1 Curs 2: Lucrul cu colectii, functii, module si pachete Python, pachetul NumPy

1.1 Collection comprehension

Pornind de la o colectie (cel mai frecvent caz: de la o lista) se poate crea o alta lista, folosind *list comprehension* - in esenta o ciclare:

In [1]:

```
lista_numere = [1, 3, 6,  21, 22, 32, 33]
lista_patrate = [i*i for i in lista_numere]
print(lista_patrate)
```

executed in 16ms, finished 21:42:13 2021-02-28

```
[1, 9, 36, 441, 484, 1024, 1089]
```

Optional, la fiecare pas al iterarii se poate lua in considerare o conditie `if` inline:

In [2]:

```
lista_patrate_doar_numere_pare = [i*i for i in lista_numere if i % 2 == 0]
print(lista_patrate_doar_numere_pare)
```

executed in 10ms, finished 21:42:13 2021-02-28

```
[36, 484, 1024]
```

... sau se foloseste si `else` pe langa `if` :

In [3]:

```
lista_patrate_sau_cuburi = [i**2 if i % 2 == 0 else i ** 3 for i in lista_numere]
print(lista_patrate_sau_cuburi)
```

executed in 9ms, finished 21:42:14 2021-02-28

```
[1, 27, 36, 9261, 484, 1024, 35937]
```

Exercitiu: daca o lista are ca elemente alte liste, cum se poate determina lista 'flattened'? De exemplu, pentru lista a1 = [[1, 2], [3, 4, 5], [10]] se doreste rezultatul a2 = [1, 2, 3, 4, 5, 10]

In [4]:

```
a1 = [[1, 2], [3, 4, 5], [10]]
a2 = [item for sublist in a1 for item in sublist]
print(a2)
```

executed in 6ms, finished 21:42:14 2021-02-28

```
[1, 2, 3, 4, 5, 10]
```

Comprehension se poate folosi si peste alte tipuri de colectii: de exemplu, putem pleca de la o lista si producem

dictionar:

In [5]:

```
lista_numere = [1, 3, 6,  21, 22, 32, 33]
dictionar_numere_si_patrate = {i:i**2 for i in lista_numere}
print(dictionar_numere_si_patrate)
```

executed in 7ms, finished 21:42:14 2021-02-28

```
{1: 1, 3: 9, 6: 36, 21: 441, 22: 484, 32: 1024, 33: 1089}
```

...sau liste de tuple:

In [6]:

```
#produs cartezian
colours = [ "red", "green", "yellow", "blue" ]
things = [ "house", "car", "tree" ]
produs_cartezian = [(colour, thing) for colour in colours for thing in things]
print(produs_cartezian)
assert len(produs_cartezian) == len(colours) * len(things)
```

executed in 9ms, finished 21:42:14 2021-02-28

```
[('red', 'house'), ('red', 'car'), ('red', 'tree'), ('green', 'house'), ('gr
een', 'car'), ('green', 'tree'), ('yellow', 'house'), ('yellow', 'car'), ('y
ellow', 'tree'), ('blue', 'house'), ('blue', 'car'), ('blue', 'tree')]
```

Mai jos sunt cateva exemple de utilizare de comprehension peste colectii.

In [7]:

```
#Conversie de temperaturi din Celsius in Fahrenheit: valoarea in Fahrenheit se obtine cu j
gradeCelsius = [-20, -10, 0, 5, 23, 35]
gradeFahrenheit = [1.8*gc + 32 for gc in gradeCelsius]
print(gradeFahrenheit)
```

executed in 8ms, finished 21:42:14 2021-02-28

```
[-4.0, 14.0, 32.0, 41.0, 73.4, 95.0]
```

In [8]:

```
#Suma patratelor numerelor de la 1 la 20
print(sum([x**2 for x in range(1, 21)]))
```

executed in 6ms, finished 21:42:14 2021-02-28

```
2870
```

In [9]:

```
#Dintr-o lista de cuvinte se mentin doar cele care nu fac parte dintr-o alta lista specif
stop_words = ["a", "about", "above", "above", "across", "after", "afterwards", "again", "
paragraph_list = ['Stopword','filtering','is','a','common','step','in','preprocessing','t
print('Initial:',paragraph_list)
filtered = [cuvant for cuvant in paragraph_list if cuvant not in stop_words]
print('\nDupa filtrare:', filtered)
```

executed in 38ms, finished 21:42:14 2021-02-28

```
Initial: ['Stopword', 'filtering', 'is', 'a', 'common', 'step', 'in', 'prepr
ocessing', 'text', 'for', 'various', 'purposes', 'This', 'is', 'a', 'list',
'of', 'several', 'different', 'stopword', 'lists', 'extracted', 'from', 'var
ious', 'search', 'engines', 'libraries', 'and', 'articles', 'There', 'is',
'a', 'surprising', 'number', 'of', 'different', 'lists']

Dupa filtrare: ['Stopword', 'filtering', 'common', 'step', 'preprocessing',
'text', 'various', 'purposes', 'This', 'list', 'different', 'stopword', 'lis
ts', 'extracted', 'various', 'search', 'engines', 'libraries', 'articles',
'There', 'surprising', 'number', 'different', 'lists']
```

1.2 Functii

Functiile sunt de trei feluri:

- Functii deja definite in limbajul Python, cum ar fi `len()`, `print()`
- Functii definite de utilizator
- Lambda functii

O functie se defineste folosind cuvantul cheie `def`. Blocul de instructiuni ce defineste corpul functiei este indentat. O functie poate sa nu returneze nimic in mod explicit (si in acest caz rezultatul returnat este considerat `None`), sau orice numar de valori.

1.2.1 Functii definite de utilizator

Urmeaza cateva exemple de functii definite de utilizatorcu comentarii:

In [10]:

```
def hello():
    print('Salutare')

hello()
```

executed in 7ms, finished 21:42:14 2021-02-28

```
Salutare
```

In [11]:

```
def hello_with_name(nume):
    '''
    Functia preia un argument si afiseaza mesajul: Salutare urmat de valoarea argumentulu
    Functia returneaza argumentul cu litere mari.
    :param nume: numele care se cere afisat
    :return: sirul din :param nume: cu litere mari
    '''
    print('Salutare ' + nume)
    return nume.upper()

nume = 'Natalia'
nume_litere_mari = hello_with_name(nume)
print(nume_litere_mari)
help(hello_with_name)
print(hello_with_name.__doc__)
```

executed in 11ms, finished 21:42:14 2021-02-28

```
Salutare Natalia
NATALIA
Help on function hello_with_name in module __main__:

hello_with_name(nume)
    Functia preia un argument si afiseaza mesajul: Salutare urmat de valoare
a argumentului.
    Functia returneaza argumentul cu litere mari.
    :param nume: numele care se cere afisat
    :return: sirul din :param nume: cu litere mari


    Functia preia un argument si afiseaza mesajul: Salutare urmat de valoare
a argumentului.
    Functia returneaza argumentul cu litere mari.
    :param nume: numele care se cere afisat
    :return: sirul din :param nume: cu litere mari
```

In [12]:

```
#exemplu de functie care returneaza mai multe valori simultan
#rezultatul este un tuplu cu doua valori
def min_max(a, b):
    if a<b:
        return a, b
    else:
        return b, a

x, y = 20, 10
min_2, max_2 = min_max(x, y)
print('Minimul este:', min_2, '; maximul este:', max_2)
```

executed in 12ms, finished 21:42:14 2021-02-28

```
Minimul este: 10 ; maximul este: 20
```

In [13]:

```
#parametrii se pot da prin numele lor urmat de egal si valoarea efectiva
min_max(a=5, b=14)
```

executed in 18ms, finished 21:42:14 2021-02-28

Out[13]:

```
(5, 14)
```

In [14]:

```
min_max(b=3, a=20)
```

executed in 13ms, finished 21:42:14 2021-02-28

Out[14]:

```
(3, 20)
```

Pot exista parametri cu valori implicite, precizati la finalul listei de parametri formali:

In [15]:

```
def greet(name, msg = "Good morning!"):
    """
    This function greets to the person with the provided message.

    If message is not provided, it defaults to "Good morning!"
    :param name: Name of the guy to be greeted
    :param msg: a message shown as greeting. It defaults to "Good morning"
    """

    print("Hello",name + ', ' + msg)

greet("Kate")
greet("Bruce","How do you do?")
# echivalent: greet(name="Bruce",msg="How do you do?")
```

executed in 13ms, finished 21:42:14 2021-02-28

```
Hello Kate, Good morning!
Hello Bruce, How do you do?
```

Putem avea n parametru care sa permita numar variabil de valori trimise la apel; acest tip de parametru se scrie cu `*` urmata de numele parametrului formal (de exemplu: `*args`)

In [16]:

```
#Functie cu numar arbitrar de argumente
def greet(*names, msg = "Good morning!"):
    for name in names:
        print('Hello', name + ', ' + msg)
greet('Dan', 'John', 'Mary')
greet('Dan', 'John', 'Mary', msg='How do you do?')
```

executed in 8ms, finished 21:42:14 2021-02-28

```
Hello Dan, Good morning!
Hello John, Good morning!
Hello Mary, Good morning!
Hello Dan, How do you do?
Hello John, How do you do?
Hello Mary, How do you do?
```

Se pot defini functii care sa manipuleze un numar variabil de parametri dati la apel sub forma de nume_parametru=valoare_parametru; denumirea traditionala este `kwargs` (keywords arguments), numele parametrului se prefixaza cu `**` :

In [17]:

```
def demo_kwargs(**kwargs):
    print(kwargs)

demo_kwargs(fruits='apples', quantity='3', measurement_unit='kg')
```

executed in 14ms, finished 21:42:14 2021-02-28

```
{'fruits': 'apples', 'quantity': '3', 'measurement_unit': 'kg'}
```

Iterarea peste perechile din kwargs se face tinant cont ca acesta este un dictionar:

In [18]:

```
def demo_kwargs_iter(**kwargs):
    for key, value in kwargs.items():
        print(key, value)

demo_kwargs_iter(fruits='apples', quantity='3', measurement_unit='kg')
```

executed in 9ms, finished 21:42:14 2021-02-28

```
fruits apples
quantity 3
measurement_unit kg
```

Acelasi efect se obtine prin despachetarea de dictionare, folosind `**` :`

In [19]:

```
dictionar_argumente = {'fruits':'apples', 'quantity':'3', 'measurement_unit':'kg'}
demo_kwargs(**dictionar_argumente)
```

executed in 6ms, finished 21:42:14 2021-02-28

```
{'fruits': 'apples', 'quantity': '3', 'measurement_unit': 'kg'}
```

Ordonarea parametrilor declarati intr-o functie este:

1. parametri formali preluati prin pozitie
2. `*args`
3. parametri cu valori asociate
4. `**kwargs`

```
def example2(arg_1, arg_2, *args, kw_1="shark", kw_2="blobfish", **kwargs):
```

1.2.2 Lambda functii

Se pot defini functii anonime (sau: lambda functii), continand o expresie, pentru care nu se considera necesara definirea unor functii separate. O lambda functie poate sa preia oricate argumente si calculeaza o expresie pe baza lor. Lambda functiile pot accesa doar parametrii trimisi (nu si pe cei globali). Se va omite cuvantul `return` , expresia calculata este cea care se returneaza automat.

In [20]:

```
suma = lambda x, y: x+y
print(suma(3, 4))
```

executed in 7ms, finished 21:42:14 2021-02-28

```
7
```

In [21]:

```
#lambda functie pentru filtrare via functia filter
lista_30 = list(range(30))
lista_filtrata = list(filter(lambda x: x%3==0, lista_30))
print(lista_filtrata)
```

executed in 7ms, finished 21:42:14 2021-02-28

```
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27]
```

In [22]:

```
#lambda functie pentru sortare:
sorted([-1, -2, -3, 2, 3, 4, -5, 6, 7, 8, 9], key=lambda x: x**2)
```

executed in 12ms, finished 21:42:14 2021-02-28

Out[22]:

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

1.2.3 Functii callback

Numele unei functii reprezinta adresa de memorie a acelei functii:

In [23]:

```
def sum_2(x, y):
    return x+y

def dif_2(x, y):
    return x - y

print(sum_2)
print(dif_2)
```

executed in 14ms, finished 21:42:14 2021-02-28

```
<function sum_2 at 0x00000156FB782CA0>
<function dif_2 at 0x00000156FB782EE0>
```

Putem folosi acest mecanism pentru a trimite functii ca parametri intr-o alta functie:

In [24]:

```
def complex_operation(x, y, to_be_called):
    return to_be_called(x, y)

print(complex_operation(2, 3, sum_2))
print(complex_operation(2, 3, dif_2))
```

executed in 7ms, finished 21:42:14 2021-02-28

```
5
-1
```

1.3 Adnotari de tipuri

Incepand cu versiunea 3.5 a limbajului Python, se pot adnota variabilele, parametrii de functii, tipul de retur al functiilor. Operatia de type annotations este optionala si este destinata a mari lizibilitatea codului. Ele sunt definite in documentul de Python Enhancement Proposal PEP484 (https://www.python.org/dev/peps/pep-0484/).

In afara de cod mai usor de citit, un alt beneficiu este analiza static de cod, de exemplu in PyCharm sau MyPy (http://mypy-lang.org), sau suport imbunatatit pentru code completion.

Adnotarile se precizeaza dupa numele variabilei, urmat de doua puncte si denumirea tipului:

In [25]:

```
age:int = 21
name:str = "Guido van Rossum"
```

executed in 6ms, finished 21:42:14 2021-02-28

Adnotarea nu inhiba in niciun fel utilizarea variabile, sau schimbarea in instructiunile urmatoare a tipului lor:

In [26]:

```
age = 21.5
```

executed in 7ms, finished 21:42:14 2021-02-28

Pentru adnotara parametrilor si a tipului functiilor se poate urmari exemplul de mai jos:

In [27]:

```
def f(name:str, age:int) -> str:
    return 'Salut ' + name + ', ai ' + str(age) + ' de ani'

f('Rafael',23)
```

executed in 11ms, finished 21:42:14 2021-02-28

Out[27]:

```
'Salut Rafael, ai 23 de ani'
```

Pentru tipuri complexe se va folosi clasa `typing`, care pune la dispozitie tipuri precum `Dict`, `Tuple`, `List`, `Set` etc.

In [28]:

```
from typing import List

def salut_multi(nume: List[str], varste: List[int]) -> None:
    for n, v in zip(nume, varste):
        print(f'Salut {n}, ai {v} de ani')

salut_multi(['Ana', 'Dan', 'Maria'], [20, 21, 22])
```

executed in 15ms, finished 21:42:14 2021-02-28

```
Salut Ana, ai 20 de ani
Salut Dan, ai 21 de ani
Salut Maria, ai 22 de ani
```

Se pot defini tipuri utilizator, folosindu-ne de cele disponibile:

In [29]:

```
from typing import Tuple

Point2D = Tuple[int, int]

def plot_point(point: Point2D) -> bool:
    ## operatii
    return True

def rotate_point(p:Point2D, angle:float) -> Point2D:
    ## operatii
    ## newPoint = ....
    return newPoint
```

executed in 8ms, finished 21:42:14 2021-02-28

Daca o variabila poate avea unul din mai multe tipuri posibile, putem urma exemplul:

In [30]:

```
from typing import Union

def print_value(value: Union[str, int, float]) -> None:
    print(value)
```

executed in 7ms, finished 21:42:14 2021-02-28

Daca o variabila poate sa fie de un anumit tip sau sa vina cu valoarea `None`, se procedeaza precum mai jos:

In [31]:

```
from typing import Optional

def f(param: Optional[str]) -> str:
    if param is not None:
        return param.upper()
    else:
        return ""
```

executed in 6ms, finished 21:42:14 2021-02-28

Pentru alte constructii: definirea de tipuri noi, callback functions, colectii generice etc. recomandam consultarea bibliografiei.

1.3.1 Bibliografie recomandata

1. PEP484 (https://www.python.org/dev/peps/pep-0484/)
2. typing — Support for type hints (https://docs.python.org/3/library/typing.html)
3. Type hints cheat sheet (Python 3) (https://mypy.readthedocs.io/en/latest/cheat_sheet_py3.html)

1.4 Module

Modulele sunt fisiere Python cu extensia .py, in care se gasesc implementari de functii, clase, declaratii de variabile. Importarea unui modul se face cu instructiunea `import`.

Exemplu: cream un modul - fisierul Python mySmartModule.py - care contine o functie ce calculeaza suma elementelor dintr-o lista:

```
#fisierul mySmartModule.py
def my_sum(lista):
    sum = 0
    for item in lista:
        sum += item
    return sum
```

Utilizarea se face cu:

```
import mySmartModule

lista = [1, 2, 3]

suma = mySmartModule.my_sum(lista)
print(suma)
```

Se poate ca modulul sa fie importat cu un nume mai scurt, sub forma:

```
import mySmartModule as msm
```

si in acest caz apelul se face cu:

```
suma = msm.my_sum(lista)
```

Putem afla ce pune la dispozitie un modul:

```
>>> dir(msm)
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '_
_package__', '__spec__', 'my_sum']
```

elementele aflate intre dublu underscore ('dunders') sunt adaugate automat de Python.

Daca se doreste ca tot ceea ce e definit intr-un modul sa fie disponibil fara a mai face prefixare cu `nume_modul.nume_entitate` , atunci se poate proceda astfel:

```
from mySmartModule import *

print(my_sum([1, 2, 3]))
```

Se recomanda insa sa se importe strict acele entitati (functii, tipuri) din modul care sunt utilizate; in felul acesta se evita suprascrierea prin import al altor entitati deja importate:

```
from mySmartModule import my_sum

print(my_sum([1, 2, 3]))
```

Ordinea de cautare a modulelor este:

1. directorul curent
2. daca nu se gaseste modulul cerut, se cauta in variabila de mediu `PYTHONPATH` , daca e definita
3. daca nu se gaseste modulul cerut, se cauta in calea implicita.

Calea de cautare implicita se gaseste in variabila `path` din modulul sistem `sys` :

In [32]:

```
import sys
print(sys.path)
```

executed in 8ms, finished 21:42:14 2021-02-28

```
['d:\\work\\school\\cursuri\\Introducere_In_Data_Science\\cursuri\\Curs2',
'C:\\anaconda3\\envs\\ids\\python38.zip', 'C:\\anaconda3\\envs\\ids\\DLLs',
'C:\\anaconda3\\envs\\ids\\lib', 'C:\\anaconda3\\envs\\ids', '', 'C:\\anacon
da3\\envs\\ids\\lib\\site-packages', 'C:\\anaconda3\\envs\\ids\\lib\\site-pa
ckages\\locket-0.2.1-py3.8.egg', 'C:\\anaconda3\\envs\\ids\\lib\\site-packag
es\\win32', 'C:\\anaconda3\\envs\\ids\\lib\\site-packages\\win32\\lib',
'C:\\anaconda3\\envs\\ids\\lib\\site-packages\\Pythonwin', 'C:\\anaconda3\\e
nvs\\ids\\lib\\site-packages\\IPython\\extensions', 'C:\\Users\\Lucian\\.ipy
thon']
```

Daca se doreste ca un modul scris de utilizator intr-un director ce nu se gaseste in lista de mai sus sa fie accesibil pentru import, atunci calea catre director trebuie adaugata la colectia `sys.path` :

In [33]:

```
sys.path.append('./my_modules/')
from mySmartModule import my_sum
print(my_sum([1, 2, 3]))
```

executed in 16ms, finished 21:42:14 2021-02-28

```
6
```

Un modul se poate folosi in doua feluri:

1. pentru a pune la dispozitie diferite implementari de functii sau de clase, sau variabile setate la anumite valori (de exemplu `math.pi`):

   ```
   import math
   print(math.pi)
   ```

2. se poate lansa de sine statator, scriind in lina de comanda: `python mySmartModule.py` . Pentru acest caz, daca se vrea ca sa se execute o anumita secventa de cod, atunci se va scrie in modulul Python `mySmartModule.py` :

   ```
   if __name__ == '__main__':
    #cod care se executa la lansarea directa a script-ului
   ```

Codul din sectiunea `if` scrisa ca mai sus nu se va executa cand modulul este importat.

Exemplu:

```
def my_sum(lista):
    sum = 0
    for item in lista:
        sum += item
    return sum

if __name__ == '__main__':
    print('Exemplu de utilizare')
    lista = list(range(100))
    print(my_sum(lista))
```

1.5 Pachete Python

Un pachet este o colectie de module. Fizic, un pachet este o structura ierarhica de directoare in care se gasesc module si alte pachete. Este obligatoriu ca in orice director care se doreste a fi vazut ca un pachet sa existe un fisier numit `__init__.py` . In prima faza, `__init__.py` poate fi gol. Plecam de la structura de directoare si fisiere:

```
---myUtils\
 |------ mySmartModule.py
 |------ __init__.py
```

Pentru importul functiei `my_sum` din fisierul `mySmartModule.py` aflat in directorul `myUtils` - care se doreste a fi pachet - s-ar scrie astfel:

```
from myUtils.mySmartModule import my_sum
print(my_sum([1, 2, 3]))
```

dar am prefera sa putem scrie:

```
from myUtils import my_sum
print(my_sum([1, 2, 30]))
```

adica sa nu mai referim modulul (fisierul) `mySmartModule` din cadrul pachetului `myUtils` . Pentru asta vom adauga in fisierul `__init__.py` din directorul `myUtils` linia:

```
from .mySmartModule import my_sum
```

unde caracterul `.` de dinaintea numelui de modul `mySmartModule` se refera la calea relativa.

In [34]:

```
from myUtils.mySmartModule import my_sum
print(my_sum([1, 2, 10, 300]))
```

executed in 14ms, finished 21:42:14 2021-02-28

```
313
```

In [35]:

```
from myUtils import my_sum
print(my_sum([1, 2, 10, 300]))
```

executed in 7ms, finished 21:42:14 2021-02-28

```
313
```

In fisierul `__init__.py` se obisnuieste sa se puna orice are legatura cu initializarea pachetului, cum ar fi incarcarea de date de pe disc in memorie sau setarea unor variabile la valori anume.

Pentru cazul in care se doreste crearea de pachete destinate comunitatii si publicarea pe PyPI, se va urma acest tutorial (https://python-packaging.readthedocs.io/en/latest/).

Alte exemple de utilizare de pachete sunt:

In [36]:

```
import re #pachet pentru expresii regulate
my_string = 'Am cumparat: mere, pere, prune... si caise'
tokens = re.split(r'\W+', my_string)
print(tokens)
```

executed in 12ms, finished 21:42:14 2021-02-28

```
['Am', 'cumparat', 'mere', 'pere', 'prune', 'si', 'caise']
```

In [37]:

```
# Serializare cu pickle
import pickle

favorite_color = { "lion": "yellow", "kitty": "red" }

pickle.dump( favorite_color, open( "save.pkl", "wb" ) )
del favorite_color #nu mai e necesar

#restaurare
favorite_color_restored = pickle.load( open( "save.pkl", "rb" ) )
print('dupa deserializare:', favorite_color_restored)

!del save.pkl #sterge fisierul pickle de pe disk
```

executed in 139ms, finished 21:42:14 2021-02-28

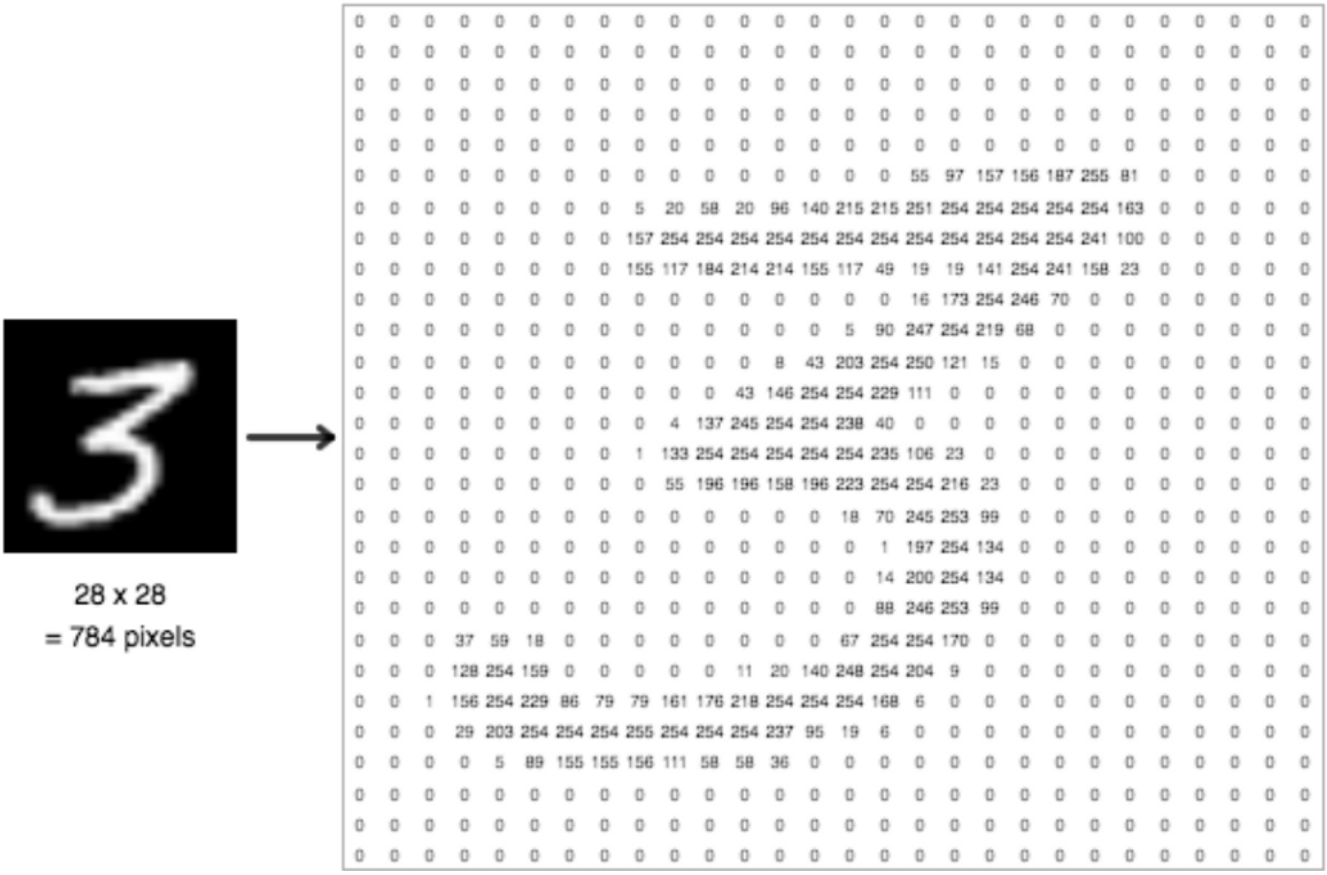
```
dupa deserializare: {'lion': 'yellow', 'kitty': 'red'}
```

1.6 Pachetul NumPy

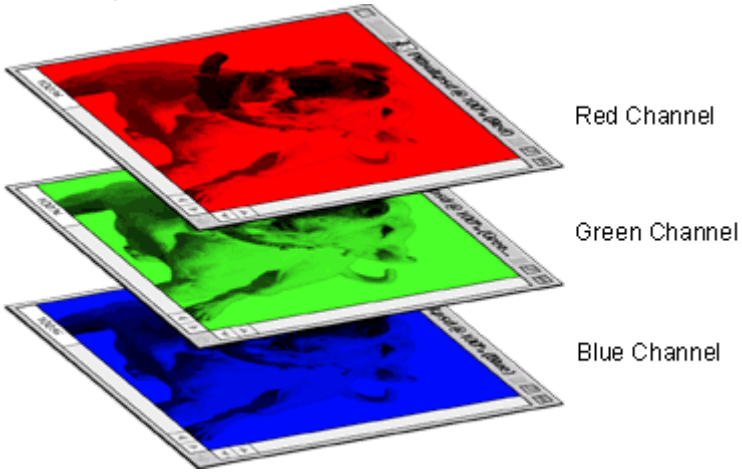
NumPy (Numerical Python) este pachetul de baza pentru calcule stiintifice in Python. Asigura suport pentru lucrul cu vectori si matrice multidimensionale, functii dedicate precum sortare, operatii din algebra liniara, procesare de semnal, calcule statistice de baza, generare de numere aleatoare etc. NumPy sta la baza multor altor pachete. Datele pe care le proceseaza trebuie sa incapa in memoria RAM. NumPy are la baza cod C compilat si optimizat.

In destul de multe situatii, datele sunt sau pot fi transformate in numere:

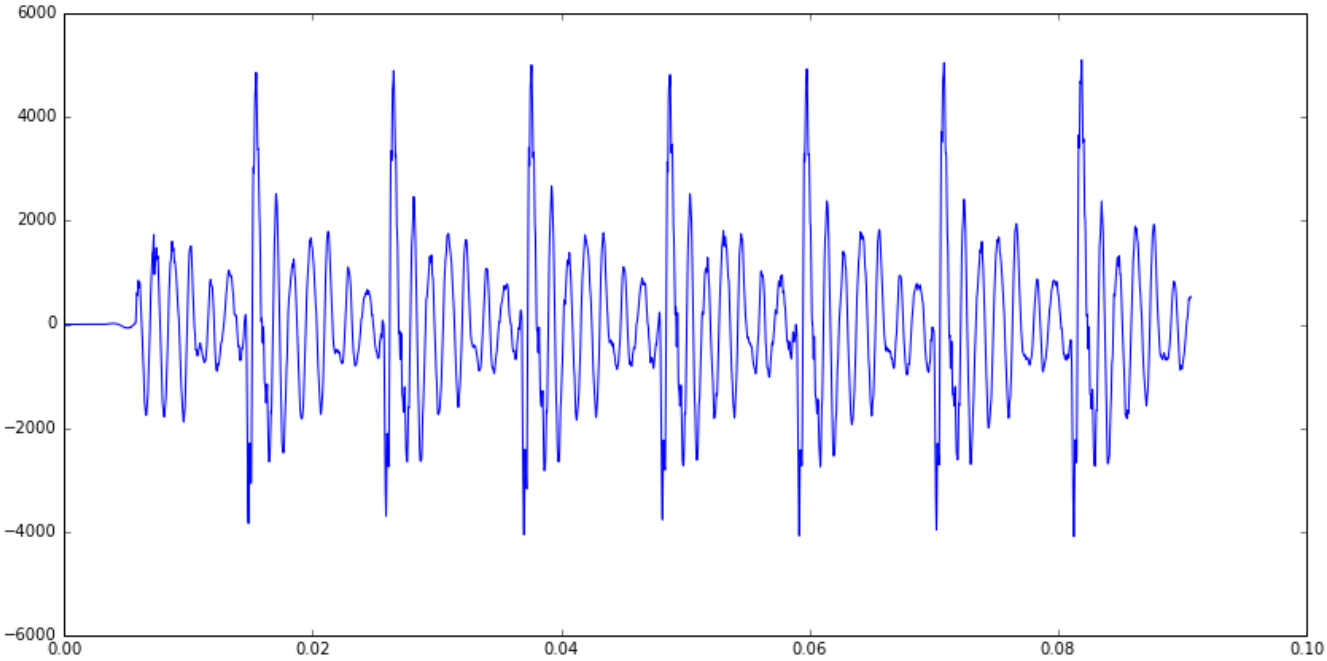
- o imagine in tonuri de gri poate fi vazuta ca o matrice bidimensionala de numere; fiecare numar reprezinta intensitatea pixelului (0 - negru, 255 - alb; frecvent se folosesc valorile scalate in intevalul [0, 1])



- o imagine color poate fi vazuta ca o matrice cu trei dimensiuni: 3 matrice bidimensionale "paralele", corespunzatoare canalelor red, green, blue; pentru fiecare canal de culoare valorile pot fi intre 0 si 255:



- un fisir audio este vazut ca unul/doi/k vectori dimensionali, corespunzatoare cazurilor: mono, stereo, k canale. Valorile numerice in cazul unui fisier wav reprezinta deplasarea membranei microfonului, discretizata in timp si amplitudine:



- un text poate fi tradus in vectori numerici prin tehnici precum Bag of words (https://en.wikipedia.org/wiki/Bag-of-words_model) sau Word2vec (https://en.wikipedia.org/wiki/Word2vec).

Reprezentarea este mult mai eficienta decat pentru listele Python; codul scris cu NumPy apeleaza biblioteci compilate in cod nativ. Daca codul este scris vectorizat, eficienta rularii e si mai mare.

Tipul cel mai comun din NumPy este `ndarray` - n-dimensional array.

In [38]:

```
#import de pachet; traditional se foloseste abrevierea np pentru numpy
import numpy as np

#crearea unui vector pornind de la o lista Python
x = np.array([1, 4, 2, 5, 3])

#tipul variabilei x; se observa ca e tip numpy
print(type(x))
#toate elementele din array sunt de acelasi tip
print(x.dtype)

#specificarea explicita a tipului de reprezentare a datelor in array
y = np.array([1, 2, 3], dtype=np.float16)
print(y.dtype)
```

executed in 391ms, finished 21:42:15 2021-02-28

```
<class 'numpy.ndarray'>
int32
float16
```

In [39]:

```
#cazuri frecvent folosite
all_zeros = np.zeros(10, dtype=int)
print(all_zeros)
#tiparire nr de elemente pe fiecare dimensiune
print(all_zeros.shape)
```

executed in 8ms, finished 21:42:15 2021-02-28

```
[0 0 0 0 0 0 0 0 0 0]
(10,)
```

In [40]:

```
#matrice 2d
mat = np.array([[1, 2, 3], [4, 5, 6]])
print(mat)
print(mat.shape)
print(mat[0, 1])
```

executed in 13ms, finished 21:42:15 2021-02-28

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

In [41]:

```
# matrice de valori constante:
mat_7 = np.ones((4, 10)) * 7
print(mat_7)
```

executed in 13ms, finished 21:42:15 2021-02-28

```
[[7. 7. 7. 7. 7. 7. 7. 7. 7. 7.]
 [7. 7. 7. 7. 7. 7. 7. 7. 7. 7.]
 [7. 7. 7. 7. 7. 7. 7. 7. 7. 7.]
 [7. 7. 7. 7. 7. 7. 7. 7. 7. 7.]]
```

Numarul de dimensiuni se determina cu:

In [42]:

```
print('Numarul de dimensiuni pentru vectorul all_zero:', all_zeros.ndim)
print('Numarul de dimensiuni pentru matricea mat:', mat.ndim)
```

executed in 12ms, finished 21:42:15 2021-02-28

```
Numarul de dimensiuni pentru vectorul all_zero: 1
Numarul de dimensiuni pentru matricea mat: 2
```

iar numarul total de elemente, respectiv dimensiunea in octeti a unui element oarecare (un ndarray are elemente de acelasi tip, intotdeauna):

In [43]:

```
print('mat size: {0}\nmat element size: {1} bytes\nmat.dtype:{2}'.format(mat.size, mat.it
```

executed in 14ms, finished 21:42:15 2021-02-28

```
mat size: 6
mat element size: 4 bytes
mat.dtype:int32
```

In [44]:

```
#cazuri comune
all_ones = np.ones((3, 5))
print(all_ones)
all_pi = np.full((3, 2), np.pi)
print(all_pi)
print(np.eye(3))
```

executed in 17ms, finished 21:42:15 2021-02-28

```
[[1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]]
[[3.14159265 3.14159265]
 [3.14159265 3.14159265]
 [3.14159265 3.14159265]]
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

In [45]:

```
#valori echidistante intr-un interval; capetele intervalului fac parte din valorile gener
print(np.linspace(0, 10, 5))
```

executed in 9ms, finished 21:42:15 2021-02-28

```
[ 0.   2.5  5.   7.5 10. ]
```

In [46]:

```
#similar cu functia range din Python: se genereaza de la primul parametru, cu pasul dat d
#ultima valoare generata fiind strict mai mica decat al doilea parametru
vector_de_valori = np.arange(0, 10, 3)
print(vector_de_valori)
print(type(vector_de_valori))
```

executed in 8ms, finished 21:42:15 2021-02-28

```
[0 3 6 9]
<class 'numpy.ndarray'>
```

In [47]:

```
#numere aleatoare
x = np.random.random((2, 3))
print(x)
```

executed in 12ms, finished 21:42:15 2021-02-28

```
[[0.50881503 0.70706784 0.09838777]
 [0.94326244 0.84431426 0.09223635]]
```

Tipurile de date folosibile pentru ndarrays sunt:

Tip	Explicatie
bool_	Boolean (True or False) stored as a byte
int_	Default integer type (same as C long; normally either int64 or int32)
intc	Identical to C int (normally int32 or int64)
intp	Integer used for indexing (same as C ssize_t; normally either int32 or int64)
int8	Byte (-128 to 127)
int16	Integer (-32768 to 32767)
int32	Integer (-2147483648 to 2147483647)
int64	Integer (-9223372036854775808 to 9223372036854775807)
uint8	Unsigned integer (0 to 255)
uint16	Unsigned integer (0 to 65535)
uint32	Unsigned integer (0 to 4294967295)
uint64	Unsigned integer (0 to 18446744073709551615)
float_	Shorthand for float64.
float16	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
float32	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
float64	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
complex_	Shorthand for complex128.
complex64	Complex number, represented by two 32-bit floats (real and imaginary components)
complex128	Complex number, represented by two 64-bit floats (real and imaginary components)

O operatie utila este schimbarea formei unui array:

In [48]:

```
#dintr=un vector intr-o matrice
vec = np.arange(10)
mat = vec.reshape(2, 5)
print(vec)
print(mat)
```

executed in 13ms, finished 21:42:15 2021-02-28

```
[0 1 2 3 4 5 6 7 8 9]
[[0 1 2 3 4]
 [5 6 7 8 9]]
```

In [49]:

```
#...si invers:
vec2 = mat.flatten()
print(vec2)
```

executed in 11ms, finished 21:42:15 2021-02-28

```
[0 1 2 3 4 5 6 7 8 9]
```

Tablourile pot fi concatenate, specificandu-se axa

In [50]:

```
a = np.array([[1, 2], [3, 4]], float)
b = np.array([[5, 6], [7,8]], float)
```

executed in 7ms, finished 21:42:15 2021-02-28

In [51]:

```
#concatenare pe verticala
stiva_verticala = np.concatenate((a, b), axis=0)
print(stiva_verticala)
```

executed in 6ms, finished 21:42:15 2021-02-28

```
[[1. 2.]
 [3. 4.]
 [5. 6.]
 [7. 8.]]
```

Conceptul de axa se defineste pentru tablourile cu mai mult de o dimensiune. Pentru un tablou cu doua dimensiuni, axa 0 parcurge pe verticala tabloul, axa 1 parcurge pe orizontala. Unele functii iau in considerare axa de lucru:

In [52]:

```
#concatenare pe orizontala
stiva_orizontala = np.concatenate((a, b), axis=1)
print(stiva_orizontala)
```

executed in 9ms, finished 21:42:15 2021-02-28

```
[[1. 2. 5. 6.]
 [3. 4. 7. 8.]]
```

In [53]:

```
#echivalent cu:
stiva_verticala = np.vstack((a, b))
stiva_orizontala = np.hstack((a, b))
print(stiva_verticala)
print(stiva_orizontala)
```

executed in 12ms, finished 21:42:15 2021-02-28

```
[[1. 2.]
 [3. 4.]
 [5. 6.]
 [7. 8.]]
[[1. 2. 5. 6.]
 [3. 4. 7. 8.]]
```

In [54]:

```
matrice = np.arange(15).reshape(3, 5)
print(matrice)
```

executed in 9ms, finished 21:42:15 2021-02-28

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
```

In [55]:

```
suma_pe_coloane= np.sum(matrice, axis=0)
print(suma_pe_coloane)
```

executed in 1.14s, finished 21:42:16 2021-02-28

```
[15 18 21 24 27]
```

In [56]:

```
suma_pe_linii= np.sum(matrice, axis=1)
print(suma_pe_linii)
```

executed in 9ms, finished 21:42:16 2021-02-28

```
[10 35 60]
```

1.6.1 Operatii cu ndarrays

Sunt implementate operatiile matematice uzuale din algebra liniara: inmultire cu scalari, adunare, scadere, inmultire de matrice.

Exemple:

- adunare
- produs Hadamard, produs matricial, produs scalar:

In [57]:

```
#inmultire cu scalar
a = np.array([[1, 2, 3], [4, 5, 6]])
print('a=\n', a)
b = a * 10
print('b=\n', b)
```

executed in 8ms, finished 21:42:16 2021-02-28

```
a=
 [[1 2 3]
 [4 5 6]]
b=
 [[10 20 30]
 [40 50 60]]
```

In [58]:

```
#adunare, scadere: +, -
suma = a + b
print(suma)
diferenta = a - b
print(diferenta)
```

executed in 8ms, finished 21:42:16 2021-02-28

```
[[11 22 33]
 [44 55 66]]
[[ -9 -18 -27]
 [-36 -45 -54]]
```

Operatorul de inmultire * este implementat altfel decat in algebra liniara: pentru doua matrice cu aceleasi

dimensiuni se face inmultirea elementelor aflate pe pozitii identice, adica: c[i, j] = a[i, j] * b[i, j]. Este asa-numitul produs Hadamard, frecvent intalnit in machine learning.

In [59]:

```
#inmultirea folosind * duce la inmultire element cu element (produs Hadamard): c[i, j] = 
c = a*b
print(c)
for i in range(c.shape[0]):  #c.shape[0] = numarul de linii ale matricei c
    for j in range(c.shape[1]): #c.shape[1] = numarul de coloane ale matricei c
        print(c[i, j] == a[i, j] * b[i, j])
```

executed in 12ms, finished 21:42:16 2021-02-28

```
[[ 10  40  90]
 [160 250 360]]
True
True
True
True
True
True
```

Operatiile folosesc biblioteci de algebra liniara, optimizate pentru microprocesoarele actuale. Se recomanda folosirea acestor implementari in loc de a face operatiile manual cu ciclari `for` :

In [60]:

```
#creare de matrice
matrix_shape = (100, 100)
a_big = np.random.random(matrix_shape)
b_big = np.random.random(matrix_shape)
```

executed in 11ms, finished 21:42:16 2021-02-28

In [61]:

```
%%timeit
c_big = np.empty_like(a_big)
for i in range(c_big.shape[0]):
    for j in range(c_big.shape[1]):
        c_big[i, j] = a_big[i, j] * b_big[i, j]
```

executed in 9.51s, finished 21:42:26 2021-02-28

```
11 ms ± 2.76 ms per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

In [62]:

```
%%timeit
c_big = a_big * b_big
```

executed in 9.42s, finished 21:42:35 2021-02-28

```
11.9 µs ± 1.06 µs per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

In [63]:

```
#'ridicarea la putere' folosind ** : fiecare element al matricei este ridicat la putere
print('matricea initiala:\n', a)
putere = a ** 2
print('dupa ridicarea la puterea 2:\n', putere)
putere_3 = np.power(a, 3)
print('dupa ridicarea la puterea 3:\n', putere_3)
```

executed in 12ms, finished 21:42:35 2021-02-28

```
matricea initiala:
 [[1 2 3]
 [4 5 6]]
dupa ridicarea la puterea 2:
 [[ 1  4  9]
 [16 25 36]]
dupa ridicarea la puterea 3:
 [[  1   8  27]
 [ 64 125 216]]
```

Se poate folosi operatorul / pentru a face impartirea punctuala (element cu element) a valorilor din doua matrice:

In [64]:

```
print('a=', a)
print('b=', b)
print('a/b=', a/b)
```

executed in 12ms, finished 21:42:35 2021-02-28

```
a= [[1 2 3]
 [4 5 6]]
b= [[10 20 30]
 [40 50 60]]
a/b= [[0.1 0.1 0.1]
 [0.1 0.1 0.1]]
```

In [65]:

```
#ridicarea la putere a unei matrice patratice, asa cum e definita in algebra liniara:
patratica = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
ridicare_la_putere = np.linalg.matrix_power(patratica, 3)
print(ridicare_la_putere)
```

executed in 10ms, finished 21:42:35 2021-02-28

```
[[ 468  576  684]
 [1062 1305 1548]
 [1656 2034 2412]]
```

Daca se apeleaza o functie matematica definita in NumPy pe un ndarray, rezultatul va fi tot un ndarray de aceeasi forma ca si intrarea, dar cu elementele calculate prin aplicarea functiei respective:

In [66]:

```
x = np.arange(6).reshape(2, 3)
print(x)
y = np.exp(x)
assert x.shape == y.shape
for i in range(0, x.shape[0]):
    for j in range(0, x.shape[1]):
        assert y[i, j] == np.exp(x[i, j])
```

executed in 13ms, finished 21:42:35 2021-02-28

```
[[0 1 2]
 [3 4 5]]
```

In [67]:

```
#produs algebric de matrice:
a = np.random.rand(3, 5)
b = np.random.rand(5, 10)
assert a.shape[1] == b.shape[0]
c = np.dot(a, b)
# se poate scrie echivalent
c = a.dot(b)
assert a.shape[0] == c.shape[0] and b.shape[1] == c.shape[1]
```

executed in 37ms, finished 21:42:35 2021-02-28

NumPy defineste o serie de functii ce pot fi utilizate: `all, any, apply_along_axis, argmax, argmin, argsort, average, bincount, ceil, clip, conj, corrcoef, cov, cross, cumprod, cumsum, diff, dot, floor, inner, inv, lexsort, max, maximum, mean, median, min, minimum, nonzero, outer, prod, re, round, sort, std, sum, trace, transpose, var, vdot, vectorize, where` - documentate aici (https://docs.scipy.org/doc/numpy-dev/reference/generated/).

1.7 Indexare

Pana acum, pentru referirea elementelor de la anumite pozitii s-au folosit indici simpli de forma:

```
vector[indice]
# sau
matrice[i, j]
```

In [69]:

```
vector = np.arange(10)
print(vector)
print('vector[4]={0}'.format(vector[4]))
```

executed in 6ms, finished 21:42:35 2021-02-28

```
[0 1 2 3 4 5 6 7 8 9]
vector[4]=4
```

In [70]:

```
matrice = np.arange(12).reshape(3, 4)
print(matrice)
print(matrice[2, 1])
```

executed in 6ms, finished 21:42:35 2021-02-28

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
9
```

Pentru matrice se poate folosi o indiciere de forma:

```
matrice[i][j]
```

dar e o varianta ineficienta fata de `matrice[i,j]` deoarece in prima varianta se face o copie temporara a linii de indice `i` a matricei si din acest obiect auxiliar se selecteaza elementul de indice `j` .

Prin `slicing` exista posibilitatea de a face referire la un intreg subset de elemente, de exemplu peentru vectori:

In [71]:

```
vector = 10 * np.arange(10)
print(vector)
print(vector[2:6]) #remarcam ca indicele din dreapta este cu rol de "exclusiv", nu contri
```

executed in 11ms, finished 21:42:35 2021-02-28

```
[ 0 10 20 30 40 50 60 70 80 90]
[20 30 40 50]
```

In [72]:

```
indici = [1, 3, 2, 7]
print(vector)
print(vector[indici])
```

executed in 12ms, finished 21:42:35 2021-02-28

```
[ 0 10 20 30 40 50 60 70 80 90]
[10 30 20 70]
```

In [73]:

```
# sau cu indici dati in progresie aritmetica
vector[2:8:2]
```

executed in 9ms, finished 21:42:35 2021-02-28

Out[73]:

```
array([20, 40, 60])
```

Pentru matrice putem folosi:

In [74]:

```
matrice = 10 * np.arange(20).reshape(4, 5)
print(matrice)
```

executed in 6ms, finished 21:42:35 2021-02-28

```
[[  0  10  20  30  40]
 [ 50  60  70  80  90]
 [100 110 120 130 140]
 [150 160 170 180 190]]
```

In [75]:

```
print(matrice[1,])
#care e tot una cu forma mai explicita:
print(matrice[1, :])
```

executed in 9ms, finished 21:42:35 2021-02-28

```
[50 60 70 80 90]
[50 60 70 80 90]
```

In [76]:

```
#putem selecta domenii de indici, pe fiecare axa
matrice[1:3, :]
```

executed in 10ms, finished 21:42:35 2021-02-28

Out[76]:

```
array([[ 50,  60,  70,  80,  90],
       [100, 110, 120, 130, 140]])
```

In [77]:

```
#indexare pe fiecare dimensiune
matrice[1:3, 2:4]
```

executed in 9ms, finished 21:42:35 2021-02-28

Out[77]:

```
array([[ 70,  80],
       [120, 130]])
```

1.7.1 Indexarea logica

Asupra elementelor unui tablou se pot aplica operatii logice; obtinem un tablou de aceeasi forma ca si tabloul initial, dar plin cu valori `True` si `False` in functie de rezultatul aplicarii operatiei logice:

In [78]:

```
a = np.array([[1,2], [3, 4], [5, 6]])
print(a)
print(a > 2)
```

executed in 9ms, finished 21:42:35 2021-02-28

```
[[1 2]
 [3 4]
 [5 6]]
[[False False]
 [ True  True]
 [ True  True]]
```

Tabloul rezultat in urma aplicarii operatiei logice poate fi folosit pentru indexare. Se vor returna doar acele elemente care satisfac conditia logica ceruta:

In [79]:

```
mai_mare_ca_2 = a > 2
print(a[mai_mare_ca_2])
#direct
print(a[a>2])
```

executed in 10ms, finished 21:42:35 2021-02-28

```
[3 4 5 6]
[3 4 5 6]
```

Daca se doresc expresii mai complicate: elemente care sunt mai mari ca 2 si mai mici ca 6, atunci:

In [80]:

```
a[np.logical_and(a > 2, a < 6)]
```

executed in 12ms, finished 21:42:35 2021-02-28

Out[80]:

```
array([3, 4, 5])
```

Mai exista: `np.logical_or` , `np.logical_not` , `np.logical_xor` .

O expresie logica utila este urmatoarea: se cere obtinerea doar a acelor elemente care sunt definite - adica nu sunt NaN:

In [81]:

```
tab = np.array([[1.0, 2.3, np.nan, 4], [10, np.nan, np.nan, 0]])
print(tab[~np.isnan(tab)])
```

executed in 9ms, finished 21:42:35 2021-02-28

```
[ 1.   2.3  4.  10.   0. ]
```

Indicierea returneaza un 'view' al tabloului, peste care se pot aplica modificari ale continutului originar:

In [82]:

```
print('Inainte:\n', tab)
tab[np.isnan(tab)] = 0.0
print('Dupa:\n', tab)
```

executed in 12ms, finished 21:42:35 2021-02-28

```
Inainte:
 [[ 1.   2.3  nan  4. ]
 [10.   nan  nan  0. ]]
Dupa:
 [[ 1.   2.3  0.   4. ]
 [10.   0.   0.   0. ]]
```

Indexarea logica permite specificarea elementelor dintr-un tablou pentru care se efectueaza anumite operatii:

In [83]:

```
# numerele pare se inmultesc cu 10, celelalte raman cum sunt
tablou = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
print('Inainte de modificare:\n', tablou)
tablou[tablou % 2 == 0] *= 10
print('Dupa modificare:\n', tablou)
```

executed in 9ms, finished 21:42:35 2021-02-28

```
Inainte de modificare:
 [[1 2 3 4]
 [5 6 7 8]]
Dupa modificare:
 [[ 1 20  3 40]
 [ 5 60  7 80]]
```

Modificareaa se poate face si doar pe o anumita axa:

In [84]:

```
tablou = np.array([[1, 2, 3, 4], [5, 6, 7, 8]], dtype=np.float)
print('Inainte de modificare\n', tablou)
#coloanele 0, 2, 3 se modifica
bool_columns = [True, False, True, True]
tablou[:, bool_columns] = (tablou[:, bool_columns] +3 )/10
print('Dupa modificare\n', tablou)
```

executed in 12ms, finished 21:42:35 2021-02-28

```
Inainte de modificare
 [[1. 2. 3. 4.]
 [5. 6. 7. 8.]]
Dupa modificare
 [[0.4 2.  0.6 0.7]
 [0.8 6.  1.  1.1]]
```

1.7.2 Bibliografie recomandata

1. https://docs.scipy.org/doc/numpy-1.13.0/glossary.html (https://docs.scipy.org/doc/numpy-1.13.0/glossary.html)

2. https://engineering.ucsb.edu/~shell/che210d/numpy.pdf (https://engineering.ucsb.edu/~shell/che210d/numpy.pdf)
3. http://www.scipy-lectures.org/intro/numpy/numpy.html#indexing-and-slicing (http://www.scipy-lectures.org/intro/numpy/numpy.html#indexing-and-slicing)

1.8 Broadcasting

Broadcasting este un mecanism prin care se permite - in anumite circumstante - operarea cu matrice de dimensiuni ce nu sunt compatibile din punct de vedere al dimensiunii. De exemplu, urmand strict definitia matematica a adunarii, matricele `a` si `b` de mai jos nu se pot aduna:

```
matrice_din_vector = vector_de_valori.reshape((2, 2))
print(matrice_din_vector.shape)
print(vector_de_valori.shape)
```

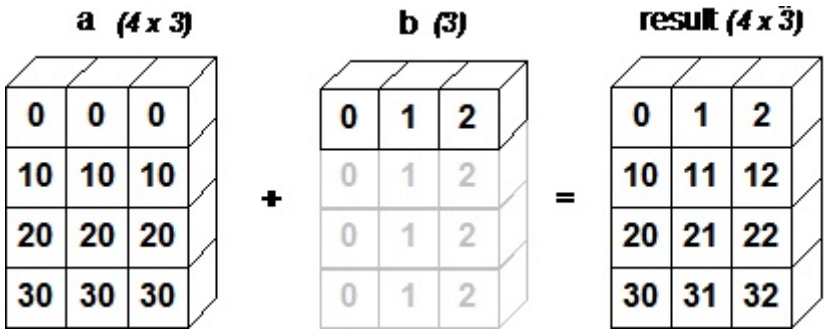
In [85]:

```
a = np.array([[0.0,0.0,0.0],[10.0,10.0,10.0],[20.0,20.0,20.0],[30.0,30.0,30.0]])
b = np.array([0.0,1.0,2.0])

print('a=\n{0}\n'.format(a))
print('b=\n{0}\n'.format(b))
```

executed in 12ms, finished 21:42:35 2021-02-28

```
a=
[[ 0.  0.  0.]
 [10. 10. 10.]
 [20. 20. 20.]
 [30. 30. 30.]]

b=
[0. 1. 2.]
```

Prin broadcasting se extine automat matrice `b` prin duplicarea (copierea) liniei:



In [86]:

```
#broadcasting
result = a + b
print('result=\n{0}\n'.format(result))
```

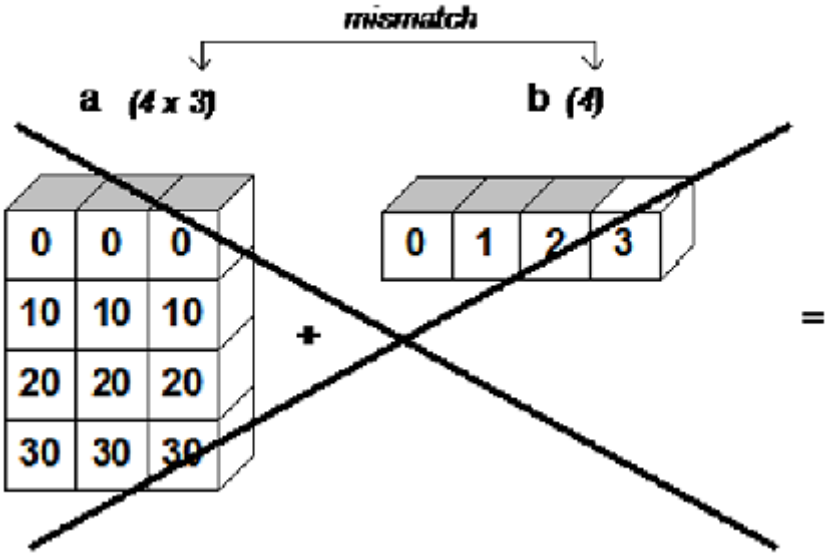
executed in 9ms, finished 21:42:35 2021-02-28

```
result=
[[ 0.  1.  2.]
 [10. 11. 12.]
 [20. 21. 22.]
 [30. 31. 32.]]
```

Cand se opereaza cu doua tablouri, NumPy compara dimensiunile - atributul `shape` - element cu element, incepand cu ultima dimensiune. Doua dimensiuni sunt compatibile cand:

1. sunt egale, sau
2. una din ele este 1

Regulile de mai sus nu sunt indeplinite, de exemplu, pentru:



sau pentru cazul simplu de mai jos:

In [87]:

```
x = np.arange(4)
y = np.ones(5)
print(x.shape, y.shape)
#print(x+y) #ValueError: operands could not be broadcast together with shapes (4,) (5,)
```

executed in 8ms, finished 21:42:35 2021-02-28

```
(4,) (5,)
```

Daca NumPy poate rezolva diferenta de dimensiune prin replicarea continutului (copiere), atunci va face acest lucru:

In [88]:

```
x = np.arange(4).reshape(4, 1)
print('x shape: ', x.shape)
print('x:\n', x)
```

executed in 10ms, finished 21:42:35 2021-02-28

```
x shape:  (4, 1)
x:
 [[0]
 [1]
 [2]
 [3]]
```

In [89]:

```
y = np.arange(5).reshape(1, 5)
print('y shape: ', y.shape)
print('y:\n', y)
```

executed in 8ms, finished 21:42:35 2021-02-28

```
y shape:  (1, 5)
y:
 [[0 1 2 3 4]]
```

In [90]:

```
z = x + y
print('z shape:', z.shape)
print('z\n', z)
```

executed in 6ms, finished 21:42:35 2021-02-28

```
z shape: (4, 5)
z
 [[0 1 2 3 4]
 [1 2 3 4 5]
 [2 3 4 5 6]
 [3 4 5 6 7]]
```

1.9 Exemplu concret

([Sursa (https://eli.thegreenplace.net/2015/broadcasting-arrays-in-numpy/)](https://eli.thegreenplace.net/2015/broadcasting-arrays-in-numpy/)) Se dau portiile de grasimi, proteine si carbohidrati dintr-un meniu. Sa se calculeze cate calorii reprezinta. Numarul de calorii se determina astfel:

1. nr de calorii pentru grasimi = 9 * grame grasimi
2. nr de calorii pentru proteine = 4 * grame proteine
3. nr de calorii pentru carbohidrati = 4 * numar grame carbohidrati

Food (1 serving)	Fats (g)	Protein (g)	Carbs (g)
Broccoli	0.3	2.5	3.5
Chicken breast	2.9	27.5	0
Banana	0.4	1.3	23.9
Raw almonds	14.4	6	2.3
...			

[9, 4, 4]

Food (1 serving)	Fats (cal)	Protein (cal)	Carbs (cal)
Broccoli	2.7	10	14
Chicken breast	26.1	110	0
Banana	3.6	5.2	95.6
Raw almonds	129.6	24	9.2
...			

Rezolvarea se fece prin aplicarea de broadcasting:

In [91]:

```
weights = np.array([
  [0.3, 2.5, 3.5],
  [2.9, 27.5, 0],
  [0.4, 1.3, 23.9],
  [14.4, 6, 2.3]])

cal_per_g = np.array([9, 4, 4])

#broadcasting
calories = weights * cal_per_g

print('Calorii:\n', calories)
```

executed in 9ms, finished 21:42:35 2021-02-28

```
Calorii:
 [[  2.7  10.   14. ]
 [ 26.1 110.    0. ]
 [  3.6   5.2  95.6]
 [129.6  24.    9.2]]
```

1.9.1 Bibliografie

Basic broadcasting: https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html (https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html)

http://scipy.github.io/old-wiki/pages/EricsBroadcastingDoc (http://scipy.github.io/old-wiki/pages/EricsBroadcastingDoc)

http://cs231n.github.io/python-numpy-tutorial/#numpy-broadcasting (http://cs231n.github.io/python-numpy-tutorial/#numpy-broadcasting)

1.10 Vectorizare

Exemple: https://www.kdnuggets.com/2017/11/forget-for-loop-data-science-code-vectorization.html (https://www.kdnuggets.com/2017/11/forget-for-loop-data-science-code-vectorization.html)

1.10.1 Exemplu

Se dau doua colectii de numere: prima contine distante parcurse, a doua timpul necesar pentru parcurgere. Se cere determinarea vitezelor corespunzatoare. Se va face implementare folosind ciclare (clasic) si vectorizare.

In [92]:

```
distante = [10, 20, 23, 14, 33, 45]
timpi = [0.3, 0.44, 0.9, 1.2, 0.7, 1.1]
```

executed in 9ms, finished 21:42:35 2021-02-28

In [93]:

```
#var 1: folosind ciclare
viteze = []
for i in range(len(distante)):
    viteze.append(distante[i] / timpi[i])

print('Viteze: ', viteze)
```

executed in 8ms, finished 21:42:35 2021-02-28

```
Viteze:  [33.333333333333336, 45.45454545454545, 25.555555555555554, 11.6666
66666666668, 47.142857142857146, 40.90909090909091]
```

In [94]:

```
# var 2: vectorizare
#vectorizarea numpy lucreaza peste tablouri, primul pas este obtinerea de tablouri din ce

distante_array = np.array(viteze)
timpi_array = np.array(timpi)

#se folosec operatii NumPy care trateaza tablourile in intregime. Codul C folosit pentru
#foloseste facilitatile de executie Single Instruction Multiple Data (SIMD) din microproc
#intregul array contine doar elemente de acelasi tip (floating point value, in acest caz)

viteze_array = distante/timpi_array

print(viteze_array)

# Pe langa asta, se stie deja ca intregul array contine doar elemente de acelasi tip (flo
# deci se evita verificarile tipurilor de date
```

executed in 9ms, finished 21:42:35 2021-02-28

```
[33.33333333 45.45454545 25.55555556 11.66666667 47.14285714 40.90909091]
```

1.10.2 Beneficii

1. Executie rapida
2. Cod mai scurt si deseori mai clar

Exemplu: sa se calculeze:

$$\sum_{i=0}^{N-1} (i\%3 - 1) \cdot i$$

In [95]:

```
#functie Python implementata naiv

N = 100000

def func_python(N):
    d = 0.0
    for i in range(N):
        d += (i%3-1) * i
    return d

print(func_python(N))
```

executed in 31ms, finished 21:42:35 2021-02-28

```
-33333.0
```

In [96]:

```
%timeit func_python(N)
```

executed in 2.17s, finished 21:42:38 2021-02-28

```
25.6 ms ± 4.51 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

In [97]:

```
#functie rescrisa folosind facilitatile NumPy si vectorizare

def func_numpy(N):
    i_array = np.arange(N)
    return ((i_array % 3 - 1 ) * i_array).sum()

print(func_numpy(N))
```

executed in 12ms, finished 21:42:38 2021-02-28

```
-33333
```

In [98]:

```
%timeit func_numpy(N)
```

executed in 11.1s, finished 21:42:49 2021-02-28

```
1.32 ms ± 70.2 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Majoritatea operatorilor si a functiilor NumPy lucreaza element cu element (sunt numite si universal functions, sau ufuncs) si intr-un mod optimizat (SIMD). Urmatoarele sunt ufuncs:

- operatori aritmetici: + - * / // % **
- operatii pe biti: & | ~ ^ >> <<
- comparatii: < <= > >= == !=
- functii matematice: np.sin, np.log, np.exp, ...
- functii speciale: `scipy.special.*`

Desi unele functii din NumPy se regasesc si in Python (ex: `sum` , `min` , `mean`), folosirea de ufunc duce la executie mai rapida:

In [99]:

```
from random import random
c = [random() for i in range(N)]
```

executed in 17ms, finished 21:42:49 2021-02-28

In [100]:

```
%timeit sum(c)
```

executed in 6.70s, finished 21:42:55 2021-02-28

```
840 µs ± 92.1 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

In [101]:

```
#vectorizare
c_array = np.array(c)
```

executed in 12ms, finished 21:42:55 2021-02-28

In [102]:

```
%timeit c_array.sum()
```

executed in 8.00s, finished 21:43:03 2021-02-28

```
97.6 µs ± 5.62 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

1.10.3 Exercitiu

Se dau n puncte in spatiul bidimensional, prin coordonatele lor memorate in 2 vectori **x** si **y**. Sa se determine care este cea mai apropiata pereche de puncte, considerand distanta Euclidiana:

$$d^2((x_i, y_i), (x_j, y_j)) = (x_i - x_j)^2 + (y_i - y_j)^2$$

In [103]:

```
n = 1000
x = np.random.random(size = n)
y = np.random.random(size = n)
```

executed in 5ms, finished 21:43:03 2021-02-28

In [104]:

```
# Varianta 1: se calculeaza matricea patratelor distantelor de dimensiune n*n. d[i, j] va
# punctul de coordonate(xi, yi) si cel de coordonate (xj, yj).
```

executed in 8ms, finished 21:43:03 2021-02-28

In [105]:

```
%%timeit

d = np.empty((n, n))
for i in range(n):
    for j in range(n):
        d[i, j] = (x[i] - x[j])**2 + (y[i]-y[j])**2
```

executed in 30.2s, finished 21:43:34 2021-02-28

```
3.77 s ± 74.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

In [106]:

```
#calculul perechii de puncte i, j cu i!= j pentru care distanta e minima
def pereche_apropiata(mat):
    n = mat.shape[0]
    #distanta dintre un punctsiel insusi este intotedauna 0; se vor exclude aceste cazuri
    i = np.arange(n)
    mat[i, i] = np.inf
    pos_flatten = np.argmin(mat)
    return pos_flatten // n, pos_flatten % n

# print(pereche_apropiata(d))
```

executed in 12ms, finished 21:43:34 2021-02-28

In [107]:

```
# Varianta 2: broadcasting, vectorizare
```

executed in 8ms, finished 21:43:34 2021-02-28

In [108]:

```
%%timeit

dx = (x[:, np.newaxis] - x[np.newaxis, :]) ** 2
dy = (y[:, np.newaxis] - y[np.newaxis, :]) ** 2

d = dx + dy
```

executed in 2.35s, finished 21:43:36 2021-02-28

```
28 ms ± 3.82 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

In [109]:

```
# decomentati %%timeit din celula anterioada daca vreti afisare
# print(pereche_apropiata(d))
```

executed in 8ms, finished 21:43:36 2021-02-28

1.10.4 Bibliografie

https://speakerdeck.com/jakevdp/losing-your-loops-fast-numerical-computing-with-numpy-pycon-2015
(https://speakerdeck.com/jakevdp/losing-your-loops-fast-numerical-computing-with-numpy-pycon-2015)

[Losing your Loops Fast Numerical Computing with NumPy (https://www.youtube.com/watch?v=EEUXKG97YRw)](https://www.youtube.com/watch?v=EEUXKG97YRw)