Lucrarea 1

Introducere în Python

Python este un limbaj de programare interpretat de uz general, dezvoltat de Guido van Rossum începând cu anul 1991. Limbajul permite gestionarea automată a memoriei (garbage collection) şi suportă multiple moduri de programare, cum ar fi programarea procedurală, obiect-orientată sau funţională. Versiunea curenta a limbajului este Python 3, introdusă in anul 2008. Această versiune nu este 100% compatibilua cu versiunea anterioară, Python 2, a cărei dezvoltare a încetat in anul 2020. Spre deosebire de alte limbaje de programare, Python foloseşte spatii albe şi indentări pentru a defini blocurile de instrucţiuni, spre deosebire de alte limbaje care î general folosesc acolade pentru acest scop.

În acest capitol sunt prezentate cateva noţiuni generale, folosind Python 3 (>= 3.6), necesare laboratorului de Teoria Transmisiunii Informaţiei.

1.1 Sintaxa Python

Codul Python poate for executat direct din linie de comandă sau poate fi salvat sub forma de fişiere `.py`. Limbajul Python este unul interpretat, nu compilat. Pentru interpretare, se poate folosi (`python` sau `python3` (în funcţie de versiunea instalată, Python 2 sau Python 3). În multe sisteme de operare Unix, ambele variante, Python 2 şi Python 3, sunt integrate în OS. În cadrul acestui laborator, vom folosi `python3`.

Un exemplu simplu de scriere a codului în linia de comandă este redat mai jos:

```
$ python3
Python 3.6.9 (default, Jul 17 2020, 12:50:27)
[GCC 8.4.0] on linux
Type "help", "copyright", "credits" or
"license" for more information.
>>> print("Hello World!")
Hello World!
```

Pentru a ieşi din linia de comandă Python, se poate folosi Ctrl + D sau quit().

În cazul salvării codului Python sub forma unui fişier cu extensia *.py*, în linia de comandă vom scrie

```
$ python3 hello_world.py
```

Rezultatul va fi următorul:

```
$ python3 hello_world.py
Hello World!
```

Python foloseşte indentarea pentru a crea blocuri de cod. Numărul de spaţii pentru indentare este la alegerea programatorului, dar cel puţin un spaţiu trebuie folosit pentru a fi creat un bloc de cod. Acelaşi număr de spaţii trebuie să fie folosit pentru tot codul în interiorul unui bloc.

Sintaxă corectă:

```
if 3 > 0:
    print("True")
```

Sintaxă invalidă:

```
if 3 > 0:
print("True")
```

1.2 Mediul de dezvoltare PyCharm

Pentru facilitarea dezvoltării aplicațiilor practice din cadrul laboratorului se recomandă utilizarea mediul de dezvoltare PyCharm, disponibil gratuit online [1].

Pentru a instala PyCharm cu suport Python 3 trebuie urmați următorii pași:

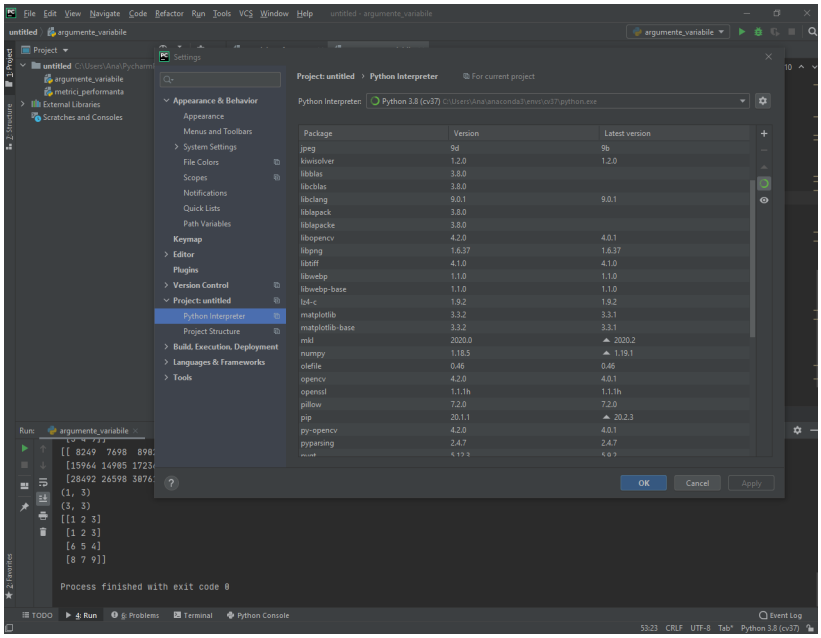
1. Descărcați un interpretor de Python (preferabil versiunea 3.6 sau una mai nouă) compatibil cu sistemul de operare pe care îl utilizați de la adresa [2].

2. Instalați interpretorul bifând opțiunea *Add Python 3.x to PATH*.

3. Adăugați interpretorul de Python în PyCharm din *Configure* → *Settings* → *Project Interpreter*

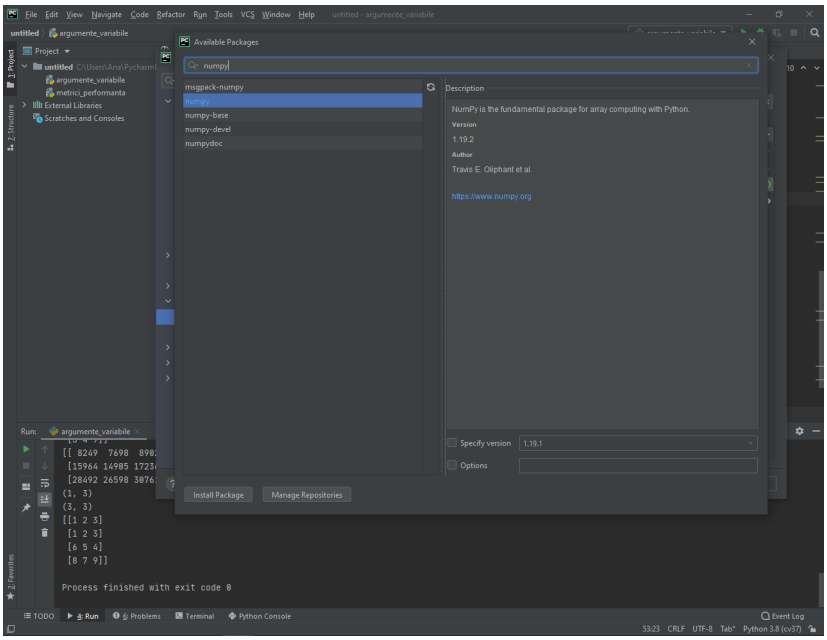
PyCharm permite instalarea, dezinstalarea și actualizarea pachetelor / bibiotecilor Python pentru un interpretor Python particular (de exemplu, NumPy, OpenCV, matplotlib). În mod implicit, PyCharm folosește `pip` pentru a gestiona pachetele. Pentru a face acest lucru, se selectează interpretorul Python fin Settings/Preferences sau se poate selecta Interpreter Settings în widget-ul Python Interpreter.

[1] `https://www.jetbrains.com/pycharm/download/`
[2] `https://www.python.org/downloads/`



Pentru a instala un pachet nou, se urmează pașii:

- Apăsați simbolul de adăugare +
- În caseta de dialog Available Packages care se deschide, se va selecta pachetul dorit (fie folosind Search, fie prin căutare directă a pachetului în listă).



- Opțional, se poate specifica versiunea dorită a pachetului (în caz că exixtă incompatibilități cu alte pachete instalate), alte opțiuni legate de `pip install`, directorul (`<path>`) în care

utilizatorul își instalează pachetele. Dacă `<path>` este gol, atunci instalarea pachetului se va face în directorul de pachete curent al interpretorului.

- După selectarea pachetului dorit, se apasă Install Package.

1.3 Variabile în Python

În Python variabilele nu au un tip predefinit (adică la două momente de timp aceeași variabilă poate avea atribuit un tip de dată diferit).
Tipurile de date principale în Python sunt:

- `int` – Pentru variabile de tip întreg;
- `float` – Pentru variabile de tip real (virgulă mobilă);
- `str` – Pentru șiruri de caractere (sau un singur caracter);
- `bool` – Pentru variabile de adevăr logic (`True` sau `False`).

Exemple:

```
x = 7
y = 9.5
prenume = 'Mihai'
q = True
```

Pentru variabilele de tip `str` nu este nici o diferență între utilizarea ghilimelelor simple ′ sau duble ”.

În Python variabilele de tip `str` sunt imutabile. Se numește imutabilă o variabilă a cărei valoare nu mai poate fi schimbată după atribuire.

Tipul de date `str` poate fi văzut ca o colecție, putând aplica asupra lui operații specifice colecțiilor.

1.4 Colecții în Python

1.4.1 list

O listă poate conține elemente ce au atribuite tipuri de date diferite. O listă se poate inițializa astfel:

```
lista = ['TTI', 2, True, 4.5]
lista_goala = []
```

Similar cu C++, accesarea elementelor din listă se face prin operatorul [ ], primul element având indexul 0. Diferența față de C++ este faptul că se pot accesa felii din listă prin specificarea indexului de început (inclusiv) și a celui de sfârșit (exclusiv). De asemenea se pot folosi și indecși negativi (în care -1 este primul element de la final, -2 este al doilea element de final, etc):

```
lista[2] = 'Programare' # Accesarea unui singur element
lista[1:3] = [5.4, False] # Accesarea unei felii din lista
lista[-2] = 3 # Accesarea celui de-al doilea element de la final
```

Pentru adăugarea de elemente în listă se pot folosi următoarele:

```
lista.append('Materia') # append adauga elementul la sfarsitul listei
lista.insert(3, False) # insert adauga elementul la pozitia ...
    specificata
```

Pentru eliminarea de elemente din listă se pot folosi următoarele:

```
lista.remove('Materia') # remove elimina primul element din lista ...
    egal cu argumentul specificat; argumentul trebuie sa existe in ...
    lista
del lista[2] # del elimina elementul cu indexul specificat
del lista[1:3] # sterge elemente din felia specificata
```

Atunci când unei variabile i se atribuie o listă (sau o altă variabilă de tip list) aceasta primește referința ei. Dacă două variabile au aceeași referință atunci când una se modifică și cealaltă variabilă suferă aceleași modificări. Pentru a copia o listă se folosește metoda copy:

```
lista_1 = [1, 2, 3]
lista_2 = lista_1
lista_3 = list_1.copy()
lista_2[1] = 4 # lista_2 = [1, 4, 3]; lista_1 = [1, 4, 3]; lista_3 ...
    = [1, 2, 3]
```

Operatorul de adunare între două liste are funcția de crearea a unei noi liste ce conține elementele celor două liste concatenate.

```
lista_1 = [1, 3]
lista_2 = [2, 4]
lista_3 = lista_1 + lista_2 # [1, 3, 2, 4]
```

Pentru a afla lungimea unei liste, se poate folosi funcția de tip build-in **len** (de exemplu, len(lista_1)).

1.4.2 tuple

Un tuplu reprezintă o listă imutabilă. Un tuplu se poate inițializa astfel:

```
tuplu = ('canale', 2, True, 4.5)
tuplu_singur = ('surse',)
```

Similar ca o listă elementele dintr-un tuplu pot fi accesate prin index, felie sau index negativ. Diferit față de o listă elementele din tuplu nu pot fi modificate sau șterse. Pentru a crea un tuplu ce conține un singur element este nevoie să fie pusă virgulă după el.

1.4.3 dict

Dicționarele repezintă colecții asociative de tip cheie - valoare. Fiecărei chei îi poate corespunde o singură valoare. Dicționarele se pot inițializa astfel:

```
dictionar = {'cod': 'Hamming', 1: False, 2: 'adevarat'}
dictionar_gol = {}
```

Pentru accesarea elementelor dintr-un dicționar se poate folosi operatorul [ ] sau metoda get.

```
i = dictionar['cod']      # Valoarea 'cod' trebuie sa existe in ...
    dictionar
j = dictionar.get(0, -1) # Daca valoarea 0 nu exista in dictionar ...
    va returna valoarea -1
```

Pentru adăugarea sau modificarea de elemente în dicționar se poate folosi următoare:

```
dictionar['cod'] = 'convolutional' # Daca cheia 'cod' exista in ...
    dictionar, se modifica valoarea ei cu 'convolutional' altfel ...
    se adauga perechea ('cod', 'convolutional')
```

Pentru eliminarea de elemente din dicționar se poate folosi următoarea:

```
del dictionar[1] # Sterge pereche cu cheia 1
```

Metodele .keys(), .values() și .items() returnează o vedere (modificările asupra dicționarului sunt reflectate în vedere) asupra cheilor, valorilor sau perechilor din dicționar. Dacă este necesar vederea poate fi convertită la o listă.

1.4.4 range

Intervalele reprezintă colecții secvențiale imutabile. Ele pot fi construite astfel:

```
interval_1 = range(4)        # [0, 1, 2, 3]
interval_2 = range(1, 5)     # [1, 2, 3, 4]
interval_2 = range(2, 6, 3) # [2, 5]
```

Comparativ cu listele, intervalele sunt secvențe optimizate pentru performanță. Elementele individuale dintr-un interval pot fi accesate prin operatorul [ ].

1.5 Instrucțiunile de control ale limbajului Python

La finalul oricărei instrucțiuni de control este necesară folosirea operatorului :. Față de C++, instrucțiunile de control nu necesită paranteze. Blocul de instrucțiuni ce se subordonează unei instrucți de control trebuie să aibă o indentare mai mare decât a instrucțiunii căreia i se subordonează.

1.5.1 Instrucțiuni condiționale

Instrucțiunile condiționale din Python sunt `if` `elif` (else + if) și `else`. Instrucțiunea `if` trebuie să fie întotdeauna prima dintr-o înșiruire de instrucțiuni condiționale, instrucțiunea `else` poate să fie doar ultima dintr-o înșiruire iar instrucțiunea `elif` se poate folosi în rest.

```
if numar % 2 == 0:
    valoare = 3
else:
    valoare = 5
```

Dacă se dorește utilizarea mai multor condiții acestea se pot combina folosind operatorul logic `or` (sau logic) sau operatorul logic `and` (și logic). Negarea unei condiții se realizează prin operatorul logic `not` (negare logică).

Pentru a verifica apartenența unei valori (sau variabile) la o colecție se poate folosi operatorul `in`:

```
text = 'Laboratorul de TTI se face in Python'
i = 'TTI se' in text       # True
j = 'n Matlab' in text      # False
```

Pentru a verifica egalitatea în referință (dacă două variabile au aceeași referință) se folosește operatorul `is`.

```
lista = [1, 2, 3]
lista_2 = lista.copy()
lista_3 = lista
a = lista_2 is lista # False
b = lista_3 is lista # True
```

1.5.2 Instrucțiuni repetitive

Instrucțiunile repetitive din Python sunt `for` și `while`.
Instrucțiunea repetitivă `for` se poate folosi doar pentru colecții, parcurgând fiecare element din colecția respectivă. Elementul folosit în iterație reprezintă o copie a elementului din colecție, deci modificarea lui nu conduce la modificarea colecției iterate. Forma generală a unei instrucțiuni `for` este:

```
for variabila in colectie:
    instructiuni
```

Instrucțiunea repetitivă `while` este similară cu cea din C++. Forma generală este:

```
while conditii:
    instructiuni
```

1.5.3 Operatorul condițional

Python permite un operator condițional (similar operatorului condițional `?  :` din C++). Operatorul condițional este folosit în locul unei instrucțiuni condiționale ce are ca scop atribuirea unei variabile. De exemplu, următoarele variante sunt echivalente:

```
if numar % 2 == 0:
    valoare = 3
else:
    valoare = 5

valoare = 3 if numar % 2 == 0 else 5
```

1.6 Instrucțiuni de comprimare

Python permite comprimarea construcțiilor de colecții într-un mod funcțional. De exemplu, următoarele variante sunt echivalente:

```
lista = [1, 2, 3, 4, 5, 6]
lista_pare_1 = []
```

```
for element in lista:
    if element % 2 == 0:
        lista_pare_1.append(element)

lista_pare_2 = [element for element in lista if element % 2 == 0]
```

1.7 Funcții

În mod similar cu variabilele, argumentele unei funcții nu au un tip predefinit, același argument putând avea două tipuri diferite la două apeluri distincte. Totuși, este permisă oferirea de indicii despre tipul unui argument sau al rezultatului dar acesta nu poate fi forțat. Pentru a oferi un indiciu de tip unui argument se folosește formularea *argument: tip* iar pentru a oferi un indiciu rezultatului se folosește formularea *functie(argumente) -> tip*. O funcție ce calculează suma a doi întregi este prezentată mai jos:

```
def suma(a: int, b: int) -> int:
    return a + b
```

Similar cu C++, se pot defini funcții cu argumente implicite. La fel ca în C++ argumentele implicite trebuie să fie cele de la sfârșit, dar față de C++ este mai ușor să oferim doar o parte din argumentele implicite prin numele lor. Totuși, odată ce am început să oferim argumente prin nume, toate argumentele ce urmează trebuie oferite prin nume, inclusiv cele neimplicite care nu au fost date. Un exemplu este prezentat mai jos:

```
def f(a, b=0, c=0):
    return 2 * a + 1.5 * b + 3 * c

n1 = f(2)           # 4.0
n2 = f(2, 1)        # 5.5
n3 = f(2, b=1)      # 5.5
n4 = f(2, c=1)      # 7.0
n5 = f(c=1, a=3)    # 9.0
```

O altă diferență majoră față de C++ este reprezentată de modul în care sunt transmise argumentele. Mai precis, ele sunt întotdeauna transmise prin atribuire; argumentele ce au tipuri de date ce realizează copiere la atribuire (precum int și float) vor fi copii ale variabilelor originale în timp ce argumentele ce au tipuri de date ce realizează referințe la atribuire (precum list și dict) vor fi referințe ale variabilelor originale.
Nu în ultimul rând, datorită modului în care este interpretat codul Python, funcțiile pot "vedea" orice variabilă din blocurile căreia i se subordonează. De exemplu, dacă dorim să afișăm pentru fiecare număr mai mic sau egal cu N care sunt divizorii lui, următoarea bucată de cod este corectă:

```
for i in range(1, N):
    for j in range(1, i + 1):
        def f():
            return i % j == 0

        if f():
            print(F'{j} este divizorul lui {i}')
```

```
    print()
```

Dacă numărul argumentelor unei funcții nu este cunoscut, numele parametrului de intrare al unei funcții poate fi precedat de * (de exemplu, *args):

```
def suma(*numere):
    s = 0
    for n in numere:
        s += n

    return s

s1 = suma(3)
s2 = suma(3, 6)
s3 = suma(3, 7, 4.65)
print(s1, s2, s3)
```

O funcție poate primi un număr arbitrar de cuvinte cheie drept argumente. În acest caz, parametrul funcției va fi precedat de ** (de exemplu, **kwargs):

```
def vcard(**contact):
    for key, item in contact.items():
        print('{}: {}'.format(key, item))

vcard(prenume='Ion', nume='Popescu')
vcard(prenume='Cristina',nume='Chiran',job='profesor')
```

1.8 Paradigma funcțională

Atunci când dezvoltarea de programe are ca scop prelucrarea de date numerice, cum este cazul acestui laborator, este recomandată folosirea paradigmei funcționale. Paradigma funcțională presupune implementarea de funcții și există o serie de construcții de bază indiferent de limbajul folosit.
În sensul ei strict, o funcție reprezintă o bucată de cod parametrizată ce poate fi refolosită și care nu produce efecte secundare (nu modifică nimic în memorie: argumente, variabile globale, etc). Acest aspect oferă argumentelor sale o garanție de imutabilitate (acesta fiind primul pas pentru paralelizarea programelor). De exemplu, doar una din următorele două exemple este funcție în sensul ei strict:

```
def functie_1(lista):
    lista.append('nou')
    return lista

def functie_2(lista):
    aux = lista.copy()
    aux.append('nou')
    return aux
```

Translatarea reprezintă o construcție de bază a paradigmei funcționale. Translatarea presupune aplicarea unei funcții pentru fiecare element dintr-o colecție. Funcția trebuie să primească un

singur argument: elementul curent. Putem calcula pătratul pentru fiecare element dintr-o listă astfel:

```
lista = [1, 4, 2, 6]
f = lambda x: x*x
lista_patrate = list(map(f, lista))
```

Reducerea reprezintă o construcție de bază a paradigmei funcționale. Reducerea presupune aplicarea unei funcții pentru fiecare element dintr-o colecție cu scopul de a obține o singură valoare redusă. Funcția trebuie să primească două argumente, și anume valoarea redusă la momentul curent și elementul curent. Putem calcula suma numerelor dintr-o listă astfel:

```
import functools

lista = [1, 4, 2, 5]
suma = functools.reduce(lambda ans, el: ans + el, lista, 0)
```

Filtrarea reprezintă o altă construcție de bază a paradigmei funcționale. Filtrarea presupune aplicarea unei funcții pentru fiecare element dintr-o colecție și păstrarea elementelor pentru care funcția întoarce valoarea de adevăr True. Funcția trebuie să primească un singur argument, și anume elementul curent și să returneze o valoare de adevăr. Putem alege numerele pare dintr-o listă astfel:

```
lista = [1, 4, 2, 5]
lista_pare = list(filter(lambda el: el % 2 == 0, lista))
```

Deși o parte din aceste rezultate se pot obține și cu ajutorul instrucțiunilor de comprimare, cele din urmă reprezintă doar un mod scurtat de a scrie instrucțiuni repetitive și condiționale în timp ce construcțiile prezentate pot fi paralelizate și sunt deja implementate paralel în biblioteci precum NumPy, SciPy, TensorFlow, etc.

1.9 Biblioteca NumPy

NumPy este o bibliotecă specializată pentru calcul vectorial, fiind una dintre cele mai populare biblioteci pentru aplicații de calcul numeric. Aceasta poate fi folosită după ce este importată în fișier, folosind deseori acronimul **np**:

```
import numpy as np
```

Vectorii din NumPy au ca tip de date np.ndarray (numpy n dimensional array). Comparativ cu listele, vectorii din NumPy au o dimensiune fixă, modificarea dimensiunii fiind costisitoare în timp deoarece presupune alocări de memorie și copieri în memorie. Pentru a crea un ndarray pornind de la o listă se poate folosi funcția *array*:

```
lista = [1, 4, 2, 5]
ndarray = np.array(lista)          # ndarray 1 dimensional (vector)

ndarray_2 = np.array([[1, 2],
                      [3, 4]])     # ndarray 2 dimensional (matrice)
```

Alte funcții pentru a crea ndarray-uri:

- **np.zeros(shape)** – crează un ndarray cu valori egale cu 0. Argumentul *shape* reprezintă un tuplu cu dimensiunile dorite. De exemplu, (3, 3) pentru o matrice 3x3 sau (5,) pentru un vector de 5 elemente;
- **np.ones(shape)** – creează un ndarray cu valori egale cu 1;
- **np.empty(shape)** – creează un ndarray gol (valorile nu sunt relevente). El trebuie să fie populat ulterior;
- **np.linspace(start, stop, num)** – creează un ndarray cu dimensiunea $(num,)$ și valori egal distanțate între start și stop. Stop poate fi exclusiv;
- **np.eye(N, M, k)** – creează un ndarray cu dimensiunea (N, M) ce are valori egale cu 0 mai puțin pe a k-a diagonală unde sunt valori de 1. Pentru $k = 0$ se obține matricea identitate;
- **np.random.standard_normal(shape)** – creează un ndarray cu valori aleatoare normale (medie 0 și distribuție 1);
- **np.random.random_integers(L, H, shape)** – creează un ndarray cu valori întregi aleatoare între L și H inclusiv.

Câteva exemple sunt date mai jos:

```
zero_array = np.zeros(3)
one_array = np.ones((1, 4))
empty_array = np.empty((2, 2))
print(zero_array)
print(one_array)
print(empty_array)

# np.arange(start, stop, pas)
range_array = np.arange(2, 10, 3)
# np.linspace(start, stop, numar_elemente)
linspace_array = np.linspace(0, 1, 11)
print(range_array)
print(linspace_array)

I3 = np.eye(3) # matricea identitate
print(I3)
```

Pentru a obține dimensiunea unui ndarray se poate folosi atributul **shape**, iar pentru a afla numărul de dimensiuni se poate folosi atributul **ndim**. Acesta este întotdeauna sub forma unui tuplu:

```
a = np.random.standard_normal((4, 2, 4))
nr_dim = a.ndim # 3
aux = a.shape # (4, 2, 4)
print('Numar de dimensiuni: ', nr_dim, ' ', 'Dimensiunile sunt: ', ...
    aux)
```

Toate operațiile matematice de bază (+, -, *, /) sunt suportate de ndarray-uri. Dacă ndarray-urile au aceeași dimensiune, operațiile se fac la nivel de element. Dacă nu au aceeași dimensiune, se va încerca răspândirea celor două ndarray-uri. Procesul de răspândire presupune următoarele:

- Dacă un ndarray are mai puține dimensiuni decât celălalt, i se completează cu 1 în față până ajunge să aibă aceleași dimensiuni;
- Se verifică dimensiunile de pe aceeași poziție: dacă cel puțin o pereche de dimensiuni sunt diferite și niciuna nu este egală cu 1, atunci nu se poate realiza răspândirea;
- Fiecare ndarray se repetă pe dimensiunile egale cu 1 până ajunge ca aceea dimensiune să fie egală cu a celuilalt;
- Se realizează operația element cu element;

Procesul de răspândire are avantaje în operațiile vectoriale cu ndarray-uri. De exemplu, dacă a este un ndarray cu 2 dimensiuni (matrice) la care dorim să adăugăm pe fiecare linie b care este un ndarray cu 1 dimensiune (vector) este suficient să scriem a + b. În mod similar, orice operație cu scalari este afectată de răspândire, ca de exemplu adăugarea la fiecare element din a valoarea 2 care se scrie ca $a + 2$.

Accesarea elementelor într-un NumPy array în Python este similară listelor, folosind indexarea și felierea.

```
a = np.random.randint(30, size=(4, 2, 4))
print(a)
print(a[2,:])
print(a[2,0,:])
print(a[2,0,0])
```

Biblioteca NumPy pune la dispoziție și o serie de operații cu array-uri. O parte din operațiile utilizate în cadrul laboratorului sunt prezentate mai jos:

- adunarea, scăderea, multiplicarea și împărțirea element-cu-element poate fi efectuată folosind operațiile matematice standard (+, -, *, /)
- np.dot() – dacă a și b sunt vectori, reprezintă produsul scalar între cei doi vectori; dacă a și b sunt matrici 2-dimensionale, reprezintă înmulțire de matrici (Atenție! Se preferă folosirea np.matmul sau a@b pentru această operație)
- ndarray.transpose() – returnează matricea transpusă
- np.linalg.matrix_power() – returnează matricea ridicată la o putere
- **np.log(x) / np.log2(x) / np.log10(x)** – calculează logaritmul în bază e / 2 / 10 pentru fiecare element din x
- **np.abs(x)** – calculează modulul fiecărui element din x
- **np.exp(x)** – calculează e la puterea fiecărui element din ndarray.

```
A = np.array([1, 2, 3])
B = np.array([[1, 2, 3], [6, 5, 4], [8, 7, 9]])
print(A + B)
print(A - B)
print(A * B)
print(A / B)
print(np.matmul(A, B))
print(B.T)  # print(B.transpose())
print(np.linalg.matrix_power(B, 4))
```

Stivuirea matricilor (prin adăugarea unor linii și coloane) poate fi făcută folosind metodele np.vstack() și np.hstack(), cu condiția ca dimensiunile să se potrivească.

```
A = np.array([1, 2, 3])
B = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(A.shape) # (3, )
print(B.shape) # (3, 3)
C = np.vstack((B, A))
D = np.hstack((B, np.array([A]).T))
print(C)
print(D)
```

O altă variantă de concatenare a matricilor este folosirea metodei np.concatenate(), cu condiția ca dimensiunile să se potrivească. Argumentul axis se folosește pentru a specifica dimensiunea în lungul căreia se face concatenarea.

```
A = np.array([[1, 2, 3]])
B = np.array([[1, 2, 3], [6, 5, 4], [8, 7, 9]])
print(A.shape) # (1, 3)
print(B.shape) # (3, 3)
c = np.concatenate((A, B), axis=0)
print(c)
```

Observați diferențele între cele două abordări.

1.10 Reprezentare grafică folosind Matplotlib

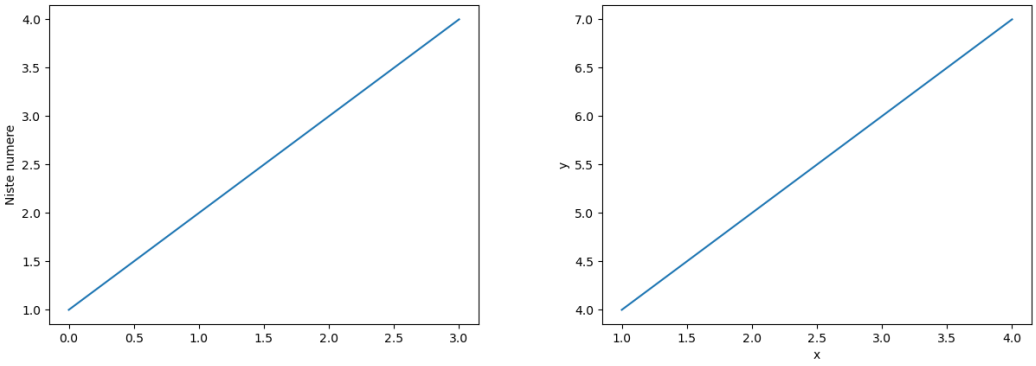
matplotlib.pyplot este o colecție de funcții care permit efectuarea unor reprezentări grafice. Un exemplu de reprezentare grafică este dat mai jos.

```
import matplotlib.pyplot as plt
import numpy as np
plt.plot([1, 2, 3, 4])
plt.ylabel('Niste numere')
plt.show()
plt.plot([1, 2, 3, 4], [4, 6, 5, 7])
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

Rezultatele sunt redate în figura 1.1.

Pentru a diferenția două grafice reprezentate în aceeași figură, se poate folosi o legendă, precum în exemplul de mai jos:

```
t = np.arange(0., 4., 0.125)
fig, ax = plt.subplots()
ax.plot(t, t, 'r—', label = 't')
ax.plot(t, t**2, 'bs', label = 't**2')
ax.plot(t, t**3, 'g^', label = 't**3')
ax.legend()
plt.show()
```



(a) (b)

Figura 1.1: Exemple Matplotlib Pyplot

Rezultatul este redat în figura 1.2.

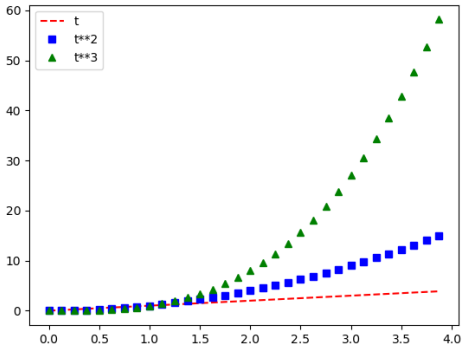


Figura 1.2: Exemplu legendă

1.11 Citirea unei imagini folosind librăria OpenCV

OpenCV "Open Source Computer Vision Library" este o bibliotecă de funcții open source pentru prelucrarea și analiza imaginilor (dar și secvențelor video), ce a pornit ca un proiect al Intel în 1999. Primii pași au fost făcuți de Gary Bradsky și Vadim Pisarevsky. OpenCV suportă un număr mare de limbaje de programare, cum ar fi C, C++, Python, Java, etc. și este disponibilă pe mai multe platforme (Linux, Windows, Android, iOS). OpenCV-Python este API-ul Python pentru OpenCV.

În cadrul laboratorului, vom folosi funcția de citire a unei imagini, și anume **cv2.imread()**. Imaginea ce se dorește a fi citită trebuie să se găsească în directorul de lucru sau se poate furniza întreaga cale către imagine. Presupunând că există o imagine 'tti.png' în directorul de lucru,

citirea unei imagini se face astfel:

```
import numpy as np
import cv2
# Incarcarea unei imagini cu mai multe niveluri de gri
img = cv2.imread('tti.jpg',cv2.IMREAD_GRAYSCALE)
```

Pentru a afișa imaginea citită, se poate folosi funcția **cv2.imshow**:

```
cv2.imshow('image',img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

1.12 Exercițiu rezolvat

Într-un oraș, c% din cetățeni sunt conservatori, s% socialiști și l% liberali. În acest oraș, la ultimele alegeri, p_c% din conservatori au votat, la fel au făcut p_s% din socialiști și p_l% din liberali. O persoană din oraș este selectată aleator. Ea afirmă că a votat la ultimele alegeri. Care este probabilitate să fie conservatoare? Scrieți o funcție care să calculeze această probabilitate în funcție de parametrii de intrare c, s, l, p_c, p_s, p_l. În main, se citesc parametrii de la tastatură (de tip int) și afișați rezultatul pentru valorile introduse. Rezultatele vor fi de tip float, cu două zecimale.

```
def get_conservator_chance(c, s, l, pc, ps, pl):
    return (100 * pc * c) / (pc * c + ps * s + pl * l)

c, s, l, pc, ps, pl = list(map(int, input().split()))

print(F'{get_conservator_chance(c, s, l, pc, ps, pl):.2f}')
```

1.13 Întrebări şi exerciţii

1. Implementați funcția *numara_caractere(text: str) -> dict*. Funcția primește ca parametru un text și returnează un dicționar cu frecvența fiecărui caracter din text. De exemplu, pentru textul 'teoria transmisiunii informatiei', funcția returnează dicționarul {'t': 3, 'e': 2, ' o': 2, 'r': 3, 'i': 8, 'a': 3, 'n': 3, 's': 2, 'm': 2, 'u': 1, 'f': 1, ' ': 2}.
2. Implementați funcția *gaseste_maxim_aparitii(lista: list) -> tuple*. Funcția primește ca parametru o listă de întregi și returnează un tuplu format din numărul care apare de cele mai multe ori și numărul său de apariții. Dacă există mai multe numere care apar de un număr maxim de ori se va alege cel mai mic. De exemplu, pentru lista [1, 4, 3, 2, 1, 4], funcția va returna (1, 2).
3. Implementați funcția *normalizeaza(vector: np.ndarray) -> np.ndarray*. Funcția primește ca parametru un vector de tip numpy array și returnează vectorul modificat astfel încât media elementelor să fie 0 și dispersia să fie 1.
4. Implementați funcția *vector_in_matrice(matrice: np.ndarray, vector: np.ndarray) -> bool*. Funcția primește ca parametri o matrice și un vector și returnează True dacă vectorul se regăsește printre liniile matricei, altfel False.