INTRODUCERE ÎN PYTHON FOLOSIND GOOGLE COLAB

Adriana STAN

Introducere în Python folosind Google Colab

Adriana STAN

Cu contribuții din partea lui

Gabriel ERDEI

Introducere în Python folosind Google Colab
Adriana STAN
Contributor: Gabriel ERDEI

ISBN
CLUJ-NAPOCA, ROMÂNIA

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

T1.1. Mediul de programare Colab

Google Colab este un mediu interactiv de programare în limbajul Python ce rulează direct în browser, fără a realiza vreo configurare pe mașina locală. Colab permite acces la resurse computaționale ce includ și putere de calcul grafic (GPU) și facilitează împărtășirea codului și colaborarea pentru redactarea acestuia. Singura cerință a Colab este ca utilizatorul să aibă asociat un cont Google. Notebook-urile create vor fi stocate automat în Google Drive într-un director creat automat și denumit `Colab Notebooks`.

De asemenea, este important de reținut faptul că în momentul inițializării unei sesiuni de Colab, acesteia i se alocă o mașină virtuală temporară și în cadrul căreia datele salvate (de exemplu fișiere încărcate în sesiune sau salvate prin intermediul codului, dar cu excepția notebook-ului) sunt șterse la încheierea sesiunii. Pentru a stoca date persistent, se poate realiza o conexiune la Google Drive, iar în acest mod conținutul acestuia devine disponibil în notebook și fișierele pot fi încărcate și salvate direct în Google Drive.

Google Colab este o extensie a IPython (Interactive Python) și a mediului Jupyter Notebook. În cadrul acestui mediu, codul este structurat în fișiere denumite *notebook-uri* cu extensia `*.ipynb`. Un notebook conține una sau mai multe *celule* de execuție. O celulă de execuție conține una sau mai multe linii de cod ce pot fi executate individual. Ordinea de execuție a celulelor poate fi aleatoare, iar programatorul poate reveni și schimba celule anterioare fără a afecta starea codului per ansamblu sau fără a fi nevoit să ruleze întreg notebook-ul din nou. Evident celulele ce depind de celula modificată vor trebui rerulate pentru ca modificările să aibă efect.

Celulele de cod sunt incluse în același domeniu de vizibilitate sau *namespace*. Acest lucru înseamnă că definirea unei variabile, funcții sau clase sau importul unui modul într-o celulă va face ca aceastea să fie disponibile și în restul celulelor din notebook-ul curent. Trebuie să subliniem faptul că în acest caz celulele trebuie rulate secvențial, în sensul că, dacă dorim să utilizăm o anumită variabilă, funcție, clasă sau modul, celula ce conține definirea lor trebuie rulată înainte de rularea celulelor ce modifică sau utilizează aceste componente ale codului.

Între celulele de cod pot fi inserate celule de text cu note explicative asociate codului. Celulele text folosesc notația de tip Markdown, dar pot fi augmentate cu notație HTML sau LaTex.

Datorită flexibilității și ușurinței de utilizare, Google Colab a devenit principalul mediu de codare pentru limbajul Python, în special în aplicații de dezvoltare a algoritmilor de inteligență artificială și învățare automată.

T1.2. Primul cod Python

În continuare vom rula o primă celulă de cod în Colab. Pentru a rula codul de mai jos se selectează celula și se apăsa butonul de rulare ce apare în stânga acesteia. Colab introduce și un set de scurtături de tastatură ce pot fi vizualizate din meniul *Tools -> Command Palette*. De exemplu, pentru rularea unei celule selectate se poate utiliza CMD/CTRL+Enter, ALT/Option+Enter sau SHIFT+Enter. Ultima combinație de taste realizează și avansarea către următoarea celulă din notebook. Combinația cu ALT va introduce automat o nouă celulă după cea curentă.

[1]:
```
# Primul exemplu de cod
print('Salut, Python!')
```

[1]:
```
Salut, Python!
```

În codul de mai sus am apelat funcția `print()` către care am transmis șirul de caractere `Salut, Python!`. Se poate observa faptul că limbajul Python permite specificarea șirurilor de caractere folosind și ghilimele simple (apostrof), dar le fel de bine putem utiliza și ghilimele duble, rezultatul fiind același.

[2]:
```
print("Salut, Python!")
```

[2]:
```
Salut, Python!
```

Pentru a verifica faptul că într-adevăr cele două simboluri sunt interschimbabile, putem afișa tipul de date asociat acestor două versiuni de definire a șirului de caractere:

[3]:
```
# Funcția `type()` returnează tipul obiectului trimis ca␣
 ↪parametru
print(type('Salut, Python!'))
print(type("Salut, Python!"))
```

[3]:
```
<class 'str'>
<class 'str'>
```

Observăm că ambele șiruri de caractere au asociată clasa `str`. Tot din acest exemplu putem observa una dintre cele mai importante caracteristici ale Python și anume că toate datele sunt obiecte. De exemplu și valorile întregi sau reale vor fi asociate claselor `int` și respectiv `float`:

[4]:
```
print(type(3))
print(type(3.14))
```

[4]:
```
<class 'int'>
<class 'float'>
```

O facilitate utilă a mediului Colab și de altfel a mediului iPython este faptul că permite afișarea valorilor obiectelor definite anterior fără a utiliza funcția `print()` prin listarea numelor variabilelor pe care dorim să le afișăm. Este necesar ca această listare să fie ultima instrucțiune din celulă.

[5]:
```
a = 3
b = 3.14
a, b
```

[5]:
```
(3, 3.14)
```

Din exemplul anterior observăm că în limbajul Python nu este necesară specificarea anterioară a tipului variabilei, așa cum facem în limbajele C/C++ sau Java. Tipul variabilei este dedus automat din formatul valorii de inițializare a acesteia. De asemenea, acest lucru înseamnă că nu putem declara un obiect fără a-l inițializa, așa cum putem în C/C++ sau Java:

```
int a;
a = 3;
```

Vom reveni mai în detaliu asupra acestor aspecte într-un tutorial viitor.

Erori

Orice programator va face erori în codul scris, astfel încât este important să înțelegem tipurile de erori ce pot să apară și cum le putem rezolva. De exemplu, dacă ar fi să greșim numele funcției `print()`:

[6]:
```
# Generarea unei erori
orint("Salut, Python!")
```

[6]:
```
---------------------------------------
NameError
Traceback (most recent call last)
<ipython-input-6-e12f2049065b> in <module>
  1 # Generarea unei erori
----> 2 orint("Salut, Python!")

NameError: name 'orint' is not defined
```

Mesajul de eroare ne va informa asupra liniei din cadrul celulei la care această eroare apare precum și tipul acesteia, în cazul de față `NameError`, interpretorul nu recunoaște funcția `orint()` deoarece aceasta nu este în lista funcțiilor predefinite sau definite anterior de către programator.

Sau, am putea uita să închidem parantezele apelului funcției:

[7]:
```
print("Salut, Python"
```

[7]:
```
  File "<ipython-input-7-e52c962515f9>", line 1
print("Salut, Python"
                      ^
SyntaxError: unexpected EOF while parsing
```

În acest caz avem o eroare de sintaxă care ne spune că pe linia 1 instrucțiunea nu este terminată corect.

[8]:
```
# Eroare de sintaxă în cod multilinie
print("Salut, Python"
print("Salut, Python")
```

[8]:
```
  File "<ipython-input-8-86d9f6177e11>", line 3
print("Salut, Python")
    ^
SyntaxError: invalid syntax
```

În cazul în care avem mai multe linii de cod, mesajul de eroare ne va indica linia ulterioară celei în care există o problemă de sintaxă. Eroarea noastră este prezentă în linia 2, dar mesajul ne indică linia 3.

Lista completă de erori definite în Python poate fi găsită pe pagina oficială a documentației limbajului.

T1.3. Bibliografie/resurse utile

Python este unul dintre cele mai bine documentate limbaje de programare și în mod evident există extrem de multe resurse disponibile ce permit aprofundarea noțiunilor introduse de acesta. Mai jos enumerăm o listă de referințe și resurse pe care le considerăm cele mai utile, abordabile și extinse din lista completă a acestora:

- Learning Python, 5th Edition, Mark Lutz, O'Reilly, 2013
- Python Cookbook, 3rd Edition, David Beazley, Brian Jones, O'Reilly, 2013
- Python.org - https://www.python.org/ - site-ul oficial al limbajului
- Python 3 Module of the Week- https://pymotw.com/3/
- Cursuri online: Coursera, CodeAcademy, Udemy

Concluzii

În acest prim tutorial de Python au fost introduse noțiunile minimale de utilizare a mediului Google Colab și de rulare a secvențelor de cod prin intermediul acestuia, precum și un set de resurse bibliografice suplimentare recomandate a fi parcurse pentru o aprofundare mai bună a limbajului. În tutorialul următor vor fi prezentate pe scurt noțiunile de bază ale limbajului și mediul de programare asociat.

Exerciții

1. Afișați textul "Salut, Ana!". Modificați textul astfel încât să includă numele vostru.
2. Definiți o variabilă ce conține șirul de caractere "Salut, Ana!" și afișați mai apoi conținutul acestei variabile folosind funcția `print()` sau doar listarea ei.
3. Verificați cu ajutorul funcției `type()` că variabila definită în exercițiul 2 are asociată clasa `str`, așadar este o variabilă de tip *șir de caractere*.

4. Definiți două variabile de tip întreg inițializate cu valorile 3 și 4 și afișați produsul lor.
5. Creați o celulă text în care să explicați rezultatele exercițiilor anterioare.

T2 Generalități ale limbajului

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 subset 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.

Tip	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=3.1)`
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	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).

Înainte de a intra în mai multe detaii 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__ __getattribute__ __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.
```

Introspecția 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__',␣
↪'__getattribute__', '__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	Descriere
`(expresie),[expresie], {key: value...}, {expresie}`	Expresii în paranteze, de asociere, afișare liste, dicționare, seturi
`x[index], x[index:index], x(arguments...), x.attribute`	Indexare, partiționare, 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)`
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`
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`

Instrucțiune	Rol/Exemplu
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

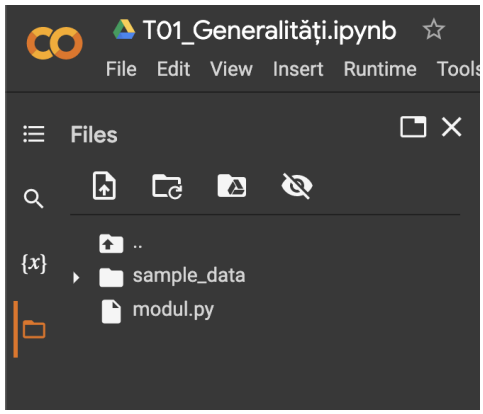
Scrierea codului Python în afara mediilor de programare interactive precum Google Colab se face în cadrul fișierelor ce au extensia `.py`. Un fișier ce conține cod Python este denumit și are rol de *modul*. Un modul va conține instanțieri de date și instrucțiuni.

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 modulului:

[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 modulului, mai există o serie de atribute predefinite disponibile pentru orice modul Pyhton.

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 specificam doar importul unui subset de atribute ale modulului:

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

În acest caz, atributele 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 atributele 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. *packets* 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). 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
```

[20]:
```
Package                       Version
----------------------------- ----------------------------
absl-py                       1.2.0
aiohttp                       3.8.1
aiosignal                     1.2.0
alabaster                     0.7.12
albumentations                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
# daca s-a adăugat calea specificata 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.

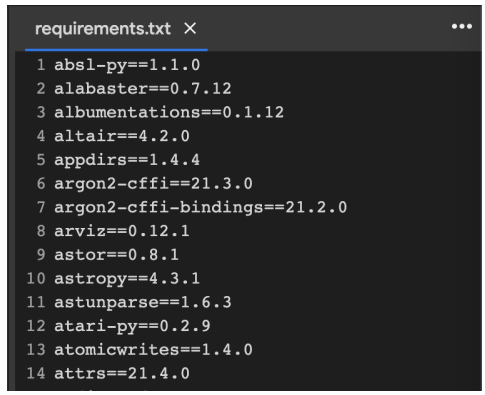
Utilizarea mediilor virtuale în Google Colab nu este necesară, deoarece fiecare notebook este un mediu de programare independent.

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:



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 acelui 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
    functia()
        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.functia)
```

[30]:
```
Help on function functia in module moduldoc:
functia()
    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.

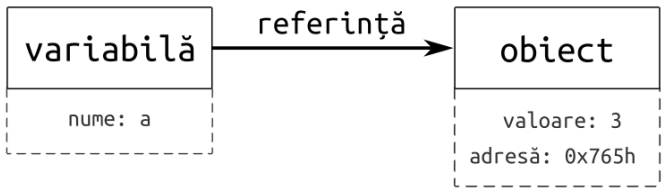
T3 Tipuri de date și operatori

T3.1. Variabile, obiecte, referințe

Una dintre particularitățile limbajului Python care poate crea confuzii la începutul utilizării limbajului se referă la faptul că toate datele în Python sunt **OBIECTE** și că, pentru tipurile de date fundamentale, modul de inițializare a variabilelor determină în mod automat tipul obiectului pe care îl referențiază. Acest concept este denumit tipizare dinamică (en. *dynamic typing*). Datorită acestui fapt, **tipurile de date sunt asociate obiectelor** și nu variabilelor, varibilele fiind doar referințe (pointeri) către spațiile de memorie unde sunt păstrate datele.

Este nevoie, astfel, să facem o diferență clară între variabile, obiecte și referințe:

- Variabilele sunt intrări în tabelul de sistem și au spații alocate pentru păstrarea legăturii (referinței) către obiecte.
- Obiectele sunt segmente de memorie alocată cu spațiu suficient pentru a stoca valorile lor;
- Referințele (pointerii) sunt legături între variabile și obiecte.



[1]:
```
# Referința variabilei și adresa obiectului
a = 3
print("Adresa referită de a: ", hex(id(a)))
print("Adresa obiectului 3: ", hex(id(3)))
```

[1]:
```
Adresa referită de a:  0xabc140
Adresa obiectului 3:  0xabc140
```

[2]:
```
# Modificarea valorii varibilei modifică referința
a = 4
```

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

[2]:
```
Adresa referită de a:  0xabc160
Adresa obiectului 3:  0xabc140
Adresa obiectului 4:  0xabc160
```

[3]:
```
# Ștergerea referinței
a = None
print("Adresa referită de a: ", hex(id(a)))
print("Adresa obiectului 3: ", hex(id(3)))
print("Adresa obiectului 4: ", hex(id(4)))
```

[3]:
```
Adresa referită de a:  0xa9c260
Adresa obiectului 3:  0xabc140
Adresa obiectului 4:  0xabc160
```

Odată cu ștergerea referinței lui a către un anumit obiect, observăm că se va face o referire la o adresă oarecare din memorie (în acest caz 0xa9c260).

Variabile

În ceea ce privește variabilele în Python, avem următoarele caracteristici:

- Variabilele sunt create atunci când li se atribuie valori prima dată;
- Variabilele sunt înlocuite cu valorile lor în expresii;
- Variabilele trebuie să refere un obiect înainte de a fi utilizate în expresii;
- Variabilele referă obiecte și nu sunt declarate înainte de utilizare (așa cum se poate face în C/C++ sau Java).

Referințe comune

Deoarece obiectele sunt cele care au alocată memorie, iar variabilele reține doar referințe către aceste zone de memorie, în cazul în care mai multe variabile au aceeași valoare, vor indica aceeași locație de memorie. Acest mecanism permite utilizarea extrem de eficientă a memoriei.

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

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

[5]:
```
# Copiem referința făcută de b
c = b
print("Adresa referită de c: ", hex(id(c)))
```

[5]:
```
Adresa referită de c:  0xabc140
```

[6]:
```
# Modificăm referința făcută de b
b = 4
print("Adresa referită de b: ", hex(id(b)))
print("Adresa referită de c: ", hex(id(c)))
```

[6]:
```
Adresa referită de b:  0xabc160
Adresa referită de c:  0xabc140
```

Eliberarea memoriei (en. *garbage collection*)

Mecanismul de eliberare a memoriei în Python este unul automat în sensul că obiectele nefolosite sunt automat de-alocate. Această dealocare se face prin numărarea referințelor ce pointează la un anumit obiect din memorie (en. *reference counting*). Dacă acest număr ajunge la 0, spațiul de memorie este eliberat.

Este important de remarcat faptul că numărarea referințelor se face relativ la tot codul ce rulează momentan în mediul Python (inclusiv biblioteca standard și module terțe), astfel încât, de exemplu, pentru valoarea 1, vom avea un număr mare de referințe:

[7]:
```
import sys
print("Numărul de referințe către obiectul 1:", sys.
 ↪getrefcount(1))
```

[7]:
```
Numărul de referințe către obiectul 1: 7025
```

Dar pentru valori mai puțin întâlnite vom avea cel puțin 3 referințe, una dintre acestea fiind legată și de variabila temporară creată pentru apelul metodei `getrefcount()`

[8]:
```
print("Numărul de referințe către obiectul 123456789:",sys.
 ↪getrefcount(123456789))
```

```
[8]: Numărul de referințe către obiectul 123456789: 3
```

Totodată este important de reținut faptul că ștergerea unei variabile nu implică și ștergerea obiectului pointat în memorie dacă acesta este referit și de alte variabile:

```
[9]: a = 987654321
     b = a
     c = a
     d = a
     print ("Nr de ref către obiectul referit de b:", sys.
      ↪getrefcount(b))
     del a
     print ("Nr de ref către obiectul referit de b după ștergerea␣
      ↪a:", sys.getrefcount(b))
     del c
     print ("Nr de ref către obiectul referit de b după ștergerea␣
      ↪c:", sys.getrefcount(b))
```

```
[9]: Nr de ref către obiectul referit de b: 6
     Nr de ref către obiectul referit de b după ștergerea a: 5
     Nr de ref către obiectul referit de b după ștergerea c: 4
```

Numărul de referințe se modifică și atunci când o variabilă referă alt obiect:

```
[10]: a = 987654321
      b = a
      c = a
      d = a
      print ("Nr de ref către obiectul referit de b:", sys.
       ↪getrefcount(b))
      a = 3
      print ("Nr de ref către obiectul referit de b după␣
       ↪modificarea a:", sys.getrefcount(b))
      c = 4
      print ("Nr de ref către obiectul referit de b după␣
       ↪modificarea c:", sys.getrefcount(b))
```

```
[10]: Nr de ref către obiectul referit de b: 6
      Nr de ref către obiectul referit de b după modificarea a: 5
      Nr de ref către obiectul referit de b după modificarea c: 4
```

Tipuri de date fundamentale (core/built-in)

În tutorialul anterior am introdus pe scurt tipurile de date fundamentale din limbajul Python:

Tip	Exemplu
Număr	`1234, 3.1415, 3+4j, 0b111, Decimal(), Fraction()`
String	`'Ana', "Maria", b'a\01c', u'An\xc4'`
Listă	`[1, [2, 'trei'], 4.5], list(range(10))`
Dicționar	`{'cheie':'valoare', 'key':'value'}, dict(cheie=3.14)`
Set	`set('abc'), {'a', 'b', 'c'}`
Tuplu	`(1, 'Ana', 'c', 3.14), tuple('Ana'), namedtuple`
Fișier	`open('fisier.txt'), open(r'C:\fisier.bin', 'wb')`
Alte tipuri	Boolean, bytes, bytearray, None

Iar în continuare vom prezenta atributele și metodele asociate acestor tipuri de date.

T3.2. Tipuri de date numerice

Una dintre cele mai des întâlnite aplicații ale limbajului Python se referă la analiza numerică sau lucrul cu structuri de date numerice de dimensiuni mari. Astfel că universul numeric din Python este extrem de extins și poate fi extins și mai mult prin utilizarea pachetului NumPy.

Cele mai importante tipuri de date numerice și funcționalități asociate ale acestora sunt:

- obiecte întregi și reale cu virgulă flotantă;
- obiecte numerice complexe;
- obiecte cu precizie fixă (zecimale);
- obiecte numere raționale (fracții);
- colecții cu operații numerice (seturi);
- valori de adevăr (booleeni): `True, False`;
- metode și module predefinite: `round(), math, random`, etc.
- expresii; precizie întreagă nelimitată; operații la nivel de bit; format hexa, octal și binar;
- extensii terțe: vectori, vizualizare, afișare, etc.

Date numerice și definirea lor

Reprezentare	Interpretare
1234, -24, 0, 99999999999999	Întreg (dimensiune nelimitată)
1.23, 1., 3.14e-10, 4E210, 4.0e+210	Numere în virgulă flotantă
0o177, 0x9ff, 0b101010	Octal, hexa, binar
3+4j, 3.0+4.0j, 3J	Numere complexe
set('spam'), {1, 2, 3, 4}	Constructori set
Decimal('1.0'), Fraction(1, 3)	Extensii de tip
bool(X), True, False	Tipul boolean și constante

[11]:
```
# Inițializarea pentru diferite tipuri de date numerice
a = 3.14
print("Tipul de date referit de a:",type(a))
b = 0x7
print("Tipul de date referit de b:",type(b))
c = 3+4j
print("Tipul de date referit de c:",type(c))
d = True
print("Tipul de date referit de d:",type(d))
```

[11]:
```
Tipul de date referit de a: <class 'float'>
Tipul de date referit de b: <class 'int'>
Tipul de date referit de c: <class 'complex'>
Tipul de date referit de d: <class 'bool'>
```

T3.2.1 Operatori numerici și precedență

Datele numerice sunt combinate în programe prin intermediul operatorilor în expresii ce pot deveni extrem de complexe. Astfel că, este important să se cunoască ordinea de execuție a operatorilor sau precedența acestora. Tabelul de mai jos prezintă această ordine de execuție, însă este expus în ordine inversă a precedenței (ultimele rânduri au precedența maximă).

Operator	Descriere
`yield x`	Funcție generator
`lambda args: expression`	Funcție lambda
`x if y else z`	Operator ternar
`x or y`	Sau logic
`x and y`	Și logic
`not x`	Negare logică
`x in y, x not in y`	Apartenență (iterabili, seturi)
`x is y, x is not y`	Identitatea obiectelor
`x < y, x <= y, x > y, x >= y`	Operatori relaționali, subset set și superset
`x == y, x != y`	Egalitate numerică

Operator	Descriere
`x | y`	Sau pe biți, reuniune set
`x ^ y`	XOR pe biți, diferență simetrică set
`x & y`	Și pe biți, intersecție seturi
`x << y, x >> y`	Deplasare stânga-dreapta pe biți
`x + y`	Adunare, concatenare
`x - y`	Scădere, diferență seturi
`x * y`	Înmulțire, repetare seturi
`x % y`	Restul împărțirii, formatare
`x / y, x // y`	Împărțire, împărțire întreagă
`-x, +x`	Negare, identitate
`~x`	Negare pe biți
`x ** y`	Ridicare la putere
`x[i]`	Indexare
`x[i:j:k]`	Partiționare
`x(...)`	Apel funcții, metode, clase
`x.attr`	Referire atribut
`(...)`	Tuplu, expresie, expresie generator
`[...]`	Listă, list comprehension
`{...}`	Dicționar, set, dictionary and set comprehension

- Parantezele pot modifica ordinea de execuție a operațiilor;
- Operatorii aplicați asupra tipurile de date mixte determină conversia implicită la tipul de date mai complex;
- Se poate forța obținerea unui anumit rezultat folosind conversie explicită (ex. `int(43.5)`);
- Conversia implicită funcționează doar pentru tipurile numerice;
- Există posibilitatea supraîncărcării operatorilor și utilizarea polimorfismului.

[12]:
```
# Operatori aritmetici
a = 7.3
b = 3
print("Negare:", -a)
print("Sumă:", a+b)
print("Diferență:", a-b)
print("Înmulțire:", a*b)
print("Împărțire:", a/b)
print("Împărțire exactă:", a//b)
print("Modulo:", a%b)
print("Ridicare la putere:", a**b)
print("Deplasare la stânga:", b<<2) # înmulțire cu 2**2
print("Deplasare la dreapta:", b>>2) # modulor cu 2**2
```

[12]:
```
Negare: -7.3
Sumă: 10.3
Diferență: 4.3
Înmulțire: 21.9
Împărțire: 2.433333333333333
Împărțire exactă: 2.0
Modulo: 1.2999999999999998
Ridicare la putere: 389.017
Deplasare la stânga: 12
Deplasare la dreapta: 0
```

Atenție la precizia de reprezentare a valorilor reale:

[13]:
```
1.1 + 2.2 == 3.3
```

[13]:
```
False
```

[14]:
```
# Operatori relaționali
a = 2.3
b = 3
print("Mai mic", a < b)
print("Mai mare", a > b)
print("Egalitate", a == b)
print("Inegalitate", a != b)
```

[14]:
```
Mai mic True
Mai mare False
```

```
Egalitate False
Inegalitate True
```

[15]:
```
# Operatori logici
a = 1
b = 0
print("Negare", not a)
print("Sau", a or b)
print("Și", a and b)
```

[15]:
```
Negare False
Sau 1
Și 0
```

[16]:
```
# Orice valoare diferită de zero e considerată adevărată
not -7, not 0
```

[16]:
```
(False, True)
```

[17]:
```
# Operatori pe biți
a = 3 # 011
b = 5 # 101
print("Negare", ~a) # 100
print("Sau pe biți", a|b) # 111
print("Și pe biți", a&b) # 001
print("XOR pe biți", a^b) # 110
```

[17]:
```
Negare -4
Sau pe biți 7
Și pe biți 1
XOR pe biți 6
```

T3.2.2 Comparații înlănțuite (chained)

Comparațiile înlănțuite se referă la utilizarea secvențială, în aceeași expresie a doi sau mai operatori relaționali sau de apartenență din lista:

```
      ">" | "<" | "==" | ">=" | "<=" | "!=" | "is" ["not"] |
["not"] "in"
```

de exemplu:

```
>> X < Y < Z
True
```

Acestea ar fi echivalente cu verificarea secvențială în cadrul unei instrucțiuni `if`:

```
if X < Y and Y < Z:
```

Conform tabelului de mai sus privind precedența operatorilor, toți operatorii relaționali au aceeași prioritate, astfel încât se vor executa secvențial:

[18]:
```
1 < 2 < 3.0 < 4
```

[18]:
```
True
```

[19]:
```
1 > 2 > 3.0 > 4
```

[19]:
```
False
```

Avantajul utilizării comparațiilor înlănțuite se referă la faptul că dacă oricare dintre comparații returnează o valoare de adevăr `False` restul operațiilor nu mai sunt evaluate. De asemenea, nu implică nicio relație între operatorii distanțați. De exemplu:

```
a < b > c
```

nu va spune nimic despre legătura dintre `a` și `c`.

Se folosesc doar operatori relaționali sau de apartenență. Alți operatori ar putea returna rezultate ciudate:

[20]:
```
1 == 2 < 3 # Echivalent cu : 1 == 2 and 2 < 3
# Dar nu echivalent cu: False < 3 (ce presupune 0 < 3, ceea␣
 ↪ce e adevărat)
```

[20]:
```
False
```

T3.2.3 Împărțire clasică și întreagă

La fel ca în limbajul C/C++, în versiunile Python 2.x, operatorul de împărțire (/) utilizat între doi operanzi întregi returnează câtul împărțirii

întregi (partea întreagă a câtului). Iar pentru cel puțin un operand de tip float, va returna câtul real al împărțirii.

În versiunile Python 3.x, operatorul de împărțire va returna întotdeauna rezultatul real al împărțirii:

[21]:
```
(5 / 2), (5 / 2.0), (5 / -2.0), (5 / -2)
```

[21]:
```
(2.5, 2.5, -2.5, -2.5)
```

Pentru a obține doar câtul împărțirii întregi, se utilizează operatorul //:

[22]:
```
(5 // 2), (5 // 2.0), (5 // -2.0), (5 // -2)
```

[22]:
```
(2, 2.0, -3.0, -3)
```

T3.2.4 Alte tipuri de date numerice

Decimal()

Permite lucrul cu valori zecimale cu precizie fixă:

[23]:
```
from decimal import Decimal
# Crearea unor obiecte de tip Decimal din șiruri de caractere
Decimal('0.1') + Decimal('0.3')
```

[23]:
```
Decimal('0.4')
```

[24]:
```
# Au număr fix de zecimale
0.1 + 0.1 + 0.1 - 0.3
```

[24]:
```
5.551115123125783e-17
```

[25]:
```
# Au număr fix de zecimale
# Implicit 28 de zecimale
Decimal(1) / Decimal(7)
```

[25]:
```
Decimal('0.14285714285714285714285714291')
```

[26]:
```
# Spre deosebire de float
0.2 + 0.4 - 0.6
```

[26]:
```
1.1102230246251565e-16
```

[27]:
```
# Se poate stabili numărul de zecimale
import decimal
decimal.getcontext().prec = 4
decimal.Decimal(1) / decimal.Decimal(7)
```

[27]:
```
Decimal('0.1429')
```

Fraction()

Permite lucrul cu reprezentări fracționare:

[28]:
```
from fractions import Fraction
a = Fraction(1, 2)
b = Fraction(4, 6)
a, b
```

[28]:
```
(Fraction(1, 2), Fraction(2, 3))
```

[29]:
```
# Afișarea cu print() se face sub formă matematică
print(a, b)
```

[29]:
```
1/2 2/3
```

[30]:
```
a + b
```

[30]:
```
Fraction(7, 6)
```

[31]:
```
# Rezultatele sunt exacte
a - b
```

[31]:
```
Fraction(-1, 6)
```

[32]:
```
# Conversia în float
float(a)
```

[32]:
```
0.5
```

[33]:
```
# Conversia din float
Fraction.from_float(1.5)
```

[33]:
```
Fraction(3, 2)
```

Boolean()

Permit lucrul cu valori de adevăr, `True/False` echivalent cu `0/1` și sunt o subclasă a tipului `int`

[34]:
```
type(True)
```

[34]:
```
bool
```

[35]:
```
isinstance(True, int)
```

[35]:
```
True
```

[36]:
```
# Aceeași valoare
True == 1
```

[36]:
```
True
```

[37]:
```
# 1 sau 0
True or False
```

[37]:
```
True
```

T3.2.5 Pachetele utile pentru lucrul cu date numerice

Pachetul math implementează un număr foarte mare de funcții și constante matematice de bază:

[38]:
```
import math
math.pi, math.e
```

[38]:
```
(3.141592653589793, 2.718281828459045)
```

[39]:
```
math.sin(2 * math.pi / 180)
```

[39]:
```
0.03489949670250097
```

[40]:
```
math.sqrt(144), math.sqrt(2)
```

[40]:
```
(12.0, 1.4142135623730951)
```

Pachetul random este util pentru generarea de numere pseudo-aleatoare:

[41]:
```
import random
#generarea unui număr aleator între 0 și 1
random.random()
```

[41]:
```
0.7932704915843354
```

[42]:
```
#generarea unui număr aleator între 1 și 10
random.randint(1, 10)
```

[42]:
```
10
```

[43]:
```
#selecție aleatoare dintr-o listă
random.choice(['Mere', 'Pere', 'Banane'])
```

[43]:
```
'Pere'
```

[44]:
```
# aleatorizarea unei liste
suite = ['inimă roșie', 'treflă', 'romb', 'inimă neagră']
random.shuffle(suite)
suite
```

[44]:
```
['inimă roșie', 'treflă', 'inimă neagră', 'romb']
```

Pachetul NumPy implementează o serie largă de structuri numerice multi-dimensionale și metode asociate acestora:

[45]:
```
import numpy as np
#vector
np.array([1, 2, 3, 4, 5, 6])
```

[45]:
```
array([1, 2, 3, 4, 5, 6])
```

[46]:
```
# matrice
np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
```

[46]:
```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

[47]:
```
# generarea unui vector cu valorile cuprinse între 2 și 9
# incrementate din 2 în 2
```

```
np.arange(2, 9, 2)
```

[47]:
```
array([2, 4, 6, 8])
```

[48]:
```
# calcul eroare medie pătratică
predictions = np.array([1.2, 2.3, 3.4])
targets = np.array([1, 2, 3])
error = (1/targets.shape[0])*np.sum(np.
 ↪square(predictions-targets))
error
```

[48]:
```
0.0966666666666666
```

T3.3. Șiruri de caractere (String)

Șirurile de caractere (String) sunt obiecte ce conțin date de tip text sau octeți (bytes). Sunt obiecte **IMUTABILE** și au asociate o serie largă de funcții predefinite. Șirurile de caractere fac parte din clasa mai mare de obiecte de top **secvențe** (en. *sequences*).

Operație	Interpretare
`S = ''`	String gol
`S = "ana"`	Definire string cu ghilimele
`S = "a\tn\ta\n"`	Secvențe escape
`S = """..."""`	String multilinie
`S1+S2`	Concatenare
`S1*2`	Repetare string
`S[i]`	Indexare string
`S[i:j]`	Partiționare
`len(S)`	Lungime string
`S.find(ss)`	Căutare substring
`S.replace(ss1, ss2)`	Înlocuire substring
`S.split(delim)`	Împărțire după delimitator
`S.lower()`	Conversie litere minuscule
`S.upper()`	Conversie litere majuscule

[49]:
```
# Definirea unui string
S = 'abc'
S
```

[49]:
```
'abc'
```

[50]:
```
# Rezultat similar
S = "abc"
S
```

[50]:
```
'abc'
```

[51]:
```
# Numărul de caractere
len('abc')
```

[51]:
```
3
```

[52]:
```
# Concatenare string-uri
'abc' + 'def'
```

[52]:
```
'abcdef'
```

[53]:
```
# Repetare, echivalent cu 'ha'+'ha'+...
'ha' * 4
```

[53]:
```
'hahahaha'
```

[54]:
```
# Verificare apartenență
S = "Ana"
"a" in S
```

[54]:
```
True
```

[55]:
```
"N" in S # Nu există, case-sensitive
```

[55]:
```
False
```

[56]:
```
# Verificare substring
'Ana' in 'Ana are mere.'
```

[56]:
```
True
```

T3.3.1 Indexarea și partiționarea șirurilor de caractere

Tipurile de date de tip secvență în Python permit indexarea avansată de forma:

```
S[i:j:k]
```

unde: - i e indexul de început (inclusiv) - j e indexul de final (exclusiv) - k e pasul de incrementare, poate fi negativ

[57]:
```
S = 'Ana are mere.'
```

[58]:
```
# Indexare
S[0], S[-2]
```

[58]:
```
('A', 'e')
```

[59]:
```
# Toate elementele
S[:]
```

[59]:
```
'Ana are mere.'
```

[60]:
```
# Elementele pornind de la indexul 2
S[2:]
```

[60]:
```
'a are mere.'
```

[61]:
```
# Elementele pornind de la indexul 2 până la indexul 5␣
↪(exclusiv)
S[2:5]
```

[61]:
```
'a a'
```

[62]:
```
# Elementele până la penultimul index (exclusiv)
S[:-2]
```

[62]:
```
'Ana are mer'
```

[63]:
```
# Tot al doilea caracter
S[::2]
```

[63]:
```
'Aaaemr.'
```

[64]:
```
# Pornind de la indexul 1, tot al doilea caracter
S[1::2]
```

[64]:
```
'n r ee'
```

[65]:
```
# Inversare
S[::-1]
```

[65]:
```
'.erem era anA'
```

[66]:
```
# Indexare inversă
S = '123456789'
S[5:1:-1]
```

[66]:
```
'6543'
```

[67]:
```
# Alternativ putem folosi funcția slice() pentru crearea␣
↪indecșilor
S[slice(1, 3)]
```

[67]:
```
'23'
```

[68]:
```
S[slice(None, None, -1)]
```

[68]:
```
'987654321'
```

T3.3.2 Modificarea stringurilor

La fel ca în alte limbaje de programare în care stringurile sunt tipuri de date individuale și nu doar tablouri de caractere, în Python nu este posibilă modificarea conținutului unui element din string in-place, adică în locația curentă de memorie:

[69]:
```
S = 'Ana'
```

[70]:
```
# Eroare!
S[0] = 'x'
```

[70]:
```
---------------------------------------------
TypeError    Traceback (most recent call last)
    <ipython-input-70-96394f2a45ed> in <module>
```

```
----> 1 S[0] = 'x' # Eroare!
TypeError: 'str' object does not support item assignment
```

Pentru a modifica un obiect de tip string, va trebui să creăm unul nou. Cu alte cuvinte, variabila S va indica o altă zonă de memorie ce conține noul string creat:

[71]:
```
S = "Ana"
print("Adresa referită de S:", hex(id(S)))
S = S + ' are'
print("Adresa referită de S după modificare:", hex(id(S)))
S
```

```
Adresa referită de S: 0x7f746bd14e70
Adresa referită de S după modificare: 0x7f746bc42570
```

[71]:
```
'Ana are'
```

[72]:
```
S = S[:3] + ' nu ' + S[-3:]
S # Se crează un nou obiect care e atribuit variabilei S
```

[72]:
```
'Ana nu are'
```

T3.3.3 Metode ale stringurilor

Stringurile au asociate o multitudine de metode, iar o listă completă poate fi regăsită în documentația oficială. În continuare vom parcurge pe scurt unele dintre cele mai des utilizate metode ale stringurilor.

[73]:
```
S = 'Ana și Mana'
```

[74]:
```
# Înlocuim aparițiile 'na' cu 're'
S = S.replace('na', 're')
S
```

[74]:
```
'Are și Mare'
```

[75]:
```
# Indexul primei apariții a substringului
S = 'Ana are mere'
S.find('are')
```

[75]: 4

[76]:
```
# Dacă nu apare substringul se returnează -1
S.find("MA")
```

[76]: -1

[77]:
```
# Împărțirea stringului după spații albe
S = 'Ana are mere'
S.split()
```

[77]: ['Ana', 'are', 'mere']

[78]:
```
# Împărțire după caractere specifice
S = 'Ana|+are|+mere'
S.split('|+')
```

[78]: ['Ana', 'are', 'mere']

[79]:
```
# Eliminare spații albe de la începutul și finalul stringului
S = "   Ana are mere!\n\t"
S.strip()
```

[79]: 'Ana are mere!'

[80]:
```
# Eliminare spații albe doar de la finalul stringului
S = "   Ana are mere!\n\t"
S.rstrip()
```

[80]: '   Ana are mere!'

[81]:
```
# Eliminare spații albe doar de la începutul stringului
S = "   Ana are mere!\n\t"
S.lstrip()
```

[81]: 'Ana are mere!\n\t'

[82]:
```
# Capitalizare
S = 'Ana are mere'
S.upper()
```

[82]:
```
'ANA ARE MERE'
```

[83]:
```
# Litere minuscule
S.lower()
```

[83]:
```
'ana are mere'
```

[84]:
```
# Verificare dacă toate elementele sunt caractere (cu␣
 ↪excepția spațiilor goale)
S = 'Ana'
S.isalpha()
```

[84]:
```
True
```

[85]:
```
S = 'Ana are'
S.isalpha()
```

[85]:
```
False
```

[86]:
```
# Verificare dacă toate elementele sunt caractere sau cifre␣
 ↪(cu excepția spațiilor goale)
S = 'Ana12'
S.isalnum()
```

[86]:
```
True
```

[87]:
```
# Intercalare caractere specifice între elementele unei␣
 ↪secvențe
S = 'bc'
S.join("aaa")
```

[87]:
```
'abcabca'
```

[88]:
```
'-'.join(['Ana', 'are', 'mere'])
```

[88]:
```
'Ana-are-mere'
```

T3.3.4 Expresii de formatare stringuri

Atunci când dorim să creăm un string mai complex pe baza altor obiecte (de cele mai multe ori pentru afișare sau scriere în fișiere) putem utiliza

expresiile de formatare.

În cadrul acestor expresii se folosesc caractere speciale ce indică tipul de date cu care acestea vor fi înlocuite în crearea stringului final, după cum urmează:

Caracter special	Tip de dată
s	String sau reprezentarea string `str()` a unui obiect
r	La fel ca s dar folosește `repr()`
c	Caracter (int sau str)
d	Zecimal (baza 10)
i	Întreg
o	Octal (baza 8)
x	Hexa (baza 16)
e	Float cu exponent
E	Float cu exponent capitalizat
f	Float zecimal
F	Float zecimal capitalizat
g	e sau f
G	E sau F
%	Literalul %

[89]:
```
'%s are %i mere' % ('Ana', 3)
```

[89]:
```
'Ana are 3 mere'
```

[90]:
```
'%e e o altă reprezentare pentru %f' %(3.14, 3.14)
```

[90]:
```
'3.140000e+00 e o altă reprezentare pentru 3.140000'
```

Putem adăuga specificații suplimentare de afișare și formatare numerică:

[91]:
```
x = 1.23456789
```

```
# Afișare pe 6 spații (completare cu spațiu gol), aliniere la stânga
# și precizie de 2 zecimale
print('%-7.2f|' %x)

# Afișare pe 5 spații (completare cu zerouri) și precizie de 2 zecimale
print('%07.2f|' %x)

# Afișare pe 6 spații cu aliniere la dreapta cu afișarea semnului
# și precizie de 2 zecimale
print('%+7.2f|' %x)
```

[91]:
```
1.23   |
0001.23|
  +1.23|
```

[92]:
```
# Afișare pe 20 de spații cu aliniere la dreapta
'%20s' %'Ana'
```

[92]:
```
'                 Ana'
```

[93]:
```
# Afișare pe 20 de spații cu aliniere la stânga
'%-20s are' %'Ana'
```

[93]:
```
'Ana                  are'
```

Metoda format()

O alternativă de formatare a stringurilor este metoda `format()`. Pentru această metodă se folosește un string șablon ce conține câmpuri de înlocuire marcate cu acolade {}. Câmpurile de înlocuire pot fi indexate prin poziție, cheie sau o combinație a acestora:

[94]:
```
# Indexare prin poziție
sablon = '{0} {1} 3 {2}'
sablon.format('Ana', 'are', 'mere')
```

[94]:
```
'Ana are 3 mere'
```

[95]:
```
# Indexare prin cheie
sablon = '{cine} are {cate} mere'
sablon.format(cine='Ana', cate='3')
```

[95]: 'Ana are 3 mere'

[96]:
```
# Indexare prin poziție și cheie
sablon = '{cine} {0} 3 {ce}'
sablon.format('are', cine='Ana', ce='mere')
```

[96]: 'Ana are 3 mere'

[97]:
```
# Indexare prin poziție relativă
sablon = '{} {} 3 {}'
sablon.format('Ana', 'are', 'mere')
```

[97]: 'Ana are 3 mere'

T3.4. Liste

Un alt tip de obiecte de tip secvență în Python sunt **listele**. Caracteristicile acestora pot fi sumarizate astfel:

- Colecții ordonate de obiecte;
- Accesate prin index/offset;
- Lungime variabilă;
- Eterogene;
- Pot fi imbricate arbitrar;
- **Mutabile**;
- Echivalent cu tablouri de referințe la obiecte.

Iar operațiile pe care le putem efectua asupra acestora sunt:

Operație	Interpretare
`L = []`	listă goală
`L = ['123', 'abc', 1.23, {}]`	patru elemente
`L = ['123', ['a', 'b']]`	liste imbricate (nested)
`L = list('ana')`	listă din elementele unui iterabil
`L = list(range(0,10)`	listă de întregi succesivi
`L[i]`	indexare
`L[i][j]`	indexare dublă
`L[i:j]`	partiționare
`len(L)`	lungimea listei
`L1 + L2`	concatenare liste
`L * 3`	repetare listă
`for x in L: print (x)`	iterare
`3 in L`	apartenență

Operație	Interpretare
`L.append(elem)`	adăugare element la final
`L.extend([elem1, elem2])`	extindere cu elemente multiple
`L.insert(i, elem)`	inserare la poziția i
`L.index(elem)`	indexul elementului
`L.count(elem)`	numărarea elementelor
`L.sort()`	sortare listă
`L.reverse()`	inversare listă
`L.copy()`	copierea elementelor
`L.clear()`	ștergerea elementelor listei
`L.pop(i)`	eliminarea elementului de pe poziția i
`L.remove(elem)`	eliminarea elementului din listă
`del L[i]`	ștergerea elementului de pe poziția i
`del L[i:j]`	ștergerea elementelor de pe pozițiile i până la j-1

[98]:
```
# Creare listă
L = [1,2,3]
```

[99]:
```
# Lungimea listei
len(L)
```

[99]: 3

[100]:
```
# Concatenare
L + [4, 5, 6]
```

[100]: [1, 2, 3, 4, 5, 6]

[101]:
```
# Repetiție
['ha'] * 4
```

[101]: ['ha', 'ha', 'ha', 'ha']

[102]:
```
# Verificare apartenență
3 in L
```

[102]: True

[103]:
```
# Iterare
for x in L:
  print (x, end=' ')
```

[103]: 1 2 3

[104]:
```
# Indexare
L = ['Ana', 'are', 'mere']
L[2]
```

[104]: 'mere'

[105]:
```
L[-2]
```

[105]: 'are'

[106]:
```
# Partiționare
L[1:]
```

[106]: ['are', 'mere']

[107]:
```
# Listă imbricată (poate fi considerată matrice)
matrice = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

[108]:
```
# Elementul de pe poziția 1 (linia din matrice)
matrice[1]
```

[108]: [4, 5, 6]

[109]:
```
# Elementul de pe poziția 1 din elementul de pe poziția 1
matrice[1][1]
```

[109]: 5

[110]:
```
# Elementul de pe poziția 0 din elementul de pe poziția 2
matrice[2][0]
```

[110]: 7

Comprehensiunea listelor

List comprehension se referă la crearea unei liste noi prin aplicarea unor operații asupra elementelor unei alte liste. Se scrie de obicei într-o linie și poate fi extrem de utilă în crearea de noi obiecte. Folosesc instrucțiunile `for` și `if` descrise în tutorialul următor.

[111]:
```
# Înmulțim fiecare caracter din 'ANA' de 4 ori
lista = [c * 4 for c in 'ANA']
lista
```

[111]: ['AAAA', 'NNNN', 'AAAA']

[112]:
```
# Noua listă conține doar elementele mai mari decât 0␣
 ↪ridicate la pătrat
lista = [-2, -1, 0, 1, 2, 3]
lista_noua = [n**2 for n in lista if n>0]
lista_noua
```

[112]: [1, 4, 9]

T3.4.1 Modificare liste in-place

Deoarece listele sunt obiecte mutabile, acestea pot fi modificate:

[113]:
```
L = ['Ana', 'are', 'mere']
print("Adresa referită de L:", hex(id(L)))
# Modificarea unui element din listă
L[0] = 'Maria'
print("Adresa referită de L după modificare:", hex(id(L)))
L
```

```
Adresa referită de L: 0x7f746bc3b870
Adresa referită de L după modificare: 0x7f746bc3b870
```

[113]: ['Maria', 'are', 'mere']

[114]:
```
# Modificarea mai multor elemente/inserare
L[2:] = ['mere','și', 'pere']
L
```

[114]: ['Maria', 'are', 'mere', 'și', 'pere']

[115]:
```
# Inserare fără înlocuire
L[1:1] = ["și",  "Ana"]
L
```

[115]: ['Maria', 'și', 'Ana', 'are', 'mere', 'și', 'pere']

[116]:
```
# Ștergere prin atribuire
L[1:3] = []
L
```

[116]: ['Maria', 'are', 'mere', 'și', 'pere']

T3.4.2 Ordonarea listelor

Ordonarea listelor se face in-place, adică se modifică lista inițială:

[117]:
```
L = ['abc', 'ABD', 'aBe']
L.sort()
L
```

[117]: ['ABD', 'aBe', 'abc']

[118]:
```
L = ['abc', 'ABD', 'aBe']
# Sortare după elementele capitalizate
L.sort(key=str.upper)
L
```

[118]: ['abc', 'ABD', 'aBe']

[119]:
```
L = ['abc', 'ABD', 'aBe']
# Sortare după elementele capitalizate și inversarea sortării
L.sort(key=str.lower, reverse=True)
L
```

[119]: ['aBe', 'ABD', 'abc']

Alte metode ale listelor

[120]:
```
L = [1, 2]
# Extinderea listei
L.extend([3, 4, 5])
L
```

[120]:
```
[1, 2, 3, 4, 5]
```

[121]:
```
# Ștergerea și returnarea ultimului element din listă
elem = L.pop()
elem, L
```

[121]:
```
(5, [1, 2, 3, 4])
```

[122]:
```
# Ștergerea și returnarea unui element de pe o anumită␣
 ↪poziție
elem = L.pop(1)
elem, L
```

[122]:
```
(2, [1, 3, 4])
```

[123]:
```
# Inversarea listei in-place
L.reverse()
L
```

[123]:
```
[4, 3, 1]
```

[124]:
```
# Indexul unui element
L = ['Ana', 'are', 'mere']
L.index('are')
```

[124]:
```
1
```

[125]:
```
# Inserarea la o anumită poziție
L.insert(1, 'nu')
L
```

[125]:
```
['Ana', 'nu', 'are', 'mere']
```

[126]:
```
# Ștergerea după valoare
L.remove('nu')
L
```

[126]:
```
['Ana', 'are', 'mere']
```

[127]:
```
L = [1,2,3,4,1,2,3]
# Numarul de apariții a unui element
L.count(1)
```

[127]:
```
2
```

[128]:
```
# Ștergerea unui element din listă
L = ['Ana', 'are', 'mere']
del L[0]
L
```

[128]:
```
['are', 'mere']
```

[129]:
```
# Ștergerea unei partiții din listă
del L[1:]
L
```

[129]:
```
['are']
```

T3.5. Seturi

Tipul de date *set* în Python sunt echivalentul seturilor din matematică. Astfel că, un set va implementa o colecție neordonată de obiecte unice. Sunt permise operații asociate din matematică: reuniune, intersecție, diferență, etc.

Seturile în Python sunt mutabile, însă obiectele conținute trebuie neapărat să fie imutabile. Un set poate conține obiecte de tip diferit (eterogene).

[130]:
```
# definirea unui set pe baza unei liste
set([1, 2, 3, 4, 3])
```

[130]:
```
{1, 2, 3, 4}
```

[131]:
```
# definirea unui set pe baza unui șir de caractere
S = set('salut')
S
```

[131]:
```
{'a', 'l', 's', 't', 'u'}
```

Se poate observa faptul că ordinea de stocare a elementelor nu este identică cu cea în care acestea au fost adăugate în set (colecție neordonată).

[132]:
```
# adăugarea unui nou obiect în set
S.add(123)
S.add(3.14)
S
```

[132]:
```
{123, 3.14, 'a', 'l', 's', 't', 'u'}
```

T3.5.1 Operații cu seturi

[133]:
```
# Intersecție
S = {1, 2, 3, 4}
S & {1, 3}
```

[133]:
```
{1, 3}
```

[134]:
```
# Reuniune
{1, 5, 6} | S
```

[134]:
```
{1, 2, 3, 4, 5, 6}
```

[135]:
```
# Diferență
S - {1, 2, 3}
```

[135]:
```
{4}
```

[136]:
```
# Incluziune
S > {1, 3} # Superset
```

[136]:
```
True
```

[137]:
```
# Inițializarea unui set gol
S = set()
```

T3.5.2 Exemple de utilizare a seturilor

[138]:
```
# Crearea unui set pornind de la o listă
L = [1, 2, 1, 3, 2, 4, 5]
set(L)
```

[138]:
```
{1, 2, 3, 4, 5}
```

[139]:
```
# Eliminarea obiectelor duplicate dintr-o listă
# Ordinea obiectelor se poate modifica
L = list(set(['a', 'b', 'c', 'a', 'd', 'b']))
L
```

[139]:
```
['a', 'd', 'b', 'c']
```

[140]:
```
# Elementele diferite din două liste
set([1, 3, 5, 7]) - set([1, 2, 4, 5, 6])
```

[140]:
```
{3, 7}
```

[141]:
```
# Elementele diferite din două șiruri de caractere
set('abcdefg') - set('abdghij')
```

[141]:
```
{'c', 'e', 'f'}
```

[142]:
```
# Verificarea egalității setului de elemente din două liste
L1 = [1,2,3]
L2 = [3,2,1]
set(L1) == set(L2)
```

[142]:
```
True
```

[143]:
```
# Ordonarea unui set
sorted(set(L2))
```

[143]:
```
[1, 2, 3]
```

T3.5.3 Frozen sets

Seturile sunt obiecte mutabile, iar în anumite cazuri acest fapt limitează utilizarea lor. Pentru a crea un set imutabil, se poate crea un așa numit *frozenset*. Restul caracteristicilor și a operațiilor asociate unui set rămân la fel.

[144]:
```
S = (1, 2, 3, 4, 5)
FS = frozenset(S)
FS
```

[144]:
```
frozenset({1, 2, 3, 4, 5})
```

T3.6. Dicționare

Listele sunt un instrument util de gestionare a colecțiilor eterogene de obiecte ce pot fi indexate după poziția lor în listă. Însă în anumite cazuri este util să putem indexa elementele unei colecții de obiecte folsind o anumită cheie. **Dicționarele** în Python permit acest lucru și pot fi caracterizate prin:

- Cel mai flexibil tip de date
- Folosesc funcții hash pentru a indexa elementele dicționarului;
- Elementele sunt indexate prin **cheie**, nu index;
- Cheile trebuie să fie unice și **hashable** (orice obiect imutabil, precum int, string, boolean, tuplu este hashable);
- Mutabile
- Colecții **neordonate**;
- Lungime variabilă;
- Eterogene;
- Imbricate arbitrar;
- Tabele de referințe la obiecte (hash);
- Nu implementează metode ale tipurilor secvență, au metodele proprii.

Începând cu Python 3.7, dicționarele rețin ordinea de inserție a elementelor.

Operație	Interpretare
`D = {}`	Dicționar gol
`D = {'nume':'Maria', 'prenume':'Popescu'}`	Dicționar cu 2 elemente
`D = {'nume':'Maria', 'note':{'mate':10, 'info':10}`	Dicționar imbricat
`D = dict(nume='Maria', prenume='Popescu')`	Definire alternativă
`D = dict([('nume', 'Maria'), ('prenume', 'Popescu')])`	Definire alternativă

Operație	Interpretare
`D = dict.fromkeys(['nume', 'prenume'])`	Definire chei
`D = dict(zip(listachei, listavalori))`	Definire alternativă
`D['nume']`	Indexare după cheie
`D['note']['mate']`	Indexare imbricată după cheie
`'elem' in D`	Verificare apartenență cheie
`D.keys()`	Listă chei
`D.values()`	Listă valori
`D.items()`	Tuplu de chei și valori
`D.copy()`	Copiere
`D.clear()`	Ștergerea tuturor elementelor
`D.update(D2)`	Reuniune după chei
`D.get(cheie, default?)`	Extragere după cheie cu valoare default în cazul în care nu există cheia
`D.pop(cheie, default?)`	Eliminare după cheie cu valoare default în cazul în care nu există cheia
`len(D)`	Numărul de elemente
`D[cheie] = val`	Atribuire valoare la cheie
`del D[cheie]`	Ștergere după cheie
`D = { k:k+2 for k in [1,2,3,4]}`	Dictionary comprehension

Să vedem și câteva exemple practice de utilizare a dicționarelor:

[145]:
```
# Creare dicționar
D = {'mere': 2, 'pere': 3, 'portocale': 4}
```

[146]:
```
# Indexare după cheie
D['pere']
```

[146]: 3

[147]:
```
# Conținutul dicționarului
D
```

[147]:
```
{'mere': 2, 'pere': 3, 'portocale': 4}
```

[148]:
```
# Numărul de elemente din dicționar
len(D)
```

[148]:
```
3
```

[149]:
```
# Verificare apartenență
'portocale' in D
```

[149]:
```
True
```

[150]:
```
# Crearea unei liste din cheile dicționarului
list(D.keys())
```

[150]:
```
['mere', 'pere', 'portocale']
```

[151]:
```
# Alternativ
list(D)
```

[151]:
```
['mere', 'pere', 'portocale']
```

[152]:
```
# Modificarea unui element
D['pere'] = ['galbene', 'verzi', 'mov']
D
```

[152]:
```
{'mere': 2, 'pere': ['galbene', 'verzi', 'mov'], 'portocale':
 ↪ 4}
```

[153]:
```
# Ștergerea unui element
del D['mere']
D
```

[153]:
```
{'pere': ['galbene', 'verzi', 'mov'], 'portocale': 4}
```

[154]:
```
# Adăugarea unui element
D['banane'] = 7
D
```

[154]:
```
{'pere': ['galbene', 'verzi', 'mov'], 'portocale': 4,␣
↪'banane': 7}
```

[155]:
```
# Lista de valori din dicționar
list(D.values())
```

[155]:
```
[['galbene', 'verzi', 'mov'], 4, 7]
```

[156]:
```
# Tuplu chei-valori
list(D.items())
```

[156]:
```
[('pere', ['galbene', 'verzi', 'mov']), ('portocale', 4),␣
↪('banane', 7)]
```

T3.6.1 Dicționare imbricate

[157]:
```
D = {'prenume': 'Maria',
     'nume': ['Popescu', 'Ionescu'],
     'note': {'mate': 10, 'info': 10}}
```

[158]:
```
D['nume']
```

[158]:
```
['Popescu', 'Ionescu']
```

[159]:
```
D['prenume'][1]
```

[159]:
```
'a'
```

[160]:
```
D['note']['info']
```

[160]:
```
10
```

T3.6.2 Alte metode de creare a dicționarelor

[161]:
```
# Definirea dinamică a cheilor
D = {}
D['prenume'] = 'Maria'
D['nume'] = 'Popescu'
D
```

[161]:
```
{'prenume': 'Maria', 'nume': 'Popescu'}
```

[162]:
```
# Definire prin argumente keyword
D = dict(nume='Maria', prenume='Popescu')
D
```

[162]:
```
{'nume': 'Maria', 'prenume': 'Popescu'}
```

[163]:
```
# Definire prin tuplu cheie-valoare
D = dict([('nume', 'Maria'), ('prenume', 'Popescu')])
D
```

[163]:
```
{'nume': 'Maria', 'prenume': 'Popescu'}
```

[164]:
```
# Creare pe baza cheilor
D = dict.fromkeys(['mere', 'pere'], 0)
D
```

[164]:
```
{'mere': 0, 'pere': 0}
```

[165]:
```
# Creare folosind funcția zip()
D = dict(zip(['mere', 'pere', 'portocale'], [2, 3, 4]))
D
```

[165]:
```
{'mere': 2, 'pere': 3, 'portocale': 4}
```

[166]:
```
# Afișare ordonată după chei
for k in sorted(D):
  print(k, D[k])
```

[166]:
```
mere 2
pere 3
portocale 4
```

T3.7. Tupluri

Tuplurile, la o primă vedere, reprezintă o alternativă imutabilă a listelor în Python. Însă utilizarea lor este diferită în programele Python și sunt mult mai eficiente din punct de vedere al utilizării memoriei. Pe scurt, tuplurile sunt:

- Colecții ordonate de obiecte;
- Accesate prin offset (index);
- **Imutabile**;
- Lungime fixă;
- Eterogene;
- Pot fi imbricate;
- Tablouri de referințe la obiecte.

O listă scurtă a operațiilor cu tupluri e prezentată în tabelul următor:

Operație	**Interpretare**
`()`	Tuplu gol
`T = ('a','b')`	Tuplu cu două elemente
`T = 'a', 'b'`	Identic cu linia anterioară
`T = ('a', ('b','c'))`	Tuplu imbricat
`T = tuple('a')`	Creare tuplu
`T[i]`	Indexare tuplu
`T[i][j]`	Indexare tuplu imbricat
`T[i:j]`	Partiționare
`len(T)`	Numărul de elemente
`T1+T2`	Concatenare
`T*2`	Repetare
`'a' in T`	Verificare apartenență

Operație	Interpretare
`T.search('a')`	Indexul unui element
`T.count('a')`	Numărul de apariții ale elementului

[167]:
```
# Creare tuplu
T = ('a', 'b', 'c', 'd')
T
```

[167]:
```
('a', 'b', 'c', 'd')
```

[168]:
```
# Creare tuplu din listă
T = tuple(['b','a'])
T
```

[168]:
```
('b', 'a')
```

[169]:
```
# Ordonare tuplu
sorted(T)
```

[169]:
```
['a', 'b']
```

Tupluri denumite

O extensie utilă a tuplurilor sunt tuplurile denumite (en. *named tuples*), în cadrul cărora se poate utiliza o cheie pentru indexarea elementelor. Deși similare cu dicționarele, tuplurile denumite sunt **imutabile**.

Tuplurile denumite sunt parte din modulul `collections` și nu sunt tipuri de date built-in.

[170]:
```
from collections import namedtuple
Rec = namedtuple('Rec', ['prenume', 'nume', 'varsta'])
maria = Rec('Maria', 'Popescu', 19)
maria
```

[170]:
```
Rec(prenume='Maria', nume='Popescu', varsta=19)
```

[171]:
```
# Accesare prin index
maria[0], maria[2]
```

[171]:
```
('Maria', 19)
```

[172]:
```
# Accesare prin atribut/cheie
maria.nume, maria.prenume
```

[172]:
```
('Popescu', 'Maria')
```

T3.8. Conversii de tip (cast)

Conversia explicită a unui obiect la un alt tip de dată este posibilă folosind funcțiile built-in asociate datelor fundamentale. Conversia este realizată doar dacă se respectă formatul tipului de date țintă.

[173]:
```
# Conversie int la string
S = str(12)
S
```

[173]:
```
'12'
```

[174]:
```
# Conversie float la string
S = str(3.14)
S
```

[174]:
```
'3.14'
```

[175]:
```
# Conversie string la int
i = int('12')
i
```

[175]:
```
12
```

[176]:
```
# Conversie string la float
f1 = float('3.14')
f2 = float('10e2')
f1, f2
```

[176]:
```
(3.14, 1000.0)
```

[177]:
```
# Eroare la conversie
i = int('3.14')
```

[177]:
```
        ------------------------------------------
        ValueError Traceback (most recent call last)

        <ipython-input-177-3f624c312617> in <module>
          1 # Eroare la conversie
    ----> 2 i = int('3.14')
    ValueError: invalid literal for int() with base 10: '3.
↪14'
```

Anumite conversii de tip sunt realizate implicit atunci când apar diferite tipuri de date în expresii. Conversia se face întotdeauna către tipul de date mai larg, doar dacă acest lucru este posibil:

[178]:
```
# Conversie implicită la float
a = 3
b = 3.14
type(a + b)
```

[178]:
```
float
```

[179]:
```
# Eroare
a = '3'
b = 3.14
a + b
```

[179]:
```
        ---------------------------------------
        TypeError Traceback (most recent call last)
        <ipython-input-179-f015462efda6> in <module>
          2 a = '3'
          3 b = 3.14
    ----> 4 a + b
    TypeError: can only concatenate str (not "float") to str
```

T3.9. Alte tipuri de date

Python mai oferă o gamă foarte largă de tipuri de date disponibile prin modulele sale.

Dată/timp

[180]:
```
# Crearea unui obiect dată
from datetime import date
date.fromisoformat('2022-08-12')
```

[180]:
```
datetime.date(2022, 8, 12)
```

[181]:
```
# Modificarea zilei
d = date(2022, 8, 12)
d.replace(day=26)
```

[181]:
```
datetime.date(2022, 8, 26)
```

[182]:
```
# Afișare în format extins
d.ctime()
```

[182]:
```
'Fri Aug 12 00:00:00 2022'
```

[183]:
```
# Azi
date.today().ctime()
```

[183]:
```
'Wed Aug 24 00:00:00 2022'
```

[184]:
```
# Formatare dată
d.strftime("%d/%m/%y")
```

[184]:
```
'12/08/22'
```

[185]:
```
# Creare obiect dată-timp
from datetime import datetime
datetime.fromisoformat('2022-08-12T12:05:23')
```

[185]:
```
datetime.datetime(2022, 8, 12, 12, 5, 23)
```

[186]:
```
# Data și ora curentă
datetime.now().ctime()
```

[186]:
```
'Wed Aug 24 14:01:42 2022'
```

[187]:
```
# Peste 10 zile
from datetime import timedelta
(datetime.now()+ timedelta(days=10)).ctime()
```

[187]:
```
'Sat Sep  3 14:01:42 2022'
```

Colecții - oferă alternative la tipurile de date built-in dicționar, listă, set și tuplu.

[188]:
```
# Counter() e un dicționar ce numără aparițiile elementelor
from collections import Counter
c = Counter()
c = Counter(['a','b','a','c','b'])
c['a']
```

[188]:
```
2
```

Concluzii

În acest tutorial am parcurs tipurile de date fundamentale disponibile în Python și am văzut modul în care acestea pot fi create, modificate și cum se aplică metodele implicite ale acestora.

Exerciții

1. Să se definească două obiecte de tip float și să se afișeze suma, diferența, produsul și câtul lor.
2. Citiți de la tastatură mai multe caractere reprezentând litere mici. Să se transforme caracterele citite în litere mari în 2 moduri: a) printr-o operație aritmetică; b) folosind o operație logică și o mască adecvată.

3. Să se definească o listă de valori întregi și să se afișeze doar valorile distincte din aceasta.
4. Să se definească un dicționar ce folosește șiruri de caractere pe post de chei și elemente float pe post de valori. Să se afișeze doar cheile dicționarului și mai apoi tupluri formate din chei și valori
5. Să se definească 2 obiecte de tip float și să se determine partea întreagă a acestora folosind: a) o operație de conversie explicită; b) o funcție asociată tipului numeric.
6. Să se genereze un număr aleator între 0 și 10000, ce reprezintă un număr de secunde. Să se calculeze reprezentarea numărului de secunde în ore, minute și secunde și să se afișeze rezultatul formatat sub forma hh:mm:ss. Alternativ, folosiți modulul `datetime`.
7. Să se definească un șir de caractere și să se verifice că acesta conține doar caractere alfa-numerice.
8. Să se genereze o listă de numere aleatoare de dimensiune 10 și să se afișeze media lor folosind pachetul NumPy.
9. Să se definească un obiect de tip string și să se afișeze reprezentarea doar cu litere majuscule, precum și reprezentarea inversă a acestuia (de ex. "maria"->"airam").

T4 Instrucțiuni

T4.1. Instrucțiuni Python

Mergem mai departe în cadrul acestui tutorial și parcurgem instrucțiunile de bază ale limbajului Python. Spre deosebire de limbajele C/C++ și Java, vom vedea faptul că există o serie de instrucțiuni ce facilitează crearea de liste într-o singură instrucțiune, precum și atribuiri mai complexe.

La baza lor, programele sunt construite din instrucțiuni și expresii. Expresiile prelucrează obiecte și sunt încadrate în instrucțiuni. Expresiile returnează un rezultat, astfel încât de cele mai multe ori se află la dreapta unui semn de atribuire. Dar pot fi utilizate independent în apeluri de funcții de exemplu sau constructori ai obiectelor.

În limbajul Python nu este necesară utilizarea unui simbol de terminare a instrucțiunii (așa cum se utilizează `';'` în multe alte limbaje de programare). Introducerea unei linii noi de text semnalizează și terminarea instrucțiunii, cu unele excepții.

[1]:
```
a = 3 # Trecerea la următoarea linie simbolizează finalul␣
↪instrucțiunii
```

Cu excepția instrucțiunilor mai complexe ce pot fi scrise pe mai multe linii. În acest caz, liniile intermediare trebuie terminate cu `'\'` fără alte caractere după acest simbol.

[2]:
```
a, b, c, d, e, f, g, h =\
    1, 2, 3, 4,\
    5, 6, 7, 8
```

Dacă adăugăm un spațiu alb, vom genera o eroare de sintaxă:

[3]:
```
a, b, c, d, e, f, g, h =\
    1, 2, 3, 4, 5, 6, 7, 8
```

[3]:
```
  File "<ipython-input-3-4879cdae8854>", line 1
a, b, c, d, e, f, g, h =\
```

```
                                        ^
    SyntaxError: unexpected character after line␣
↪continuation character
```

O excepție de la această regulă se referă la apelul funcțiilor, unde nu e necesară terminarea liniei cu '\':

[4]:
```
def functie(a,b,c,d):
  return a+b+c+d

functie (a=1,
   b=2,
   c=3,
   d=4)
```

[4]:
```
10
```

Precum și definirea datelor de tip secvență:

[5]:
```
lista = [1,
     2,
     3,
     4]
```

Este posibilă scrierea mai multor instrucțiuni pe aceeași linie prin separarea lor prin ';', cu excepția instrucțiunilor compuse.

[6]:
```
a = 3; b = 4;
```

Instrucțiuni compuse

Instrucțiunile compuse în Python se demarchează prin indentare. Pentru a începe o instrucțiune compusă, se utilizează simbolul două puncte, ':'. Finalul indentării marchează finalul instrucțiunii compuse

[7]:
```
if 2 > 3:
  print ("Ramura True")
else:
  print ("Ramura False")
  print ("Mai adăugăm o linie")

print("Am ieșit din instrucțiunea compusă")
```

```
[7]: Ramura False
     Mai adăugăm o linie
     Am ieșit din instrucțiunea compusă
```

Pe scurt, lista de instrucțiuni disponibile în Python este prezentată în tabelul următor:

Instrucțiune	**Rol/Exemplu**
Atribuire	Crearea de referințe: `s, n = 'Ana', 3`
Apel și alte expresii	Rulare funcții: `suma(3, 4)`
Apeluri print()	Afișare obiecte: `print(obiect)`
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`
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')`

Instrucțiune	Rol/Exemplu
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`

În cele ce urmează vom prezenta pe scurt utilizarea și caracteristicile acestor instrucțiuni din limbajul Python.

T4.2. Atribuiri

Atribuirile se referă la crearea unei noi referințe către un obiect. Se realizează prin utilizarea simbolului egal, `'='`:

```
referință = expresie_obiect
```

Referințele sunt denumite în mod comun *variabile*. Astfel că în Python, variabilele sunt create la atribuire, iar acestea nu pot fi folosite înainte de a fi atribuite. Anumite operații crează atribuiri implicit.

Reguli pentru denumirea variabilelor

- Sunt formate din (underscore sau literă) + (oricâte litere, cifre sau underscore);
- Sunt case-sensitive;
- Cuvintele rezervate nu pot fi utilizate ca identificatori.

Convenții de denumire

- Variabilele ce încep cu underscore `'_'` nu sunt importați prin `from module import *`;
- Variabilele de tipul `__X__` sunt identificatori utilizați de sistem;
- Metodele încadrate de dunder `__` sunt denumite magic/dunder methods și sunt apelate implicit de obiect la execuția anumitor acțiuni;
- Variabilele ce încep cu dunder `__` și nu se termină cu underscore `_` sunt variabile pseudoprivate ale claselor;
- Variabila `_` în cadrul sesiunilor interactive păstrează ultimul rezultat calculat.

Cuvinte rezervate

Următorii identificatori sunt cuvinte rezervate și nu pot fi utilizați pentru a defini variabile de program:

`False`	`await`	`else`	`import`	`pass`
`None`	`break`	`except`	`in`	`raise`

True	class	finally	is	return
and	continue	for	lambda	try
as	def	from	nonlocal	while
assert	del	global	not	with
async	elif	if	or	yield

T4.2.1 Atribuiri de bază

[8]:
```
a = 1 # Atribuire de bază
a
```

[8]:
```
1
```

[9]:
```
b = 2
b
```

[9]:
```
2
```

[10]:
```
# Atribuire prin tuplu, echivalent cu a = 1; b = 2;
a, b = 1, 2
print(a)
print(b)
```

[10]:
```
1
2
```

[11]:
```
# Atribuire prin listă
[a, b] = [1, 2]
print(a)
print(b)
```

[11]:
```
1
2
```

[12]:
```
# Interschimbare variabile
a, b = b, a
a, b
```

[12]:
```
(2, 1)
```

[13]:
```
# Atribuire tuplu la listă de variabile
[a, b, c] = (1, 2, 3)
c, b, a
```

[13]:
```
(3, 2, 1)
```

[14]:
```
(a, b, c) = "ABC" # Atribuirea unui string către un tuplu
a, b, c           # mecanismul este denumit și despachetarea␣
 ↪secvențelor (en. sequence unpacking)
```

[14]:
```
('A', 'B', 'C')
```

T4.2.2 Atribuiri secvențiale avansate

[15]:
```
s = 'mere'
a, b, c = s[0], s[1], s[2:] # Indexare și segmentare
a, b, c
```

[15]:
```
('m', 'e', 're')
```

[16]:
```
a, b, c = list(s[:2]) + [s[2:]] # Segmentare și concatenare
a, b, c
```

[16]:
```
('m', 'e', 're')
```

[17]:
```
a, b = s[:2] # similar
c = s[2:]
a, b, c
```

[17]:
```
('m', 'e', 're')
```

[18]:
```
(a, b), c = s[:2], s[2:] # Secvențe imbricate
a, b, c
```

[18]:
```
('m', 'e', 're')
```

Atribuiri multiple

[19]:
```
a = b = c = 'mere'
a, b, c
```

[19]:
```
('mere', 'mere', 'mere')
```

[20]:
```
# ATENȚIE la obiectele mutable!!
a = b = [1,2]
b.append(42)
a, b # Se modifică ambele variabile
```

[20]:
```
([1, 2, 42], [1, 2, 42])
```

Atribuiri compuse

Se realizează in-place, ceea ce înseamnă că referința (variabila) inițială își va modifica valoarea sau obiectul referit.

[21]:
```
a = [1, 2]
b = a
a += [3, 4] # extindem a cu valorile 3 și 4
a, b        # b va conține aceleași valori ca a, deoarece
 ↪referă același obiect
```

[21]:
```
([1, 2, 3, 4], [1, 2, 3, 4])
```

T4.2.3 Despachetarea extinsă a secvențelor

În cazul în care dorim să despachetăm secvențe într-un mod mai complex, putem utiliza așa numitele variabile *star named*. În acest caz, valorile din secvență ce nu au un corespondent direct în lista de variabile, vor fi atribuite acestei variabile star named.

Pentru despachetarea simplă, am văzut deja un exemplu de tipul:

[22]:
```
seq = [1, 2, 3, 4]
a, b, c, d = seq
print(a, b, c, d)
```

[22]:
```
1 2 3 4
```

Ce se întâmplă atunci când numărul de elemente din secvență este mai mare decât numărul de variabile?

[23]:
```
# Eroare
a, b = seq
```

[23]:
```
------------------------------
ValueError     Traceback (most recent call last)
<ipython-input-23-566ff93241e9> in <module>
  1 # Eroare
----> 2 a, b = seq
ValueError: too many values to unpack (expected 2)
```

În acest caz, pentru ultima variabilă din listă și doar pentru aceasta, putem folosi star name. Ca urmare, toate elementele din secvență ce nu sunt atribuite variabilelor anterioare din listă, vor fi atribuite acestei ultime variabile star name:

[24]:
```
seq = [1, 2, 3, 4]
a, *b = seq
print (a)
print (b) # b va conține toate elementele din listă ce nu au␣
 ↪fost atribuite
```

[24]:
```
1
[2, 3, 4]
```

Este important de menționat faptul că, deși s-ar putea face o atribuire simplă în funcție de numărul de elemente din secvență, variabila star name va fi întotdeauna o listă (posibil goală):

[25]:
```
# d va prelua ultimul element sub formă de listă
seq = [1, 2, 3, 4]
a, b, c, *d = seq
print(a, b, c, d)
```

[25]:
```
1 2 3 [4]
```

[26]:
```
# *e va fi o listă goală
a, b, c, d, *e = seq
print(a, b, c, d, e)
```

[26]:
```
1 2 3 4 []
```

T4.3. Instrucțiunea vidă

În timpul dezvoltării unei aplicații pot să existe funcții, metode, clase, etc. ce nu sunt implementate momentan, dar care trebuie să fie declarate în cod. Cu alte cuvinte aceste funcții, metode, clase nu fac nimic momentan. Pentru a putea realiza acest lucru avem la dispoziție instrucțiunea vidă `pass`. Aceasta nu are niciun rezultat, ci este folosită ca înlocuitor pentru codul scris ulterior:

[27]:
```
def func():
  pass

func() # Apelul funcției
```

[28]:
```
class C:
  pass

obj = C() # Instanțierea unui obiect din clasa C
```

Începând cu Python 3.0 putem folosi ca alternativă elipsa `'...'`

[29]:
```
def func():
    ...
func()


class C:
  ...
obj = C()
```

T4.4. Funcția print()

Funcția `print()` afișează reprezentarea text a unui obiect. Este important de menționat aici faptul că reprezentarea text a unui obiect (mai complex) poate fi diferită de conținutul atributelor sale. Implicit se va apela metoda `__str__()` asociată obiectului și care poate fi suprascrisă în clasele proprii.

În Python 2.x `print` era o instrucțiune: `print a`. Iar în Python 3.x este o funcție built-in ce returnează `None`: `print (a)`.

Forma completă a funcției `print()` în Python 3.x este:

```
print([object, ...][, sep=' '][, end='\n']
      [, file=sys.stdout][, flush=False])
```

Să vedem câteva exemple:

[30]:
```
a = 'mere'
b = 1
c = ['pere']
print(a, b, c)
```

[30]:
```
mere 1 ['pere']
```

[31]:
```
print(a, b, c, sep='') # Eliminăm separatorul
```

[31]:
```
mere1['pere']
```

[32]:
```
print(a, b, c, sep=', ') # Separator special
```

[32]:
```
mere, 1, ['pere']
```

Rezultatul funcției `print()` este afișat în mod implicit în `stdout`. Putem însă redirecționa această afișare către `stderr` sau către un fișier:

[33]:
```
# Afișăm în stderr
import sys
print("Stderr", file=sys.stderr)
```

[33]:
```
Stderr
```

[34]:
```
# Scriem într-un fișier
print(a, b, c, file=open('out.txt', 'w'))
```

În mod uzual, funcția print va folosi metodele de formatare a stringurilor pentru a crea mesaje ce combină șirurile de caractere cu variabilele din program:

[35]:
```
a = 2
b = 3
print("Suma numerelor %d și %d este %s." %(a,b,a+b))
```

[35]:
```
Suma numerelor 2 și 3 este 5.
```

[36]:
```
s1 = 'Ana'
s2 = 'mere'
print ("%s are %s. " %(s1, s2))
print ("Inversul propoziției este: \"%s era %s.\"" %(s1[::
 ↪-1], s2[::-1]))
```

[36]:
```
Ana are mere.
Inversul propoziției este: "anA era erem."
```

T4.5. Instrucțiunea IF

Instrucțiunea IF este o instrucțiune de decizie, compusă, cu următoarea formă generală:

```
if test1:
   statements1
elif test2:
    statements2
else:
   statements3
```

`test1` și `test2`, precum și alte expresii incluse pe clauzele de `if` sau `else` trebuie să returneze o valoare de adevăr sau booleană, după cum urmează:

- Toate obiectele au o valoare booleană implicită;
- Orice număr diferit de zero și orice obiect nenul este `True`;
- Numerele egale cu 0, obiectele nule (goale) și obiectul special `None` sunt considerate `False`;
- Comparațiile și testele de egalitate sunt aplicate recursiv asupra structurilor de date;
- Comparațiile și testele de egalitate returnează `True` sau `False` (versiuni custom ale 1 și 0);
- Operatorii booleeni `and` și `or` returnează un obiect de tip `True` sau `False`;
- Operatorii booleni înlănțuiți nu se mai execută dacă se știe deja valoarea rezultatului.

Să vedem câteva exemple deutilizare a instrucțiunii IF:

[37]:
```
if 1:
  print('Ramura if')
else:
  print('Ramura else')
```

[37]:
```
Ramura if
```

[38]:
```
if False:
  print('Ramura if')
else:
  print('Ramura else')
```

[38]:
```
Ramura else
```

[39]:
```
a = 3
if a == 1:
  print("a are valoarea 1")
elif a == 2:
  print("a are valoarea 2")
else:
  print("a are altă valoare în afară de 1 sau 2")
```

[39]:
```
a are altă valoare în afară de 1 sau 2
```

În Python nu există echivalent direct pentru instrucțiunea `switch`. Însă poate fi substituită în 2 moduri. Pentru versiuni Python < 3.10 putem utiliza un dicționar, dar nu se pot executa instrucțiuni suplimentare, se returnează doar o valoare:

[40]:
```
cases = {'ana': 10,
         'ionut': 9,
         'maria': 8,
         'george': 7}

choice = 'maria'
print(cases[choice])
```

[40]:
```
8
```

În Python 3.10 a fost introdusă instrucțiunea `match` sub forma:

```
match choice:
   case <pattern_1>:
       <action_1>
   case <pattern_2>:
       <action_2>
   case <pattern_3>:
       <action_3>
```

```
    case _:
        <action_wildcard>
```

Pentru a utiliza această instrucțiune e necesar Python 3.10, în mod evident. Putem verifica versiunea de Python ce rulează momentan cu:

[41]:
```
!python --version
```

```
Python 3.7.13
```

Dacă celula de mai sus afișează Python 3.10.*, puteți rula celula următoare, altfel veți primi o eroare de sintaxă.

[42]:
```
choice = 'adriana'
match choice:
  case "ana":
    print ("10")
  case "ionut":
    print ("9")
  case "maria":
    print ("8")
  case "george":
    print ("7")
  case _: # Default
    print ("Nu am informatii despre aceasta persoana")
```

T4.5.1 Operatorul ternar

Similar cu operatorul ternar din C/C++, `expr?ramura_true:ramura_false`, în Python avem implementat acest operator folosind instrucțiunea `if` scrisă într-o singură linie:

```
R = Y if X else Z
```

Ceea ce ar fi echivalent cu:

```
if X:
   R = Y
 else:
   R = Z
```

[43]:
```
# Operator ternar cu if
a = 4
b = 10 if a < 3 else 11
print (b)
```

[43]:
```
11
```

[44]:
```
s = 'ana'
t = ' are mere' if s == 'ana' else ' are pere'
print (s+t)
```

[44]:
```
ana are mere
```

T4.5.2 Instrucțiuni IF imbricate

Spre deosebire de alte libaje de programare în care poate deveni destul de complicat de urmărit ramurile de `else if` sau `else` asociate unei instrucțiuni `if`, în Python, indentarea face acest lucru mult mai simplu:

[45]:
```
a = 5
b = 4
c = 3
if a > b:
  if a > c:
    print ("Max = a:", a)
  else:
    print ("Max = c:", c)
else:
  if b > c:
    print ("Max = b:", b)
  else:
    print ("Max = c:", c)
```

[45]:
```
Max = a: 5
```

T4.6. Instrucțiunea WHILE

Trecem mai departe la instrucțiunile ciclice sau de buclare. Prima instrucțiune de acest fel este instrucțiunea `while` ce are forma generală:

```
while condiție: # condiția de test a buclei
   instrucțiuni  # corpul buclei
 else:           # ramură else opțională
   instrucțiuni  # se execută dacă nu s-a ieșit cu break
```

Să vedem un prim exemplu:

[46]:
```
a = 10
while a: # Atât timp cât a!=0
  print(a, end=' ')
  a-=1
```

[46]:
```
10 9 8 7 6 5 4 3 2 1
```

E important ca în interiorul buclei, condiția de test să fie modificată. Altfel, obținem bucle infinite. Pentru exemplul următor va trebui să opriți forțat execuția celulei folosind iconița de stop din stânga acesteia:

[47]:
```
a = 11
while a: # atât timp cât a != 0
  print(a, end=' ')
  a-=2
```

[47]:
```
11 9 7 5 3 1 -1 -3 -5 -7 -9 -11 -13 -15  ...
```

Evident că există cazuri în care nu ne dorim ca o buclă să fie executată până ce condiția de test devine falsă sau să executăm tot corpul de instrucțiuni. Pentru aceasta, avem la dispoziție instrucțiunile de salt: `break` și `continue`

- `break` - iese din bucla ce o încapsulează;
- `continue` - sare la începutul buclei ce o încapsulează.

[48]:
```
a = 11
while a:
  if a < 5:
    break # Se iese din while cand a devine 5
  print(a, end=' ')
  a-=1
```

[48]:
```
11 10 9 8 7 6 5
```

[49]:
```
a = 11
while a:
  a-=1
  if a < 5:
    continue # Se sare peste următoarele instrucțiuni cand a␣
 ↪devine 5
  print(a, end=' ')

print("\na la iesirea din bucla este:", a)
```

[49]:
```
10 9 8 7 6 5
a la iesirea din bucla este: 0
```

Tot în bucla `while` avem ramura de `else`, care nu este comună multor altor limbaje de programare. Această ramură se execută la ieșirea normală din buclă și nu se execută atunci când ieșim cu o instrucțiune de salt de tip `break`:

[50]:
```
a = 11
while a:
  a-=1
else:
  print ("Am ajuns pe ramura de else!")
```

[50]:
```
Am ajuns pe ramura de else!
```

[51]:
```
a = 11
while a:
  a-=1
  if a < 5:
    print ("Iesim din bucla fara a trece prin ramura de␣
 ↪else")
```

```
    break
else:
  print ("Am ajuns pe ramura de else!")
```

[51]: Iesim din bucla fara a trece prin ramura de else

T4.7. Instrucțiunea FOR

O altă instrucțiune de ciclare (buclă) este instrucțiunea `for` cu forma generală dată de:

```
for val in obiect_iterabil:
   instrucțiuni
 else:
   instrucțiuni #se execută doar la ieșirea normală din for
```

Obiectul utilizat în antetul instrucțiunii (`obiect_iterabil`) trebuie să fie **iterabil**!!! Aceasta înseamnă că e fie un obiect de tip secvență, fie un obiect ce implementează mecanisme de iterare. Vom reveni spre finalul acestui tutorial asupra iteratorilor.

`val` poate fi modificată în cadrul buclei `for`, dar va reveni la următoarea valoare din obiectul iterabil în iterația următoare. La ieșire din buclă, `val` va stoca ultima valoare utilizată în buclă.

[52]:
```
# for peste o listă
for x in ["ana", "are", "mere"]:
  print(x, end=' ')
```

[52]:
```
ana are mere
```

[53]:
```
suma = 0
for x in [1, 2, 3, 4]: # iterare peste lista
  suma += x
print("Suma: ", suma)
```

[53]:
```
Suma:  10
```

[54]:
```
# Iterare peste string
S = "Python"
for c in S:
```

```
  print(c, end=' ')
```

[54]:
```
P y t h o n
```

[55]:
```
# Iterare peste tuplu
T = ('a', 'b', 'c')
for x in T:
  print(x, end=' ')
```

[55]:
```
a b c
```

[56]:
```
# Despachetare tuplu
T = [(1, 2), (3, 4), (5, 6)]
for (a, b) in T:
  print(a, b)
```

[56]:
```
1 2
3 4
5 6
```

[57]:
```
# Iterare folosind chei din dicționar
D = {'a': 1, 'b': 2, 'c': 3}
for key in D:
  print(key, D[key])
```

[57]:
```
a 1
b 2
c 3
```

[58]:
```
# Iterare folosind chei și valori din dicționar
D = {'a': 1, 'b': 2, 'c': 3}
for (key, value) in D.items():
  print(key, value)
```

[58]:
```
a 1
b 2
c 3
```

T4.7.1 Ramura else

Ca în cazul instrucțiunii WHILE, avem la dispoziție ramura `else` a instrucțiunii `for` ce se execută doar la ieșirea normală din buclă (fără salt):

[59]:
```
# Verificăm existența unei chei în dicționar
D = {'a': 1, 'b': 2, 'c': 3}
valoare = 4
for key in D:
  if D[key] == valoare:
    print ("Valoarea a fost găsită")
    break
else:
  # Dacă nu s-a apelat break
  print ("Valoarea nu a fost găsită", valoare)
```

[59]:
```
Valoarea nu a fost găsită 4
```

[60]:
```
# Forțăm ieșirea prin break
D = {'a': 1, 'b': 2, 'c': 3}
valoare = 2
for key in D:
  if D[key] == valoare:
    print ("Valoarea a fost găsită")
    break
else:
  # Nu se execută ramura else
  print ("Valoarea nu a fost găsită", valoare)
```

[60]:
```
Valoarea a fost găsită
```

T4.7.2 Bucle for imbricate

[61]:
```
litere = ['a', 'b', 'c']
cifre = [1, 2, 3]

for l in litere: # Iterăm peste lista litere
  for c in cifre: # Iterăm peste lista cifre
    print (l,c)
```

[61]:
```
a 1
a 2
a 3
b 1
b 2
b 3
c 1
c 2
c 3
```

T4.7.3 FOR și WHILE pentru citire din fișiere

În majoritatea aplicațiilor va fi nevoie să citim sau să scriem date din/în fișiere. Folosind buclele `while` sau `for`, putem realiza acest lucru extrem de simplu.

[62]:
```
# Creăm un fișier pe care să îl citim
%%writefile test.txt
Salut.
Ce mai faci?
```

[62]:
```
Writing test.txt
```

[63]:
```
# Folosind bucla while
file = open('test.txt')
while True:
  char = file.read(1) # Citim caracter cu caracter
  if not char:
    break  # Un string gol înseamnă finalul fișierului
  print(char)
```

[63]:
```
S
a
l
u
t
.

C
e
```

```
m
a
i

f
a
c
i
?
```

[64]:
```
# Folosind bucla for
for char in open('test.txt').read():
  print(char)
```

[64]:
```
S
a
l
u
t
.

C
e

m
a
i

f
a
c
i
?
```

[65]:
```
# Citire linie cu linie
file = open('test.txt')
while True:
  line = file.readline()
  if not line: break
  print(line.rstrip())
```

[65]:
```
Salut.
Ce mai faci?
```

[66]:
```
# Citim inițial toate liniile și doar le afișăm pe rând
for line in open('test.txt').readlines():
  print(line.rstrip())
```

[66]:
```
Salut.
Ce mai faci?
```

[67]:
```
# Echivalent cu
for line in open('test.txt'):
  print(line.rstrip())
```

[67]:
```
Salut.
Ce mai faci?
```

T4.8. Funcții suplimentare pentru bucle

T4.8.1 Funcția range()

Pentru buclele for am văzut până acum faptul că avem nevoie de un obiect iterabil astfel încât să poată fi rulate. Însă de cele mai multe ori avem nevoie de un iterator de numere simplu pe baza căruia să parcurgem bucla de un număr fix de ori. Pentru aceasta avem la dispoziție funcția `range()`:

```
range(start, stop, step)
```

Funcția va genera numerele cuprinse între `start` (inclusiv) și `stop` (exclusiv) cu un pas dat de `step`. Start este implicit 0, iar step este implicit +1:

[68]:
```
# Numerele de la 0 la 4
list(range(5))
```

[68]:
```
[0, 1, 2, 3, 4]
```

[69]:
```
# Numerele de la 2 la 4
list(range(2, 5))
```

[69]:
```
[2, 3, 4]
```

[70]:
```
# Numerele de la 0 la 9 incrementate cu 2 la ficare pas
list(range(0, 10, 2))
```

[70]:
```
[0, 2, 4, 6, 8]
```

[71]:
```
# Numerele de la -5 la 4
list(range(-5, 5))
```

[71]:
```
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
```

[72]:
```
# Numerele de la 5 la -4 decrementate cu 2 la fiecare pas
list(range(5, -5, -2))
```

[72]:
```
[5, 3, 1, -1, -3]
```

[73]:
```
# Utilizare în for
for i in range(3):
  print(i)
```

```
0
1
2
```

Range versus segmentare

Pentru tipurile de date secvență, `range()` poate fi înlocuit cu metodele de partiționare aplicate asupra lor:

[74]:
```
S = 'abcde'
# Indecși de la 0 la lungimea S, incrementați cu 2
list(range(0, len(S), 2))
```

[74]:
```
[0, 2, 4]
```

[75]:
```
# Afișăm tot al doilea caracter din S folosind range()
for i in range(0,len(S),2):
  print(S[i])
```

[75]:
```
a
c
e
```

[76]:
```
# Afișăm tot al doilea caracter din S folosind partiționarea␣
↪stringului
for c in S[::2]:
  print(c)
```

[76]:
```
a
c
e
```

T4.8.2 Funcția zip()

Funcția `zip()` permite combinarea mai multor date de tip secvență într-una singură. Secvența rezultată va fi compusă din elementele secvențelor

individuale aflate pe aceeași poziție ordinală:

[77]:
```
# Combinăm elementele a două liste
L1 = [1,2,3,4]
L2 = ['a', 'b', 'c', 'd']
list(zip(L1, L2)) # Primul element din L1 combinat cu␣
 ↪primult element din L2...
```

[77]:
```
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]
```

Mai sus folosim contructorul de listă deoarece `zip()` returnează o secvență iterabilă și nu poate fi afișat în mod direct:

[78]:
```
zip(L1,L2)
```

[78]:
```
<zip at 0x7f82fec4a230>
```

[79]:
```
# Utilizare în for
for (x, y) in zip(L1, L2):
  print(x, y)
```

[79]:
```
1 a
2 b
3 c
4 d
```

[80]:
```
# Secvențele sunt truncate la dimensiunea celei mai scurte
S1 = 'abc'
S2 = '12345'
list(zip(S1, S2))
```

[80]:
```
[('a', '1'), ('b', '2'), ('c', '3')]
```

Crearea dicționarelor cu funcția zip()

Cu ajutorul funcției `zip()` putem crea rapid dicționare în cazul în care știm cheile și valorile asociate acestor chei sub formă de secvențe:

[81]:
```
# Creăm o asociere între listele chei și valori
chei = ['ana', 'are', 'mere']
valori = [1, 2, 3]
list(zip(chei, valori))
```

[81]:
```
[('ana', 1), ('are', 2), ('mere', 3)]
```

[82]:
```
# Creăm un dicționar pornind de la cele 2 liste
D = {}
for (k, v) in zip(chei, valori):
  D[k] = v
D
```

[82]:
```
{'ana': 1, 'are': 2, 'mere': 3}
```

[83]:
```
# Putem scrie într-o singură linie folosind dictionary␣
↪comprehension
D = {k: v for (k, v) in zip(chei, valori)}
D
```

[83]:
```
{'ana': 1, 'are': 2, 'mere': 3}
```

[84]:
```
# Sau putem folosi constructorul dict()
D = dict(zip(chei,valori))
D
```

[84]:
```
{'ana': 1, 'are': 2, 'mere': 3}
```

T4.8.3 Funcția map()

Funcția `map()` va aplica o funcție specificat[asupra fiecărui element dintr-o secvență:

[85]:
```
# Calculăm valoarea ASCII a fiecărui caracter din string
list(map(ord, 'mere'))
```

[85]:
```
[109, 101, 114, 101]
```

[86]:
```
# Creăm o listă ce conține cubul valorilor din lista inițială
valori = [1, 2, 3, 4, 5]
def cub(n):
    return n**3

list(map(cub, valori))
```

Putem folosi și funcții ce iau mai mulți parametri. În acest caz va trebui să furnizăm iterabili pentru fiecare parametru în parte:

[87]:
```
# Ridicăm fiecare element din lista baza la puterea␣
↪specificată în putere
baza = [2,2,2,2]
putere = [1,2,3,4]
list(map(pow, baza, putere))
```

[87]:
```
[2, 4, 8, 16]
```

T4.8.4 Funcția enumerate()

Funcția `enumerate()` va prelua fiecare element dintr-o secvență, precum și indexul acestui element în secvență:

[88]:
```
S = 'mere'
for (index, elem) in enumerate(S):
          print(elem, 'apare la indexul', index)
```

[88]:
```
m apare la indexul 0
e apare la indexul 1
r apare la indexul 2
e apare la indexul 3
```

[89]:
```
# Iterăm liniile din fișier
for (index, linie) in enumerate(open('test.txt')):
         print('Linia %s: %s' % (index, linie.strip()))
```

[89]:
```
Linia 0: Salut.
Linia 1: Ce mai faci?
```

T4.9. Iteratori

Un **obiect iterabil** este o generalizare a noțiunii de secvență. Poate să fie o secvență stocată fizic (precum liste, tuplu, dicționare) sau un obiect ce produce valorile din secvență pe rând.

Orice obiect ce are atașată o metodă `__next__` pentru a avansa la următorul rezultat și care aruncă excepția `StopIteration` la finalul seriei de rezultate este considerat un iterator în Python. Un astfel de obiect poate fi utilizat într-o buclă for.

Iteratori de fișiere

[90]:
```
%%writefile input.txt
Linia 1
Linia 2
Linia 3
Linia 4
```

[90]:
```
Writing input.txt
```

[91]:
```
# Citim tot conținutul fișierului deodată
open('input.txt').read()
```

[91]:
```
'Linia 1\nLinia 2\nLinia 3\nLinia 4\n'
```

[92]:
```
# Extragem pe rând liniile din fișier folosind __next__
f = open('input.txt')
f.__next__()
```

[92]:
```
'Linia 1\n'
```

[93]:
```
f.__next__() # Următoarea linie din fișier
```

[93]:
```
'Linia 2\n'
```

T4.10. Comprehensiunea secvențelor

O facilitate extrem de puternică a limbajului Python și a secvențelor de obiecte se referă la mecanismul de comprehensiune (en. *comprehension*). Acest mecanism implică crearea obiectelor de tip secvență folosind o înlănțuire de operații și funcții scrise într-o singură linie de cod:

```
lista = [expresie for var in input_list
          if (var satisface condiția)]

dict = {cheie:valoare for (cheie, valoare) in iterabil
          if (cheie, valoare satisfac condiția)}

set = {expresie for var in input_list
          if (var satisface condiția)}
```

Echivalentul pentru acest mecanism ar fi utilizarea unei bucle `for` combinată cu instrucțiuni `if`.

[94]:
```
# Varianta standard
L = [1, 2, 3, 4, 5]
for i in range(len(L)):
  L[i] += 10
L
```

[94]:
```
[11, 12, 13, 14, 15]
```

[95]:
```
# Comprehensiune listă
L = [x + 10 for x in L]
L
```

[95]:
```
[21, 22, 23, 24, 25]
```

[96]:
```
# Creăm o listă cu caracterele din string
S = 'Ana123'
```

```
L = [c for c in S]
L
```

[96]: ['A', 'n', 'a', '1', '2', '3']

[97]:
```
# Creăm o listă cu caracterele majuscule din string
[c.upper() for c in S]
```

[97]: ['A', 'N', 'A', '1', '2', '3']

[98]:
```
# Creăm o listă cu literele majuscule din string
[c.upper() for c in S if c.isalpha()]
```

[98]: ['A', 'N', 'A']

[99]:
```
# Creăm o listă cu literele mari din lista de stringuri
# Folosim 2 bucle for în comprehensiune
L1 = ["Ana are mere", "Ionuț are pere", "Mihai are␣
 ↪portocale"]
L2 = [c for word in L1 for c in word if c.isupper()]
L2
```

[99]: ['A', 'I', 'M']

[100]:
```
# Comprehensiune dicționar - cubul elementelor pare din lista
L = [1, 2, 3, 4, 5, 6, 7]
D = {var:var ** 3 for var in L if var % 2 != 0}
D
```

[100]: {1: 1, 3: 27, 5: 125, 7: 343}

[101]:
```
# Comprehensiune set
L = [1, 2, 3, 4, 5, 5]
S = {x+10 for x in L}
S
```

[101]: {11, 12, 13, 14, 15}

Concluzii

În acest tutorial am încercat să introducem cât mai multe detalii esențiale ale utilizării instrucțiunilor de bază în limbajul Python. În tutorialul următor vom extinde utilizarea acestor instrucțiuni pentru crearea funcțiilor, a modulelor și pachetelor.

Exerciții

1. Să se afișeze valoarea lui Pi obținută din modulul `math` cu o precizie de 10 zecimale și aliniere la dreapta pe 20 de poziții.
2. Să se detemine maximul a trei numere folosind instrucțiunea `if`.
3. Să se afișeze primele 20 de valori din șirul Fibonacci.
4. Să se scrie un program ce afișează tot al doilea caracter dintr-o listă de șiruri de caractere.

```
L = ["Ana", "Maria", "Popescu", "Ionescu", "Vasile",␣
 ↪"Gheorghe"]
```

5. Să se scrie un program care determină numărul de cifre care compun un număr întreg.
6. Să se creeze o listă folosind mecanismul de comprehensiune ce conține doar numerele ce sunt pătrate perfecte dintr-o altă listă.
7. Să se creeze un dicționar prin mecanismul de comprehensiune ce folosește chei extrase dintr-o listă de stringuri, iar valorile asociate cheilor sunt indecșii la care apare caracterul 'a' în cheie. Cheile sunt doar acele stringuri ce conțin doar caractere alfabetice

```
L = ["Ana", "Maria", "Popescu", "Ion12", "Vasile34",␣
 ↪"Gheorghe"]
# Output: {'Ana': 2, 'Maria': 1, 'Popescu': -1, 'Gheorghe':␣
 ↪-1}
```

Referințe suplimentare

- Comprehensiune avansată - online.
- Iteratori - online.

T5 Funcții. Module

T5.1. Funcții

Reutilizarea codului este un aspect extrem de important în eficientizarea dezvoltării aplicațiilor. Definirea unui set de funcții reutilizabile, grupate în module sau pachete poate face acest lucru în mod facil. Acest tutorial prezintă aspectele legate de aceste noțiuni evidențiind în mod special flexibilitatea definirii funcțiilor în limbajul Python.

Funcțiile sunt seturi de instrucțiuni ce pot fi rulate de mai multe ori în decursul unui program și care de obicei returnează un rezultat pe baza unor parametri dați la intrare.

Definirea unei funcții se face prin utilizarea cuvântului cheie `def` urmat de numele funcției, lista de argumente și simbolul două puncte `:`. Nu se specifică datele de retur, iar codul aferent funcției este indentat.

```
def nume_functie(argumente):
    instrucțiuni
    return valoare
```

[1]:
```
# Definirea unei funcții
def functia_mea():
  print ("Prima mea funcție!")
```

Apelul (rularea) funcției se realizează prin numele funcției urmat de lista de argumente efective (dacă există):

[2]:
```
# Apel funcție
functia_mea()
```

[2]:
```
Prima mea funcție!
```

În cazul în care funcție returnează o valoare, aceasta va fi specificată folosind instrucțiunea `return`:

[3]:
```
# Funcție ce returnează o valoare
def functia_mea():
  return "Salut!"

functia_mea()
```

[3]:
```
'Salut!'
```

Lista de parametri ai funcției nu trebuie să conțină și tipul acestora, iar la apel se înlocuiesc cu parametri efectivi:

[4]:
```
# Funcție ce ia două argumente la intrare
def suma(a,b):
  return a+b

suma(1,2)
```

[4]:
```
3
```

Din acest motiv, funcțiile pot fi apelate cu diferite tipuri de obiecte, atât timp cât operațiile din interiorul funcției pot fi aplicate asupra acestor obiecte

[5]:
```
# Apel funcție cu diferite tipuri de obiecte trimise ca␣
 ↪argumente
print(suma(1,2))
print(suma(3.14, 1.93))
print(suma("Salut!", " Ce mai faci?"))
```

[5]:
```
3
5.07
Salut! Ce mai faci?
```

Din exemplu anterior putem observa o altă caracteristică importantă în Python legată de **polimorfismul operanzilor**. Acest lucru se referă la faptul că rezultatul unei operații depinde de operanzi, iar în Python toate operațiile sunt polimorfice atât timp cât obiectele asupra cărora sunt aplicate au definite comportamentele asociate.

Obiecte funcții

Funcțiile sunt de fapt obiecte, astfel încât numele lor nu este relevant pentru cod și se poate atribui un nou nume unei funcții fără a avea vreun efect programatic:

[6]:
```
def suma(a,b):
  return a+b
suma(1,2)
```

[6]: 3

[7]:
```
# Atribuim funcția unui alt obiect funcție
o_alta_suma = suma
# Apelăm noul obiect
o_alta_suma(2,3)
```

[7]: 5

Deoarece funcțiile sunt obiecte, se permite asocierea de atribute unui obiect funcție. Acest lucru poate părea destul de ciudat la o primă vedere pentru un programator ce utilizează alte limbaje de programare în mod uzual:

[8]:
```
# Atribuim un atribut unui obiect funcție
suma.attr = 3
suma.attr
```

[8]: 3

T5.2. Transmiterea argumentelor

Obiectele trimise ca parametri la apelul funcțiilor vor fi copiate sau referite de variabilele locale din funcție. Argumentele imutabile sunt transmise prin **valoare**, iar argumentele mutabile sunt transmise prin **referință**.

Modificarea unui obiect mutabil în cadrul unei funcții poate să afecteze obiectul trimis la apel!

[9]:
```
# Argumente imutabile
def f(a): # Se face o copie a obiectului trimis ca argument
  a = 99 # Modificăm valoarea locală

b = 88
f(b) # a din functie va fi o copie a obiectului trimis ca␣
 ↪argument
print(b) # b nu se modifică
```

[9]:
```
88
```

[10]:
```
# Argumente mutabile
def f(a, b):
  a = 2          # Modificăm copia locală
  b[0] = 'Mara' # Modificăm obiectul referit

m = 1
l = ["Ana", "are"]
f(m, l) # Trimitem obiecte mutabile și imutabile
m, l    # m nu se modifică, l se modifică
```

[10]:
```
(1, ['Mara', 'are'])
```

Evitarea modificării argumentelor

Pentru a evita modificarea obiectelor mutabile, se poate transmite o copie a acestora către funcție:

[11]:
```
l = ["Ana", "are"]
f(m, l[:]) # Trimitem o copie a lui l la apel
m, l
```

[11]:
```
(1, ['Ana', 'are'])
```

[12]:
```
# Sau modificăm funcția să lucreze cu o copie a obiectului␣
 ↪mutabil
def f(a, b):
  b = b[:]       # Facem o copie a obiectului trimis către␣
 ↪funcție
  a = 2
  b[0] = 'Mara' # Modifică doar copia listei

l = ["Ana", "are"]
f(m, l)
m, l
```

[12]:
```
(1, ['Ana', 'are'])
```

T5.3. Argumente ++

O caracteristică importantă a limbajului Python se referă la flexibilitatea listei de argumente ce pot fi transmise către funcții. O listă completă a metodelor de utilizare a argumentelor este prezentată în tabelele următoare, atât pentru partea de apel de funcții cât și pentru definirea lor.

La apelul funcției:

Sintaxă	Interpretare
`func(var)`	Argument pozițional
`func(nume=var)`	Argument de tip keyword identificat prin numele parametrului
`func(*pargs)`	Se trimit toate obiectele din iterabil ca argumente individuale poziționale
`func(**kargs)`	Se trimit toate perechile de cheie-valoare din dicționar ca argumente individuale de tip keyword

La definirea funcției:

Sintaxă	Interpretare
`def func(var)`	Argument normal, identifică orice valoare trimisă prin poziție sau nume
`def func(nume=var)`	Valoare implicită a argumentului dacă nu se transmite nicio valoare
`def func(*nume)`	Identifică și colectează toate argumentele poziționale într-un tuplu
`def func(**nume)`	Identifică și colectează toate argumentele de tip keyword într-un dicționar

Sintaxă	Interpretare
`def func(*rest, nume)`	Argumente ce trebuie transmise doar în apeluri de tip keyword (Python3.x)
`def func(*, nume=val)`	Argumente ce trebuie transmise doar în apeluri de tip keyword (Python3.x)

T5.3.1 Ordinea argumentelor

Ca urmare a complexității modului și tipului de transmisie a argumentelor către funcții, trebuie respectată o anumită ordine a argumentelor atât la apel, cât și la definirea funcției:

La apel:

- argumente poziționale,
- argumente keyword,
- argumentul *pargs,
- argumentul **kargs

În antet:

- argumente poziționale,
- argumente cu valori implicite,
- *pargs (sau * in Python 3.x),
- argumente keyword, * **kargs.

În Python 3.x au fost introduse și declarațiile de funcții ce permit utilizarea doar a argumentelor de tip keyword.

[13]:
```
# Argumente poziționale
def f(a, b, c):
  print(a, b, c)

# Ordinea argumentelor la apel contează
f(1, 2, 3)
f(2, 1, 3)
```

[13]:
```
1 2 3
2 1 3
```

[14]:
```
# Argumente keyword
# Se poate specifica numele argumentului și valoarea␣
 ↪transmisă
f(c=3, b=2, a=1)
f(b=3, a=1, c=2)
```

[14]:
```
1 2 3
1 3 2
```

[15]:
```
# Combinare argumente poziționale cu argumente keyword
f(1, c=3, b=2) # a primește valoare pe baza poziției, b și c␣
 ↪sunt trimise prin nume
```

[15]:
```
1 2 3
```

T5.3.2 Valori implicite ale argumentelor

[16]:
```
# Definim valori implicite pentru a, b și c
def f(a=1, b=2, c=3):
  print(a, b, c)

# Se utilizează valorile implicite pentru argumentele␣
 ↪netransmise
f()
f(2)
f(a=2)
```

[16]:
```
1 2 3
2 2 3
2 2 3
```

[17]:
```
# Suprascriem pozițional valorile implicite
f(1, 4) # c va lua valoarea implicită
f(1, 4, 5)
```

[17]:
```
1 4 3
1 4 5
```

[18]:
```
# Specificăm ce valoare implicită suprascriem
f(1, c=6) # a va lua valoare pozițional
```

[18]:
```
1 2 6
```

Valori implicite mutabile

În cazul în care folosim obiecte mutabile pentru valorile implicite, același obiect este folosit la fiecare apel al funcției ce utilizează doar valori implicite:

[19]:
```
def f(a=[]):
  a.append(1)
  print(a)

f()
f()
f([3]) # Trimitem o altă listă
```

[19]:
```
[1]
[1, 1]
[3, 1]
```

T5.3.3 Apel cu listă variabilă de argumente

Python permite ca numărul de argumente transmis către funcție să fie variabil:

[20]:
```
# Funcție cu listă variabilă de argumente poziționale
def f(*pargs):
  print(pargs)
# Apel fără argumente
f()
```

[20]:
```
()
```

[21]:
```
# Apel cu un argument
f(1)
```

[21]:
```
(1,)
```

[22]:
```
# Apel cu 4 argumente
f(1, 2, 3, 4)
```

[22]:
```
(1, 2, 3, 4)
```

[23]:
```
# Listă variabilă de argumente keyword
def f(**kargs):
  print(kargs)

# Apel fără argumente
f()
```

[23]:
```
{}
```

[24]:
```
# Apel cu două argumente keyword
# Argumentele sunt reținute sub formă de dicționar
f(a=1, b=2)
```

[24]:
```
{'a': 1, 'b': 2}
```

[25]:
```
# Combinare argument poziționat cu listă variabilă de␣
 ↪argumente poziționale
# și listă variabilă de argumente keyword
def f(a, *pargs, **kargs):
  print(a, pargs, kargs)
```

[26]:
```
f(1, 2, 3, x=1, y=2)
```

[26]:
```
1 (2, 3) {'x': 1, 'y': 2}
```

Despachetarea argumentelor

Despachetarea argumentelor (en. *unpacking arguments*) se referă la modul în care putem transmite lista de argumente către funcții folosind dicționare:

[27]:
```
def f(a, *pargs, **kargs):
  print(a, pargs, kargs)

f(1, 2, 3, x=1, y=2)
```

[27]:
```
1 (2, 3) {'x': 1, 'y': 2}
```

[28]:
```
# Definim un dicționar pentru lista de argumente de apel
kargs = {'a': 1, 'b': 2, 'c': 3}
kargs['d'] = 4
# Apelăm funcția folosind dicționarul definit anterior
f(**kargs)
```

```
[28]: 1 () {'b': 2, 'c': 3, 'd': 4}
```

T5.4. Funcții - elemente avansate

Înainte de a discuta elemente avansate legate de funcții, este important să reținem anumite principii de bază pentru codarea funcțiilor:

- Funcțiile nu trebuie să se bazeze pe elemente din afara lor, sunt elemente de sine-stătătoare și trebuie să fie cât mai simple, să servească un singur scop;
- Utilizarea variabilelor globale trebuie minimizată;
- Obiectele mutabile nu trebuie modificate decât dacă apelantul se așteaptă la asta;

T5.4.1 Funcții recursive

Funcțiile recursive au în corpul lor un apel la funcția definită curent. Este important ca în cadrul funcțiilor recursive să existe o condiție finală (ultimul pas din recursivitate), în caz contrar recursivitatea devine infinită și codul rămâne blocat în această funcție.

[29]:
```
# Calcul factorial() recursiv
def factorial(n):
    if n == 1:
        return 1
    else:
        return (n * factorial(n-1))

factorial(10)
```

[29]:
```
3628800
```

[30]:
```
# Calcul fibonacci() recursiv
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1 or n == 2:
        return 1
```

```
    else:
        return fibonacci(n-1) + fibonacci(n-2)

fibonacci(10)
```

[30]: 55

T5.4.2 Obiecte funcții

Am menționat și la începutul acestui tutorial faptul că funcțiile în Python sunt tot obiecte. Acest lucru înseamnă că le putem trata ca atare:

[31]:
```
def func(s):
  print(s)
func('Salut') # Apel direct
```

[31]: Salut

[32]:
```
alta_func = func # Creăm o nouă referință la obiectul func
alta_func('Pa')  # Apel prin noul obiect creat
```

[32]: Pa

Astfel că putem crea o funcție ce apelează funcții transmise ca argumente:

[33]:
```
def indirect(func, arg):
  func(arg) # Apelăm funcția trimisă ca argument

indirect(func, 'Ce mai faci?')
```

[33]: Ce mai faci?

[34]:
```
# Definim o altă funcție
def func2(L):
  print(L[0]*L[1])

# Și o apelăm prin funcția indirect()
indirect(func2, ["Ha", 3])
```

[34]: HaHaHa

T5.4.3 Introspecția în funcții

Fiind obiecte, funcțiile au asociate o serie de atribute ce permit introspecția, de exemplu afișarea numelui funcției apelate:

[35]:
```
def func(s):
  print(s)
func.__name__
```

[35]:
```
'func'
```

[36]:
```
# Creăm o nouă referință la obiect
alta_func = func
# Numele funcției rămâne același
alta_func.__name__
```

[36]:
```
'func'
```

Putem afișa toate atributele implicite ale obiectului funcție apelând `dir()`:

[37]:
```
' '.join(dir(func))
```

[37]:
```
'__annotations__ __call__ __class__ __closure__ __code__␣
↪__defaults__ __delattr__ __dict__ __dir__ __doc__ __eq__␣
↪__format__ __ge__ __get__ __getattribute__ __globals__␣
↪__gt__ __hash__ __init__ __init_subclass__ __kwdefaults__␣
↪__le__ __lt__ __module__ __name__ __ne__ __new__␣
↪__qualname__ __reduce__ __reduce_ex__ __repr__ __setattr__␣
↪__sizeof__ __str__ __subclasshook__'
```

Sau numele argumentelor funcției:

[38]:
```
func.__code__.co_varnames
```

[38]:
```
('s',)
```

Sau numărul argumentelor:

[39]:
```
func.__code__.co_argcount
```

[39]:
```
1
```

T5.4.4 Adnotări ale funcțiilor - Python 3.x

Datorită faptului că Python nu necesită specificarea tipului obiectelor transmise ca parametrii către funcții sau tipul returnat de acestea, uneori poate fi destul de complicat de citit codul. În Python 3.x au fost introduse așa numitele adnotări ale funcțiilor. Acestea nu au vreun efect programatic, ci doar informează utilizatorul codului despre tipul de date necesar apelului unei funcții sau alte condiții și mesaje ajutătoare pentru programator.

[40]:
```
import math
# Funcția ia la intrare un int și returnează floar
def arie(raza: int) -> float:
    return 2 * math.pi * raza ** 2

arie(2)
```

[40]:
```
25.132741228718345
```

Este important de reiterat faptul că adnotările nu au vreun efect asupra codului și faptul că parametrul funcției anterioare este adnotat cu `int`, nu însemnă că funcția nu poate fi apelată cu alt tip de obiect atâta timp cât acesta poate fi utilizat în corpul funcției

[41]:
```
# Apelăm cu argument float
arie(2.5)
```

[41]:
```
39.269908169872416
```

Adnotările nu sunt folosite doar pentru specificarea tipului argumentelor funcției, ci pot fi doar anumite mesaje informative:

[42]:
```
def arie(raza:'Raza cercului') -> float:
  return 2 * math.pi * raza ** 2

arie(2)
```

[42]:
```
25.132741228718345
```

Pentru a vizualiza aceste adnotări fără a avea acces la codul sursă, putem folosi atributul `__annotations__` al obiectelor funcție:

[43]:
```
arie.__annotations__
```

[43]:
```
{'raza': 'Raza cercului', 'return': float}
```

Restul aspectelor legate de argumentele funcțiilor rămân valide, precum valorile implicite ale parametrilor:

[44]:
```
def arie(raza: int = 2):
  return 2 * math.pi * raza ** 2

# Apelăm cu argument implicit
arie()
```

[44]:
```
25.132741228718345
```

[45]:
```
# Apelăm cu argument pozițional
arie (10)
```

[45]:
```
628.3185307179587
```

[46]:
```
# Apelăm cu argument keyword:
arie (raza=12)
```

[46]:
```
904.7786842338604
```

Trebuie să facem o distincție aici între documentația și adnotarea unei funcții. Documentația este de obicei un text elaborat prin intermediul căruia se specifică întreaga funcționalitate a unei funcții, alături de parametri de intrare și ieșire. Adnotările sunt mesaje informative scurte și doar ajută la o mai bună utilizare a funcției.

T5.5. Funcții lambda

Funcțiile lambda sunt de fapt expresii ce returnează o funcție ce poate fi mai apoi atribuită unui alt obiect funcție. Corpul funcției este constituit dintr-o singură expresie și au utilitate în reducerea numărului de linii de cod și utilizarea funcțiilor în construcții mai complexe.

Forma generală a unei funcții lambda este:

```
lambda arg1, arg2,... argN : expresie cu argumente
```

Să vedem câteva exemple:

[47]:
```
# Definirea standard a unei funcții
def suma(a, b):
  return a+b
suma(1,2)
```

[47]: 3

[48]:
```
# Alternativa lambda
suma = lambda a, b: a+b
# Apelăm în mod standard
suma(1,2)
```

[48]: 3

[49]:
```
# Putem utiliza și valori implicite
suma = (lambda a=1, b=2: a + b)
suma()
```

[49]: 3

Funcțiile lambda ne permit definirea unor secvențe de tip listă sau dicționar ce conțin diferite funcții și care pot fi apelate direct din indexarea secvenței:

[50]:
```
# Definim o listă de funcții lambda
L = [lambda x: x ** 2,
     lambda x: x ** 3,
     lambda x: x ** 4]
# Apelăm pe rând funcțiile din listă asupra unui argument
for f in L:
  print(f(2))
```

```
4
8
16
```

[51]:
```
# Sau putem apela direct funcția din lista definită anterior
L[1](3)
```

[51]: 27

[52]:
```
# Definim un dicționar de funcții lambda
D = {'patrat': (lambda x: x ** 2),
     'cub': (lambda x: x ** 3)}
# Apelăm funcția indexată de cheia 'cub'
D['cub'](3)
```

[52]: 27

Sau putem crea funcții ce returnează alte funcții lambda

[53]:
```
# Funcție ce returnează o funcție
def increment(x):
  return (lambda y: y + x)

# Creăm un nou obiect funcție
# Funcția din interiorul increment() devine lambda y: y+2
increment2 = increment(2)
# Apelăm noua funcție
increment2(6)
```

[53]: 8

Un element și mai avansat legat de funcții lambda se referă la imbricarea acestora. Definițiile din celula anterioară pot fi înlocuite de:

[54]:
```
increment = (lambda x: (lambda y: y + x))
increment2 = increment(2)
increment2(6)
```

[54]: 8

Sau mai abstract:

[55]:
```
((lambda x: (lambda y: y + x))(2))(6)
```

[55]: 8

T5.6. Programarea funcțională

Paradigmă de programare în care programele sunt construite prin aplicarea și compunerea funcțiilor. Funcțiile pot fi atribuite unor variabile, transmise ca parametri către alte funcții și returnate din funcții

În Python cele mai des utilizate funcții sunt: `map`, `filter`, `reduce` și se aplică asupra obiectelor **iterabile**

T5.6.1 MAP()

`map()` aplică funcția specificată ca prim parametru asupra secvenței iterabile:

[56]:
```
# Definirea standard
valori = [1, 2, 3, 4]
patrate = []
for x in valori:
  patrate.append(x ** 2)
patrate
```

[56]:
```
[1, 4, 9, 16]
```

[57]:
```
# Definim o funcție ce va fi aplicată prin map()
def patrat(x):
  return x ** 2
# Aplicăm patrat() asupra listei de valori
list(map(patrat, valori))
```

[57]:
```
[1, 4, 9, 16]
```

[58]:
```
# Sau folosim direct o funcție lambda
list(map((lambda x: x ** 2), valori))
```

[58]:
```
[1, 4, 9, 16]
```

[59]:
```
# În map() putem utiliza și funcții ce preiau argumente␣
 ↪multiple
def putere (a, b):
  return a ** b
list(map(putere, [1, 2, 3], [2, 3, 4]))
```

[59]:
```
[1, 8, 81]
```

T5.6.2 FILTER()

Selectează elementele unui obiect iterabil pe baza unei funcții de test

[60]:
```
L = [-2, -1, 0, 1, 2]
# Filtrăm doar valorile mai mari decât 0
list(filter((lambda x: x > 0), L))
```

[60]:
```
[1, 2]
```

T5.6.3 REDUCE()

Returnează un singur rezultat pornind de la un obiect iterabil. Se vor prelua pe rând elementele iterabilului și se va aplica funcția specificată, reținând întretimp rezultatul anterior:

[61]:
```
# În Python3.x trebuie importată funcția reduce()
from functools import reduce
L = [1, 2, 3, 4]
# Calculăm suma elementelor din L
reduce((lambda x, y: x + y), L)
```

[61]:
```
10
```

[62]:
```
# Calculăm produsul elementelor din L
reduce((lambda x, y: x * y), L)
```

[62]:
```
24
```

T5.6.4 Modulul operator

Tot în cadrul paradigmei de programare funcțională, Python oferă posibilitatea utilizării operatorilor sub formă de funcții prin intermediul modulului

operator. Lista funcțiilor asociate operatorilor poate fi consultată aici.

[63]:
```
import operator
# Mai mic decât
operator.lt(3,5)
```

[63]:
```
True
```

[64]:
```
# Împărțire exactă
operator.truediv(10,3)
```

[64]:
```
3.3333333333333335
```

[65]:
```
# Modificare elemente din listă
L = [1, 2, 3, 4, 5]
operator.setitem(L, slice(2,3), [9,10])
L
```

[65]:
```
[1, 2, 9, 10, 4, 5]
```

T5.7. Generatori

În anumite aplicații, datorită volumului mare de valori ce poate fi returnat de o funcție, este de dorit ca aceste valori să fie returnate secvențial. Pentru aceasta, avem la dispoziție **funcții și expresii generator** în Python. Atât funcțiile, cât și expresiile generator vor returna rezultatele pe rând, ceea ce înseamnă că execuția funcției sau a expresiei este suspendată până când codul apelant are nevoie de următoarea valoare. Drept urmare, generatorii sunt mult mai eficienți din punct de vedere al utilizării memoriei.

Pentru a crea un generator, vom folosi instrucțiunea `yield` în loc de `return` în definirea funcției:

[66]:
```
# Definim un generator
def patrate(n):
  for i in range(n):
    yield i ** 2

for i in patrate(5):
  # Se preiau pe rând valorile returnate de funcția generator
  print(i)
```

[66]:
```
0
1
4
9
16
```

[67]:
```
# Creăm un generator
x = patrate(4)
x
```

[67]:
```
<generator object patrate at 0x7f02d482aad0>
```

[68]:
```
# Extragem pe rând valorile din acesta folosind next()
next(x)
```

[68]: 0

[69]:
```
next(x)
```

[69]: 1

[70]:
```
next(x)
```

[70]: 4

Expresiile generator sunt similare cu comprehensiunea, dar nu returnează întreagă secvență, ci fiecare element pe rând:

[71]:
```
# Comprehensiune listă
L = [x ** 2 for x in range(4)]
L
```

[71]: [0, 1, 4, 9]

[72]:
```
# Generator
G = (x ** 2 for x in range(4))
G
```

[72]: <generator object <genexpr> at 0x7f02d482a950>

[73]:
```
next(G)
```

[73]: 0

[74]:
```
next(G)
```

[74]: 1

Funcțiile și expresiile generator pot fi iterate într-o singură instanță (en. *single iteration objects*), o singură dată, ceea ce înseamnă că nu pot fi iterate simultan la diferite poziții:

[75]:
```
G = (x * 2 for x in range(3))
I1 = iter(G) # Iterăm generatorul
next(I1)
```

[75]: 0

[76]:
```
next(I1)
```

[76]:
```
2
```

[77]:
```
# Creăm un al doilea iterator
I2 = iter(G)
# Dar acesta reține poziția iteratorului anterior
next(I2)
```

[77]:
```
4
```

[78]:
```
list(I1) # Extragem restul elementelor din generator
```

[78]:
```
[]
```

[79]:
```
# La epuizarea generatorului se aruncă o excepție
next(I1)
```

[79]:
```
        ---------------------------------------
        StopIteration  Traceback (most recent call last)
        <ipython-input-79-624144ae50ae> in <module>
          1 # La epuizarea generatorului se aruncă o excepție
    ----> 2 next(I1)
        StopIteration:
```

Python 3.3+ yield from

Începând cu Python 3.3 avem la dispoziție instrucțiunea `yield from` ce utilizează un obiect generator pentru a returna elementele. Pot fi utilizate mai multe instrucțiuni `yield from` în cadrul aceleiași funcții:

[80]:
```
def doi_generatori(N):
  yield from range(N)
  yield from (x ** 2 for x in range(N))

# Se returnează lista concatenată a elementelor celor 2␣
 ↪generatori
list(doi_generatori(4))
```

[80]:
```
[0, 1, 2, 3, 0, 1, 4, 9]
```

T5.8. Module

Fiecare fișier ce conține cod Python este un **modul**. Un modul determină așa-numitul *namespace* sau spațiul de vizibilitate a obiectelor. Modulele importă alte module pentru a folosi funcționalitățile implementate de acestea din urmă.

La realizarea unui import se caută mai întâi sursa modulului, se compilează în bytecode și se rulează și crează obiectele definite.

[81]:
```
# Creăm un modul
%%writefile modul.py
a = 3
b = 7
def test():
   print ("Salut")
```

[81]:
```
Writing modul.py
```

[82]:
```
# Importăm modulul și folosim variabilele și funcția definită
import modul
print (modul.a, modul.b)
modul.test()
```

[82]:
```
3 7
Salut
```

Modulele pot fi rulate și independent și se rulează de fapt tot codul ce există în afara funcțiilor sau claselor:

[83]:
```
# Creăm un modul
%%writefile alt_modul.py
a = 3
b = 7
def test():
  print ("Salut")
```

```
# Se vor executa instrucțiunile de mai jos
test()
print(a+b)
```

[83]:
```
Writing alt_modul.py
```

[84]:
```
# Rulăm modulul independent (din linia de comandă)
!python alt_modul.py
```

[84]:
```
Salut
10
```

Definirea anterioară a modulului nu este una corectă, deoarece și la import se va rula același cod:

[85]:
```
import alt_modul
```

[85]:
```
Salut
10
```

Astfel că e util să facem distincția între importul unui modul și rularea sa independentă. Pentru aceasta avem la dispoziție atributul `__name__`. Verificarea rulării independente se facem prin valoarea `__main__` a acestui atribut:

[86]:
```
# Creăm un modul
%%writefile modul_nou.py
a = 3
b = 7
def test():
  print ("Salut")

# Verificam daca rulam independent modulul
if __name__ == '__main__':
  print(a+b)
  test()
```

[86]:
```
Writing modul_nou.py
```

[87]:
```
# Rezultatul e același
!python modul_nou.py
```

[87]:
```
10
Salut
```

[88]:
```
# La import nu se rulează codul din if
import modul_nou
```

Rularea independentă a modulelor e utilă pentru testarea acestora. Codul de testare apare de obicei în instrucțiunea compusă `if __name__ == "__main__":`.

Calea de căutare a modulelor

Căile în care sunt căutate modulele importate respectă următoarea ierarhie:

1. Directorul de bază (home) al aplicației;
2. Căile din variabila PYTHONPATH (dacă e setată);
3. Directoarele în care e stocată librăria standard Python;
4. Conținutul fișierelor `.pth` (dacă există);
5. Directorul de bază (home) al site-packages pentru module terțe.

Rezultă lista stocată în variabila `sys.path`:

[89]:
```
import sys
sys.path
```

[89]:
```
['/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']
```

Pentru a adăuga un director nou la calea de căutare a pachetelor/modulelor, trebuie să modificăm atributul `path` al modulului `sys`:

[90]:
```
import sys
print(sys.path)
sys.path.append('/usr/adriana')
print(sys.path)
```

```
[90]: ['/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']
      ['/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', '/usr/adriana']
```

Fișiere bytecode *.pyc

Până la Python 3.1, fișierele compilate erau stocate în același director precum codul sursă, dar foloseau extensia `*.pyc`. Începând cu Python 3.2+ fișierele sunt stocate într-un subdirector `__pycache__` pentru a separa sursa de fișierul compilat. Folosesc tot extensia `*.pyc`, dar adăugă informații referitoare la versiunea de Python cu care au fost create. În ambele situații, fișierele sunt recompilate dacă s-a modificat codul sursă asociat lor.

T5.9. Pachete

Când grupăm mai multe module Python în cadrul aceluiași director, creăm de fapt un **pachet** Python. Pachetul va crea un nou namespace ce corespunde ierarhiei de directoare creată.

La import trebuie să specificăm calea relativă față de codul executat, până la modulul dorit:

```
import dir1.dir2.modul
from dir1.dir2.modul import x
```

[91]:
```
# Creăm două subdirectoare
!mkdir dir1
!mkdir dir1/dir2
```

[92]:
```
# Creăm un modul în cel de-al doilea subdirector
%%writefile dir1/dir2/submodul.py
a = 3
b = 4
```

```
Writing dir1/dir2/submodul.py
```

[93]:
```
# Importăm submodulul
import dir1.dir2.submodul
dir1.dir2.submodul.a, dir1.dir2.submodul.b
```

[93]:
```
(3, 4)
```

[94]:
```
# E mai eficient să folosim un alias
import dir1.dir2.submodul as s
s.a, s.b
```

[94]:
```
(3, 4)
```

Căi relative cu from

La importuri făcute cu from, putem utiliza căi relative, unde `.` se referă la directorul curent, iar `..` la directorul părinte. Dacă avem o structură de directoare de tipul

```
dir1 /
   main.py
   app.py
 app.py
```

Putem realiza următoarele importuri de module din `main.py`

```
from .  import app  # mod1/app.py
from .. import app  # ../app.py
```

__init__.py (Python <3.3)

Pentru ca un director oarecare să fie tratat ca pachet până la versiunea Python 3.3, acesta trebuie să includă un fișier denumit `__init__.py` ce conține codul de inițializare pentru pachetul respectiv, dar putea fi și gol. De cele mai multe ori acest fișier implementa comportamentul la importurile ce folosesc `from`, precum și lista `__all__` ce include submodulele ce trebuie importate. Fișierele `__init__.py` nu sunt create pentru a fi rulate independent.

De exemplu, pentru o structură de directoare de tipul: `dir0/dir1/dir2/modul.py`

Și un import:

```
import dir1.dir2.modul
```

- `dir1` și `dir2` trebuie să conțină `__init__.py`;
- `dir0` nu e necesar să conțină `__init__.py`. Acest fișier va fi ignorat dacă există;
- `dir0`, dar nu `dir0/dir1`, trebuie să existe în calea de căutare a modulelor `sys.path`.

Rezultă o structură de tipul:

```
dir0\ # Container on module search path
   dir1\
```

```
    __init__.py
  dir2\
    __init__.py
    mod.py
```

Ascunderea atributelor (_X, __all__)

Pentru a nu expune anumite obiecte către modulele apelante, există două metode de a le proteja într-o oarecare măsură:

1. Variabilele ce încep cu _ nu sunt importate print `from modul import *`. Însă acest variabile sunt disponibile la import simplu;
2. Definirea listei `__all__` la nivelul superior al modulului.

[95]:
```
%%writefile amodul.py
a = 1
_b = 2
c = 3
_d = 4
```

[95]:
```
Writing amodul.py
```

[96]:
```
# Se aduc doar variabilele ce nu încep cu _
from amodul import *
a, c
```

[96]:
```
(1, 3)
```

[97]:
```
# Eroare
_b
```

[97]:
```
---------------------------------------
NameError     Traceback (most recent call last)
<ipython-input-97-ba5b04df1ee1> in <module>
  1 # Eroare
----> 2 _b
NameError: name '_b' is not defined
```

[98]:
```
import amodul # La import simplu avem acces la toate␣
↪variabilele
amodul._b
```

[98]:
```
2
```

[99]:
```
# Dacă definim lista __all__, aceasta are precedență asupra␣
 ↪_X
# Atenție la utilizarea ghilimelelor la definirea __all__
%%writefile all_def.py
__all__ = ['x', '_z', '_t']
x, y, _z, _t = 1, 2, 3, 4
```

[99]:
```
Writing all_def.py
```

[100]:
```
# Se aduc doar variabilele definite în __all__
from all_def import *
x, _z
```

[100]:
```
(1, 3)
```

[101]:
```
# Eroare
y
```

[101]:
```
---------------------------------
NameError    Traceback (most recent call last)
<ipython-input-101-9558686eed14> in <module>
  1 # Eroare
----> 2 y
NameError: name 'y' is not defined
```

[102]:
```
# Fără wildcard putem aduce toate variabilele
from all_def import x, y, _z, _t
x, y, _z, _t
```

[102]:
```
(1, 2, 3, 4)
```

[103]:
```
import all_def
all_def.x, all_def.y, all_def._z, all_def._t
```

[103]:
```
(1, 2, 3, 4)
```

T5.10. Spațiul de nume

Vizibilitatea variabilelor este dată de următoarea ierarhie, denumită și **LEGB** (en. *local, enclosing, global, built-in*): * variabile locale (în funcție) ce nu sunt definite global; * variabile definite în funcțiile încapsulatoare (oricâte ar fi acestea), pornind de la cea mai interioară spre cele exterioare; * variabile globale (în modul) definite la începutul modulului sau precedate de cuvântul cheie `global` în alte funcții; * variabile build-in (în Python) definite în biblioteca standard.

Să vedem câteva exemple:

[104]:
```
i = 10
def test():
  # Utilizăm variabila globală
  print(i)

test()
```

[104]: 10

[105]:
```
i = 10
def test():
  # Definim o variabilă locală funcției
  i = 12

test()
# Variabila globală nu se modifică
i
```

[105]: 10

[106]:
```
i = 10
def test_outer():
  i = 12
  def test_inner():
```

```
    # test_inner va folosi variabila definită în test_outer
    print ("I in test_inner: ", i)
  test_inner()
  print ("I in test_outer: ", i)

test_outer()
# variabila din modul rămâne neschimbată
print ("I in modul: ", i)
```

[106]:
```
I in test_inner:  12
I in test_outer:  12
I in modul:  10
```

[107]:
```
def outer_1():
  a = 3
  def outer_2():
    b = 4
    def inner():
      # Inner are acces la variabilele definite în funcțiile␣
 ↪încapsulatoare
      print(a+b)
    inner()
  outer_2()
outer_1()
```

[107]:
```
7
```

Global și non-local

Cuvântul cheie `global` poate fi utilizat pentru a ne referi la variabile din codul încapsulator sau pentru a crea noi variabile la nivelul codului încapsulator:

[108]:
```
# Creăm o variabilă globală din interiorul unei funcții
def test():
  global j
  j = 24
test()
# Putem utiliza j chiar dacă a fost definit în funcție
j
```

[108]:
```
24
```

[109]:
```
# Modificăm o variabilă din exteriorul funcției
k = 10
def test():
  global k
  k = 20

test()
k
```

[109]:
```
20
```

Non-local este similar cu global, dar are utilitate doar în interiorul unei funcții. Spre deosebire de `global`, variabilele `non-local` nu pot fi create dinamic, ele trebuie să există în codul încapsulator.

[110]:
```
# Eroare, ii nu este definit anterior într-o funcție
ii = 10
def test():
  nonlocal ii
  ii = 12
test()
ii
```

[110]:
```
  File "<ipython-input-116-4669ba2d1853>", line 4
    nonlocal ii
    ^
SyntaxError: no binding for nonlocal 'ii' found
```

[111]:
```
# Încapsulăm codul anterior într-o altă funcție
def outer():
  ii = 10
  def test():
    nonlocal ii
    ii = 12
  test()
  print (ii)

outer()
```

[111]:
```
12
```

Built-in scope

Pentru a accesa lista de variabile și funcții definite în librăria standard (en. *built-in*) putem rula următoarea secvență de cod:

[112]:
```
import builtins
' '.join(dir(builtins))
```

[112]:
```
'ArithmeticError AssertionError AttributeError BaseException␣
↪BlockingIOError BrokenPipeError BufferError BytesWarning␣
↪ChildProcessError ConnectionAbortedError ConnectionError␣
↪ConnectionRefusedError ConnectionResetError␣
↪DeprecationWarning EOFError Ellipsis EnvironmentError␣
↪Exception False FileExistsError FileNotFoundError␣
↪FloatingPointError FutureWarning GeneratorExit IOError␣
↪ImportError ImportWarning IndentationError IndexError␣
↪InterruptedError IsADirectoryError KeyError␣
↪KeyboardInterrupt LookupError MemoryError␣
↪ModuleNotFoundError NameError None NotADirectoryError␣
↪NotImplemented NotImplementedError OSError OverflowError␣
↪PendingDeprecationWarning PermissionError␣
↪ProcessLookupError RecursionError ReferenceError␣
↪ResourceWarning RuntimeError RuntimeWarning␣
↪StopAsyncIteration StopIteration SyntaxError SyntaxWarning␣
↪SystemError SystemExit TabError TimeoutError True␣
↪TypeError UnboundLocalError UnicodeDecodeError␣
↪UnicodeEncodeError UnicodeError UnicodeTranslateError␣
↪UnicodeWarning UserWarning ValueError Warning␣
↪ZeroDivisionError __IPYTHON__ __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 display divmod enumerate eval␣
↪exec execfile filter float format frozenset get_ipython␣
↪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␣
↪range repr reversed round runfile set setattr slice sorted␣
↪staticmethod str sum super tuple type vars zip'
```

Variabilele built-in sunt automat disponibile în codul scris și de aceea e important să nu le suprascriem:

[113]:
```
open = 'Ana'      # Variabilă locală ce ascunde built-in
open('data.txt') # Open nu mai este funcția ce permite␣
 ↪deschiderea fișierelor
```

[113]:
```
        ---------------------------------
        TypeError    Traceback (most recent call last)
        <ipython-input-119-1eea51cff875> in <module>
    1 open = 'Ana'     # Variabilă locală ce ascunde built-in
---> 2 open('data.txt') # Open nu mai este funcția ce␣
 ↪permite deschiderea fișierelor
        TypeError: 'str' object is not callable
```

Concluzii

În acest tutorial am descoperit modul de definire a funcțiilor și organizarea codului Python în module și pachete. Tutorialul următor are în vedere introducerea aspectelor legate de programarea obiectuală (OOP).

Exerciții

1. Definiți o funcție ce returnează numărul de apariții ale unui caracter într-un string.
2. Definiți o funcție ce concatenează oricâte stringuri sunt date la intrarea sa.
3. Definiți o funcție ce rezolvă ecuații de gradul 2. Funcția primește ca argumente coeficienții ecuației.
4. Definiți o listă de funcții lambda ce returnează: tot al doilea caracter dintr-un string; stringul cu litere majuscule; poziția pe care se găsește un anumit caracter dat la intrare. Apelați toate funcțiile din listă pe rând în cadrul unei bucle for.
5. Definiți o funcție ce calculează media a trei note sprecificate la intrare. Dacă la apel nu se trimit toate notele, se vor folosi valori implicite egale cu 4. Apelați funcția cu diferite combinații de argumente poziționale și keyword.
6. Definiți o funcție recursivă ce afișează suma primelor N numere naturale.

T6 Programare obiectuală

T6.1. Clase

Programarea orientată pe obiecte (en. *OOP - Object Oriented Programming*) are o serie de avantaje, așa cum sunt ele definite și în alte limbaje obiectuale:

- moștenire;
- compunere;
- instanțe multiple;
- specializare prin moștenire;
- supraîncărcarea operatorilor;
- polimorfism;
- reducerea redundanței codului;
- încapsulare.

Toate aceste aspecte sunt reprezentate în limbajul Python ce permite astfel și implementarea paradigmei de programare obiectuală pe lângă cea funcțională pe care am văzut-o în tutorialul anterior.

T6.1.1 Definire. Instanțe. Atribute. Metode.

Pentru a defini o clasă în Python se utilizează cuvântul cheie `class`, iar sintaxa generală este:

```
class nume(clasadebaza1,clasadebaza2):
 atribut = valoare
 def metodă(self,...):
   self.atribut = valoare
```

Să vedem un prim exemplu:

[1]:
```
# Creăm o clasă fără corp
class Persoana:
  pass
```

Pentru a instanția un obiect, folosim numele clasei urmat de paranteze rotunde:

[2]:
```
# Instanțiem un obiect din clasă
P = Persoana()
# Afișăm tipul obiectului
type(P)
```

[2]:
```
__main__.Persoana
```

Atribute

Atributele de clasă se definesc simplu, la fel ca orice altă variabilă și pot fi accesate mai apoi prin numele obiectului urmat de punct `.` și numele atributului:

[3]:
```
class Persoana:
  # Două atribute ale clasei
  a = 3
  b = 4

P = Persoana()
# Accesăm atributele obiectului
P.a, P.b
```

[3]:
```
(3, 4)
```

În Python toate atributele sunt publice și virtuale (echivalent C++). Există, însă, o convenție de notare a atributelor private folosind `_atribut`. Dar această notație nu are valoare programatică, ci doar informează programatorul ce utilizează codul, că acele atribute nu sunt proiectate pentru a fi utilizate în afara claselor.

[4]:
```
class Persoana:
    _a = 3

P = Persoana()
# Putem folosi atributul _a
P._a
```

[4]: 3

O funcționalitate a atributelor claselor în Python se referă la modificarea numelor variabilelor (en. *name mangling*). Atunci când o variabilă începe cu dunder __, numele său este automat extins de interpretor pentru a include și numele clasei: `__atribut` devine `_Clasa__atribut`. Este folosit pentru a nu ascunde/suprascrie atribute din ierarhia de moștenire.

[5]:
```
class Persoana:
    __nume = "Ana"

P = Persoana()
# Afișăm atributele obiectului
print(dir(P))
```

[5]:
```
['_Persoana__nume', '__class__', '__delattr__', '__dict__',␣
 ↪'__dir__', '__doc__', '__eq__', '__format__', '__ge__',␣
 ↪'__getattribute__', '__gt__', '__hash__', '__init__',␣
 ↪'__init_subclass__', '__le__', '__lt__', '__module__',␣
 ↪'__ne__', '__new__', '__reduce__', '__reduce_ex__',␣
 ↪'__repr__', '__setattr__', '__sizeof__', '__str__',␣
 ↪'__subclasshook__', '__weakref__']
```

Obsevăm că, la fel ca în cazul tipurilor de date built-in, avem o serie de atribute predefinite pentru orice obiect instanțiat din clase definite de programator. Vom reveni asupra unora dintre acestea ulterior.

Accesul la acest tip de atribute se face prin numele complet:

[6]:
```
print(P._Persoana__nume)
```

[6]: Ana

[7]:
```
# Eroare
print(P.__nume)
```

[7]:
```
---------------------------------
AttributeError   Traceback (most recent call last)
<ipython-input-7-3bf65905cfa5> in <module>
  1 # Eroare
----> 2 print(P.__nume)
```

```
    AttributeError: 'Persoana' object has no attribute␣
↪'__nume'
```

Metode

Funcțiile definite în clase sunt denumite **metode**. Au aceleași funcționalități ca funcțiile de bază, doar că vor conține ca prim argument, o referință la instanța curentă (cu excepția metodelor statice și a celor de clasă). Referința la instanța curentă se face prin variabila `self`:

[8]:
```
# Metodă a instanței
class Persoana:
  nume = "Ana"
  varsta = 19
  def print_info(self, nume, varsta):
    print ("Nume: %s, varsta: %d" %(self.nume, self.varsta))

P = Persoana()
P.print_info("Ana", 19)
```

[8]:
```
Nume: Ana, varsta: 19
```

[9]:
```
# Nu putem apela metoda prin numele clasei doar
Persoana.print_info("Ana", 19)
```

[9]:
```
    ------------------------------------
    TypeError  Traceback (most recent call last)
    <ipython-input-9-3ab5d6f7fb95> in <module>
      1 # Nu putem apela metoda prin numele clasei doar
      ----> 2 Persoana.print_info("Ana", 19)
TypeError: print_info() missing 1 required positional␣
↪argument: 'varsta'
```

Din eroarea generată înțelegem faptul că, odată cu apelarea metodei prin intermediul numelui clasei, nu se transmite și referința la obiectul curent (`self`), ci doar argumentele `Ana` și `19`. Definiția metodei așteaptă 3 argumente, `self`, `nume` și `varsta`.

Dacă dorim să apelăm o anumită metodă prin intermediul numelui clasei, deși nu este recomandat, putem folosi următoarea instrucțiune în care trimitem și o instanță a clasei:

[10]:
```
Persoana.print_info(P, "Ana", 19)
```

[10]:
```
        ---------------------------------
        AttributeError  Traceback (most recent call last)
        <ipython-input-8-bc56ff4238f9> in <module>
    ----> 1 Persoana.print_info(P, "Ana", 19)
    AttributeError: type object 'Persoana' has no attribute␣
↪'print_info'
```

Constructori

Constructorii sunt metode speciale ale unei clase cu nume predefinit, `__init__()`. Constructorii sunt apelați automat la instanțierea unui nou obiect din clasa respectivă. De obicei sunt folosiți pentru a defini atributele instanțelor și pentru a rula alte metode necesare la inițializarea obiectului curent. În cadrul constructorului trebuie utilizată o referință la obiectul curent, specificată prin argumentul `self`.

Așa cum am văzut în exemplele anterioare, nu este necesar să fie definit un constructor explicit, existând oricum unul implicit. Fără a defini, însă, un constructor, devine mai complicată personalizarea diferitelor instanțe ale obiectului.

[11]:
```
class Persoana:
  # Constructor explicit
  def __init__(self, nume, varsta):
    self.nume = nume
    self.varsta = varsta
  # Metodă a instanței
  def print_info(self):
    print ("Nume: %s, varsta: %d" %(self.nume, self.varsta))

# Definim două obiecte cu atribute diferite
P1 = Persoana("Ana", 19)
P1.print_info()
P2 = Persoana("Maria", 20)
P2.print_info()
```

[11]:
```
Nume: Ana, varsta: 19
Nume: Maria, varsta: 20
```

În Python **NU** putem avea mai mulți constructori într-o clasă. Dacă sunt definite mai multe metode `__init__()`, doar ultima va fi apelată.

Dacă, însă, dorim să avem comportamente diferite în funcție de numărul de obiecte transmise la instanțierea unui obiect, putem utiliza valori implicite pentru argumentele constructorului:

[12]:
```
class Persoana:
  # Constructor explicit cu valori implicite
  def __init__(self, nume = "UNK", varsta = -1):
    self.nume = nume
    self.varsta = varsta

  # Metodă a instanței
  def print_info(self):
    print ("Nume: %s, varsta: %d" %(self.nume, self.varsta))

# Utilizăm valorile implicite pentru atribute
P = Persoana()
P.print_info()
# Dăm valori atributului nume
P1 = Persoana(nume="Ionuț")
P1.print_info()
# Dăm valori atributului varsta
P2 = Persoana(varsta=21)
P2.print_info()
# Dăm valori ambelor atribute
P3 = Persoana("Mihai", 22)
P3.print_info()
```

[12]:
```
Nume: UNK, varsta: -1
Nume: Ionuț, varsta: -1
Nume: UNK, varsta: 21
Nume: Mihai, varsta: 22
```

- Instrucțiunea `class` creează un obiect clasă și îi atribuie un nume;
- Atribuirile din interiorul clasei creează atribute de clasă;
- Atributele clasei definesc starea și comportamentul unui obiect;
- Obiectele instanță sunt elemente concrete;
- Instanțele sunt create prin constructorul clasei;
- Fiecare obiect are asociate atributele instanței;
- Referirea la instanța curentă se face prin `self` (convenție).
- Clasele sunt atribute ale modulelor;
- Pot fi definite mai multe clase în cadrul aceluiași modul

- La instanțierea obiectelor din module externe celui curent, trebuie urmărită ierarhia modulului.

Referitor la `self`, putem utiliza și alt identificator pentru a face referire la instanța curentă, doar că nu este recomandat:

[13]:
```
class Persoana:
  # Folosim this in loc de self pentru instanța curentă
  def __init__(this, nume = "", varsta = -1):
    this.nume = nume
    this.varsta = varsta

  def print_info(this):
    print ("Nume: %s, varsta: %d" %(this.nume, this.varsta))

P = Persoana()
P.print_info()
```

[13]:
```
Nume: , varsta: -1
```

T6.1.2 Moștenire

Pentru a moșteni alte clase în Python, din punct de vedere al sintaxei trebuie doar să le enumerăm între paranteze după numele clasei:

```
class ClasaDerivata(ClasaBaza1, ClasaBaza2):
    ...
```

Accesul la metodele claselor de bază se face prin `super()` sau prin numele clasei de bază:

[14]:
```
# Clasa de baza
class Baza:
  def __init__(self):
    self.tip = "om"
    self.gen = "feminin"

  def print_base_info(self):
    print("Tip: %s, gen: %s" %(self.tip, self.gen))

# Clasa derivata
class Persoana(Baza):
```

```
  def __init__(self, nume = "", varsta = -1):
    # Apel constructor clasa de baza
    Baza.__init__(self)
    self.nume = nume
    self.varsta = varsta

  def print_info(self):
    # Folosim și atribute ale clasei de bază
    print ("Nume: %s, varsta: %d, tip: %s, gen: %s" %(self.
 ↪nume, self.varsta, self.tip, self.gen))
    # Apel metodă din clasa de baza prin super()
    super().print_base_info()
    # Apel metodă din clasa de baza prin numele clasei
    Baza.print_base_info(self)

P = Persoana("Ana", 20)
P.print_info()
# Apelam o metoda a clasei de bază prin intermediul␣
 ↪instanței derivate
P.print_base_info()
# Folosim un atribut din clasa de bază
P.tip
```

[14]:
```
Nume: Ana, varsta: 20, tip: om, gen: feminin
Tip: om, gen: feminin
Tip: om, gen: feminin
Tip: om, gen: feminin
'om'
```

Ordinea de rezoluție a metodelor

Ordinea de rezoluție a metodelor (en. *MRO - Method resolution order*) se referă la modul în care interpretorul determină metoda ce trebuie apelată din ierarhia de moștenire. Procesul decurge astfel: se caută metoda în clasa curentă, iar mai apoi în clasele de bază, în ordinea enumerării lor la definirea clasei curente.

[15]:
```
class A:
    def met(self):
        print("Met() din A")
class B:
    def met(self):
```

```
        print("Met() din B")

# Clasa C moștenește A și B și redefinește met()
class C(A,B):
  def met(self):
    print("Met() din C")
# Clasa D moștenește A și B și nu redefinește met()
class D(A,B):
        pass
# Inversăm ordinea moștenirii
class E(A,B):
        pass

# Obiect din clasa C
O1 = C()
O1.met()

# Obiect din clasa D
O2 = D()
O2.met()

# Obiect din clasa E
O3 = E()
O3.met()
```

[15]:
```
Met() din C
Met() din A
Met() din A
```

Același principiu se aplică și atunci când apelăm metode prin intermediul `super()`, ordinea de enumerare a claselor moștenite determină metoda apelată.

Clase abstracte

Clasele abstracte sunt acele clase ce au cel puțin o metodă ce nu este definită și nu pot fi instanțiate. Sunt de fapt o bază pentru clasele derivate.

În Python nu există un mecanism implicit de definire a claselor abstracte, ci se poate realiza prin utilizarea modulului `abc` (Abstract Base Class). Metodele neimplementate ale clasei abstracte vor fi decorate cu `@abstractmethod`, iar clasele derivate va trebui să implementeze aceste metode:

[16]:
```
# Definim o clasă abstractă ce conține o metodă abstractă și una implementată
from abc import ABC, abstractmethod

class FormaGeometrica(ABC):
  @abstractmethod
  def perimetru(self, L:list):
    pass

  def salut(self):
    return "Salut, sunt o formă geometrică de tipul: "
```

[17]:
```
# Nu o putem instanția
FG = FormaGeometrica()
```

[17]:
```
---------------------------------
TypeError Traceback (most recent call last)
<ipython-input-15-5c0b048ad3ae> in <module>
  1 # Nu o putem instanția
----> 2 FG = FormaGeometrica()
TypeError: Can't instantiate abstract class FormaGeometrica with abstract methods perimetru
```

[18]:
```
# Definim clase derivate ce vor implementa metoda abstractă
class Triunghi(FormaGeometrica):
  def perimetru(self, L:list):
    return L[0]+L[1]+L[2]
  def salut(self):
    # Apelăm metoda din clasa de bază
    print(super().salut()+"triunghi")

class Dreptunghi(FormaGeometrica):
  def perimetru(self, L:list):
    return L[0]+L[1]+L[2]+L[3]

  def salut(self):
    # Apelăm metoda din clasa de bază
    print(super().salut()+"dreptunghi")


T = Triunghi()
```

```
print(T.perimetru([2,3,4]))
T.salut()

D = Dreptunghi()
print(D.perimetru([2,3,2,3]))
D.salut()
```

[18]:
```
9
Salut, sunt o formă geometrică de tipul: triunghi
10
Salut, sunt o formă geometrică de tipul: dreptunghi
```

Introspecția în clase

Introspecția se poate aplica și asupra claselor Python, unde putem folosi atribute precum:

- `instance.__class__` - clasa din care face parte instanța
- `class.__name__` - numele clasei
- `class.__bases__` - clasele din ierarhie
- `object.__dict__` - dicționar cu lista de atribute asociată obiectului

[19]:
```
# Creăm un modul ce conține o clasă
%%writefile person.py
class Persoana:
  # Constructor explicit
  def __init__(self, nume, varsta):
    self.nume = nume
    self.varsta = varsta
  # Metodă a instanței
  def print_info(self):
    print ("Nume: %s, varsta: %d" %(self.nume, self.varsta))
```

[19]:
```
Writing person.py
```

[20]:
```
# Importăm modulul
from person import Persoana
ana = Persoana('Ana', 19)
# Afișăm tipul obiectului
type(ana) # Se afișează și numele modulului
```

[20]:
```
person.Persoana
```

[21]:
```
# Afișăm numele modulului din care face parte clasa␣
 ↪obiectului
ana.__module__
```

[21]:
```
'person'
```

[22]:
```
# Afișăm clasa asociată obiectului cu numele modulului inclus
ana.__class__
```

[22]:
```
person.Persoana
```

[23]:
```
# Afișăm doar numele clasei asociate obiectului
ana.__class__.__name__
```

[23]:
```
'Persoana'
```

[24]:
```
# Afișăm atributele asociate obiectului din clasa Persoana
list(ana.__dict__.keys())
```

[24]:
```
['nume', 'varsta']
```

[25]:
```
# Afișăm atributele și volorile lor folosind O.__dict__
for key in ana.__dict__:
  print(key, '=', ana.__dict__[key])
```

[25]:
```
nume = Ana
varsta = 19
```

[26]:
```
# Afișăm atributele obiectului folosind __dict__ și getattr
for key in ana.__dict__:
  print(key, '=', getattr(ana, key))
```

[26]:
```
nume = Ana
varsta = 19
```

T6.1.3 Metode statice și de clasă

Metodele statice și de clasă pot fi apelate fără a instanția clasa. Diferența este că, metodele statice funcționează ca simple funcții în interiorul unei clase, fără a fi atașate unei instanțe și fără a avea acces la starea clasei în mod direct. Metodele de clasă primesc ca prim argument referința la clasă

în locul unei instanțe, ceea ce înseamnă că pot modifica starea per ansamblu a clasei, de exemplu un atribut ce aparține tuturor instanțelor clasei. De cele mai multe ori, metodele de clasă returnează un obiect din clasa curentă și au un comportament similar cu al constructorilor.

Specificarea faptului că o metodă este statică sau de clasă se face prin aplicarea metodelor `staticmethod()` sau `classmethod()` asupra obiectului metodă. Sau folosind decoratorii `@staticmethod` sau `@classmethod`. Decoratorii vor fi introduși într-o secțiune următoare.

Metode statice

[27]:
```
# Definim o metodă statică în clasă
class Persoana:
  def print_info():
    print("Salut!")

  # Specificăm faptul că metoda este statică
  print_info = staticmethod(print_info)

# Apelăm metoda prin instanță
P1 = Persoana()
P1.print_info()
# Apelăm metoda prin numele clasei
Persoana.print_info()
```

[27]:
```
Salut!
Salut!
```

Metode de clasă

[28]:
```
class Persoana:
  numar_persoane = 0
  def __init__(self):
    # Atribut al clasei
    Persoana.numar_persoane += 1

  # Metodă de clasă, primește argument o clasă
  def print_info(cls):
    print("Numarul de persoane: %d" % cls.numar_persoane)
  # Specificăm faptul că print_info e metodă de clasă
  print_info = classmethod(print_info)
```

```
a = Persoana()
# Se transmite automat clasa către metodă
a.print_info()
b = Persoana()
b.print_info()
# Apelăm prin numele clasei
Persoana.print_info()
```

[28]:
```
Numarul de persoane: 1
Numarul de persoane: 2
Numarul de persoane: 2
```

[29]:
```
# Metodă de clasă factory
class Persoana:
  def __init__(self, nume, varsta):
    self.nume = nume
    self.varsta = varsta
  def print_info(self):
    print ("Nume: %s, varsta: %d" %(self.nume, self.varsta))

  # Metodă de clasă
  def from_string(cls, S):
    return cls(S.split('-')[0], int(S.split('-')[1]))
  from_string = classmethod(from_string)

# Instanțiem un obiect prin intermediul metodei de clasă
P = Persoana.from_string("Ana-19")
P.print_info()
```

[29]:
```
Nume: Ana, varsta: 19
```

T6.1.4 Supraîncărcarea operatorilor

De cele mai multe ori, pentru obiectele definite de programator, operatorii standard nu pot fi aplicați, deoarece nu există un mecanism clar de aplicare a lor. De exemplu, ce înseamnă că un obiect este mai mare decât altul sau că două obiecte sunt egale sau diferite.

Mecanismul de supraîncărcare a operatorilor asociat claselor permite modificarea comportamentului de bază al operatorilor built-in atunci când aceștia sunt aplicați asupra obiectelor definite de programator.

Să vedem câteva exemple:

[30]:
```
# Suprascriem operatorul scădere
class Numar:
  def __init__(self, val):
    self.val = val

  def __sub__(self, sub):
    return Numar(self.val - sub) # Rezultatul e o nouă␣
 ↪instantă

O1 = Numar(10) # se apelează Numar.__init__(O1, 10)
O2 = O1 - 5    # se apelează Numar.__sub__(O1, 5)
O2.val         # O2 e o altă instanță a Numar
```

[30]: 5

[31]:
```
# Suprascriem operatorul de indexare
class Numar:
  def __getitem__(self, index):
    return index+1
O = Numar()
O[2]        # Se apelează O.__getitem__(2)
```

[31]: 3

[32]:
```
# Suprascriem operatorul de returnare valoare atribut
class Persoana:
  def __getattr__(self, attrname):
    if attrname == 'varsta':
      return 19
    else:
      return -1
O = Persoana()
O.varsta  # Se apelează O.__getattr__("varsta")
```

[32]: 19

[33]:
```
# Pentru alte atribute se returnează -1
O.nume
```

[33]:
```
-1
```

[34]:
```
# Inclusiv cele ce nu sunt definite în clasă
O.prenume
```

[34]:
```
-1
```

[35]:
```
# Suprascriem operatorul de setare atribut
class Persoana:
  def __setattr__(self, atribut, val):
    if atribut == 'varsta':
      self.__dict__[atribut] = val + 10 # Modificăm valoarea␣
 ↪de atribuire
    else:
      raise AttributeError(atribut + ' nu poate fi␣
 ↪modificat')
O = Persoana()
O.varsta = 19 # Se apeleaza O.__setattr__('varsta',19)
O.varsta
```

[35]:
```
29
```

[36]:
```
# Eroare
O.nume = 'Ana'
```

[36]:
```
    ------------------------------
    AttributeError    Traceback (most recent call last)
    <ipython-input-34-ea68eec40c8e> in <module>
          1 # Eroare
    ----> 2 O.nume = 'Ana'

    <ipython-input-33-0d12a5fd57ac> in __setattr__(self,␣
 ↪atribut, val)
          5       self.__dict__[atribut] = val + 10
          6     else:
    ----> 7       raise AttributeError(atribut + ' nu poate␣
 ↪fi modificat')
          8 O = Persoana()
          9 O.varsta = 19
        AttributeError: nume nu poate fi modificat
```

Modificarea reprezentărilor text ale obiectelor

- `__str__` este folosită de `print()`;
- `__repr__` este folosită de alte procese și ar trebui să afișeze o reprezentare ce poate fi utilizată la crearea unei noi instanțe a aceleiași clase.

[37]:
```
# Versiuni implicite pentru repr și str
class Persoana():
  def __init__(self,nume, varsta):
    self.nume=nume
    self.varsta=varsta

O = Persoana('Ana', 19)
print(("Reprezentarea __str__: ", O))
"Reprezentarea __repr__: ", O
```

[37]:
```
('Reprezentarea __str__: ', <__main__.Persoana object at␣
↪0x7fc101597c10>)
('Reprezentarea __repr__: ', <__main__.Persoana at␣
↪0x7fc101597c10>)
```

[38]:
```
# Modificăm __str__ și __repr__
class Persoana():
  def __init__(self,nume, varsta):
    self.nume=nume
    self.varsta=varsta
  def __str__(self):
    return 'Numele este %s.' % self.nume # User-friendly␣
↪string
  def __repr__(self):
    return 'Numele și vârsta sunt (%s,%s)' % (self.nume,␣
↪self.varsta)

# Instanțiem un obiect
O = Persoana('Ana', 19)
print(("Reprezentarea __str__:", O)) # Se apeleaza __str__
"Reprezentarea __repr__:", O # Se apeleaza __repr__
```

[38]:
```
('Reprezentarea __str__:', Numele și vârsta sunt (Ana,19))
('Reprezentarea __repr__:', Numele și vârsta sunt (Ana,19))
```

[39]:
```
# Putem apela și explicit funcțiile asociate acestor␣
 ↪operatori
str(O), repr(O)
```

[39]:
```
('Numele este Ana.', 'Numele și vârsta sunt (Ana,19)')
```

Operatori relaționali

Nu există relații implicite între obiecte. Dacă două obiecte nu sunt ==, nu înseamnă că != va fi adevărat. De aceea e util uneori să definim aceste relații. Se permite supraîncărcarea tuturor operatorilor relaționali.

[40]:
```
# Supraîncărcare operatori relaționali
class Persoana:
  def __init__(self, nume, varsta):
    self.nume=nume
    self.varsta=varsta
  # Mai mare decât
  def __gt__(self, val):
    return self.varsta > val
  # Mai mic decât
  def __lt__(self, val):
    return self.varsta < val

O = Persoana('Ana', 19)
print(O > 12) # Se apeleaza O.__gt__(12)
print(O < 12) # Se apeleaza O.__lt__(12)
```

[40]:
```
True
False
```

Operatorul de ștergere

[41]:
```
# Suprascriem operatorul de ștergere a unui obiect
class Persoana:
  def __init__(self, nume, varsta):
    self.nume=nume
    self.varsta=varsta

  def __del__(self):
    print('Persoana ' + self.nume + ' a dispărut.')
```

```
PP = Persoana('Maria', 18)
# Ștergem obiectul
del PP # Se apelează PP.__del__()
```

[41]:
```
Persoana Maria a dispărut.
```

Datorită mecanismului de funcționare al Google Colab prin care garbage collection să nu fie aplicat imediat, s-ar putea ca la rularea celulei anterioare să nu se șteargă obiectul. Pentru a ne asigura că acest lucru se întâmplă, putem crea un script cu codul anterior și să îl rulăm independent.

[42]:
```
%%writefile test_del.py
# Suprascriem operatorul de ștergere a unui obiect
class Persoana:
  def __init__(self, nume, varsta):
    self.nume=nume
    self.varsta=varsta

  def __del__(self):
    print('Persoana ' + self.nume + ' a dispărut.')

PP = Persoana('Maria', 18)
# Ștergem obiectul
del PP # Se apelează PP.__del__()
```

[42]:
```
Writing test_del.py
```

[43]:
```
!python test_del.py
```

[43]:
```
Persoana Maria a dispărut.
```

T6.2. Decoratori

Decoratorii sunt un tipar de design în Python ce permite adăugarea unor funcționalități asupra unor obiecte, fără a modifica structura obiectelor. Decoratorii pot fi definiți prin intermediul funcțiilor sau claselor și sunt aplicați asupra unor funcții sau metode. Numele decoratorului e precedat de arond, '@' și apar înaintea definirii funcției.

Există o serie de decoratori predefiniți. De exemplu, metodele statice sau de clasă pot fi specificate și prin intermediul decoratorilor:

[44]:
```
class Persoana:
  # Metodă a instanței
  def imeth(self, x):
    print([self, x])

  # Metodă statică
  @staticmethod
  def smeth(x):
    print([x])

  # Metodă de clasă
  @classmethod
  def cmeth(cls, x):
    print([cls, x])
```

Putem crea noi o funcție decorator, va trebui să creăm o funcție ce ia ca parametru o altă funcție și returnează rezultatul modificat al funcției parametru:

[45]:
```
# Funcție decorator
def litere_mari(functie):
    def modificare():
        func = functie()
```

```
        litere_mari = func.upper()
        return litere_mari
    return modificare

def salut():
    return 'salut'

# Versiunea standard de aplicare a înlănțuirii de funcții
decorate = litere_mari(salut)
decorate()
```

[45]: 'SALUT'

[46]:
```
# Versiunea cu decorator
@litere_mari
def salut():
    return 'salut'

salut()
```

[46]: 'SALUT'

Se pot aplica și mai mulți decoratori asupra aceleiași funcții:

[47]:
```
# Definim un nou decorator
def multiplicare(functie):
    def modificare():
        return functie() * 3
    return modificare

# Aplicaăm ambii decoratori asupra funcției salut()
@multiplicare
@litere_mari
def salut():
    return 'salut'
salut()
```

[47]: 'SALUTSALUTSALUT'

În cazul claselor decorator se aplică același principiu, dar va trebui să specificăm comportamentul decoratorului în cadrul metodei `__call__`. Această metodă este apelată atunci când utilizăm o instanță a clasei ca

apelator, fapt ce poate părea ciudat inițial, dar putem să asociem acest mecanism unui apel de funcție. Cu alte cuvinte, considerăm instanța ca fiind o funcție, iar la apel se execută codul definit în metoda `__call__`

[48]:
```
class A:
    def __init__(self):
        print("Apel constructor")

    def __call__(self, a, b):
        print("Apel __call__")
        print("Suma valorilor este:", a+b)

O = A() # Se apelează constructorul
O(3, 4) # Tratăm obiectul ca apelator. Se apelează __call__
```

[48]:
```
Apel constructor
Apel __call__
Suma valorilor este: 7
```

[49]:
```
# Definim o clasă decorator
class LitereMari:
    def __init__(self, functie):
        self.functie = functie

    def __call__(self):
        return self.functie().upper()

# Aplicăm decoratorul asupra funcției
@LitereMari
def salut():
    return "salut"

print(salut())
```

[49]:
```
SALUT
```

[50]:
```
# Echivalent cu:
O = LitereMari(salut)
O()
```

[50]:
```
'SALUT'
```

T6.3. Excepții

Excepțiile sunt situații ce pot să apară în rularea codului și pe care programatorul le poate anticipa. Astfel încât, acesta poate să adauge o metodă de tratare a excepției pentru ca aplicația să ruleze în continuare fără probleme sau să informeze utilizatorul într-un mod adecvat despre situația apărută.

Pentru tratarea excepțiilor în Python avem la dispoziție următoarele instrucțiuni:

- `try/except` - prinde și tratează excepții ridicate de Python sau de programator
- `try/finally` - realizează acțiuni de "curățare"/finalizare și dacă au apărut excepții și dacă nu
- `raise` - ridică o excepție manual în cod
- `assert` - ridică o excepție condiționată în cod
- `with/as` - manageri de context din Python2.6+

Sintaxa generală pentru tratarea unei excepții este:

```
 try:
   # Secventa de cod ce poate arunca o excepție
except Exceptia1:
   # Tratarea excepției1
except Exceptia2 as e:
   # Tratarea excepției2
except (Exceptia3, Excepția4):
   # Tratarea excepției3 și excepției4
except (Exceptia5, Excepția6) as e:
   # Tratarea excepției4 și excepției6
except:
   # Prinde toate excepțiile ce nu au fost tratate anterior
...
else:
```

```
    # Se execută dacă nu au apărul excepții
finally:
    # Se execută oricum la ieșirea din bloc
```

Ramurile `except` trebuie să trateze excepțiile particulare mai întâi și mai apoi cele generale. Ramura `else` se execută doar dacă nu au apărul excepții în blocul `try`.

`Finally` este rulat oricum: • a apărut o excepție ce a fost tratată; • a apărut o excepție ce nu a fost tratată; • nu a apărut nicio excepție; • a apărut o excepție în una dintre ramurile `except`.

Să vedem câteva exemple:

[51]:
```
# Forțăm o excepție de împărțire cu 0
try:
  a = 3/0
except ArithmeticError as e:
  print("Împărțire cu 0")
  # Afișăm mesajul asociat excepției
  print(e)
```

[51]:
```
Împărțire cu 0
division by zero
```

[52]:
```
# Nu generăm nicio excepție în blocul try
try:
  a = 2+3
except:
  print("A apărut o excepție")
else:
  print("Nu a apărul nicio excepție")
```

[52]:
```
Nu a apărul nicio excepție
```

[53]:
```
# Adăugăm blocul finally
try:
  a = 2+3
except:
  print("A apărut o excepție")
else:
  print("Nu a apărut nicio excepție")
```

```
finally:
  print("Afișăm oricum acest mesaj")
```

[53]:
```
Nu a apărut nicio excepție
Afișăm oricum acest mesaj
```

[54]:
```
# Creăm două excepții în blocul try
try:
  a = 3/0
  f = open("fisier_inexistent.txt")
# Este tratată doar prima excepție apărută în cod
except ArithmeticError as e:
  print(e)
except FileNotFoundError as e:
  print(e)
except:
  print("A apărut o eroare necunoscută")
```

[54]:
```
division by zero
```

Din codul anterior ar trebui să ne fie clar faptul că fiecare secvență de cod ce poate arunca o excepție va trebui să fie încadrată de un bloc `try-except` propriu.

[55]:
```
# Blocuri try imbricate
try:
  a = 3/1
  try:
    f = open("fisier_inexistent.txt")
  except FileNotFoundError as e:
    print(e)
except ArithmeticError as e:
  print(e)
except:
  print("A apărut o eroare necunoscută")
```

[55]:
```
[Errno 2] No such file or directory: 'fisier_inexistent.txt'
```

În cazul în care nu tratăm excepțiile, acestea duc la terminarea abruptă a execuției codului, iar de cele mai multe ori mesajul afișat nu este informativ pentru utilizatorul final.

[56]:
```
# Încercăm deschiderea unui fișier inexistent
f = open('fisier_inexistent.txt')

for line in f.readlines():
  print(line)
```

[56]:
```
        ---------------------------------
        FileNotFoundError Traceback (most recent call last)
        <ipython-input-54-a24e5c385791> in <module>
          1 # Încercăm deschiderea unui fișier inexistent
    ----> 2 f = open('fisier_inexistent.txt')
          3
          4 for line in f.readlines():
          5   print(line)
FileNotFoundError: [Errno 2] No such file or directory:␣
 ↪'fisier_inexistent.txt'
```

[57]:
```
# Tratăm excepția
try:
  f = open('fisier_inexistent.txt')

  for line in f.readlines():
    print(line)
except FileNotFoundError:
  print("Fisierul nu există")

print("Codul continuă să ruleze cu următoarea instrucțiune␣
 ↪după try-except")
```

[57]:
```
Fisierul nu există
Codul continuă să ruleze cu următoarea instrucțiune după␣
 ↪try-except
```

T6.3.1 Ridicarea excepțiilor

Forțarea apariției unei excepții în cod sau ridicarea excepțiilor se poate realiza folosind instrucțiunea `raise` ce poate fi urmată de:

- o instantă a unei clase excepție `raise instance`;
- o clasă excepție, se va crea automat o instanță a clasei, `raise class`;
- nimic și atunci se va ridica cea mai recentă excepție apărută, `raise`.

[58]:
```
# Ridicăm o instanță a unei clase excepție
def func():
  ie = ArithmeticError()
  raise ie

try:
  func()
except ArithmeticError as e:
  print("A apărut o excepție în cod")
```

[58]:
```
A apărut o excepție în cod
```

[59]:
```
# Ridicăm o clasă excepție, se crează automat o instanță a ei
def func():
  raise ArithmeticError()

try:
  func()
except ArithmeticError:
   print("A apărut o excepție în cod")
```

[59]:
```
A apărut o excepție în cod
```

[60]:
```
# Ridicăm cea mai recentă excepție apărută
def func():
  raise ArithmeticError()

try:
  func()
except ArithmeticError as e:
   print("A apărut o excepție în cod")
   # Cea mai recentă excepție
   raise
```

[60]:
```
A apărut o excepție în cod
      ----------------------------
      ArithmeticError Traceback (most recent call last)
      <ipython-input-58-8cdebadf3b9a> in <module>
        4
        5 try:
  ----> 6   func()
```

```
      7 except ArithmeticError as e:
      8    print("A apărut o excepție în cod")

    <ipython-input-58-8cdebadf3b9a> in func()
      1 # Ridicăm cea mai recentă excepție apărută
      2 def func():
----> 3   raise ArithmeticError()
      4
      5 try:
    ArithmeticError:
```

T6.3.2 Clase excepție definite de utilizator

În cazul în care dorim să programăm o excepție specifică, va trebui să definim o clasă ce moștenește clasa `Exception`. Clasele nou definite pot fi moștenite la rândul lor. Convenția de denumire a excepțiilor proprii este ca acestea să se termine cu stringul `Error`

[61]:
```
# Definim o excepție proprie
class ExceptiaMeaError(Exception):
  def __str__(self):
    return "Valoarea nu poate fi negativă"

def func(val):
  if val<0:
    # Ridicăm excepția proprie
    raise ExceptiaMeaError

try:
  func(-1)
except ExceptiaMeaError as e:
  print(e)
```

[62]:
```
Valoarea nu poate fi negativă
```

[62]:
```
# Excepții derivate
class ExceptiaMeaError(Exception):
  def __str__(self):
    return "Valoarea nu poate fi negativă"

# Moștenim excepția proprie
```

```
class ExceptiaMeaDerivataError(ExceptiaMeaError):
  def __str__(self):
    return "Valoarea nu poate fi mai mică decât -5"


def func(val):
  if val<-5:
    # Ridicăm excepția derivată
    raise ExceptiaMeaDerivataError
  elif val<0:
    # Ridicăm excepția de bază
    raise ExceptiaMeaError

try:
  func(-10)
except ExceptiaMeaDerivataError as e:
  print(e)


# Excepția de bază prinde și excepțiile derivate
try:
  func(-10)
except ExceptiaMeaError as e:
  print(e)
```

[62]:
```
Valoarea nu poate fi mai mică decât -5
Valoarea nu poate fi mai mică decât -5
```

Ce ar trebui încadrat de `try-except`:

- Operații ce pot eșua în general, de exemplu acces la fișiere, sockets;
- Totuși nu toate operațiile ce pot eșua ar trebui tratate de excepții, în special cele ce ar cauza rularea greșită a programului pe mai departe;
- Trebuie implementate acțiuni de finalizare prin `try-finally` pentru a garanta execuția lor;
- Uneori e utilă încadrarea unei întregi funcții într-o instrucțiune `try-except` și nu segmentarea excepției în cadrul funcției;
- A se evita utilizarea unei ramuri `except` generale (goale);
- A se evita utilizarea unor excepții foarte specifice, ci mai degrabă se utilizează clase de excepții.

T6.4. Aserțiuni

Aserțiunile sunt instrucțiuni ce verifică dacă anumite condiții din cod sunt îndeplinite. Sunt folosite mai degrabă în partea de testare (en. *debugging*) a codului. Aserțiunile mai pot fi utilizate și pentru ridicarea excepțiilor condiționate.

Sintaxa generală este:

```
assert test, mesaj
```

`mesaj` e opțional. Se ridică un `AssertionError` dacă `test` e `False`.

Aserțiunile pot fi dezactivate atunci când codul este trimis către clienții finali.

[63]:
```
# Forțăm un AssertionError
a = 3
assert a < 0
```

[63]:
```
---------------------------------
AssertionError   Traceback (most recent call last)
<ipython-input-61-b412d00636e3> in <module>
  1 # Forțăm un AssertionError
  2 a = 3
----> 3 assert a < 0
AssertionError:
```

[64]:
```
# Assert corect, nu se afișează nimic
assert a==3
```

[65]:
```
# Afișăm un mesaj asociat aserțiunii
assert a < 0, 'Valoarea lui a trebuie să fie negativă'
```

[65]:
```
        ------------------------------
        AssertionError    Traceback (most recent call last)
        <ipython-input-63-da6753282976> in <module>
          1 # Afișăm un mesaj asociat aserțiunii
    ----> 2 assert a < 0, 'Valoarea lui a trebuie să fie␣
 ↪negativă'
        AssertionError: Valoarea lui a trebuie să fie␣
 ↪negativă
```

[66]:
```
# Putem utiliza valorile variabilelor testate în mesajul␣
 ↪afișat
numar = -2
assert numar > 0, \
    f"Numar trebuie să fie mai mare decât 0, valoarea sa␣
 ↪este: {numar}"
```

[66]:
```
    ---------------------------------
    AssertionError  Traceback (most recent call last)
    <ipython-input-64-99d2e021c142> in <module>
    2 numar = -2
    3 assert numar > 0, \
----> 4  f"Numar trebuie să fie mai mare decât 0, valoarea␣
 ↪sa este: {numar}"
    AssertionError: Numar trebuie să fie mai mare decât 0,␣
 ↪valoarea sa este: -2
```

[67]:
```
numar = 3.14
assert isinstance(numar, int),\
   f"Numar trebuie să fie întreg, valoarea sa este: {numar}"
```

[67]:
```
    ---------------------------------
    AssertionError      Traceback (most recent call last)
    <ipython-input-65-67fcd610eb6f> in <module>
    1 numar = 3.14
    2 assert isinstance(numar, int),\
----> 3    f"Numar trebuie să fie întreg, valoarea sa este:␣
 ↪{numar}"
    AssertionError: Numar trebuie să fie întreg, valoarea sa␣
 ↪este: 3.14
```

T6.5. Manageri de context: with/as

Managerii de context reprezintă secvențe de cod ce pot să atribuie și să elibereze resurse în puncte specifice ale codului. Pot fi văzuți ca o alternativă simplificată a blocurilor `try-except`. Cel mai des întâlnit manager de context este instrucțiunea `with`, cu sintaxa generală:

```
with expression [as variable]:
        with-block
```

Rezultatul expresiei trebuie să implementeze așa numitul context management protocol. Expresia poate să ruleze o secvență de cod înainte și după execuția blocului `with`. Variabilei nu trebuie să îi fie atribuit rezultatul expresiei.

Context Management Protocol - funcționare

- Expresia este evaluată și rezultă un obiect de tip context manager ce trebuie să aibă asociate metodele `__enter__` și `__exit__`;
- Metoda `__enter__` este apelată, iar rezultatul returnat este atribuit clauzei `as` (dacă există);
- Se execută `with-block`;
- Dacă `with-block` ridică o excepție, metoda `__exit__` este apelată pe baza detaliilor excepției
- Dacă această metodă returnează o valoare `False`, excepția este din nou ridicată, altfel excepția este terminată. Excepția ar trebui ridicată din nou pentru a fi propagată în afara instrucțiunii `with`;
- Dacă `with-block` nu ridică o excepție, metoda `__exit__` este oricum apelată, dar parametrii trimiși către ea sunt `None`.

Cea mai des întâlnită utilizare a managerilor de context este la manipularea fișierelor și asigură faptul că la ieșirea din blocul `with/as` fișierul este închis, indiferent ce se întâmplă în blocul `with-block`:

[68]:
```
# Citire standard
f = open("fis.txt", "w")
print (f.write("Salut!"))
f.close()
# Nu e sigur că fișierul e închis dacă a apărut o excepție␣
 ↪la scriere
```

[68]: 6

[69]:
```
# Implementare corectă cu try-except-finally
f = open("fis.txt", "w")

try:
    f.write("Salut!")
except Exception as e:
    print("A apărut o excepție la scriere")
finally:
    # Ne asigurăm că fișierul e închis în orice condiții
    f.close()
```

[70]:
```
with open("fis.txt", "w") as f:
  print("Salut!")
# La ieșierea din bloc, fișierul este închis automat, chiar␣
 ↪dacă a apărut o excepție
```

[70]: Salut!

Manageri de context multipli, Python3.1+

Începând cu Python3.1, se permite utilizarea mai multor manageri de context în aceeași instrucțiune `with/as`:

```
with A() as a, B() as b:
 with-block
```

Este echivalent cu:

```
with A() as a:
 with B() as b:
   with-block
```

[71]:
```
%%writefile fisier1.txt
Linia1.1.: Salut
Linia1.2.: Ce mai faci?
```

[71]:
```
Writing fisier1.txt
```

[72]:
```
%%writefile fisier2.txt
Linia2.1.: Hello
Linia2.2.: How are you?
```

[72]:
```
Writing fisier2.txt
```

[73]:
```
with open('fisier1.txt') as f1, open('fisier.txt') as f2:
  for l1,l2 in zip(f1, f2):
          print(l1.strip(), '|', l2.strip())
```

[73]:
```
Linia1.1.: Salut | Linia2.1.: Hello
Linia1.2.: Ce mai faci? | Linia2.2.: How are you?
```

Concluzii

Acest tutorial a prezentat sintaxa și conceptele asociate programării obiectuale în Python. De asemenea, au fost introduse aspecte legate de decoratori, excepții, aserțiuni și manageri de context. Tutorialul următor va prezenta modul de lucru cu fișiere de intrare/ieșire și fișiere cu format standard (CVS, JSON, XML, etc.)

Exerciții

1. Să se definească o clasă denumită `Model` cu atributele de instanță `valoare` și `radical`. Clasa conține o metodă statică ce calculează radicalul unui număr primit ca atribut. De asemenea, clasa mai conține o metodă de clasă ce permite instanțierea unui nou obiect pornind de la o valoarea numerică oarecare și care apelează metoda statică pentru definirea atributului `radical`.

2. Să se creeze o funcție decorator ce modifică întotdeauna valoarea numerică returnată de o funcție prin rotunjire la cel mai apropiat întreg.

3. Scrieți o clasă care modelează o matrice de valori întregi. Atât dimensiunile matricii cât și tabloul bidimensional de elemente sunt atribute

pseudoprivate în clasă, accesate prin intermediul unor metode setter și getter. Includeți în clasă metode de afișare formatată a matricii, de calcul și retur a numărului de grupuri de elemente (9 valori învecinate), care nu diferă cu mai mult de 5% față de un anumit prag primit ca atribut la apelul metodei. Instanțiați clasa și testați metodele.

4. Creați propria clasă excepție ce se aruncă atunci când într-un string există caractere non-ASCII. Atașați un mesaj corespunzător excepției și scrieți un cod de testare.

5. Scrieți o aplicație care definește o clasă de verificare a unei chei de autentificare. Cheia de autentificare este de tipul: `XXXXX-XXXXX-XXXXX-XXXXX`, unde X reprezintă un caracter ce poate fi cifră sau literă. Cheia are exact 4 grupuri de caractere a câte 5 caractere fiecare, separate prin caracterul '-'. De asemenea, numărul de cifre trebuie să fie mai mare decât numărul de litere, iar numărul de litere nu poate să fie 0. În cazul în care nu este îndeplinită cel puțin o condiție din cele menționate anterior, se aruncă o excepție proprie cu mesajul: "Cheie de autentificare incorectă!". Toate verificările se fac în momentul instanțierii unui nou obiect din clasa definită.

Referințe suplimentare

- Decoratorul `@property` - online.

T7 Lucrul cu fișiere

T7.1. Fișiere

În tutorialele anterioare am văzut deja o serie de exemple de citire a datelor din fișiere. În cadrul tutorialului curent vom extinde aceste metode cu metode de scriere, precum și cu metode de prelucrare a unor tipuri de fișiere standard, precum CSV, JSON sau XML.

Începem tutorialul cu citirea și scrierea fișierelor text simple. Pentru citire/scriere avem la dispoziție funcția built-in `open()` cu formatul complet:

```
f = open(file, mode='r', buffering=- 1, encoding=None,
         errors=None, newline=None, closefd=True, opener=None)
```

Modul de deschidere al fișierului ne va furniza și operațiile pe care le putem efectua prin intermediul obiectului asociat: citire și/sau scriere.

Pentru a putea testa citirea, vom crea mai întâi un fișier folosind acțiunile magice din Google Colab:

[1]:
```
%%writefile input.txt
Salut!
Ce mai faci?
```

[1]:
```
Writing input.txt
```

Metodele asociate citirii pentru un obiect de tip fișier sunt:

- `read()` - citește tot conținutul
- `readline()` - citește linie cu linie

Și returnează șirul de caractere citit.

[2]:
```
# Citim întreg fișierul ca string
f = open('input.txt', 'rt')
```

```
data = f.read()
data
```

[2]:
```
'Salut!\nCe mai faci?\n'
```

[3]:
```
# E recomandat să folsim manageri de context pentru␣
 ↪prelucrarea fișierelor
with open('input.txt', 'rt') as f:
  data = f.read()
print (data)
```

[3]:
```
Salut!
Ce mai faci?
```

[4]:
```
# Nu este nevoie să specificăm explicit modul de citire și␣
 ↪cel text
# Acestea sunt implicite pentru funcția open()
with open('input.txt') as f:
  data = f.read()
print (data)
```

[4]:
```
Salut!
Ce mai faci?
```

[5]:
```
# Citim fișierul linie cu line
with open('input.txt') as f:
  for line in f.readlines():
    print (line)
```

[5]:
```
Salut!

Ce mai faci?
```

[6]:
```
# Același rezultat poate fi obținut parcugând obiectul fișier
with open('input.txt') as f:
  for line in f:
    print (line) # Separatorul de rând (\n) e conținut în␣
 ↪linia citită
```

[6]:
```
Salut!

Ce mai faci?
```

Pentru a putea scrie într-un fișier folosim aceeași funcție `open()`, dar specificând modul `w`. Metodele asociate obiectului fișier pentru scriere sunt:

- `write()` - scrie textul ca atare
- `writeline()` - adăugă un rând nou (\n) după scrierea textului

Și iau ca argument de intrare un șir de caractere.

[7]:
```
# Scriem un string într-un fișier
s = "Ana are mere.\nȘi pere."
with open('output.txt', 'wt') as f:
  f.write(s)
```

[8]:
```
# Putem realiza scrierea și cu ajutorul funcției print()
with open('output.txt', 'w') as f:
  print(s, file=f)
```

Putem verifica scrierea fișierului din zona de fișiere a mediului Colab. Ar trebui să apară un nou fișier denumit `output.txt`.

Dacă dorim să scriem într-un fișier doar dacă acesta nu există, putem folosi modul `x`. În acest mod, vom primi o eroare de tip `FileExistsError` dacă fișierul există deja. Implicit, în modul `w` fișierul se crează dacă nu există și se suprascrie dacă există.

Dacă rulați celula de mai jos de 2 ori, veți primi o eroare:

[9]:
```
with open('test.txt', 'x') as f:
  f.write('Salut')
```

Fișiere binare

Pentru fișiere binare avem la dispoziție modul `b` al funcției `open()`:

[10]:
```
# Scriem un fisier binar
with open('output.bin', 'wb') as f:
  f.write(b'Salut')

# Citim conținutul său
```

```
with open('output.bin', 'rb') as f:
  data = f.read()
data
```

[10]: b'Salut'

Date binare (string vs. byte)

Citirea din fișiere se poate face fie prin date de tip string, fie la nivel de byte, ce poate fi ulterior interpretat ca fiind un caracter (ASCII).

[11]:
```
t = 'Pe luni!'
for c in t:
  print(c)

type(c)
```

[11]:
```
P
e

l
u
n
i
!

str
```

[12]:
```
# Byte string
b = b'Pe luni!'
for c in b:
  print(c)

type(c)
```

[12]:
```
80
101
32
108
117
110
105
```

```
33
```

```
int
```

T7.1.1 Manipulare căi

În lucrul cu fișiere este necesară și manipularea căilor către acestea. În Python avem la dispoziție modulul `os.path` ce are implementate o serie de metode utile pentru crearea, segmentare, determinarea căilor și a tipului fișierelor sau directoarelor:

[13]:
```
# Determinăm numele fișierului din cale
import os
path = '/Users/adriana/Data/data.csv'
os.path.basename(path)
```

[13]:
```
'data.csv'
```

[14]:
```
# Determinăm calea de directoare
os.path.dirname(path)
```

[14]:
```
'/Users/adriana/Data'
```

[15]:
```
# Creăm o nouă cale prin combinarea mai multor directoare și␣
 ↪un fișier
os.path.join('tmp', 'data', os.path.basename(path))
```

[15]:
```
'tmp/data/data.csv'
```

[16]:
```
# Determinăm calea completă către directorul utilizatorului
path = '~/Data/data.csv'
os.path.expanduser(path)
```

[16]:
```
'/root/Data/data.csv'
```

[17]:
```
# Detemină̆m numele și extensia fișierului
path = 'data.csv'
os.path.splitext(path)
```

[17]:
```
('data', '.csv')
```

[18]:
```
# Verificăm existența unui fișier
os.path.exists('/content/output.txt')
```

[18]:
```
True
```

[19]:
```
# Verificăm existența unui director
os.path.exists('/content/')
```

[19]:
```
True
```

[20]:
```
# Verificăm dacă este fișier
os.path.isfile('/content/output.txt')
```

[20]:
```
True
```

[21]:
```
# Verificăm dacă este director
os.path.isdir('/content/')
```

[21]:
```
True
```

Listare directoare

Pentru a determina conținutul unei căi ne putem folosi de alte funcții ale pachetului built-in os

[22]:
```
# Listăm conținutul directorului curent
os.listdir('.')
```

[22]:
```
['.config', 'test.txt', 'output.bin', 'output.txt', 'input.
↪txt', 'sample_data']
```

[23]:
```
# Listăm conținutul unui director oarecare
os.listdir('/content/sample_data')
```

[23]:
```
['anscombe.json',
 'README.md',
 'california_housing_train.csv',
 'california_housing_test.csv',
 'mnist_test.csv',
 'mnist_train_small.csv']
```

[24]:
```
# Creăm o listă cu directoarele din /content/
[name for name in os.listdir('/content/') if os.path.
 ↪isdir(os.path.join('/content/', name))]
```

[24]:
```
['.config', 'sample_data']
```

[25]:
```
# Creăm o listă cu fișierele cu extensia txt din /content/
[name for name in  os.listdir('/content/') if name.
 ↪endswith('.txt')]
```

[25]:
```
['test.txt', 'output.txt', 'input.txt']
```

T7.1.2 Mutare/copiere fișiere

În cazul în care dorim să manipulăm fizic directoare și fișiere pe disc, putem utiliza pachetul built-in `shutil`:

[26]:
```
import shutil
# Copiem un fișier sub un alt nume
shutil.copy('output.txt', 'newtext.txt')
os.listdir()
```

[26]:
```
['.config',
 'newtext.txt',
 'test.txt',
 'output.bin',
 'output.txt',
 'input.txt',
 'sample_data']
```

[27]:
```
# Copiem un director
shutil.copytree('sample_data', 'new_data')
os.listdir()
```

[27]:
```
['.config',
 'new_data',
 'newtext.txt',
 'test.txt',
 'output.bin',
 'output.txt',
 'input.txt',
```

```
 'sample_data']
```

[28]:
```
# Mutăm un fișier/director cu suprascriere fără avertisment
# Dacă fișierul output.txt există va fi suprascris
shutil.move('newtext.txt', 'output.txt')
os.listdir()
```

[28]:
```
['.config',
 'new_data',
 'test.txt',
 'output.bin',
 'output.txt',
 'input.txt',
 'sample_data']
```

T7.1.3 Fișiere/directoare temporare

În anumite aplicații este util să avem o serie de fișiere și directoare ce există doar atât timp cât aplicația rulează, ele neavând o existență persistentă pe disc. În acest caz putem folosi modulul `tempfile` pentru a le crea și manipula.

Fișiere temporare anonime

[29]:
```
# Creăm un fișier temporar anonim în care scriem date
from tempfile import TemporaryFile
with TemporaryFile('w+t') as f:
  # Sciere/citire
  f.write('Salut!\n')
  f.write('Test\n')
  # Revenim la începutul fișierului
  f.seek(0)
  # Citim conținutul
  data = f.read()
  print(data)

# La ieșirea din context manager fișierul e distrus
```

[29]:
```
Salut!
Test
```

Fișiere temporare denumite

[30]:
```
# Creăm un fișier temporar denumit
from tempfile import NamedTemporaryFile
with NamedTemporaryFile('w+t') as f:
  # Numele este aleator
  print('Fișierul denumit este:', f.name)

# Fișierul este distrus
```

[30]:
```
Fișierul denumit este: /tmp/tmpk0p9krwt
```

Directoare temporare

[31]:
```
# Creăm un director temporar anonim și un fișer test in␣
 ↪interiorul său
from tempfile import TemporaryDirectory
with TemporaryDirectory() as dirname:
  print('Directorul este:', dirname)
  # Creăm un fișier în director, scriem și citim
  with open(os.path.join(dirname,'test'), 'w+') as f:
    f.write("Salut!")
    f.seek(0)
    print(f.read())

# Directorul este distrus
```

[31]:
```
Directorul este: /tmp/tmpwd485g53
Salut!
```

T7.2. Serializarea obiectelor

Atunci când dorim stocarea persistentă a atributelor/conținutul unor obiecte mai complexe din aplicația noastră, este util ca acestea să poată fi scrise pe disc în mod direct și mai apoi citite. Putem realiza acest lucru în Python, folosind modulul `pickle`. Dar e important de menționat faptul că s-ar putea ca diferitele versiune de serializare să nu fie compatibile, astfel încât e important să reținem și versiunea cu care obiectele au fost serializate.

[32]:
```
# Serializarea unei liste
import pickle
obiect = [1,2,3,4,5,6,7,8]

with open('obiect_serializat.pkl', 'wb') as f:
  # Serializăm obiectul
  pickle.dump(obiect, f)
```

[33]:
```
# Citirea obiectului din fișier
with  open('obiect_serializat.pkl', 'rb') as f:
  obiect = pickle.load(f)
obiect
```

[33]:
```
[1, 2, 3, 4, 5, 6, 7, 8]
```

O alternativă la serializarea către un fișiere, este serializarea către string:

[34]:
```
# Serializare către string
s = pickle.dumps(obiect)
print (s)
# Restaurare din string
obiect = pickle.loads(s)
obiect
```

```
[34]: b'\x80\x03]q\x00(K\x01K\x02K\x03K\x04K\x05K\x06K\x07K\x08e.'

      [1, 2, 3, 4, 5, 6, 7, 8]
```

T7.3. Fișiere speciale

În aplicațiile practice, de cele mai multe ori vom folosi fișiere cu un format predefinit pentru a fi mai ușoară partajarea informațiilor stocate în acestea și în afara aplicației. Cele mai des întâlnite formate de fișiere sunt CSV, JSON și XML, descrise în continuare.

Pe de altă parte, intern aplicației, dar tot respectând un format predefinit, va trebui să utilizăm și fișiere pentru scrierea log-urilor aplicației sau stocarea informațiilor de configurare a aplicației. Și aceste tipuri de fișiere sunt descrise în secțiunile următoare.

T7.3.1 Fișiere CSV

Fișierele CSV - Comma Separated Values sunt de fapt fișiere text ce au un format tabular în care câmpurile sunt separate prin virgulă. Prima linie poate fi utilizată pentru a specifica antetul (en. *header*) tabelului:

[35]:
```
%%writefile stocks.csv
Symbol,Price,Date,Time,Change,Volume
"AA",39.48,"6/11/2007","9:36am",-0.18,181800
"AIG",71.38,"6/11/2007","9:36am",-0.15,195500
"AXP",62.58,"6/11/2007","9:36am",-0.46,935000
"BA",98.31,"6/11/2007","9:36am",+0.12,104800
"C",53.08,"6/11/2007","9:36am",-0.25,360900
"CAT",78.29,"6/11/2007","9:36am",-0.23,225400
```

[35]:
```
Writing stocks.csv
```

Pentru a lucra eficient cu acest tip de fișiere, putem utiliza pachetul built-in `csv`:

[36]:
```
# Citim antetul fișierului csv
import csv
```

```
f = open('stocks.csv')
f_csv = csv.reader(f)
headers = next(f_csv)
headers
```

[36]:
```
['Symbol', 'Price', 'Date', 'Time', 'Change', 'Volume']
```

[37]:
```
# Citim pe rând liniile din fișier
for row in f_csv:
  print(row)
```

[37]:
```
['AA', '39.48', '6/11/2007', '9:36am', '-0.18', '181800']
['AIG', '71.38', '6/11/2007', '9:36am', '-0.15', '195500']
['AXP', '62.58', '6/11/2007', '9:36am', '-0.46', '935000']
['BA', '98.31', '6/11/2007', '9:36am', '+0.12', '104800']
['C', '53.08', '6/11/2007', '9:36am', '-0.25', '360900']
['CAT', '78.29', '6/11/2007', '9:36am', '-0.23', '225400']
```

O metodă mai ușoară de a lucra cu datele dintr-un fișier CSV este dacă pentru fiecare rând din tabel putem extrage un anumit câmp în funcție de antetul fișierului. Pentru aceasta avem la dispoziție clasa `DictReader()` a modulului `csv`. Prin intermediul unui obiect de tip `DictReader()` vom putem mai apoi să referim elementele unui rând sub formă de dicționar:

[38]:
```
# Creăm un DictReader() din conținutul fișierului
import csv
with open('stocks.csv') as f:
  f_csv = csv.DictReader(f)
  rows = [row for row in f_csv]

print(rows[3]['Symbol'], rows[2]['Time'])
```

[38]:
```
BA 9:36am
```

Scrierea unor date tabulare în fișiere de tip CSV se face în mod similar cu scrierea într-un fișier text simplu:

[39]:
```
# Scrierea unui fișier csv
headers = ['Symbol','Price','Date','Time','Change','Volume']
rows = [('AA', 39.48, '6/11/2007', '9:36am', -0.18, 181800),
  ('AIG', 71.38, '6/11/2007', '9:36am', -0.15, 195500),
```

```
  ('AXP', 62.58, '6/11/2007', '9:36am', -0.46, 935000),
]

with open('stocks_out.csv','w') as f:
  f_csv = csv.writer(f)
  f_csv.writerow(headers) # Scriem antetul
  f_csv.writerows(rows)   # Scriem rândurile
```

T7.3.2 Fișiere JSON

Fișierele JSON - Java Script Object Notation reprezintă un format de fișier extrem de des utilizat în comunicarea cu API-uri: transmitere de interogări (en. *requests*) și recepție răspuns. Pentru a manipula astfel de fișiere, în Python avem la dispoziție modulul `json`. Formatul fișierelor și modul de lucru cu ele în Python este similară cu lucrul cu dicționare.

[40]:
```
%%writefile data.json
{"studenti":[
    {"prenume": "Maria", "nume": "Popescu", "varsta": 19},
    {"prenume": "Ana", "nume": "Ionescu", "varsta": 20}],
  "profesori":[
    {"prenume": "Ion", "nume": "Pop", "varsta": 42},
    {"prenume": "Mihai", "nume": "Vaile", "varsta": 35}]

}
```

[40]:
```
Writing data.json
```

[41]:
```
# Citim fișierul json
import json
with open('data.json', 'r') as f:
  data = json.load(f)

data
```

[41]:
```
{'studenti': [{'prenume': 'Maria', 'nume': 'Popescu',␣
 ↪'varsta': 19},
  {'prenume': 'Ana', 'nume': 'Ionescu', 'varsta': 20}],
 'profesori': [{'prenume': 'Ion', 'nume': 'Pop', 'varsta':␣
 ↪42},
  {'prenume': 'Mihai', 'nume': 'Vaile', 'varsta': 35}]}
```

[42]:
```
# Accesăm valoarea elementele din câmpul studenți
data['studenti']
```

[42]:
```
[{'prenume': 'Maria', 'nume': 'Popescu', 'varsta': 19},
 {'prenume': 'Ana', 'nume': 'Ionescu', 'varsta': 20}]
```

[43]:
```
# Accesăm primul element din câmpul studenți
data['studenti'][0]
```

[43]:
```
{'prenume': 'Maria', 'nume': 'Popescu', 'varsta': 19}
```

[44]:
```
# Creare date json dintr-un dicționare
data = {
  'prenume' : 'Maria',
  'nume' : 'Popescu',
  'varsta' : 19
}

# Creăm un json string din dicționarul anterior
json_str = json.dumps(data)

# Scriem în fișier
with open('data_out.json', 'w') as f:
  json.dump(data, f)
```

Apel API

[45]:
```
# Citim datele dintr-un fișier JSON online generat de un API
from urllib.request import urlopen
import json
from pprint import pprint

u = urlopen('https://catfact.ninja/fact')
rasp = json.loads(u.read().decode('utf-8'))
pprint(rasp['fact'])
```

[45]:
```
('Some Siamese cats appear cross-eyed because the nerves␣
 ↪from the left side of the brain go to mostly the right eye␣
 ↪and the nerves from the right side of the brain go mostly␣
 ↪to the left eye. This causes some double vision, which the␣
 ↪cat tries to correct by ‘‘crossing’’ its eyes.')
```

T7.3.3 Fișiere XML

Fișierele XML (Extensible Markup Language) sunt din punct de vedere istoric, print primele formate de fișiere cu structură standard, ușor de citit atât de oameni, cât și de calculatoare. Fișierele au o structură ierarhică bazată pe taguri și atribute.

Pentru a parsa fișiere XML în Python avem la dispoziție mai multe module, precum `xml` sau `beautifulsoup`. Ca în cazul fișierelor JSON, trebuie să cunoaștem structura și tag-urile utilizate de către fișierul XML sau schema fișierului.

[46]:
```
# Parsăm un fișier XML online
from urllib.request import urlopen
from xml.etree.ElementTree import parse

u = urlopen('http://planet.python.org/rss20.xml')
doc = parse(u)
# Extragem tag-urile de interes
for item in doc.iterfind('channel/item'):
  title = item.findtext('title')
  date = item.findtext('pubDate')
  print(title, date)
```

[46]:
```
Python for Beginners: Delete Attribute From an Object in␣
↪Python Wed, 24 Aug 2022 13:00:00+0000
John Ludhi/nbshare.io: Join or Merge Lists In Python Wed, 24␣
↪Aug 2022 10:38:18+0000
PyCoder's Weekly: Issue #539 (Aug. 23, 2022) Tue, 23 Aug␣
↪2022 19:30:00 +0000
Real Python: Building a URL Shortener With FastAPI and␣
↪Python Tue, 23 Aug 2022 14:00:00 +0000
The Digital Cat: Data Partitioning and Consistent Hashing␣
↪Tue, 23 Aug 2022 12:00:00 +0000
```

Creare XML din dicționar

[47]:
```
# Creăm un fișier XML pe baza unui dicționar
from xml.etree.ElementTree import Element, tostring
import xml.etree.ElementTree as ET

def dict_to_xml(tag, d):
```

```
  elem = Element(tag)
  for key, val in d.items():
    child = Element(key)
    child.text = val
    elem.append(child)
  return elem

s = {'prenume': 'Maria', 'nume': 'Popescu', 'varsta':'19'}
e = dict_to_xml('date', s)

print(tostring(e))
```

[47]:
```
b'<date><prenume>Maria</prenume><nume>Popescu</
 ↪nume><varsta>19</varsta></date>'
```

T7.3.4 Fișiere de logging

Pentru orice aplicație cu utilizatori multipli și care are nevoie să fie disponibilă în mod continuu este important să se salveze log-uri ale acesteia și eventualele erori ce pot să apară. În Python, avem la dispoziție pachetul `logging` pentru această facilitate:

[48]:
```
import logging
# Ne asigurăm că nu avem stabilite alte căi pentru loguri
for handler in logging.root.handlers[:]:
    logging.root.removeHandler(handler)

# Configurăm logger-ul pentru a scrie în fișierul app.log
# informația din acest fișier este scrisă în continuare fără␣
 ↪a șterge info existent
# și stabilim nivelul de salvare a logurilor la cel mai mic,␣
 ↪DEBUG
logging.basicConfig(filename='app.log', level=logging.DEBUG)
```

[49]:
```
# Salvăm logguri
logging.critical('Nu se poate accesa aplicația')
logging.error('Nu am găsit fișierul')
logging.warning('Feature deprecated')
filename = "input.txt"
logging.info('S-a deschis fișierul %s', filename)
logging.debug('Am ajuns aici!')
```

[50]:
```
# Afișăm conținutul
with open('app.log') as f:
  print(f.read())
```

[50]:
```
CRITICAL:root:Nu se poate accesa aplicația
ERROR:root:Nu am găsit fișierul
WARNING:root:Feature deprecated
INFO:root:S-a deschis fișierul input.txt
DEBUG:root:Am ajuns aici!
```

Dacă rerulați celulele de mai sus veți observa că se vor scrie mesajele de log unul după altul.

T7.3.5 Fișiere de configurare

Fișierele de configurare (.ini) reprezintă fișiere text simple cu o serie de demarcaje standard în interiorul cărora se pot realiza adaptări ale anumitor parametri ai aplicației, fără a fi nevoie de o intervenție în cod. De cele mai multe ori, aceste fișiere sunt modificate automat la instalare cu date privind, de exemplu, calea către aplicație și dependențele acesteia, sistemul de operare, licență, etc.

Fișierele de configurare nu au alt scop decât să permită aplicației să ruleze cât mai bine pe mașina clientului și sunt citite de obicei la fiecare lansare a aplicației.

Dacă se folosesc astfel de fișiere pentru aplicația dezvoltată, este nevoie ca informația din acestea să fie adusă în cod. Acest lucru se poate face în Python folosind pachetul `configparser`. Creăm mai întâi un astfel de fișier și îl vom parsa ulterior:

[51]:
```
%%writefile config.ini
; config.ini
[installation]
library=%(prefix)s/lib
include=%(prefix)s/include
bin=%(prefix)s/bin
prefix=/usr/local
[debug]
log_errors=true
show_warnings=False
```

```
[server]
port: 8080
nworkers: 32
pid-file=/tmp/spam.pid
root=/www/root
```

[51]:
```
Writing config.ini
```

[52]:
```
# Parsăm fișierul creat anterior
from configparser import ConfigParser
cfg = ConfigParser()
cfg.read('config.ini')
```

[52]:
```
['config.ini']
```

[53]:
```
# Determinăm secțiunile acestui fișier
cfg.sections()
```

[53]:
```
['installation', 'debug', 'server']
```

[54]:
```
# Extragem din sectiunea installation, valoarea parametrului␣
 ↪library
cfg.get('installation','library')
```

[54]:
```
'/usr/local/lib'
```

[55]:
```
# Extragem din sectiunea debug valoarea parametrului␣
 ↪log_errors
cfg.getboolean('debug','log_errors')
```

[55]:
```
True
```

În anumite cazuri este nevoie ca aceste fișiere de configurare să poată fi modificate din aplicație, iar pentru aceasta avem la dispoziția funcția `set()`:

[56]:
```
# Modificăm parametrul port din secțiunea server
cfg.set('server','port','9000')
```

[57]:
```
# Modificăm parametrul log_errors din secțiunea debug
cfg.set('debug','log_errors','False')
```

[58]:
```
# Afișăm modificările în stdout
import sys
cfg.write(sys.stdout)
```

[58]:
```
[installation]
library = %(prefix)s/lib
include = %(prefix)s/include
bin = %(prefix)s/bin
prefix = /usr/local

[debug]
log_errors = False
show_warnings = False

[server]
port = 9000
nworkers = 32
pid-file = /tmp/spam.pid
root = /www/root
```

T7.4. Lucrul cu baze de date

Deși Python nu este un limbaj recunoscut pentru eficiența lucrului cu baze de date, se pot realiza conexiuni la baze de date de tip SQLite folosind pachetul `sqlite3`:

[59]:
```
import sqlite3
# Ne conectăm la baza de date și creăm cursorul
db = sqlite3.connect('database.db')
c = db.cursor()
# Creăm un nou tabel
c.execute('create table exemplu (nume, prenume, nota)')
db.commit()
```

[60]:
```
# Definim datele pe care dorim să le introducem în tabel
values = [
  ('Pop', 'Ionut', 10),
  ('Popescu', 'Maria', 9),
  ('Ionescu', 'George', 9),
  ('Ivan', 'Elena', 10),
]

# Introducem valorile în tabel
c.executemany('insert into exemplu values (?,?,?)', values)
db.commit()
```

[61]:
```
# Extragem toate rândurile din tabel
for row in db.execute('select * from exemplu'):
  print(row)
```

[61]:
```
('Pop', 'Ionut', 10)
('Popescu', 'Maria', 9)
('Ionescu', 'George', 9)
```

```
('Ivan', 'Elena', 10)
```

[62]:
```
# Extragem rândurile din table în care nota este mai mare␣
 ↪decât 9
min_nota = 9
for row in db.execute('select * from exemplu where nota > ?
 ↪',(min_nota,)):
  print(row)
```

[62]:
```
('Pop', 'Ionut', 10)
('Ivan', 'Elena', 10)
```

T7.5. Expresii regulare (regex)

Expresiile regulare sunt un mecanism extrem de puternic de identificare a tiparelor în șiruri de caractere. Au o sintaxă specifică prin intermediul căreia se pot specifica de exemplu seturi de caractere, numărul de caractere căutat, moduri de reprezentare a caracterelor, etc. Mai multe informații puteți găsi la acest link.

În Python, putem folosi expresii regulare prin pachetul `re`:

[63]:
```
import re

# Identificăm secvența "exemplu:" urmată de oricare 3␣
 ↪caractere
str = 'Un exemplu:ana!!'
match = re.search(r'exemplu:\w\w\w', str)
# Dacă s-a găsit secvența va fi disponibilă ca rezultat al␣
 ↪metodei group()
if match:
  print('Am găsit:', match.group())
else:
  print('Nu am găsit șirul de caractere.')
```

[63]:
```
Am găsit: exemplu:ana
```

[64]:
```
# Identificăm grupul de litere "nnn"
match = re.search(r'nnn', 'annna')
match.group()
```

[64]:
```
'nnn'
```

[65]:
```
# Identificăm oricare două caractere urmate de litera g
match = re.search(r'..a', 'annna')
```

```
match.group()
```

[65]: 'nna'

[66]:
```
# Identificăm prima cifră din șir (dacă există)
match = re.search(r'[0-9]', 'a123n456a')
match.group()
```

[66]: '1'

[67]:
```
# Identificăm toate cifrele
re.findall(r'[0-9]', 'a123n456a')
```

[67]: ['1', '2', '3', '4', '5', '6']

[68]:
```
## Identificăm caracterul a urmat de oricâte litere n␣
 ↪consecutive (minim 1)
match = re.search(r'an+', 'annna')
match.group()
```

[68]: 'annn'

[69]:
```
# Căutăm adrese de e-mail
str = 'Adresa de e-mail adriana@utcluj.ro '
match = re.search(r'[a-z-]+@[a-z\.]+', str)
match.group()
```

[70]: 'adriana@utcluj.ro'

Concluzii

În acest tutorial am încercat să introducem cât mai multe detalii esențiale ale utilizării instrucțiunilor de bază în limbajul Python. În tutorialul următor vom extinde utilizarea acestor instrucțiuni pentru crearea funcțiilor și a modulelor.

Exerciții

1. Creați un fișier ce conține informații legate de vreme sub forma: localitate, temperatură_medie, precipitații_medii. Citiți informațiile din acesta și afișați doar localitățile pentru care temperatura este mai

mare decât 20 de grade. Scrieți în alt fișier localitățile pentru care precipitațiile medii sunt mai mici decât 10 l/m2.

2. Determinați lista completă de directoare și fișiere din directorul curent. Afișați-o ordonată alfabetic.
3. Folosiți un fișier temporar în care să scrieți pe rând valorile șirului Fibonacci. Folosiți o valoare mare pentru numărul de elemente. Citiți fișierul și afișați elementele din șirul Fibonacci în ordine inversă.
4. Scrieți o expresie regulară ce identifică adrese web de forma `www.exemplu.com/index.html`.
5. În Google Colab, în mașina virtuală curentă există un director `sample_data/` ce conține fișierul `california_housing_train.csv`. Citiți conținutul fișierului și afișați numărul de intrări din acesta.
6. Căutați online un API ce răspunde în format JSON. Apelați API-ul și afișați rezultatul parsat.

Referințe suplimentare

- Modulul `pandas` pentru lucrul cu date tabulare - online.
- Acces la resurse web prin modulul `urllib` - online.