of substring `old`

↳ replaced by `new`.

`count`

Maximum number of occurrences to replace.

-1 (the default value) means replace all occurrences.

If the optional argument `count` is given, only the first `count` occurrences are replaced.

Introspectia obiectelor

Python include mecanismul de [introspecție](#), prin intermediul căruia se pot determina caracteristici ale obiectelor utilizate în cod. Din acest mecanism fac parte funcții precum `type(Object)`, `dir(Object)` sau `hasattr(Object)`. În tutorialele viitoare vom vedea mecanismul de introspecție aplicat și funcțiilor și claselor, care de altfel sunt tot obiecte în Python.

```
[9]: S = "abc"
     # Tipul obiectului
     print(type(S))
     # Lista metodelor asociate
     print(dir(S))
     # Verificăm dacă obiectul S are asociat atributul 'length'
     print(hasattr(S, 'length'))
```

```
[9]: <class 'str'>
     ['__add__', '__class__', '__contains__', '__delattr__',
     → '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
     → '__getattr__', '__getitem__', '__getnewargs__',
     → '__gt__', '__hash__', '__init__', '__init_subclass__',
     → '__iter__', '__le__', '__len__', '__lt__', '__mod__',
     → '__mul__', '__ne__', '__new__', '__reduce__',
     → '__reduce_ex__', '__repr__', '__rmod__', '__rmul__',
     → '__setattr__', '__sizeof__', '__str__',
     → '__subclasshook__', 'capitalize', 'casefold', 'center',
     → 'count', 'encode', 'endswith', 'expandtabs', 'find',
     → 'format', 'format_map', 'index', 'isalnum', 'isalpha',
     → 'isascii', 'isdecimal', 'isdigit', 'isidentifier',
     → 'islower', 'isnumeric', 'isprintable', 'isspace',
     → 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip',
     → 'maketrans', 'partition', 'replace', 'rfind', 'rindex',
     → 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split',
     → 'splitlines', 'startswith', 'strip', 'swapcase', 'title',
     → 'translate', 'upper', 'zfill']
     False
```

T2.1.3 Introducere în operatori

Operatorii în Python, la fel ca în orice alt limbaj de programare leagă datele în cadrul expresiilor. Din nou, ca în alte limbaje de programare, ordinea de execuție a operatorilor în expresii complexe este dată de așa numita precedență. În Python [tabelul de precedență](#) al operatorilor este următorul:

Operator	Exemplu
(expressions...),[expressions...], {key: value...}, {expressions...}	Expresii în paranteze, de asociere, afișare liste, dicționare, seturi
x[index], x[index:index], x(arguments...), x.attribute	Indexare, partitionare, apel, referirea atributelor
await x	Await
**	Ridicare la putere
+x, -x, ~x	Operator unar +/- și negare pe biți
*, @, /, //, %	Înmulțire, matrice, împărțire, împărțire întreagă, modulo
+, -	Adunare și scădere
<<, >>	Deplasare la stânga/dreapta
&	Și pe biți
^	Sau exclusiv pe biți
	Sau pe biți
in, not in, is, is not, <, <=, >, >=, !=, ==	Comparații, inclusiv testarea calității de membru și a identității
not x	Negare booleană
and	Și boolean
or	Sau boolean
if – else	Expresie condițională
lambda	Funcție anonimă (lambda)
:=	Atribuire

T2.1.4 Introducere în instrucțiuni

Pe scurt, lista de instrucțiuni disponibile în Python este prezentată în tabelul următor. Instrucțiunile pot fi simple (ex. apeluri de funcții) sau compuse (ex. if/elif/else).

Instrucțiune	Rol	Exemplu
Atribuire	Crearea de referințe	a, b = 'Ana', 'Maria'
Apel și alte expresii	Rulare funcții	suma(3, 4)
Apeluri print	Afișare obiecte	print(obiect)

Instrucțiune	Rol	Exemplu
if/elif/else	Selectare acțiuni	if True: print(text)
for/else	Bucle	for x in lista: print(x)
while/else	Bucle	while x > 0: print('Salut')
pass	Instrucțiune vidă	while True: pass
break	Ieșire din buclă	while True: if conditie: break
continue	Continuare buclă	while True: if conditie: continue
def	Funcții și metode	def suma(a, b): print(a+b)
return	Revenire din funcții	def suma(a, b): return a+b
yield	Funcții generator	def gen(n): for i in n: yield i*2
global	Spații de nume	global x, y; x = 'a'
nonlocal	Spații de nume (3.x)	nonlocal x; x = 'a'
import	Import module	import sys
from	Acces la componente ale modulului	from modul import clasa
class	Definire clase de obiecte	class C(A,B):
try/except/ finally	Prindere excepții	try: actiune; except: print('Exceptie')
raise	Aruncare excepții	raise Exceptie
assert	Aserțiuni	assert X > 0, 'X negativ'
with/as	Manager de context (3.X, 2.6>)	with open('fisier') as f: pass)
del	Ștergere referințe	del Obj

Asupra tipurilor de date, a operatorilor și instrucțiunilor vom reveni cu mai multe detalii în tutorialele următoare. Trecem acum spre zona de organizare a mediului de lucru pentru aplicațiile Python și utilizarea bibliotecii standard și a documentației.

T2.2. Organizarea mediului de lucru Python

T2.2.1 Pachete și module

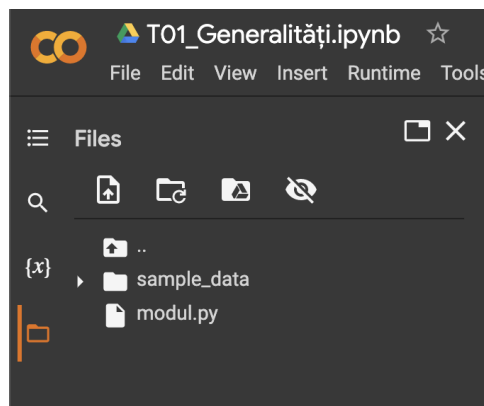
Scrierea codului Python în afara mediilor de programare interactive precum Google Colab se face în cadrul fișierelor ce au extensia `.py`. Un fișier ce conține cod Python este denumit și are rol de *modul*. Un modul va conține instanțieri de date și instrucțiuni.

De exemplu, putem crea chiar în Colab un astfel de modul, folosind [funcțiile magice IPython](#). Codul de mai jos va crea un fișier în mașina virtuală curentă denumit `modul.py`.

```
[10]: %%writefile modul.py
      a = 3
      b = 7
      for i in range(5):
          print(i)
```

[10]: Overwriting modul.py

Acest modul a fost scris în sesiunea curentă de Colab și poate fi vizualizat prin selectarea din bara de meniu din stânga notebook-ului a tab-ului Files.



Iar acum putem importa conținutul acestui modul în notebook-ul curent:

```
[11]: import modul
```

```
[11]: 0
      1
      2
      3
      4
```

La import, codul ce accesibil în afara claselor sau a funcțiilor este executat automat.

Un modul este un set de identificatori denumit și *namespace*, iar identificatorii din modul sunt denumiți *atribute*. După import, putem să folosim și valorile atributelor a și b ale modulului:

```
[12]: notebook_a = modul.a
      notebook_b = modul.b
      print(notebook_a + notebook_b)
```

```
[12]: 10
```

Numele fișierului în care este stocat un modul e disponibil ca atribut: `__name__`

```
[13]: modul.__name__
```

```
[13]: 'modul'
```

Iar prin intermediul funcției `dir()` putem afla lista de atribute ale modului:

```
[14]: ' '.join(dir(modul))
```

```
[14]: '__builtins__ __cached__ __doc__ __file__ __loader__
      ↪__name__ __package__
      __spec__ a b i'
```

Putem să remarcăm faptul că pe lângă atributele definite în codul modului, mai există o serie de atribute predefinite disponibile pentru orice modul Python.

Pentru importarea modulelor mai avem disponibile două metode. Una prin care redenumim modulul în cadrul codului curent sau îi creăm un așa numit alias:

```
import modul as alias
```

```
[15]: import numpy as np
      np.__name__
```

```
[15]: 'numpy'
```

Și una prin care putem specifica doar importul unui subset de attribute ale modulului:

```
from modul import class/function/attribute
```

În acest caz, attributele respective vor fi disponibile fără a specifica numele modulului înainte de acestea. Cu alte cuvinte, realizăm importul atributelor în spațiul de nume curent.

```
[16]: from numpy import sort
      sort([1,9,2,8])
```

```
[16]: array([1, 2, 8, 9])
```

O a treia metodă (nerecomandată) de import a conținutului unui pachet este prin utilizarea caracterului wildcard * ce permite importul tuturor atributelor disponibile în pachet în spațiul de nume curent.

```
from modul import *
```

Pentru acest tip de import vom discuta în tutorialul referitor la pachete și modul despre așa numita listă `__all__` inclusă în fișierul de inițializare `__init__.py` și care specifică attributele ce pot fi importate prin această instrucțiune.

Rulare independentă a modulelor

Evident că putem rula conținutul unui modulul ca script independent.



În Colab pentru [instrucțiunile de linie de comandă \(shell\)](#) este necesară utilizarea semnului `!` înaintea instrucțiunii.

```
[17]: !python modul.py
```

```
[17]: 0
      1
      2
      3
      4
```

La fel, se va executa codul disponibil în modul aflat în afara funcțiilor și claselor.

Pachete

Modulele cu funcționalități similare sunt organizate în **pachete** (en. *packages* sau *dotted module names*). Acest lucru înseamnă că modulele de nivel ierarhic similar vor fi stocate în directoare de același nivel. De exemplu:

pachet/	Pachetul de nivel înalt
__init__.py	Cod pentru inițializarea pachetului
subpachet1/	Subpachet 1
__init__.py	
modul1_1.py	
modul1_2.py	
...	
subpachet2/	Subpachet 2
__init__.py	
modul2_1.py	
modul2_2.py	
...	

Accesul la subpachete se face prin numele pachetului urmat de '.', numele subpachetului și apoi funcția sau clasa apelată. De aici și numele de *dotted module names*.

Fișierul `__init__.py` se folosește pentru a informa interpretorul că directoarele în care există trebuie tratate ca subpachete. Fișierul e de cele mai multe ori gol, dar poate fi utilizat și pentru anumite inițializări/definiții la importul modulului.

T2.2.2 Biblioteca standard Python

Biblioteca standard Python include un număr destul de mare de pachete predefinite, ceea ce face ca scrierea aplicațiilor complexe să se rezume de cele mai multe ori la cunoașterea acestor pachete și funcționalitățile lor.

PIP

Pe lângă pachetele Python de bază, pachete create de alți programatori sunt incluse în [The Python Package Index \(PyPI\)](https://pypi.org/). Orice programator poate publica pachetul propriu pe PyPI.

Pentru a instala pachete din PyPI se pot utiliza următoarele comenzi:

```
>> python3 -m pip install UnPachet
>> pip install UnPachet
>> pip install UnPachet==version.number
>> pip install "UnPachet>=minimum.version"
```

```
[18]: # Instalam un pachet de prelucrare audio
      # https://librosa.org/
      !pip install librosa
```

```
[18]: Looking in indexes: https://pypi.org/simple, https://
      ↳us-python.pkg.dev/colab-wheels/public/simple/
      Collecting librosa
        Downloading librosa-0.9.2-py3-none-any.whl (214 kB)
        ...
      Installing collected packages: librosa
      Successfully installed librosa-0.9.2
```

Iar pentru dezinstalare putem utiliza:

```
!pip uninstall UnPachet
```

```
[19]: !pip uninstall librosa
```

```
[19]: Found existing installation: librosa 0.9.2
      Uninstalling librosa-0.9.2:
        Would remove:
          /usr/local/lib/python3.7/dist-packages/librosa-0.9.2.
          ↳dist-info/*
          /usr/local/lib/python3.7/dist-packages/librosa/*
      Proceed (y/n)? y
      Successfully uninstalled librosa-0.9.2
```

În orice moment putem vizualiza lista completă a pachetelor disponibile în mediul de programare Python curent prin `pip list`.



În Google Colab această listă este extrem de extinsă deoarece mediul este pregătit pentru diferite aplicații de învățare automată și calcul numeric.

```
[20]: # Afișăm primele 10 pachete instalate
!pip list
```

Package	Version
absl-py	1.2.0
aiohttp	3.8.1
aiosignal	1.2.0
alabaster	0.7.12
alumentations	1.2.1
altair	4.2.0
appdirs	1.4.4
arviz	0.12.1
astor	0.8.1
astropy	4.3.1

System Path

Calea către pachetele externe bibliotecii standard este păstrată în `sys.path`. Această listă de căi este parcursă de interpretor pentru a găsi pachetele referite în cod de către programator.

În momentul instalării unui pachet folosind utilitarul `pip`, calea sa este automat adăugată la `sys.path`.

```
[21]: import sys
print(sys.path)
```

```
[21]: ['/content', '/env/python', '/usr/lib/python37.zip', '/usr/
→lib/python3.7',
'/usr/lib/python3.7/lib-dynload', '', '/usr/local/lib/
→python3.7/dist-packages',
'/usr/lib/python3/dist-packages', '/usr/local/lib/python3.7/
→dist-
packages/IPython/extensions', '/root/.ipython']
```

Dacă dorim să adăugăm noi o cale proprie, putem folosi funcția `append`:

```
[22]: sys.path.append('/user/adriana/modul1/')  
      # Afișăm doar ultimele intrări din sys.path pentru a verifica  
      # dacă s-a adăugat calea specificată de noi  
      print(sys.path[-2:])
```

```
[22]: ['/root/.ipython', '/user/adriana/modul1/']
```

T2.2.3 Python virtual environment (venv)

De cele mai multe ori, aplicațiile dezvoltate de programatori necesită instalarea unui set de module externe. În momentul în care aplicațiile sunt transmise către clienți, aceste module trebuie cunoscute și specificate în clar. Iar dacă un programator realizează mai multe aplicații în paralel este utilă separarea mediilor de programare în cadrul aceleiași mașini fizice.

Pentru a putea separa mediul de lucru de pe o anumită mașină de dezvoltare, Python pune la dispoziție **Python virtual environment**. Acest mediu virtual permite separarea aplicațiilor, astfel încât fiecare dintre acestea să ruleze independent, iar totalitatea modulelor de care depind să fie cunoscută. Este important de specificat aici faptul că orice IDE de dezvoltare Python va crea automat astfel de medii virtuale.

Pentru a crea un mediu virtual din linia de comandă putem rula:

```
python -m venv tutorial-env
```

S-a creat o mașină virtuală Python denumită `tutorial-env`. Putem activa această mașină prin intermediul:

```
source tutorial-env/bin/activate
```

După rularea comenzii, se va crea un director denumit `tutorial-env` ce conține toate sursele necesare și pachetele instalate. După activarea mediului, se va modifica indicatorul liniei de comandă pentru a reflecta mediul virtual utilizat momentan.



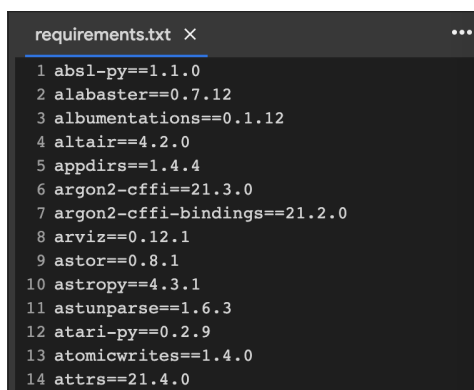
Utilizarea mediilor virtuale în Google Colab nu este necesară, deoarece fiecare notebook este un mediu de programare independent.

T2.2.4 Fișierul de dependențe

La distribuirea unei aplicații, menționam anterior faptul că este nevoie să se specifice setul de pachete și versiuni ale acestora necesare rulării aplicației. Dacă utilizăm un mediu virtual această listă de pachete poate fi obținută foarte ușor prin comanda:

```
[23]: !pip freeze > requirements.txt
```

După rularea comenzii, fișierul `requirements.txt` va conține informație de tipul:

A screenshot of a terminal window with a dark background. The title bar of the window says "requirements.txt x" and there are three dots on the right. The terminal displays a list of 14 dependencies, each on a new line, numbered from 1 to 14. The dependencies are: 1 abs1-py==1.1.0, 2 alabaster==0.7.12, 3 alumentations==0.1.12, 4 altair==4.2.0, 5 appdirs==1.4.4, 6 argon2-cffi==21.3.0, 7 argon2-cffi-bindings==21.2.0, 8 arviz==0.12.1, 9 astor==0.8.1, 10 astropy==4.3.1, 11 astunparse==1.6.3, 12 atari-py==0.2.9, 13 atomicwrites==1.4.0, and 14 attrs==21.4.0.

```
requirements.txt x
1 abs1-py==1.1.0
2 alabaster==0.7.12
3 alumentations==0.1.12
4 altair==4.2.0
5 appdirs==1.4.4
6 argon2-cffi==21.3.0
7 argon2-cffi-bindings==21.2.0
8 arviz==0.12.1
9 astor==0.8.1
10 astropy==4.3.1
11 astunparse==1.6.3
12 atari-py==0.2.9
13 atomicwrites==1.4.0
14 attrs==21.4.0
```

Destinatarul final al aplicației poate mai apoi să instaleze automat toate aceste pachete prin instalarea listei de module specificată în fișierul creat. Numele `requirements.txt` nu este impus, dar este o convenție de denumire a sa.

```
[24]: # Creăm o lista de pachete
%%writefile requirements.txt
librosa
pywer
```

```
[24]: Overwriting requirements.txt
```

```
[25]: !pip install -r requirements.txt
```

```
[25]: Looking in indexes: https://pypi.org/simple, https://
      ↳us-python.pkg.dev/colab-
wheels/public/simple/
Collecting librosa
Using cached librosa-0.9.2-py3-none-any.whl (214 kB)
```

```
Collecting pywer
  Downloading pywer-0.1.1-py3-none-any.whl (3.6 kB)
...
Successfully installed librosa-0.9.2 pywer-0.1.1
```

T2.2.5 Documentația codului

Orice cod scris în mod profesional trebuie să conțină o documentație aferentă. Cel mai ușor de redactat această documentație este atunci când ea rezidă direct în cod. Din acest punct de vedere, Python permite crearea documentației prin scrierea ca prime instrucțiuni în cadrul modulelor, claselor sau funcțiilor a unor explicații privind utilizarea lor sub forma unui comentariu încadrat de ghilimele triple `"""`.

Aceste instrucțiuni devin automat atributul `__doc__` al celui modul/clasă/funcție.

```
[26]: %%writefile moduldoc.py
      """Documentația modulului"""
      class Clasa:
          """Documentația clasei"""
          def metoda(self):
              """Documentația metodei"""
      def functia():
          """Documentația funcției"""
```

[26]: Writing moduldoc.py

```
[27]: import moduldoc
      help(moduldoc)
```

[27]: Help on module moduldoc:

```
NAME
    moduldoc - Documentația modulului

CLASSES
    builtins.object
        Clasa

    class Clasa(builtins.object)
```

```

| Documentația clasei
|
| Methods defined here:
|
| metoda(self)
|     Documentația metodei
|
| -----
| Data descriptors defined here:
|
| __dict__
|     dictionary for instance variables (if defined)
|
| __weakref__
|     list of weak references to the object (if
→defined)

```

FUNCTIONS

```

funcția()
    Documentația funcției

```

FILE

```

/content/moduldoc.py

```

```
[28]: help(moduldoc.Clasa)
```

[28]: Help on class Clasa in module moduldoc:

```

class Clasa(builtins.object)
| Documentația clasei
|
| Methods defined here:
|
| metoda(self)
|     Documentația metodei
|
|
| -----
| Data descriptors defined here:
|
| __dict__
|     dictionary for instance variables (if defined)

```

```
|  
|  __weakref__  
|      list of weak references to the object (if defined)
```

```
[29]: help(moduldoc.Clasa.metoda)
```

```
[29]: Help on function metoda in module moduldoc:  
      metoda(self)  
          Documentația metodei
```

```
[30]: help(moduldoc.functie)
```

```
[30]: Help on function functie in module moduldoc:  
      functie()  
          Documentația funcției
```

Concluzii

În acest tutorial am realizat o introducere destul de abruptă asupra noțiunilor fundamentale ale limbajului Python și a modului de redactare și organizare a codului. Vom reveni în detaliu asupra majorității acestor aspecte în tutorialele următoare, însă considerăm importantă o viziune globală a lucrurilor pe care trebuie să le aprofundăm și la fel de importantă posibilitatea de a redacta cod Python minimal cât mai rapir.

Este important de remarcat simplitatea redactării codului și a organizării acestuia, fapt ce îl face unul dintre cele mai ușor de învățat și utilizat limbaje de programare.

Exerciții

1. Definiți un obiect de tip `float` și verificați cu ajutorul funcției `id()` faptul că este de tip **imutabil**.
2. Consultați lista de metode predefinite ale obiectului de tip `float` definit în exercițiul 1. Verificați programatic dacă metoda `split()` face parte din această listă.
3. Afișați la ecran documentația funcției `split()` a unui șir de caractere.
4. Instalați pachetul `flask` folosind `pip`. Verificați că instalarea a avut succes folosind `!pip list`.

5. Salvați lista pachetelor instalate pentru notebook-ul curent într-un fișier numit `requirements-notebook.txt` folosind `!pip freeze`.
6. Scrieți într-un fișier denumit `prime.py` un modul care printează primele 10 numere prime. Importați modulul `prime` în notebook-ul curent și verificați că numerele printate sunt corecte. Rulați independent modulul `prime` din linia de comandă folosind `!python`.

Referințe suplimentare

- [Evoluția celor mai populare limbaje de programare](#) - online.
- [O scurtă istorie a limbajelor de programare](#) - online.

Tipuri de date și operatori

T3

T4

Instrucțiuni

T5

Funcții. Module



Programare obiectuală

T7

Lucrul cu fişiere

