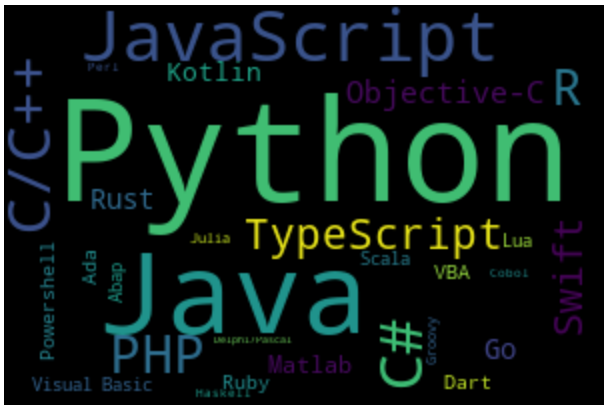
Curs 1: Introducere in Python, cu exemple

Bibliografie

1. Hans Petter Langtangen, *A Primer on Scientific Programming with Python*, Springer-Verlag, 2016
2. Charles Severance, *Python for Everybody (PY4E)*
3. Al Sweigart, *Automate the Boring Stuff with Python*
4. *Python 3 Patterns, Recipes, and Idioms*
5. Leonardo Giordani, *Clean Architectures in Python*, Editia a doua
6. Allen B. Downey, *Think Python, 3rd edition*, O'Reilly Media
7. Learn to think computationally and write programs to tackle useful problems
8. Zed A. Shaw, Learn Python the Hard Way, Addison-Wesley Professional, 2017
9. Google's Python Class

Limbajul Python

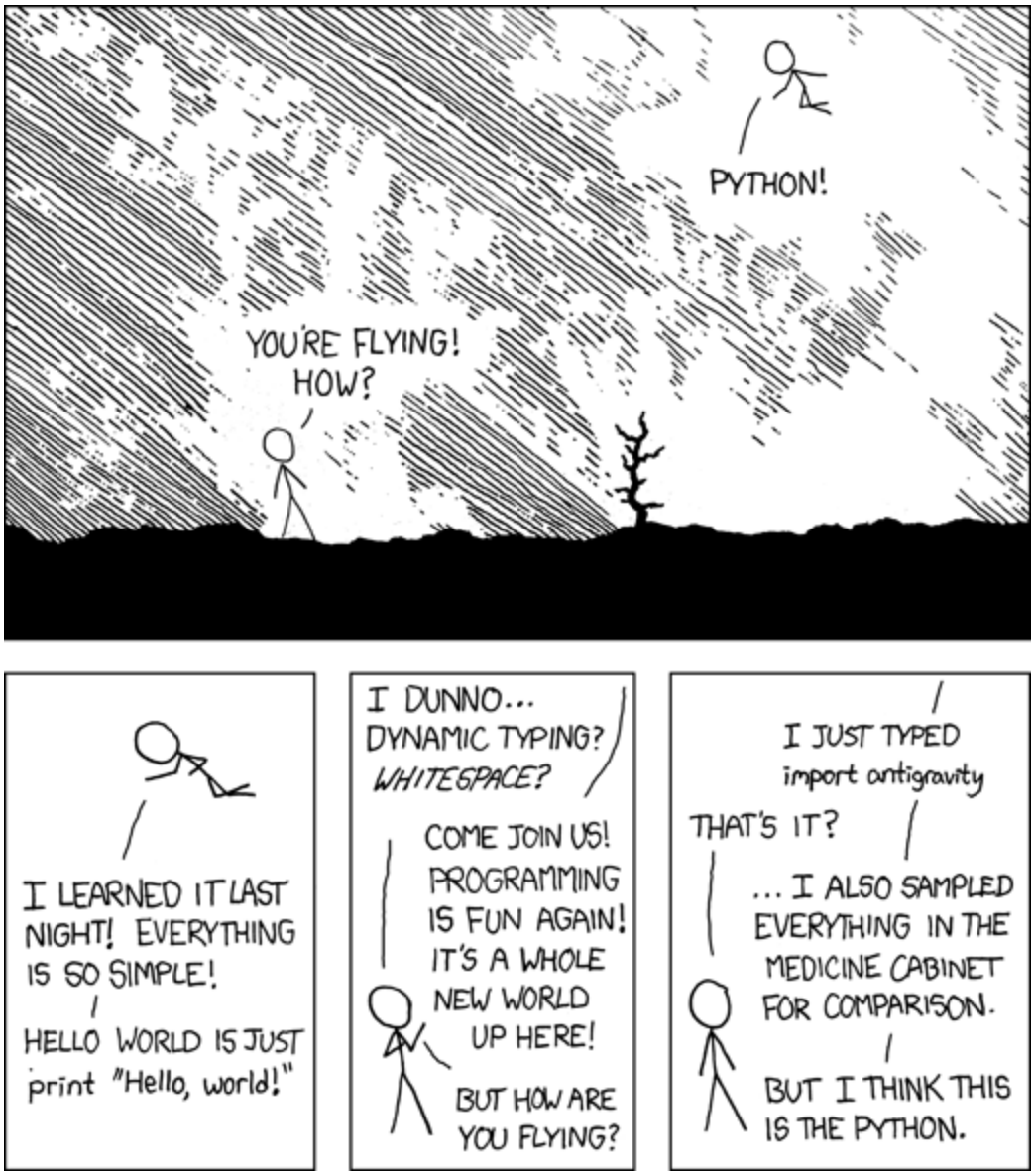
- Limbaj open source, gratuit
- In septembrie 2024: primul limbaj ca popularitate in TIOBE Index, devansand C, C++, Java
- In aceeasi luna, indexul PYPL raporteaza Python pe locul al intai



(Sursa: http://pypl.github.io/PYPL.html, septembrie 2024)

- Versiunea stabila actuala de Python este 3.12.6.; Python 2.x se considera depasit si e trecut pe linie moarta
- Se poate instala manual sau folosind o distributie de tipul Anaconda
  - A se vedea despre distributiile Anaconda, Miniconda, Anaconda Enterprise
  - Exista o lista consistenta de distributii de Python

- Python este limbaj interpretat si netipizat
- Permite programare imperativa, orientata pe obiect, functionala, metaprogramare
- Exista o paleta larga de biblioteci deja implementate pentru Python, usurand considerabil dezvoltarea de prototipuri sau chiar de aplicatii comerciale larg raspandite.



(Sursa: https://xkcd.com/353/)

- Peste 500000 de pachete dezvoltate
- Python este in adoptare crescanda pentru proiecte comerciale:
  - Can Your Enterprise choose Python for Software Development?
  - Full Stack Python
  - Which Internet companies use Python?
  - Who uses Python?
  - Is Python a Good Choice for Entreprise Projects?

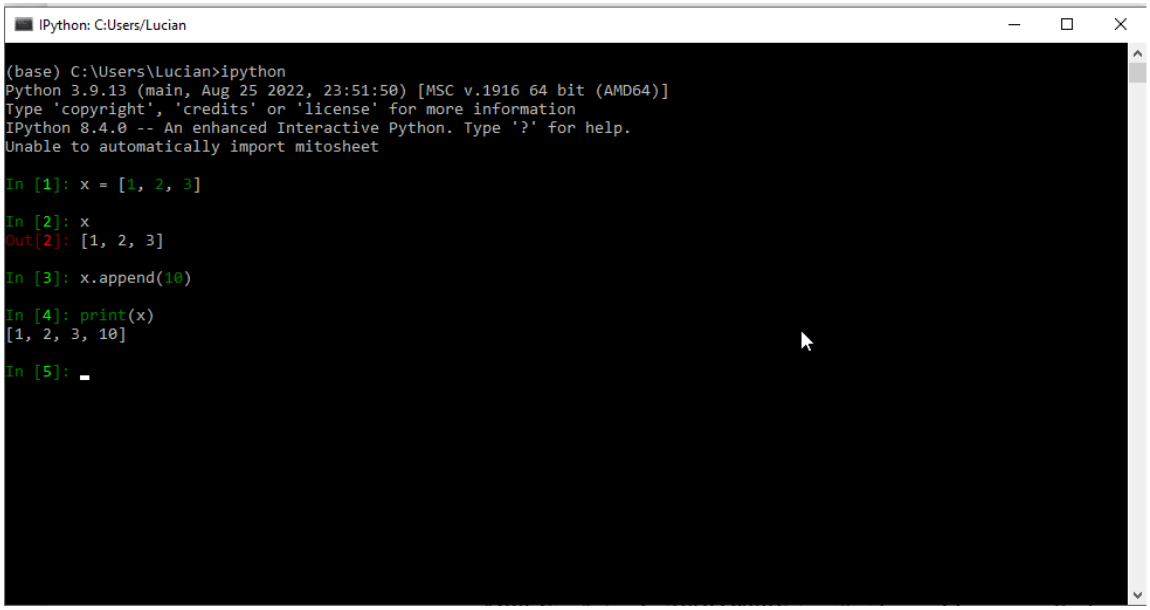
De ce Python?

Iata cateva motive solide pentru a utiliza Python, conform The unexpected effectiveness of Python in science:

1. Limbajul vine "cu toate bateriile incluse": multitudinea de biblioteci il face perfect pentru o gama larga de proiecte. Cercetatorii care il folosesc activeaza in astronomie, fizica, bioinformatica, chimie, stiinte cognitive etc. - si pentru toate aceste domenii exista biblioteci care simplifica mult dezvoltarea de prototipuri. Prin comparatie, limbaje traditionale precum C/C++/Java/Python vin cu mai putin suport disponibil, iar uneori includerea unui pachet/biblioteci poate fi o experienta consumatoare de timp.
2. Capacitatea de interoperare cu alte limbaje este iarasi un plus. Exista punti de comunicare (bridges) care permit apelul de cod Python din alte limbaje sau invers. Desigur, aceasta facilitate se regaseste si in alte limbaje.
3. Natura dinamica a limbajului poate fi inteligent speculata: o functie poate sa preia o colectie de valori organizata in diverse moduri si cateva linii de cod functioneaza la fel de bine peste toate acestea.
4. Python incurajeaza un stil de lucru experimental, via Read, Evaluate, Print, and Loop (REPL): poate e nevoie de cateva incercari pentru a ajunge la instructiunile potrivite pentru o anumita functionalitate. Incercarea unei alte instructiuni nu necesita recompilarea sau rerularea codului anterior, ci vine in completare - pana cand ajungi la ce doresti sa obtii.

Moduri de utilizare

1. Codul poate fi scris intr-un fisier text cu extensia py. Lansarea codului se face cu comanda `python nume_script.py`
2. Se poate folosi interpretorul python sau ipython, in care modul de lucru este REPL. REPL permite o prototipizare rapida si evolutia codului in mod incremental. Interpretorul ipython este o varianta imbunatatita a interpretorului python.



Mai multe detalii pentru lucrul cu interpretorul ipython se gasesc in bibliografie: debug, magic commands, lucru cu interpretorul de comenzi etc.

1. Jupyter notebook - codul este scris in browser, cu suport pentru completare automata de cod. Fisierele rezultate sunt cu extensia ipynb.

O lista de notebooks se gaseste aici. Resurse de lucru cu Jupyter notebooks aici

1. Folosind diverse framework-uri (*i.e.* Flask), codul Python poate fi folosit pentru crearea de pagini Web, REST endpoints - sau mai simplu, functiile definite in Jupyter notebook pot fi expuse ca servicii web REST etc.

Variabile, tipuri de date

Clasica afisare de mesaj "Hello world" se obtine cu *functia* `print()`:

In [2]:
```
print("Hello world!")
# sau: print('Hello world')
```

```
Hello world!
```

Pentru Python versiunea 2, afisarea se face folosind *instructiunea* print:

```
print "Hello world!"
```

Se observa lipsa parantezelor.

In ambele cazuri delimitarea sirurilor de caractere se poate face cu apostroafe sau ghilimele.

Variabilele nu se declara in prealabil impreuna cu tipul lor - Python este un limbaj slab tipizat. Natura unei variabile se deduce din valoarea care ii este asociata:

In [3]:
```
x = 3 # x e variabila de tip intreg
y = "abcd" # y este variabila de tip sir de caractere
```

```
# se obisnuieste ca numele compuse ale variabilelor sa fie despartite prin _:
nume_complet = "Popescu Ion"
```

Python este case sensitive: Nume_complet si nume_complet refera variabile diferite.
Atribuirea se face folosind semnul egal, asa cum s-a vazut mai sus.

Tipurile concrete ale variabilelor poate fi aflat la rulare:

In [4]:
```
# functia type
print(type(x))
print(type(y))
```

```
<class 'int'>
<class 'str'>
```

Variabilelor li se pot da nume incepand cu caracter sau _, continuand cu caractere, _ sau cifre.
Urmatoarele cuvinte rezervate nu pot fi folosite ca nume de variabile:

```
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
```

Tipuri numerice

In [5]:
```
height = 1.79
weight = 78
body_mass_index = weight / height ** 2 # la numitor este inaltimea la patrat
print("Indicele de masa corporala este: ", body_mass_index)
```

```
Indicele de masa corporala este:  24.343809494085704
```

Pentru tipurile intregi, operatorii utilizabili sunt:

- +
- -
- *
- / (impartire cu rezultat in virgula mobila; de retinut ca in Python 2.7 intepretarea operatorul / este diferita fata de cea din Python 3)
- // (impartire intreaga)
- % (modulo)
- ** (ridicare la putere)

In [6]:
```
print(7/3)
print(7//3)
```

```
print(7%3)
print(7**3)
```

```
2.3333333333333335
2
1
343
```

Exista si numere intregi lungi (long integer), precizia lor fiind limitata de memoria alocata procesului:

In [7]:
```
big_number = 1*2*3*4*5*6*7*8*9*10*11*12*13*14*15*16*17*18*19*20*21*22*23*24*25*26*2
print('100!=', big_number)
```

```
100!= 9332621544394415268169923885626670049071596826438162146859296389521759999322991560894146397615651828625369792082722375825118521091686400000000000000000000000
```

Pentru tipurile in virgula mobila, operatorii sunt cei de mai sus mai putin impartire intreaga. Daca un operand este in virgula mobila, operatia returneaza valoare in virgula mobila.

In [8]:
```
print(5.0-5)
```

```
0.0
```

Numerele se pot compara folosind operatorii:

- <, >, <=, >=
- == (egalitate), != (diferit)

Exista suport nativ pentru numere complexe:

In [9]:
```
z = 1 + 4j
print(z)
print(type(z))
print(z.conjugate())
z2 = z * z.conjugate()
print(z2)
```

```
(1+4j)
<class 'complex'>
(1-4j)
(17+0j)
```

Sunt admise conversii intre tipuri numerice:

In [10]:
```
a = 3.997
b = int(a)
print(b)
print(float(b))
```

```
3
3.0
```

In [11]:
```
# Alte exemple: https://www.tutorialspoint.com/python/python_numbers.htm
```

Siruri de caractere

Sirurile de caractere se delimiteaza cu apostroafe sau cu ghilimele. Pentru literalii de tip sir de caractere, exista posibilitatea de a scrie valori care nu necesita secvente escape:

In [12]:
```
a = r'adresa \\server\share\....' # remarcam prefixul r
print(a)

# the hard way...
a = 'adresa \\\\server\\share\\....'
print(a)

# caractere Unicode
a = u"ĂÎÂȘȚ aăîâșț" # se remarca prefixul u
print(a)
```

```
adresa \\server\share\....
adresa \\server\share\....
ĂÎÂȘȚ aăîâșț
```

..sau stringuri multilinie, folosind delimitare cu trei ghilimele sau trei apostroafe:

In [13]:
```
versuri = '''If you can keep your head when all about you
    Are losing theirs and blaming it on you,
If you can trust yourself when all men doubt you,
    But make allowance for their doubting too;
If you can wait and not be tired by waiting,
    Or being lied about, don’t deal in lies,
Or being hated, don’t give way to hating,
    And yet don’t look too good, nor talk too wise:

If you can dream—and not make dreams your master;
    If you can think—and not make thoughts your aim;
If you can meet with Triumph and Disaster
    And treat those two impostors just the same;
If you can bear to hear the truth you’ve spoken
    Twisted by knaves to make a trap for fools,
Or watch the things you gave your life to, broken,
    And stoop and build ’em up with worn-out tools:'''
print(versuri)
```

```
If you can keep your head when all about you
    Are losing theirs and blaming it on you,
If you can trust yourself when all men doubt you,
    But make allowance for their doubting too;
If you can wait and not be tired by waiting,
    Or being lied about, don’t deal in lies,
Or being hated, don’t give way to hating,
    And yet don’t look too good, nor talk too wise:

If you can dream—and not make dreams your master;
    If you can think—and not make thoughts your aim;
If you can meet with Triumph and Disaster
    And treat those two impostors just the same;
If you can bear to hear the truth you’ve spoken
    Twisted by knaves to make a trap for fools,
Or watch the things you gave your life to, broken,
    And stoop and build ’em up with worn-out tools:
```

Sirurile de caractere se pot concatena cu + si suporta urmatoarele operatii si functii:

In [14]:
```
sir = "Ana " + "are " + "mere"
print(sir)
```

```
Ana are mere
```

In [15]:
```
print('Lungimea sirului: ', len(sir))
print('Primul element din sir: ', sir[0])
print('Ultimul element din sir: ', sir[len(sir)-1])
print("Ultimul element, in stil Python: ", sir[-1])
print("Penultimul element din sir, in stil Python: ", sir[-2]) # !!!
```

```
Lungimea sirului:  12
Primul element din sir:  A
Ultimul element din sir:  e
Ultimul element, in stil Python:  e
Penultimul element din sir, in stil Python:  r
```

In [16]:
```
# Conversia unei variabile de tip non-string in string:
x = 3
mesaj = "numarul este " + str(x)
print(mesaj)
```

```
numarul este 3
```

In [17]:
```
# metode si operatii posibile pe obiect string:
print(sir.lower())
print('slicing:', sir[4:7])
print(sir.find('nu are'))
print(sir.replace('Ana', 'Ion'))
```

```
ana are mere
slicing: are
-1
Ion are mere
```

In [18]:
```
multi_ana = 'Ana' * 10
print(multi_ana)
```

```
AnaAnaAnaAnaAnaAnaAnaAnaAnaAna
```

In [19]:
```
cuvantul_pere_este_in_sir = 'pere' in sir
print(cuvantul_pere_este_in_sir)
cuvantul_pere_nu_este_in_sir = 'pere' not in sir
print(cuvantul_pere_nu_este_in_sir)
```

```
False
True
```

In [20]:
```
tokens = sir.split(' ')
print(type(tokens))
print(tokens)
print(",".join(['Ana', 'Vasile', 'Dana', 'Ion']))
```

```
<class 'list'>
['Ana', 'are', 'mere']
Ana,Vasile,Dana,Ion
```

Demna de mentionat este functia `eval`, care preia un string de caractere si returneaza un obiect rezultat prin evaluare:

In [21]:
```
a = 3
b = 4
expresie = '(a+b)/(a**2 + b**2 + 1)'
print(eval(expresie))
```

```
0.2692307692307692
```

Recomandam citirea documentatiei oficiale pentru tipul de date string.

Python contine tipul `bool`, util pentru reprezentarea valorilor de adevar `True` si `False`:

In [22]:
```
x = True
print(type(x))
```

```
<class 'bool'>
```

Se pot face conversii de la tipuri numerice la bool si viceversa: valoarea 0 este asociata lui `False`, orice non-zero lui `True`. Invers, `True` se traduce in 1 si `False` in 0:

In [23]:
```
x = bool(-1)
print('-1 ca bool: ', x)
x = bool(0.0)
print('0.0 ca bool: ', x)
b = True
print('True ca int:', int(b))
b = False
print('False ca int:', int(b))
```

```
-1 ca bool:  True
0.0 ca bool:  False
True ca int: 1
False ca int: 0
```

Operatorii aplicabili pe valori bool sunt:

In [24]:
```
print(True and False)
print(True or False)
print(not False)
# nu exista xor, dar...
a, b = True, False
a_xor_b = a != b
print(a_xor_b)
```

```
False
True
True
True
```

Liste

In interiorul unei liste se pot pune oricate elemente, nu neaparat de acelasi tip. Elementele se despart prin virgula; capetele listei sunt marcate prin paranteze drepte:

In [25]:
```
lista = [10, 20, 30, 40]
print('Lungimea listei este', len(lista))
print('Tipul listei este', type(lista))
# Lista eterogena: string si int
lista2 = ['Ana', 234]
```

```
Lungimea listei este 4
Tipul listei este <class 'list'>
```

Elementele unei liste se acceseaza pe baza de indici, incepand de la 0 si terminand cu `len(lista)-1`. Ultimul element se poate referi cu indicele '-1':

In [26]:
```
lista = [10, 20, 30]
print('Ultimul element este:', lista[-1])
print('Penultimul element este:', lista[-2])
```

```
Ultimul element este: 30
Penultimul element este: 20
```

O lista se poate obtine prin repetarea unei constructii de un numar dorit de ori:

In [27]:
```
lista = [1, 2, 3] * 5 # efect similar demonstrat anterior pe string 'Ana'
print(lista)
```

```
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Putem prelua portiuni intregi din lista, nu doar de pe pozitii individuale, folosind 'decuparea' (slicing), sub forma: lista[k:l]. Va rezulta o lista formata din elementele lista[k], lista[k+1], ..., lista[l-1] (se remarca marginea din dreapta a sirului: nu e l, ci l-1). Daca se doreste ca ordinea elementelor sa fie inversata, se foloseste lista[l:k:-1]; elementul de indice l va fi inclus in rezultat, elementul de indice k se va omite:

In [28]:
```
lista = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(lista)
sliced = lista[3:8]
```

```
print(sliced)
sliced_reversed = lista[8:3:-1]
print(sliced_reversed)
```

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

In [29]:
```
lista = [10, 20, 30, 40]
```

Formele acceptate pentru slicing sunt:

- a[start:end] # items start through end-1
- a[start:] # items start through the rest of the array
- a[:end] # items from the beginning through end-1
- a[:] # a copy of the whole array

In special ultima forma este interesanta: se obtine o clona a listei a. In lipsa acestui mecanism, doua variabile care indica spre aceeasi lista pot duce la modificari mutual vizibile:

In [30]:
```
# aliasing
a = [1, 2, 3]
b = a
print('a=', a)
print('b=', b)
a[0] *= -1
print('Dupa modificare via a: b=', b)
# a si b refera aceeasi zona de memorie
```

```
a= [1, 2, 3]
b= [1, 2, 3]
Dupa modificare via a: b= [-1, 2, 3]
```

In [31]:
```
# copiere de lista
a = [1, 2, 3]
b = a[:]
print('a=', a)
print('b=', b)
a[0] *= -1
print('Dupa modificare: a=', a)
print('Dupa modificare: b=', b)
# Concluzia: a si b refera zone de memorie diferite
```

```
a= [1, 2, 3]
b= [1, 2, 3]
Dupa modificare: a= [-1, 2, 3]
Dupa modificare: b= [1, 2, 3]
```

Copierea unei liste se poate face si cu metoda `copy`:

In [32]:
```
# copiere de lista cu metoda copy
a = [1, 2, 3]
b = a.copy()
print('a=', a)
```

```
print('b=', b)
a[0] *= -1
print('Dupa modificare: a=', a)
print('Dupa modificare: b=', b)
# Concluzia: a si b refera zone de memorie diferite
```

```
a= [1, 2, 3]
b= [1, 2, 3]
Dupa modificare: a= [-1, 2, 3]
Dupa modificare: b= [1, 2, 3]
```

Listele pot fi concatenate:

In [33]:
```
a = [1, 2, 3]
b = [10, 20, 30]
c = a+b
print(c)
```

```
[1, 2, 3, 10, 20, 30]
```

La o lista se pot adauga elemente:

In [34]:
```
a = [1, 2, 3]
a.append(100)
print(a)
```

```
[1, 2, 3, 100]
```

Se recomanda ca daca pentru o lista se cunoaste capacitatea maxima sau chiar exacta, sa se prealoce, in loc sa se foloseasca adaugari repetate:

In [35]:
```
a = [None] * 10
#....
a[3] = 'Ana are mere'
print(a)
```

```
[None, None, None, 'Ana are mere', None, None, None, None, None, None]
```

Sortarea unei liste se face cu functia `sorted`, care optional poate sa preia:

- un parametru numit `reverse` de tip boolean, care precizeaza daca se face sortare descrescatoare
- un parametru numit `key`, care face referinta la o functie ce determina politica de sortare

In [36]:
```
# Sortare crescatoare
a = [5, 1, 4, 3]
print(sorted(a))
```

```
[1, 3, 4, 5]
```

In [37]:
```
# Sortare descrescatoare
print(sorted(a, reverse=True))
```

```
[5, 4, 3, 1]
```

In [38]:
```
# Sortare cu criteriu specificat
siruri = ['ccc', 'aaaa', 'd', 'bb']
print(sorted(siruri, key=len))
```

```
['d', 'bb', 'ccc', 'aaaa']
```

Functia `sorted` returneaza o noua lista, lasand pe cea originara nemodificata. Daca se doreste ca sortarea sa se faca in cadrul liste originare, se va folosi metoda `sort` a tipului lista. Metoda `sort` returneaza `None`.

In [39]:
```
siruri = ['ccc', 'aaaa', 'd', 'bb']
siruri.sort()
print(siruri)
```

```
['aaaa', 'bb', 'ccc', 'd']
```

Stergerea unui element dintr-o lista, de pe o pozitie (indice) specificat se face cu `del`:

In [40]:
```
lista = [10, 20, 30]
del lista[1]
print(lista)
```

```
[10, 30]
```

Testarea existentei unui element intr-o lista se face cu operatorul `in`:

In [41]:
```
lista = ['Ana', 'are', 'mere']
print('portocale' in lista )
```

```
False
```

Minimul si maximul unei liste se obtin cu functiile `min` respectiv `max`.

Alte metode apartinand tipului list sunt exemplificate mai jos:

In [42]:
```
lista = [10, 20, 30, 10]
# De cate ori apare elementul 10 in lista?
print('De cate ori apare elementul 10 in lista', lista.count(10))
# Care e primul index in care un anumit element apare in lista?
print('Care e primul index in care un anumit element apare in lista?', lista.index(
# daca pentru metoda index se specifica un element care nu exista, se arunca except
# print(lista.index(100000))
# ValueError: 100000 is not in list
# Inserare elementului 100 pe indexul 2 in lista:
lista.insert(2, 100)
print('Inserare elementului 100 pe indexul 2 in lista', lista)
# Inversarea ordinii elementelor dintr-o lista, in-place
lista.reverse()
print('Inversarea ordinii elementelor dintr-o lista, in-place', lista)
# Stergerea unui element din lista, prima aparitie
lista.remove(10)
print('Stergerea unui element din lista, prima aparitie', lista)
lista.clear()
print('Lista a fost golita:', lista)
```

```
De cate ori apare elementul 10 in lista 2
Care e primul index in care un anumit element apare in lista? 1
Inserare elementului 100 pe indexul 2 in lista [10, 20, 100, 30, 10]
Inversarea ordinii elementelor dintr-o lista, in-place [10, 30, 100, 20, 10]
Stergerea unui element din lista, prima aparitie [30, 100, 20, 10]
Lista a fost golita: []
```

Pentru testarea faptului ca toate elementele (respectiv: macar un element al) unei liste de valori boolene sunt cu valoarea `true`, se va folosi functia `all` (respectiv: `any`).

In [43]:
```
lista = [True, False, True]
print('Macar unul e True:', any(lista))
print('Toate sunt True:', all(lista))
```

```
Macar unul e True: True
Toate sunt True: False
```

Daca lista este goala, functia `all` returneaza `True`, iar `any` - `False`.

In [44]:
```
print('Toate sunt True:',all([]))
print('Macar unul e True:',any([]))
```

```
Toate sunt True: True
Macar unul e True: False
```

O metoda deosebit de eleganta de procesare a elementelor unei liste este prin list comprehension, ce se va prezenta in cadrul sectiunii de instructiuni.

Tupluri

In timp ce o lista permite modificarea continutului sau, un tuplu reprezinta o colectie ordonata imuabila. Elementele se separa cu virgula, capetele tuplului se marcheaza de regula cu paranteze rotunde, sau pot lipsi.

In [45]:
```
tuplu1 = ('informatica', 3, 'dimineata')
tuplu2 = 'chimie', 2, 'seara'
print(tuplu1)
```

```
('informatica', 3, 'dimineata')
```

In [46]:
```
print(tuplu1[0])
```

```
informatica
```

In [47]:
```
print(tuplu1[-1])
```

```
dimineata
```

In [48]:
```
print(len(tuplu1))
```

```
3
```

In [49]:
```
tuplu_gol = ()
print(len(tuplu_gol))
```

```
0
```

In [50]:
```
la_tuplul_cu_doar_o_valoare_trebuie_adaugata_virgula_la_sfarsit = (42,)
print(len(la_tuplul_cu_doar_o_valoare_trebuie_adaugata_virgula_la_sfarsit))
```

```
1
```

Elementele unui tuplu pot fi preluate individual in variabile:

In [51]:
```
print(tuplu1)
a, b, c = tuplu1
print(a)
print(b)
print(c)
```

```
('informatica', 3, 'dimineata')
informatica
3
dimineata
```

In [52]:
```
# Tuplele se pot concatena
tuplu1 + tuplu2
```

Out[52]:
```
('informatica', 3, 'dimineata', 'chimie', 2, 'seara')
```

In [53]:
```
# Convertirea unei liste in tuplu:
lista = [1, 2, 3, 4, 5]
tuplu = tuple(lista)
print(type(tuplu))
print(tuplu)
```

```
<class 'tuple'>
(1, 2, 3, 4, 5)
```

Dictionare

Dictionarele sunt colectii de asocieri intre chei si valori. Colectia de tip dictionar se demarcheaza cu acolade. Elementele se acceseaza specificand intre paranteze drepte valoarea cheii.

In [54]:
```
geografie = {} # dictionar gol
geografie = {'Romania': 'Bucuresti', 'Serbia': 'Belgrad'}
print('Lungimea este:', len(geografie))
key = 'Romania'
print('Valoarea pentru cheia', key, 'este',geografie[key])
geografie['Grecia'] = 'Atena' # adaugare de pereche cheie-valoare in dictionar
# daca se foloseste o cheie care exista, valoarea asociata va fi suprascrisa cu cea
geografie['Grecia'] = 'Athens'
# accesarea folosind o cheie invalilda duce la eroare (KeyError)
# geografie['Franta']
# pentru a evita eroare la rulare, se poate folosi metoda get; daca cheia nu e gasi
print(geografie.get('Franta'))
# daca se doreste returnarea unei valori prestabilite in cazul lipsei cheii, metoda
print(geografie.get('Franta', '<Nu exista in dictionar>'))
```

```
Lungimea este: 2
Valoarea pentru cheia Romania este Bucuresti
None
<Nu exista in dictionar>
```

Colectiile cheilor unui dictionar se determina cu metoda `keys`, respectiv `values`:

In [55]:
```
print('Chei:', geografie.keys())
print('Valori:', geografie.values())
```

```
Chei: dict_keys(['Romania', 'Serbia', 'Grecia'])
Valori: dict_values(['Bucuresti', 'Belgrad', 'Athens'])
```

Oricare din colectii poate fi transformata intr-o lista prin apelul constructorului `list()` care poate prelua o colectie oarecare:

In [56]:
```
print(list(geografie.keys()))
print(list(geografie.values()))
```

```
['Romania', 'Serbia', 'Grecia']
['Bucuresti', 'Belgrad', 'Athens']
```

Stergerea unei chei din dictionar, impreuna cu valoarea asociata, se face cu instructiunea `del`:

In [57]:
```
del geografie['Grecia']
print(geografie)
```

```
{'Romania': 'Bucuresti', 'Serbia': 'Belgrad'}
```

Testarea faptului ca o anumita cheie se afla intr-un dictionar se face cu operatorul `in`:

In [58]:
```
print('Grecia' in geografie)
```

```
False
```

Golirea unui dictionar se face, precum la alte tipuri colectie cu metoda `clear`:

In [59]:
```
# geografie.clear()
```

Colectia perechilor (cheie, valoare) a unui dictionar se obtine cu metoda `items()`:

In [60]:
```
print(geografie.items())
print(list(geografie.items()))
```

```
dict_items([('Romania', 'Bucuresti'), ('Serbia', 'Belgrad')])
[('Romania', 'Bucuresti'), ('Serbia', 'Belgrad')]
```

Daca la un dictionar se doreste adaugarea perechilor (cheie-valoare) ale altui dictionar se foloseste metoda `update`:

In [61]:
```
geografie_asia = {'China':'Beijing', 'India':'New Delhi'}
geografie.update(geografie_asia)
print(geografie)
```

```
{'Romania': 'Bucuresti', 'Serbia': 'Belgrad', 'China': 'Beijing', 'India': 'New Delh
i'}
```

Clonarea unui dictionar se face cu metoda `copy`:

In [62]:
```
geografie_copie = geografie.copy()
geografie_copie['India'] = 'New----Delhi'
print(geografie['India'], ',', geografie_copie['India'])
```

```
New Delhi , New----Delhi
```

O prezentare detaliata si bine structurata pentru dictionare se gaseste aici.

Alte colectii

Se foloseste foarte frecvent functia `range` care produce o secventa de numere. Formele de utilizare sunt:

- `range(n)` produce colectia 0, 1, ..., n-1
- `range(start, stop)` produce colectia start, start+1, ..., stop-1
- `range(start, stop, step)` produce secventa in functie de semnul lui `step`:
  - start, start+step, start+2*step, ... k, unde k este cel mai mare intreg mai mic decat stop, care se obtine prin adaugarea unui multiplu intreg al pasului step la start
  - start, start+step, start+2*step, ...k unde k este cel mai mare intreg mai mare decat stop, obtinut prin adaugarea unui multiplu intreg al pasului step la start

In [63]:
```
for i in range(2, 10):
    print(i, end=' ')
```

```
2 3 4 5 6 7 8 9
```

In [64]:
```
for i in range(2, 10, 3):
    print(i, end=' ')
```

```
2 5 8
```

In [65]:
```
for i in range(10, 2, -2):
    print(i, end=' ')
```

```
10 8 6 4
```

Functia `enumerate` porneste de la o colectie si da acces simultan atat la indicele elementului din colectie, cat si la elementul curent:

In [66]:
```
lista1 = ['a', 'b', 'c']
for i, item in enumerate(lista1):
    print(i, item)
```

```
0 a
1 b
2 c
```

Functia `zip` permite 'imperecherea' a doua liste

In [67]:
```
lista2 = ['m', 'n', 'p']

for pereche in zip(lista1, lista2):
    print(pereche)
```

```
('a', 'm')
('b', 'n')
('c', 'p')
```

Un alt tip de date frecvent utilizat este tipul multime (set), cu prezentare in extenso aici.

Instructiuni, comentarii

Blocuri de instructiuni

Instructiunile se scriu de regula cate una pe linie. Daca pentru o instructiune prea lunga se doreset continuarea ei pe linia urmatoare, se pune la finalul liniei caracterul \ si se continua pe rand nou:

In [68]:
```
a = 1 + 2 + 3 \
          + 4
print(a)
```

```
10
```

Pentru colectii, continuarea pe rand nou nu necesita caracter backslash:

In [69]:
```
lista_lunga = [1, 2, 3,
               4, 5, 6]
```

Se pot scrie mai multe instructiuni pe o linie, despartindu-se cu caracterul ';'. Acest stil insa e nerecomandat

In [70]:
```
print('1'); print('2')
```

```
1
2
```

Un bloc de instructiuni va folosi aceeasi indentare. Se poate folosi orice pentru indentare (tab, sau mereu un acelasi numar de spatii), dar stilul de indentare trebuie sa fie unitar. Un astfel de bloc de instructiuni se termina cu prima linie care nu e indentata in acelasi stil ca si blocul.

In [71]:
```
if 1 + 1 == 3:
    print('Nu se afiseaza')
print('Linia aceasta nu face parte din blocul corespunzator instructiunii if')
```

```
Linia aceasta nu face parte din blocul corespunzator instructiunii if
```

Comentarii

Comentariile sunt fie pe o singura linie, incepand de la caracterul # pana la finalul liniei, fie folosind apostroafe sau ghilimele, de trei ori la inceput si la sfarsit de comentariu:

In [72]:
```
"""Comentariu
foarte lung"""
```

Out[72]:
```
'Comentariu\nfoarte lung'
```

Atribuirea

Atribuirea a fost exemplificata mai sus. Mai avem variantele:

In [73]:
```
# Atribuire multipla cu o valoare
a = b = c = 3
```

In [74]:
```
# Atribuire multipla cu mai multe valori simultan
a, b = 2, 3
print(a, b)
```

```
2 3
```

Ultima forma poate fi speculata astfel: daca `a` si `b` sunt doua variabile si se doreste interschimbarea valorilor lor, atunci putem scrie:

In [75]:
```
print('Inainte de interschimbare: ', a, b)
a, b = b, a
print('Dupa interschimbare: ', a, b)
```

```
Inainte de interschimbare:  2 3
Dupa interschimbare:  3 2
```

Instructiunea `if`

Formele uzuale sunt:

```
if expresie:
    bloc 1
else
    bloc 2
```

Partea `else` poate sa lipseasca. Blocurile de instructiuni ce urmeaza dupa `if` si `else` sunt indentate.

In [76]:
```
var1 = 100
var2 = 200
if var1 > var2:
   print("In if - Got a true expression value")
   print(var1)
else:
```

```
    print("In else - Got a false expression value")
    print(var2)
```

```
In else - Got a false expression value
200
```

O instructiune `if` poate conine multiple teste, folosind `elif`:

```
if expression1:
   statement(s)
elif expression2:
   statement(s)
elif expression3:
   statement(s)
else:
   statement(s)
```

Se evalueaza in ordine conditiile `expression1`, `expression2` etc. pana la prima care e gasita ca fiind adevarata; blocul ei este executat si restul se ignora

In [77]:
```
var = 100
if var == 200:
    print("1 - Got a true expression value")
    print(var)
elif var == 150:
    print("2 - Got a true expression value")
    print(var)
elif var == 100:
    print("3 - Got a true expression value")
    print(var)
else:
    print("4 - Got a false expression value")
    print(var)
```

```
3 - Got a true expression value
100
```

Instructiunea `if` se poate folosi si inline, in forma:

```
expression_if_true if condition else expression_if_false
```

Exemplu:

In [78]:
```
CNP = '5678901234567'
gen_masculin = True if int(CNP[0]) % 2 == 1 else False
print('gen masculin:', gen_masculin)
```

```
gen masculin: True
```

Ciclare: for

Ciclarea cu `for` este folosita pentru a itera o instructiune sau un bloc de instructiuni peste o colectie cunoscuta.

In [79]:
```
for i in range(0, 3):
```

```
    print('i=', i)
```

```
i= 0
i= 1
i= 2
```

In [80]:
```
# Exemplu: calculul factorialului unui numar
n = 100
p = 1
for i in range(1, n+1):
    p *= i
print(n, '!=', p)
```

```
100 != 933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000000
```

In [81]:
```
# %%timeit
# n = 100
# list_numbers = [str(i) for i in range(1, n+1)]
# expression = '*'.join(list_numbers)
# print(eval(expression))
```

In [82]:
```
lista_nume = ['Ana', 'Dan', 'Rares', 'Dana']
for nume in lista_nume:
    print(nume)
```

```
Ana
Dan
Rares
Dana
```

In [83]:
```
geografie = {'Romania': 'Bucuresti', 'Serbia': 'Belgrad', 'Grecia':'Atena'}
for key in geografie:
    print(geografie[key])
```

```
Bucuresti
Belgrad
Atena
```

In [84]:
```
# iterare cu doua elemente, peste colectie de perechi
for key, value in geografie.items():
    print('Capitala pentru', key, 'este', value)
```

```
Capitala pentru Romania este Bucuresti
Capitala pentru Serbia este Belgrad
Capitala pentru Grecia este Atena
```

In [85]:
```
for index, element in enumerate(lista_nume):
    print(index, element)
```

```
0 Ana
1 Dan
2 Rares
3 Dana
```

In [86]:
```
# enumerare peste doua colectii 'impreunate'
lista_ani = [20, 21, 22, 23]
```

```
for nume, ani in zip(lista_nume, lista_ani):
    print(nume, 'are', ani, 'ani')
```

```
Ana are 20 ani
Dan are 21 ani
Rares are 22 ani
Dana are 23 ani
```

Se poate forta iesirea dintr-un ciclu `for` cu instructiunea `break`. Se poate sari peste o parte din blocul unui ciclu `for` si trece la ciclarea urmatoare folosind `continue`:

In [87]:
```
for i in range(100):
    if i > 5:
        break
    print(i)
```

```
0
1
2
3
4
5
```

In [88]:
```
for i in range(10):
    if i % 2 == 0:
        continue
    print(i)
```

```
1
3
5
7
9
```

O forma aparte a instructiunii `for` in Python este aceea in care la final se pune `else`: daca nu s-a executat `break` in corpul ciclului, atunci se executa blocul de dupa `else`:

In [89]:
```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print( n, 'equals', x, '*', n//x)
            break
    else:
        # loop fell through without finding a factor
        print(n, 'is a prime number')
```

```
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

Ciclare: while

Folosind `while` se cicleaza peste un bloc de instructiuni atata timp cat o anumita conditie este adevarata:

```
while test_expression:
    body of while
```

In [90]:
```
n = 10

# initialize sum and counter
my_sum = 0
i = 1

while i <= n:
    my_sum = my_sum + i
    i = i+1    # update counter

# print the sum
print("The sum is", my_sum)
```

```
The sum is 55
```

Se poate ca o instructiune `while` sa se termine cu `else`, al carei bloc se executa cand expresia de test devine falsa, dar numai daca nu s-a executat un `break`.