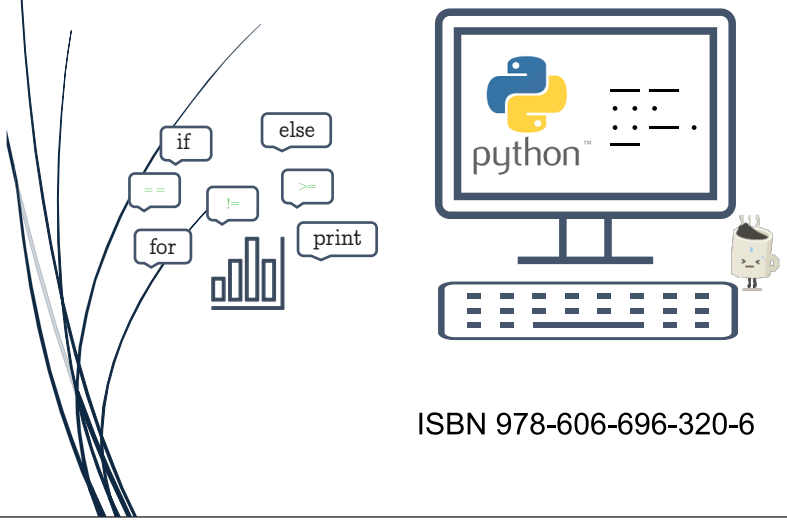
LIMBAJUL PYTHON ÎN APLICAȚII INGINEREȘTI. LUCRĂRI PRACTICE

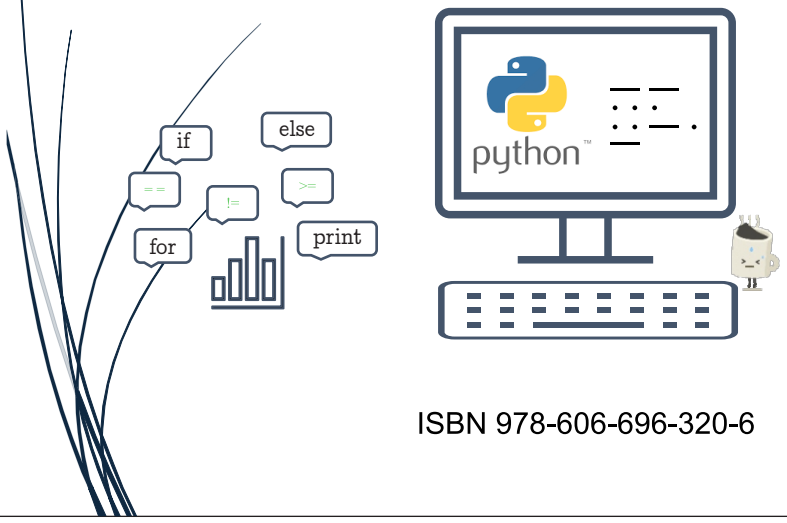
Iulian VASILIEV
Larisa DIACONU
Marian BARBU



ISBN 978-606-696-320-6

LIMBAJUL PYTHON ÎN APLICAȚII INGINEREȘTI. LUCRĂRI PRACTICE

Iulian VASILIEV
Larisa DIACONU
Marian BARBU



ISBN 978-606-696-320-6

Iulian VASILIEV
Larisa DIACONU
Marian BARBU

LIMBAJUL PYTHON ÎN APLICAȚII INGINEREȘTI

LUCRĂRI PRACTICE

GALATI UNIVERSITY PRESS
2024

Copyright © 2024 Galaţi University Press
Toate drepturile rezervate. Nicio parte a acestei publicaţii nu poate fi reprodusă în nicio formă, fără acordul scris al editurii.

Colecţia Ştiinţe Inginereşti

Galaţi University Press – Cod CNCSIS 281
Editura Universităţii „Dunărea de Jos”
Str. Domnească, nr. 47, 800008 – Galaţi, ROMÂNIA
Tel. 0336 130 139; Fax: 00 40 236 461 353
gup@ugal.ro

Referenți științifici:
Conf. Dr. Daniela CERNEGA
Prof. Dr. Sergiu Viorel CARAMAN

ISBN 978-606-696-320-6

Tipărit la Editura Universităţii “Dunărea de Jos” din Galaţi - GALATI UNIVERSITY PRESS.

PREFAȚĂ

Acest îndrumar de lucrări practice își propune să vă familiarizeze și să vă îndrume în explorarea detaliată a diverselor aspecte ale limbajului Python, oferind un ghid comprehensiv pentru dezvoltarea abilităților în programare și utilizarea acestora în aplicații inginerești.

Îndrumarul acoperă o gamă largă de subiecte, începând de la aspectele fundamentale ale limbajului Python, precum funcții și structuri de date, până la teme avansate, cum ar fi: programarea concurentă, calculul numeric și simbolic. Structurat în capitole clare și practice, acesta este destinat să servească drept ghid util pentru studenți, programatori și ingineri, fie că sunt la început de drum sau doresc să-și aprofundeze cunoștințele.

Sperăm ca această resursă să vă servească asemenea unui ghid de încredere în călătoria dumneavoastră în domeniul aplicațiilor inginerești cu Python. Prin abordarea practică și structurarea tematică, intenționăm să vă oferim instrumentele necesare pentru a vă dezvolta abilitățile și înțelegerea într-un mod coerent și eficient.

Succes în explorarea acestui domeniu captivant!

Autorii,

dr. ing. Iulian Vasiliev,

dr. ing. Larisa Diaconu,

prof. dr. ing. Marian Barbu

CUPRINS

Lucrarea 1	Introducere în *Python*	5
Lucrarea 2	Funcții. Structuri de date	28
Lucrarea 3	Librăria *NumPy*	43
Lucrarea 4	Librăria *Pandas*	58
Lucrarea 5	Librăria *Matplotlib*	75
Lucrarea 6	Calcul simbolic. Librăria *SymPy*	89
Lucrarea 7	Calcul numeric. Librăria *SciPy*	99
Lucrarea 8	Funcționalități avansate ale limbajului *Python*	113
Lucrarea 9	Programare orientată pe obiect în *Python*	126
Lucrarea 10	Programare concurentă în *Python*	140
Bibliografie		152

Lucrarea 1. Introducere în Python

A. Obiectivele lucrării

1. Prezentarea generală a mediului de dezvoltare ***PyCharm***
2. Prezentarea modului de instalare a librăriilor ***Python***
3. Prezentarea bazelor limbajului ***Python***
4. Prezentarea modului de definire și afișare a variabilelor în ***Python***
5. Prezentarea operatorilor de bază în ***Python***

B. Ce este Python?

Python este un limbaj de programare de nivel înalt, de utilitate generală, putând fi folosit în aproape orice domeniu (dezvoltarea de aplicații mobile/PC/web, dezvoltarea de jocuri video, dezvoltarea de aplicații industriale sau de laborator: inteligență artificială, procesare de date, controlul sistemelor, calcul ingineresc etc.) [1].

Printre avantajele acestui limbaj de programare se numără [2]:

1. Este considerat un limbaj de nivel înalt, adică instrucțiunile acestuia sunt apropiate de limbajul uman.
2. Codul nu este compilat, ci interpretat, economisindu-se astfel timp atât la dezvoltarea, cât și la depanarea aplicațiilor.
3. Există un număr foarte mare de librării ce sunt disponibile în mod gratuit.
4. Oferă posibilitatea mai multor tipuri de programare: programare structurată, programare funcțională și programare orientată pe obiect.

C. Instalare

Descărcarea kit-ului de instalare se poate face din secțiunea „***Downloads***” a site-ului oficial Python: https://www.python.org/. Se recomandă instalarea versiunii ***Python 3.9*** pentru buna funcționare a exemplelor de aplicații din acest îndrumar.

Există mai multe metode de a utiliza Python:

1. Comenzile pot fi introduse în mod interactiv. Pentru această metodă se va deschide o fereastră de „***Command Prompt***” sau „***PowerShell***” în care se va introduce comanda **`python`**. După apariția prompt-ului **`>>>`**, se pot introduce instrucțiuni ***Python*** ce se vor executa în momentul apăsării tastei ***Enter***. *Figura 1* prezintă un exemplu de utilizare a limbajului ***Python*** în mod interactiv.

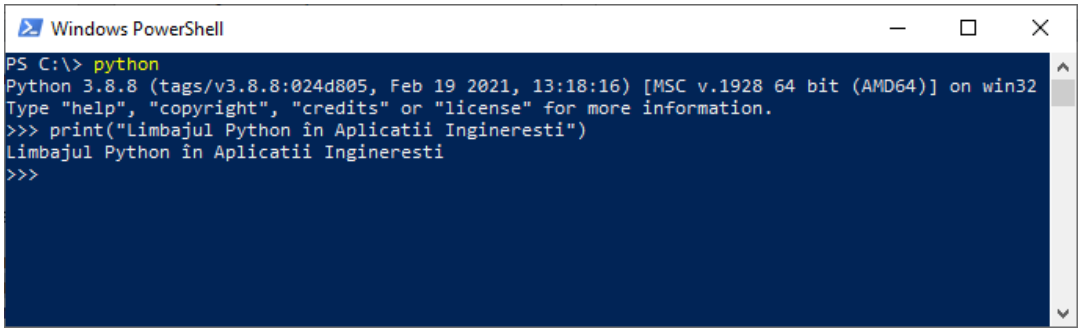


Figura 1 – Exemplu de utilizare a limbajului Python în mod interactiv

2. Instrucțiunile ***Python*** pot fi scrise într-un fișier script cu extensia **`.py`** folosind orice editor de text și apoi rulate din „***Command Prompt***” sau „***PowerShell***”, introducând comanda **`python`** urmată de numele fișierului script, incluzând extensia. *Figura 2* prezintă un astfel de exemplu.

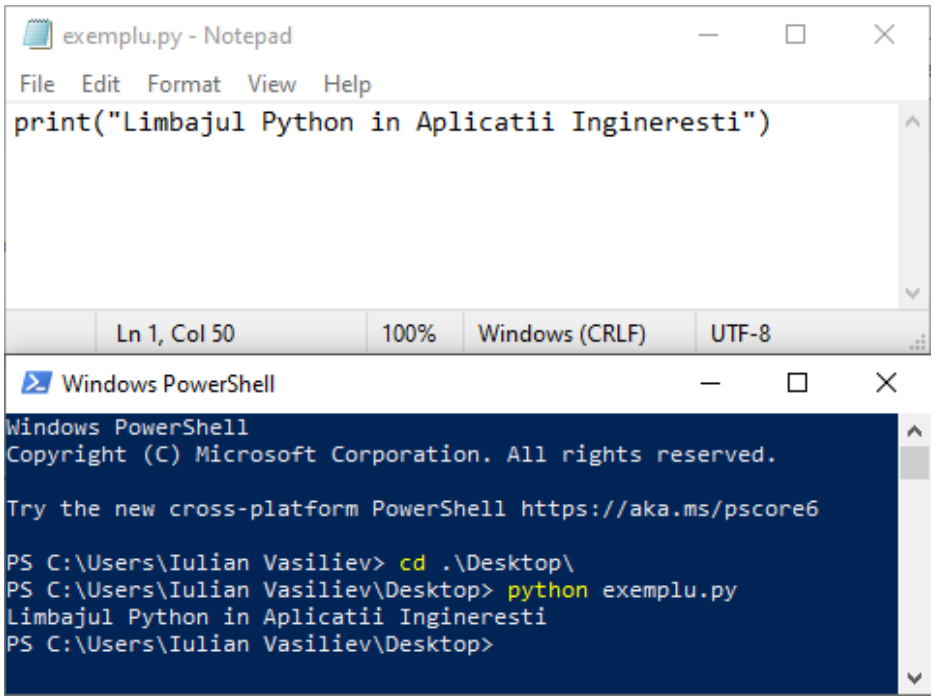


Figura 2 – Exemplu de rulare a scripturilor Python din linie comandă

3. Utilizarea unui mediu de dezvoltare (IDE – Integrated Development Environment) care facilitează dezvoltarea aplicațiilor prin îmbinarea, într-o singură interfață, a unei serii de instrumente pentru dezvoltare, precum: managementul aplicației, editor de cod sursă, managementul librăriilor, depanare etc. În cadrul acestui laborator, se va folosi versiunea community a mediului de dezvoltare ***PyCharm***. Descărcarea kit-ului de instalare al acestui IDE se poate face din secțiunea „***Downloads***” a site-ului oficial PyCharm: https://www.jetbrains.com/pycharm/.

D. Prezentarea mediului de dezvoltare PyCharm

La deschiderea mediului de dezvoltare PyCharm, veți fi întâmpinați de fereastra de bun venit (*Figura 3*), care va permite crearea unui nou proiect sau deschiderea unui proiect deja existent. În cazul în care, la rularea anterioară, proiectul în care s-a lucrat nu a fost închis, PyCharm va deschide direct acel proiect. În acest caz, pentru a ajunge la fereastra de bun venit se va închide proiectul deschis din meniul „***File***” → „***Close project***”.

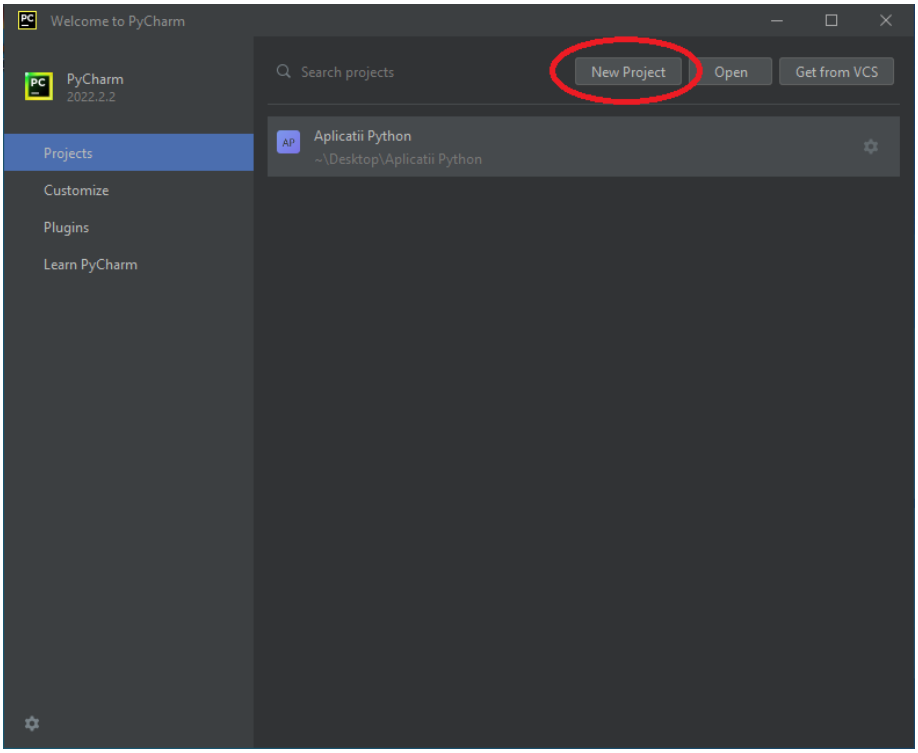


Figura 3 – Fereastra de bun venit a mediului de dezvoltare PyCharm

În fereastra de bun venit, prin apăsarea butonului „***New Project***” (încercuit cu roșu în *Figura 3*), va apărea fereastra ce permite crearea unui nou proiect (*Figura 4*). În această fereastră se poate seta, la câmpul „***Location***”, locația proiectului, după care, prin apăsarea butonului „***Create***”, noul proiect va fi creat și va apărea fereastra ce va permite editarea codului proiectului creat (*Figura 5*).

Această fereastră conține mai multe sub-ferestre („***views***”) de bază. În partea stângă, se află sub-fereastra „***Project***” ce permite navigarea prin fișierele și directoarele proiectului, iar în partea dreaptă avem editorul de cod propriu-zis. Din partea de jos se pot deschide o serie de alte sub-ferestre cu diferite utilități.

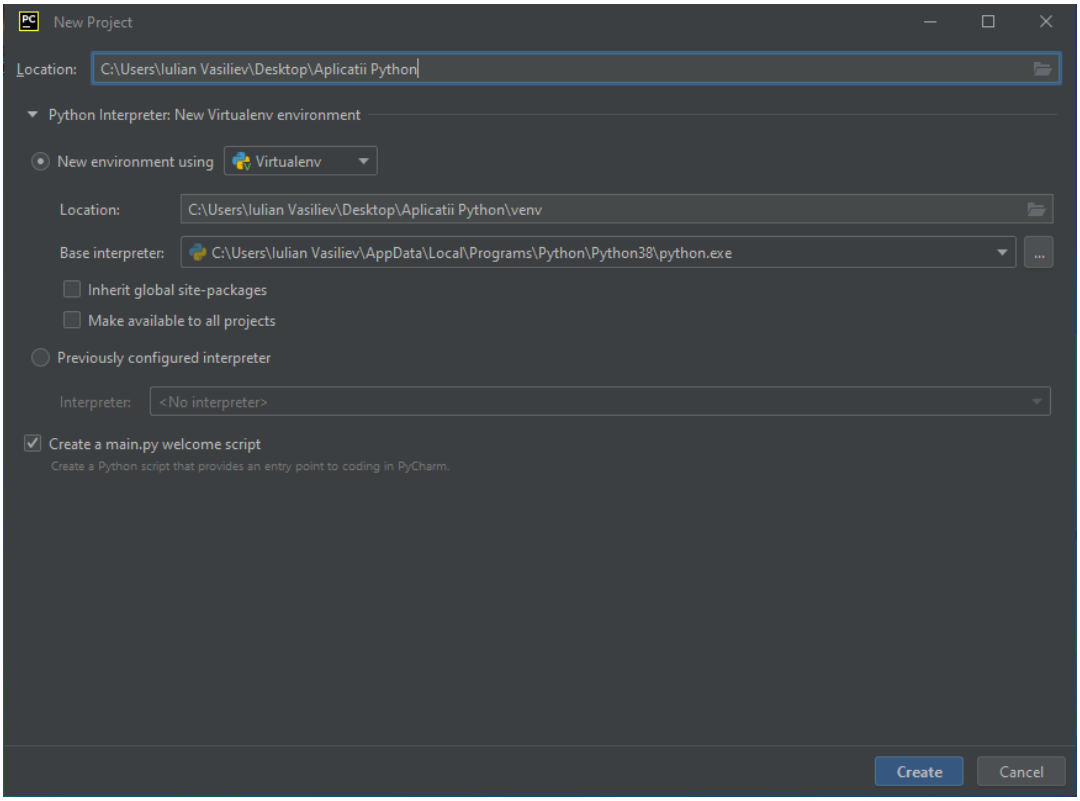


Figura 4 – Fereastră pentru crearea unui nou proiect în mediul de dezvoltare PyCharm

Când se creează un nou proiect, se creează implicit și un script „`main.py`” care este fișierul principal al proiectului. În mod implicit, acest fișier conține o funcție numită `print_hi` ce primește ca parametru o variabilă `name` și care, apelată cu parametrul „`PyCharm`”, va afișa în sub-fereastra „***Run***” mesajul „`Hi, PyCharm`”. Rularea proiectului se poate face accesând din meniu „***Run***” → „***Run 'main'***”. La

rularea proiectului, va apărea în partea de jos sub-fereastra „***Run***” în care se vor afișa rezultatele rulării (*Figura 6*).

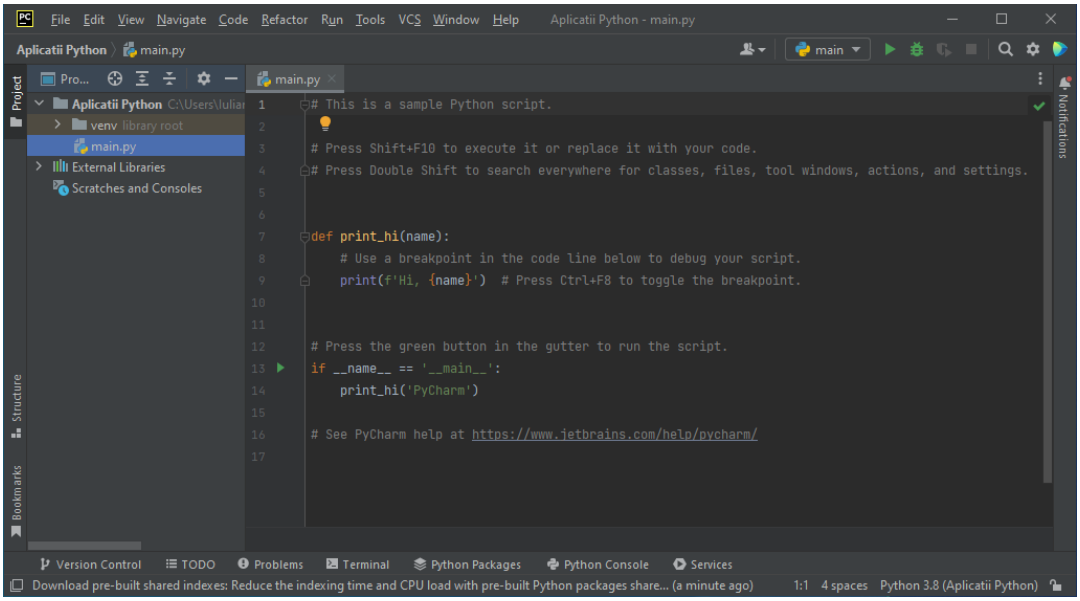


Figura 5 – Fereastra de bază a mediului de dezvoltare PyCharm

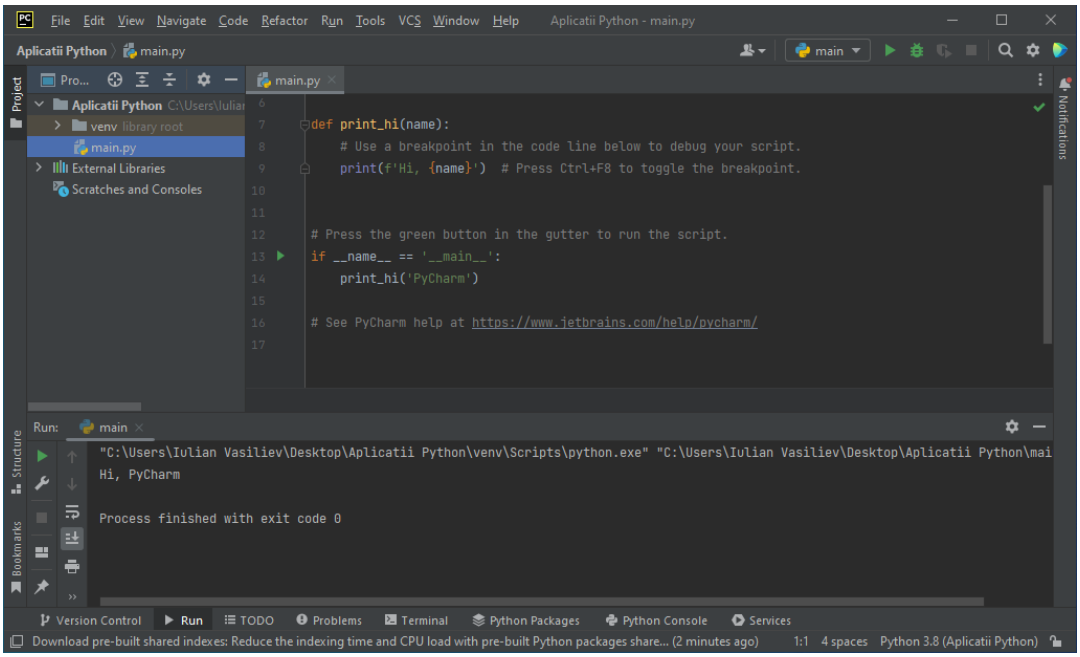


Figura 6 – Rezultatul rulării proiectului implicit în mediul de dezvoltare PyCharm

E. Instalarea librăriilor

Extinderea capacităților de bază ale limbajului ***Python*** se face cu ajutorul librăriilor. Librăriile sunt colecții de module specializate pe anumite funcționalități (ex: ***numpy*** – lucru cu matrice multi-dimensionale, ***Matplotlib*** – afișarea grafică a datelor etc.). Aceste librării sunt disponibile în mod gratuit și se instalează diferit în funcție de modul de rulare al aplicațiilor ***Python***.

Astfel, în cazul în care ***Python*** se utilizează în mod interactiv sau scripturile sunt rulate din linie comandă, instalarea unei librării se face din „***Command Prompt***” sau „***PowerShell***” folosind comanda `pip3 install` urmată de numele librăriei ce se dorește instalată. Un exemplu se poate observa în Figura 7.

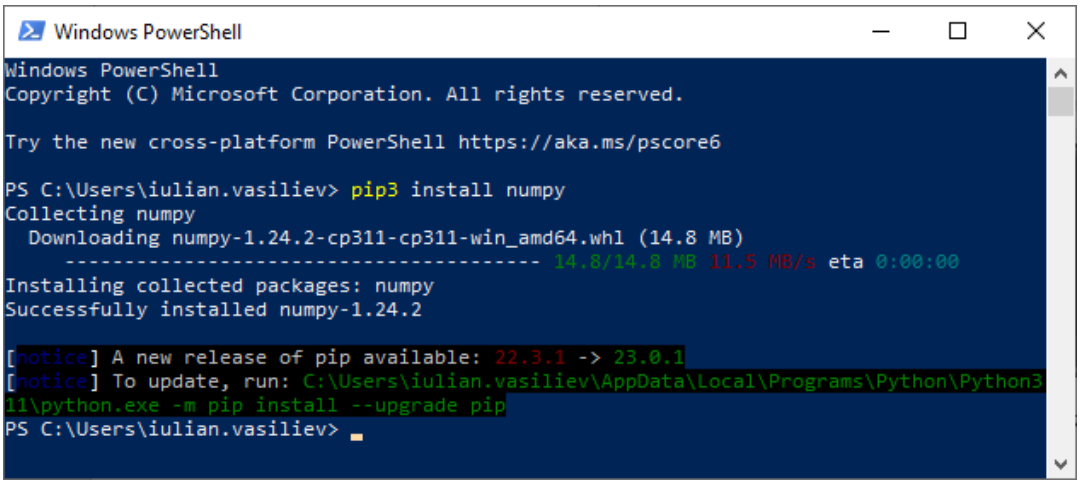


Figura 7 – Exemplu de instalare a unei librării Python folosind utilitarul pip3

În cazul în care se utilizează un mediu de dezvoltare pentru crearea aplicațiilor ***Python***, instalarea unei librării poate diferi de la IDE la IDE. În cazul PyCharm, instalarea unei librării implică o serie de pași (notați cu roșu în *Figura 8*):

1. Se accesează sub-fereastra „***Python Packages***” din partea de jos a ferestrei principale.
2. Vom scrie numele librăriei în câmpul de căutare al sub-ferestrei.
3. Se selectează librăria ce se dorește instalată.
4. Se apasă butonul „***Install package***” din partea dreaptă a sub-ferestrei.

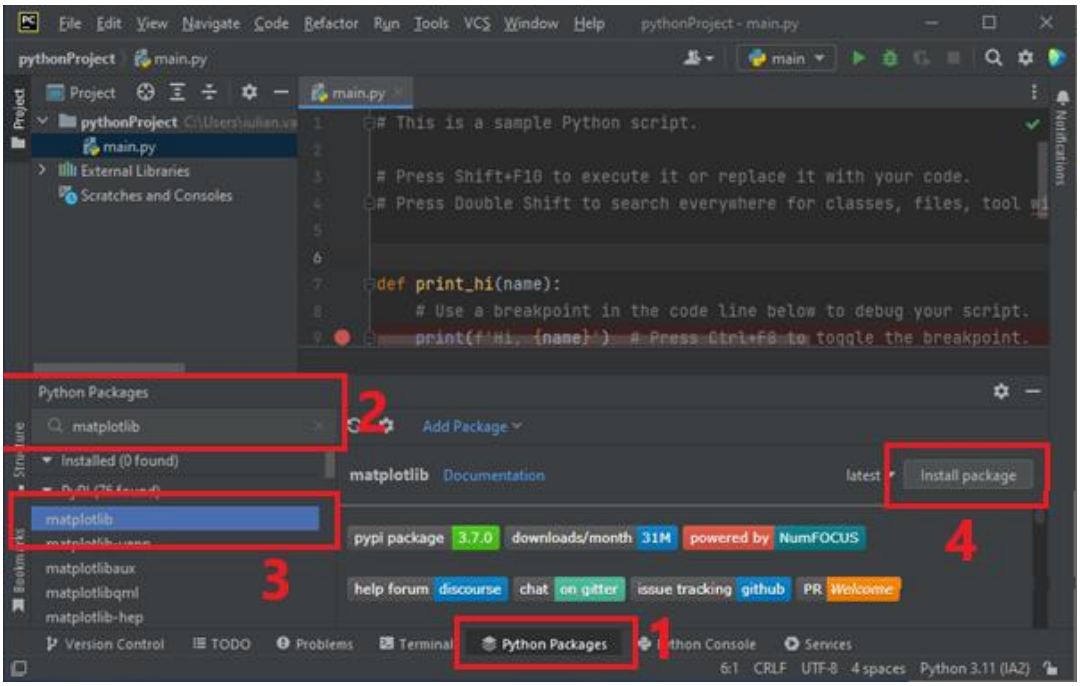


Figura 8 – Procedura de instalare a unei librării Python în IDE-ul PyCharm

F. Bazele programării în limbajul Python

Una dintre cele mai simple instrucțiuni de bază ale limbajului ***Python*** este funcția `print` ce poate fi folosită la afișarea mesajelor în consolă. De asemenea, în limbajul ***Python*** există două tipuri de comentarii în cod: comentarii pe o singură linie, acestea fiind precedate de simbolul `#` și comentarii pe mai multe linii, acestea fiind cuprinse între seturi de ghilimelele `"""` [1]. Exemplu:

main.py

```
print("Introducere Python")
# Exemplu de comentariu pe o singură linie
"""
Exemplu de comentariu
pe mai multe linii
"""
```

rezultat

```
Introducere Python
```

Așa cum se observă în exemplul anterior, spre deosebire de alte limbaje, în acesta nu este necesar să punem simbolul `;` (punct și virgulă) la sfârșitul liniilor de cod. Un lucru important de menționat este că în limbajul ***Python***, indentarea este foarte importantă. Astfel, o linie de cod este indentată doar dacă ea aparține interiorului unei structuri de control (condițională, buclă, funcție etc.). ***Python*** nu utilizează acolade pentru a defini blocurile interioare structurilor de control, ci indentarea [1]. Aceasta se poate face atât cu spații, cât și folosind caracterul Tab. Implicit, ***PyCharm*** se va ocupa singur de indentare, folosind 4 spații. O indentare greșită va duce la returnarea unei erori la rularea aplicației implementate. Exemplu de utilizare a indentării pentru definirea codului unei structuri condiționale:

main.py

```
a = 1
if a == 2:
    # cod interior structurii if
    print("Sunt in structura if")
    # alte instructiuni in interiorul structurii if
# cod in afara structurii if
print("OK")
```

rezultat

```
OK
```

Exemplu de utilizare a unei indentări greșite:

main.py

```
def test():
    x = 7
     y = 2  # Aceasta este o indentare greșită

    if x < y:
        print("x este mai mic decât y")
    else:
print("x nu este mai mic decât y") # Aceasta este o indentare
 greșită

test()
```

rezultat

```
  File "C:\User\Desktop\test.py", line 3
    y = 10  # Aceasta este o indentare greșită
IndentationError: unexpected indent
```

O indentare corectă ar arata altfel:

main.py

```
def test():
    x = 7
    y = 2

    if x < y:
        print("x este mai mic decât y")
    else:
        print("x nu este mai mic decât y")
test()
```

rezultat

```
x nu este mai mic decât y
```

a) Tipuri de date și definirea variabilelor în Python

Python oferă mai multe tipuri de date, dintre care cele mai simple sunt cele numerice și șirurile de caractere. Datorită orientării spre obiecte a acestui limbaj, orice variabilă ***Python*** poate fi văzută ca un obiect. De asemenea, este important de menționat că în ***Python*** nu trebuie declarat tipul de variabilă înainte. Astfel, pentru a defini și atribui o valoare unei variabile se va scrie numele variabilei, urmat de simbolul = și de valoarea pe care o atribuim.

Important [1]:

- Fiecare variabilă trebuie să aibă nume unic;
- Nu sunt permise spații în numele variabilelor;
- Numele variabilelor ar trebui să înceapă cu literă sau cu „_”;
- Cifrele sunt permise în numele variabilelor (dar nu ca prim caracter);
- Variabilele sunt CASE-SENSITIVE;
- Pentru a respecta convențiile de denumire în Python, variabilele ar trebui să fie denumite folosind formatele: Snake Case, Camel Case;
- Variabilele își pot schimba valorile și tipul de date în timpul execuției programului (suprascriere).

Exemple:

main.py

```
# Variabilă care începe cu literă sau cu "_"
varsta_tatalui = 45
_varsta_tatalui = 54


# Numere în numele variabilelor
camera_10 = "Recepție"


# Snake Case
prima_variabla = 10
nr_total = 100
nume_utilizator = "PopescuIon"

# Camel Case
primaVariabla = 20
nrTotal = 200
numeUtilizator = "PopescuIon"
```

Limbajul ***Python*** oferă suport pentru trei tipuri de date numerice: numere întregi, numere în virgulă mobilă și numere complexe [1]. Exemplu:

main.py

```
# numere intregi
nr_intreg = 42
print(nr_intreg)

# numere in virgula mobila
nr_real1 = 7.0
print(nr_real1)
nr_real2 = float(7)
print(nr_real2)

# numere complexe
nr_complex = 7 + 0j
print(nr_complex)
```

rezultat

```
42
7.0
7.0
(7+0j)
```

Un alt tip de date de bază este șirul de caractere, numit și string. Acesta se poate defini folosind atât ghilimele simple, cât și ghilimele duble. Exemplu:

main.py

```
sir_de_caractere = "Introducere Python"
print(sir_de_caractere)
```

rezultat

```
Introducere Python
```

Un alt tip de date disponibil în ***Python*** este lista, reprezentând un vector unidimensional ce poate conține orice număr de elemente de orice tip. Valoarea acesteia se poate inițializa enumerând valorile elementelor acesteia, separate de virgulă între un set de paranteze drepte. Ulterior, un element al listei se poate accesa scriind numele variabilei urmat de indexul elementului între paranteze pătrate. În ***Python***, numărarea index-urilor începe de la 0. Exemplu:

main.py

```
lista = [3, 2, 1]
print(lista)
print(lista[1])
```

rezultat

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

b) Operatori matematici de bază în Python

Toți operatorii matematici de bază (adunare, scădere, înmulțire, împărțire, dar și ridicare la putere) pot fi folosiți în ***Python*** între oricare tipuri de date numerice, fiind incluși în limbajul de bază. Pentru adunare, scădere, înmulțire și împărțire operatorii sunt aceiași ca în celelalte limbaje de programare (`+`, `-`, `*` și `/`). Pentru ridicare la putere, în ***Python*** operatorul este „`**`”. Un alt operator matematic ce poate fi folosit în ***Python*** este operatorul modulo (%) ce va calcula restul unei împărțiri. Ordinea operațiilor este cea matematică, având, de asemenea, posibilitatea grupării cu ajutorul parantezelor rotunde. Exemplu:

main.py

```
rezultat = 5 + 2 * 4 ** (2 - 1) - (3 + 2j) / 1
print(rezultat)
```

rezultat

```
(10-2j)
```

Dintre operatorii aritmetici de bază, doi pot fi folosiți și cu șiruri de caractere. Adunarea se poate face între două sau mai multe șiruri de caractere, rezultatul fiind concatenarea acestor șiruri, iar înmulțirea se poate face între un șir de caractere și un număr întreg, rezultatul fiind multiplicarea șirului de caractere de un număr de ori egal cu operatorul de tip număr întreg. Exemplu:

main.py

```
print("Introducere" + " " + " Python")
print("Laborator" * 3)
```

rezultat

```
Introducere Python
LaboratorLaboratorLaborator
```

Operatorul de adunare nu poate fi pus între un șir de caractere și un număr. Dacă dorim să inserăm valoarea unei variabile în interiorul unui șir de caractere se va folosi una dintre funcționalitățile de bază ale limbajului, și anume „***Literal String Interpolation***” [1]. Pentru aceasta, se va pune litera **`f`** înainte de ghilimelele ce definesc șirul de caractere. Astfel, atunci când dorim să inserăm valoarea unei variabile în acel șir de caractere se va scrie numele variabilei între acolade. Exemplu de utilizarea „***Literal String Interpolation***”:

main.py

```
nume = "Introducere Python"
nr_laborator = 1

print(f"Acesta este laboratorul numarul {nr_laborator} de
{nume}")
```

rezultat

```
Acesta este laboratorul numarul 1 de Introducere Python
```

c) Funcții matematice de bază

Marea majoritate a funcțiilor matematice de bază nu este disponibilă direct în limbajul de bază, fiind necesară utilizarea librăriei ***math*** (librăria este pre-instalată). Pentru a importa o librărie în codul aplicației se va folosi instrucțiunea `import` urmată de numele librăriei. Astfel, funcțiile și constantele definite de acea librărie vor putea fi accesate scriind numele librăriei, urmat de simbolul punct și de numele constantei / funcției dorite.

Librăria ***math*** pune la dispoziție mai multe constante matematice și funcții matematice. Cele mai uzuale se regăsesc în tabelul următor [2]:

Tabelul 1 - Constante și funcții matematice de bază din librăria `math`

Constante Matematice	
`math.pi`	Constanta matematică π
`math.e`	Constanta matematică *e* (numărul lui Euler)
Funcții pentru reprezentarea numerelor	
`math.ceil(x)`	Cel mai mic număr întreg mai mare sau egal cu **x**
`math.floor(x)`	Cel mai mare număr întreg mai mic sau egal cu **x**
`round(x)`	Cel mai apropiat întreg de numărul **x** – funcția este în limbajul de bază (nu trebuie importată librăria ***math***)
`abs(x)`	Modulul numărului **x** – funcția este în limbajul de bază
Funcții exponențiale și logaritmice	
`math.exp(x)`	**e** la puterea **x**
`math.log(x, b)`	Logaritm în bază **b** din **x** (dacă **b** lipsește, va calcula logaritm natural)
`math.log2(x)`	Logaritm în baza 2 din **x**
`math.log10(x)`	Logaritm în baza 10 din **x**
`math.sqrt(x)`	Rădăcina pătrată a lui **x**
Funcții trigonometrice	
`math.sin(x)`	Sinus de **x** (**x** în radiani)
`math.cos(x)`	Cosinus de **x** (**x** în radiani)
`math.tan(x)`	Tangent de **x** (**x** în radiani)
`math.asin(x)`	Arc sinus de **x** în radiani (rezultatul între $-\pi/2$ și $\pi/2$)
`math.acos(x)`	Arc cosinus de **x** în radiani (rezultatul între 0 și π)

`math.atan(x)`	Arc tangent de **x** în radiani (rezultatul între $-\pi/2$ și $\pi/2$)
`math.atan2(y, x)`	Arc tangent de **y** / **x** în radiani (rezultatul între $-\pi$ și π)
Funcții de conversie unghiulară	
`math.degrees(x)`	Convertește în grade unghiul **x** specificat în radiani
`math.radians(x)`	Convertește în radiani unghiul **x** specificat în grade

Spre exemplu, dacă se dorește calcularea lui $\sin(2 \cdot \pi/3)$ se va scrie:

main.py

```
import math
print(math.sin(2 * math.pi / 3))
```

rezultat

```
0.8660254037844387
```

d) Operatori logici

Python utilizează logica booleană pentru a evalua condiții. Astfel, acest limbaj folosește două valori booleene ce pot fi returnate la evaluarea unei expresii logice (condiții): `True` și `False`. Operatorii ce permit compararea a două valori sunt asemănători cu cei din alte limbaje. Considerând variabila `x` având valoarea 2, operatorii de comparație se pot observa în tabelul următor:

Tabelul 2 - Operatori logici în Python

Operator	Descriere	Exemplu	Rezultat
==	egal cu	`print(x == 2)`	`True`
!=	diferit de	`print(x != 2)`	`False`
<	mai mic ca	`print(x < 2)`	`False`
<=	mai mic sau egal cu	`print(x <= 2)`	`True`
>	mai mare ca	`print(x > 2)`	`False`
>=	mai mare sau egal cu	`print(x >= 2)`	`True`

Un alt operator logic definit de ***Python*** este operatorul `in` ce se poate utiliza pentru a verifica dacă o anumită valoare se regăsește într-un obiect iterabil. Exemplu:

main.py

```
lista_forme = ["triunghi", "patrat", "cerc"]
print("triunghi" in lista_forme)
print("hexagon" in lista_forme)
```

rezultat

```
True
False
```

Pentru a construi expresii booleene complexe, limbajul ***Python*** pune la dispoziție 3 operatori booleeni, conform tabelului următor:

Tabelul 3 - Operatori booleeni în Python

Operator	Descriere	Exemplu	Rezultat
`not`	negarea expresiei precedate de operator	`not True`	`False`
`or`	operația de „sau logic” între două expresii	`True or False`	`True`
`and`	operația de „și logic” între două expresii	`True and False`	`False`

e) Structura condițională

Așa cum a fost specificat mai devreme, în ***Python*** blocurile interioare structurilor de control nu sunt definite utilizând acolade ci indentare. Structura condițională permite executarea unor secvențe de cod doar dacă anumite condiții sunt îndeplinite. Această structură folosește 3 cuvinte cheie (`if`, `elif` și `else`) pentru definirea ramurilor acesteia. După definirea fiecărei ramuri, înainte de blocul de cod aferent acelei ramuri, este necesar să se pună simbolul „două puncte”.

Sintaxă generală a structurii condiționale:

main.py

```
if expr_logica1:
    # bloc de cod - ramura if
    ...
elif expr_logica2: # else if
    # bloc de cod - ramura elif
    ...
# alte ramuri elif
else:
    # bloc de cod - ramura else
    ...
```

Blocul de cod interior ramurii `if` se execută doar dacă expresia logică `expr_logica1` dată acestei ramuri are valoarea `True`. Blocul de cod interior ramurii `elif` se execută doar dacă expresia logică `expr_logica2` dată acestei ramuri are valoarea `True` și dacă expresiile logice date ramurilor precedente au avut valoarea `False`. Blocul de cod aferent ramurii `else` se execută dacă nici una dintre expresiile logice aferente ramurilor precedente nu a avut valoarea `True`. Este important de menționat faptul că ramurile `elif` și `else` pot lipsi din structurile condiționale și faptul că pot exista mai multe ramuri `elif`.

main.py

```
x = 33
if x % 2 == 0:
    print("x este divizibil cu 2!")
elif x % 5 == 0:
    print("x este divizibil cu 5!")
else:
    print("x nu este divizibil nici cu 2, nici cu 5!")
```

rezultat

```
x nu este divizibil nici cu 2, nici cu 5!
```

f) Structuri repetitive

Python oferă două tipuri de structuri repetitive: bucle `while` – se utilizează pentru a repeta un bloc de cod atât timp cât o condiție este adevărată, și bucle `for` – se utilizează pentru a repeta o secvență de cod pentru fiecare element al unui obiect iterabil.

Exemplu de utilizare a structurii repetitive `while`:

main.py

```
a = 0
while a < 4:
    print(f"Iteratia {a}")
    a = a + 1
```

rezultat

```
Iteratia 0
Iteratia 1
Iteratia 2
Iteratia 3
```

Exemplu de utilizare a structurii repetitive `for`:

main.py

```
lista_forme = ["triunghi", "patrat", "cerc"]
for forma in lista_forme:
    print(forma)
```

rezultat

```
triunghi
patrat
cerc
```

Funcția `range` poate fi folosită pentru a crea un obiect iterabil ce conține o secvență de numere. Această funcție poate primi 3 parametri:

Tabelul 4 - Parametrii funcției `range`

Parametru	Descriere	Observație
`start`	valoarea de start a secvenței	poate lipsi, caz în care se consideră valoarea implicită 0
`stop`	valoarea de stop a secvenței	nu poate lipsi; de asemenea, valoarea de stop a secvenței nu este inclusă în secvență
`step`	pasul de incrementare	poate lipsi, valoarea implicită 1

Exemplu:

main.py

```
for x in range(1, 11, 2):
    print(x)
```

rezultat

```
1
3
5
7
9
```

În buclele `while` și `for` se pot utiliza două comenzi pentru a opri execuția blocului de cod:

Tabelul 5 - Comenzi ce se pot utiliza în interiorul buclelor

Comanda	Descriere
`break`	oprește execuția blocului de cod și iese din buclă
`continue`	oprește execuția blocului de cod și continuă cu următoarea iterație a buclei

Exemple:

main.py

```
# Exemplul 1: bucla se va opri cand cnt ajunge la 4
cnt = 0
while True:
    print(cnt)
    cnt += 1
    if cnt >= 4:
        break
# Exemplul 2: Va afisa doar numerele pare
for cnt in range(10):
    if cnt % 2 != 0:
        continue
    print(cnt)
```

rezultat

```
1
3
5
7
9
```

G. Cerințe

1. Deschideți mediul de dezvoltare PyCharm și creați un proiect cu numele dumneavoastră pe Desktop. Familiarizați-vă cu mediul de dezvoltare. Realizați din PyCharm, în interiorul proiectului creat, un director numit „`Lab1`” în care constituiți 14 fișiere Python: `ex2.py`, `ex3.py`, `ex4.py`, `ex5.py`, `ex17.py`, `ex18.py`, `ex19.py`, `ex20.py`, `ex21.py`, `ex22.py`, `ex23.py`, `ex24.py`, `ex25.py`, `ex26.py`. Rezolvați următoarele exerciții câte unul în fiecare fișier creat. Observație: pentru a rula un anumit fișier în PyCharm, veți da click-dreapta pe el în sub-fereastra „***Project***” și apoi „***Run nume_fisier”***”.

2. Scrieți un program Python care să afișeze numele vostru.

3. Creați o variabilă de tip număr întreg, numită *`vârstă`* și afișați-o în consolă.

4. Creați o variabilă de tip șir de caractere, numită *`nume`* și afișați-o în consolă.
5. Creați o variabilă de tip număr în virgulă mobilă, numită *`reducere`* și afișați-o în consolă.

6. Cum introduceți comentarii în Python?
   a) `# Acesta este un comentariu`
   b) `// Acesta este un comentariu`
   c) `/* Acesta este un comentariu */`

7. Care este extensia corectă a unui fișier Python?
   a) *`.pt`*
   b) *`.pyth`*
   c) *`.pyt`*
   d) *`.py`*

8. În Python `'Salut'` este același lucru cu `"Salut"` ?
   a) Adevărat
   b) Fals

9. Ce operator este folosit pentru înmulțirea numerelor?
   a) `x`

b) `#`
c) `%`
d) `*`

10. Ce operator poate fi folosit pentru a compara două valori?
    a) `=`
    b) `><`
    c) `==`
    d) `<>`

11. Cum creezi o variabilă numită „fruct” și îi atribui valoarea „măr”?
    a) `var fruct = "măr"`
    b) `fruct = 'măr'`
    c) `variabile fruct 'măr'`
    d) `fruct_string = 'măr'`

12. Care dintre următoarele: atribuie valoarea numerică 5 variabilei „a”?
    a) `a = '5'`
    b) `a = "5"`
    c) `a = 5`

13. Cum începi să scrii structura „`if`” în Python?
    a) `if a > b:`
    b) `if (a > b)`
    c) `if a>y then`

14. Cum începi să scrii structura „`while`” în Python?
    a) `while a > b:`
    b) `while a > b{`
    c) `while (a > b)`
    d) `a > b while {`

15. Cum începi să scrii structura „`for`” în Python?
    a) `for a in b:`
    b) `for each a in b:`
    c) `for a > b:`

16. Care declarație este folosită pentru a opri o buclă?
    a) `exit`

b) `stop`
c) `return`
d) `break`

17. Stocați valoarea numerică 5 într-o variabilă numită „`x`” și valoarea numerică 10 într-o variabilă numită „`y`”. Salvați rezultatul lui `x*y` într-o variabilă separată numită „`rezultat`”. În cele din urmă afișați rezultatul astfel încât să arate astfel:>>> 5*10=50

18. Stocați valoarea 12 într-o variabilă numită „`nume`”.
    a) Verificați dacă valoarea este pozitivă.
    b) Verificați dacă valoarea este mai mare de 5.
    c) Verificați dacă valoarea este mai mică de 25.
    d) Verificați dacă valoarea se află între 0 și 20.

19. Având în vedere exemplele de mai jos, scrie într-un comentariu rezultatul așteptat al fiecărei declarații de print comentate înainte de a rula codul. De-comentați declarațiile `print` și verificați rezultatele.

    a) Exemplul 1:
    ```
    a = True
    b = False
    # print(not(a))
    # print(not(b))
    ```

    b) Exemplu 2:
    ```
    a = True
    b = False
    # x = not(a)
    # y = not(b)
    # print(a or b)
    # print(x or y)
    # print(a or x)
    # print(x or b)
    ```

    c) Exemplu 3:
    ```
    a = False
    b = False
    # x = not(a)
    # y = not(b)
    ```

```
# print(a and b)
# print(a and x)
# print(y and b)
# print(x and y)
```

20. Având în vedere codul de mai jos, scrieți într-un comentariu rezultatul așteptat a fiecărei instrucțiuni de print comentate înainte de a rula codul. De-comentați declarațiile print și verificați rezultatele.

```
x = 10
y = x
z = 10

# print(x is y)
# print(x is not y)

# print(x is z)
# print(x is not z)
# print(z is x)
# print(z is not x)

# print(y is z)
# print(y is not z)
```

21. Să se calculeze și afișeze rezultatele:

a) $5 + 4 \cdot 2$

b) $\frac{2}{15} + 3^2 \cdot 2$

c) $8 - 9 \cdot (1 - 3)$

d) $1 - \frac{3}{7} + 2$

e) $\frac{2+5^3}{3-\frac{1}{9}}$

f) $\left(\frac{\frac{2+3}{3}+1}{3^2+2^3}\right)^{\frac{1}{2}}$

g) $15.601 - 12.8$

h) $3.256^{1.34}$

22. Stocați valoarea 12 într-o variabilă numită „`a`” și valoarea 3 într-o variabilă numită „`b`”. Calculați suma, diferența, înmulțirea, împărțirea și ridicarea la putere a variabilelor stocate și salvați rezultatul pentru fiecare situație într-o variabilă separată numită „`rezultat`”. Afișați rezultatele folosind „***Literal String Interpolation***”. Exemplu: în cazul sumei se va afișa „`12 + 3 = 15`”.

23. Pentru $a = 22$, $b = a - 8.5$ și $c = 3 \cdot b + 1$, calculați:

a) $c \cdot \frac{a}{b} + \frac{c^2}{2+2 \cdot c} - (b^2 - c) \cdot (c + a^{\frac{1}{2}})$

b) $\frac{c^2+a^2}{b-c} - (a - c + b)^{\frac{1}{3}}$

24. Să se calculeze și să se afișeze pe ecran rezultatele:

a) $\frac{1+\sqrt{2}}{4}$

b) $\left|e^{-1} - 2 \cdot \sin\frac{\pi}{6}\right|$

c) $\pi^{0.5} + (1 + \tan 60°)^3$

d) $2 \cdot \cot^2\frac{\pi}{6} - \cos\frac{\pi}{2}$

e) $\frac{\lg(86)+\sin(\cos(\sqrt{22.55}))}{\sqrt[3]{\ln(100)-\log_4(\sin\frac{7 \cdot \pi}{3})}}$

25. Să se calculeze și să se afișeze rezultatul sumelor:

a) $\sum_{x=1}^{100} \frac{1}{x^2}$

b) $\sum_{x=1}^{5} x^2 - x$

c) $\sum_{x=4}^{15} \frac{2^{x+1}}{x-1}$

26. Implementați instrucțiunile pentru a afișa toate numerele în intervalul [0 132] care sunt atât pare cât și divizibile cu 11.

Lucrarea 2. Funcții. Structuri de date

A. Obiectivele lucrării

1. Familiarizarea cu sintaxa și modul de declarare a funcțiilor în ***Python***
2. Prezentarea generală a structurilor de date disponibile în limbajul de bază
3. Familiarizarea cu instrucțiunile necesare pentru a lucra cu fișiere
4. Prezentarea formatelor disponibile pentru salvarea datelor

B. Funcții

Funcțiile sunt o metodă de împărțire a codului în secvențe ce au anumite funcționalități și care pot fi apoi refolosite, salvând astfel timp la implementarea aplicațiilor. În ***Python***, definirea unei funcții se face folosind cuvântul cheie `def`, urmat de numele funcției, parametrii de intrare ai acesteia, separați de virgulă și plasați între paranteze rotunde și simbolul „:” (două puncte). După aceasta urmează blocul de cod aferent funcției (ca și la structurile de control, acesta trebuie indentat pentru a sugera interpretorului faptul că este cod interior funcției).

Exemplu de definire și apelare a unei funcții care să afișeze pe ecran mesajul „Introducere Python”:

main.py

```
def afiseaza_titlu():
    # continutul functiei
    print("Introducere Python")

# apelarea functiei
afiseaza_titlu()
```

rezultat

```
Introducere Python
```

Funcțiile pot avea și parametri de intrare (unul sau mai mulți) – variabile ce pot fi transmise funcției în momentul apelării acesteia. Exemplu:

main.py

```
def numara(x, y):
    for i in range(x, y + 1):
        print(i)

# apelarea functiei
numara(2, 7)
```

rezultat

```
2
3
4
5
6
7
```

În plus, funcțiile pot, folosind cuvântul cheie `return`, să furnizeze o valoare la sfârșitul execuției acestora. Exemplu:

main.py

```
def farenheit_to_celsius(degF):
    degC = (degF - 32) * 5 / 9
    return degC

print(farenheit_to_celsius(50))

val_degC = farenheit_to_celsius(66.3)
print(val_degC)
```

rezultat

```
10.0
19.055555555555557
```

C. Structuri de date

Python oferă în limbajul de bază o serie de structuri de date: liste, tupluri, seturi și dicționare [1].

a) Liste

Lista este un vector unidimensional ce poate conține orice număr de elemente de orice tip. Valoarea acesteia se poate inițializa enumerând valorile elementelor acesteia, separate de virgulă între un set de paranteze drepte. Ulterior, un element al listei se poate accesa scriind numele variabilei urmat de indexul elementului între paranteze pătrate. In ***Python*** numărarea index-urilor începe de la 0, dar se pot folosi și indecși negativi, caz în care elementele vor fi numărate de la sfârșitul listei spre început (e.g. indexul -1 reprezintă ultimul element al listei, indexul -2 reprezintă penultimul element etc.) [1].

Exemplu:

main.py

```
lista = ["triunghi", "patrat", "cerc"]

print(lista[1]) # elementul cu indexul 1 ("patrat")
print(lista[2]) # elementul cu indexul 2 ("cerc")
print(lista[-1]) # ultimul element ("cerc")
lista[0] = "hexagon" # modificare primul element
print(lista)

print(lista[0:2]) # elementele cu indecșii 0 și 1
```

rezultat

```
patrat
cerc
cerc
['hexagon', 'patrat', 'cerc']
['hexagon', 'patrat']
```

Conținutul unei liste poate fi alterat folosind o serie de metode ceea ce înseamnă că listele sunt MUTABILE. Modul în care se poate utiliza o metodă de alterare: se scrie numele listei, urmat de simbolul punct și de numele metodei (`lista.append`, unde `lista` reprezintă numele listei). Cele mai uzuale metode pentru liste se regăsesc în tabelul următor [1]:

Tabelul 1 - Metode pentru alterarea listelor

Metoda	Descriere
`append(x)`	adaugă valoarea `x` la sfârșitul listei
`insert(i, x)`	inserează un element cu valoarea `x` la poziția `i`
`extend(l)`	adaugă elementele din lista `l` la sfârșitul listei
`pop(i)`	șterge elementul de pe poziția `i` și returnează valoarea acestuia
`clear()`	șterge toate elementele listei
`sort()`	sortează lista

Exemple de utilizare a unor metode de alterare a conținutului unei liste:

main.py

```
lista = [1, 2, 3]
print(lista)
# [1, 2, 3]

lista.append(4)
print(lista)
# [1, 2, 3, 4]

lista.insert(1, 5)
print(lista)
# [1, 5, 2, 3, 4]

lista.pop(2)
print(lista)
# [1, 5, 3, 4]

lista.clear()
print(lista)
# []
```

Pentru a determina numărul de elemente dintr-o listă, se va folosi funcția `len` dându-i ca parametru lista. Exemplu:

main.py

```
lista = [7, 4, 5, 12, -2.2]
print(len(lista))
```

rezultat

```
5
```

b) Tupluri

Un tuplu este un vector unidimensional ce poate conține orice număr de elemente de orice tip. Valoarea acestuia se poate inițializa enumerând valorile elementelor acesteia, separate de virgulă între un set de paranteze rotunde. Ulterior, un element al listei se poate accesa scriind numele variabilei urmat de indexul elementului între paranteze pătrate. Indexarea elementelor se poate face ca la liste, utilizând inclusiv indecși negativi [1].

Diferența dintre un tuplu și o listă este că, un tuplu, odată definit, nu mai poate suferi modificări ale elementelor, lucru care înseamnă că tuplurile sunt IMMUTABILE. Conversia unui tuplu într-o listă se poate face utilizând funcția `list` căreia îi dăm ca parametru tuplul.

Comparativ cu listele, tuplurile se definesc utilizând paranteze rotunde.

Exemplu de definire a unui tuplu, accesarea elementelor acestuia și convertirea în listă:

main.py

```
tuplu = (1, 2, 3, 4, 5)

print(tuplu)
print(tuplu[1])
print(tuplu[-2])

lista = list(tuplu)
print(lista)
```

rezultat

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

c) Seturi

Un set este un vector unidimensional ce poate conține orice număr de elemente de orice tip. Valoarea acestuia se poate inițializa enumerând valorile elementelor acestuia, separate de virgulă între un set de acolade. Diferența dintre un set și o listă

este că elementele setului nu au o anumită ordine, deci nu pot fi indexate. De asemenea, deoarece elementele nu pot fi indexate, nu se pot modifica valorile elementelor deja existente. În plus, un set nu poate avea două elemente cu aceeași valoare [1].

Exemplu de definire a unui set:

main.py

```
my_set = {7, 4, 6, 2, 3, 4}
print(my_set)
```

rezultat

```
{2, 3, 4, 6, 7}
```

Se pot adăuga elemente noi sau se pot șterge elemente dintr-un set, folosind metodele din tabelul următor:

Tabelul 2 - Metode pentru alterarea seturilor

Metoda	Descriere
`add(x)`	adaugă elementul `x` în set
`update(y)`	adaugă elementele din iterabilul `y` (lista/tuplu/set) în set
`remove(x)`	șterge elementul `x` din set

Exemplu:

main.py

```
my_set = {7, 4, 6, 2, 3, 7}

print(my_set)

my_set.add(-1)
print(my_set)

my_set.remove(7)
print(my_set)
```

rezultat

```
{2, 3, 4, 6, 7}
{2, 3, 4, 6, 7, -1}
{2, 3, 4, 6, -1}
```

Seturile pot fi văzute ca mulțimi matematice, de aceea ***Python*** implementează o serie de operații cu mulțimi sub forma unor metode pentru seturi [1]:

Tabelul 3 - Metode pentru operații cu mulțimi (seturi)

Metoda	**Descriere**
`x.intersection(y)`	returnează un set ce conține elementele comune seturilor `x` și `y`
`x.union(y)`	returnează un set ce conține toate elementele din seturile `x` și `y`
`x.difference(y)`	returnează un set ce conține toate elementele din setul `x` care nu sunt și în setul `y`
`x. issuperset(y)`	returnează `True` doar dacă toate elementele din setul `y` se află și în setul `x`
`x. issubset(y)`	returnează `True` doar dacă toate elementele din setul `x` se află și în setul `y`

Exemplu:

main.py

```
my_set1 = {1, 2, 3, 4}
my_set2 = {3, 4, 5, 6}

print(my_set1.intersection(my_set2))
print(my_set1.union(my_set2))
print(my_set1.difference(my_set2))
```

rezultat

```
{3, 4}
{1, 2, 3, 4, 5, 6}
{1, 2}
```

d) Dicționare

Dicționarele sunt utilizate la stocarea perechilor de date de tipul cheie – valoare. Cheile trebuie să fie unice și de obicei de tip șir de caractere (deși se acceptă și tipuri numerice), în timp ce valorile pot fi de orice tip, inclusiv liste, obiecte sau chiar alte dicționare. Dicționarele se definesc între acolade, perechile de tip cheie – valoare fiind separate de virgulă, în timp ce între cheie și valoare se pune simbolul „:” (două

puncte). Accesarea elementelor unui dicționar se face scriind numele dicționarului și, între paranteze pătrate, cheia căreia dorim să îi accesăm valoarea. Dacă încercăm să modificăm valoarea unei chei care nu există în dicționar, atunci se va adăuga o nouă pereche cheie – valoare în dicționar. Ștergerea unei perechi din dicționar se face folosind instrucțiunea `del` așa cum se observă în exemplul următor [1]:

main.py

```
nota = {"Colocviu": 3, "Prezenta": 1, "Examen": 6}
print(nota)
print(nota["Examen"]) # accesam un element

nota["Tema"] = 1 # adaugam element nou
print(nota)

del nota["Prezenta"] # stergem element
print(nota)
```

rezultat

```
{'Colocviu': 3, 'Prezenta': 1, 'Examen': 6}
6
{'Colocviu': 3, 'Prezenta': 1, 'Examen': 6, 'Tema': 1}
{'Colocviu': 3, 'Examen': 6, 'Tema': 1}
```

D. List Comprehension

„List comprehension” este o modalitate de definire a unei noi liste pe baza elementelor unei alte liste, oferind o sintaxă mai scurtă decât dacă s-ar defini în mod clasic. Definirea unei noi liste folosind „list comprehension” se face scriind între paranteze pătrate expresia de generare a elementelor noii liste urmată de iterarea listei deja existente, utilizând *for*, și, opțional, condiționarea elementelor, utilizând *if* [1].

Spre exemplu, se dă o listă `lista1` conținând valorile $[-5, -2, 0, 1, 12, -1, 5, 7]$. Se dorește să se definească o nouă listă care să conțină rădăcina pătrată a elementelor mai mari sau egale cu 0 din `lista1`.

În mod clasic, problema s-ar rezolva astfel:

main.py

```
lista1 = [-5, -2, 0, 1, 12, -1, 5, 7]
lista2 = []

for elem in lista1:
    if elem >= 0:
        lista2.append(elem ** (1/2))

print(lista2)
```

rezultat

```
[0.0,      1.0,      3.4641016151377544,      2.23606797749979,
2.6457513110645907]
```

Utilizând „list comprehension”, codul s-ar reduce la:

main.py

```
lista1 = [-5, -2, 0, 1, 12, -1, 5, 7]
lista2 = [elem ** (1/2) for elem in lista1 if elem >= 0]

print(lista2)
```

rezultat

```
[0.0,      1.0,      3.4641016151377544,      2.23606797749979,
2.6457513110645907]
```

E. Salvarea datelor

a) Lucrul cu fișiere

Pentru a lucra cu fișiere în ***Python*** (citire / salvare date) există mai multe metode. Una dintre ele este utilizând structura `with` [1]. Exemplu:

main.py

```
with open("exemplu.txt", "r") as my_file:
    # scriem / citim fisierul in interiorul structurii with
    ...
```

Această structură va crea o variabilă `my_file` ce va face referire la fișierul deschis cu funcția `open`. Această funcție primește doi parametri: calea spre fișier și modul de deschidere a fișierului, conform tabelului de mai jos [1]:

Tabelul 4 - Moduri de deschidere a fișierelor în Python

Mod	Descriere
`r`	deschide un fișier doar pentru citire; dacă fișierul nu există va rezulta o eroare
`w`	deschide un fișier pentru scriere; dacă fișierul nu există, îl va crea. Dacă fișierul există, conținutul acestuia va fi suprascris
`a`	deschide un fișier pentru scriere; dacă fișierul nu există, îl va crea. Dacă fișierul există, scrierea se va face după conținutul deja existent
`x`	deschide un fișier pentru scriere; dacă fișierul există deja, va rezulta o eroare

Astfel, pentru citire și scriere, se vor folosi o serie de metode disponibile pentru variabila `my_file`:

Tabelul 5 - Metode pentru citirea și scrierea fișierelor

Mod	Metoda	Descriere
Citire	`my_file.read()`	va returna tot conținutul fișierului
	`my_file.read(x)`	va returna primele `x` caractere din fișier
	`my_file.readline()`	va returna o linie din fișier; dacă se va mai apela din nou această metodă, va rezulta următoarea linie, și așa mai departe
Scriere	`my_file.write(text)`	va scrie valoarea șirului de caractere `text` în fișier

Exemplu:

main.py

```
with open("test.txt", "w") as my_file:
    my_file.write("Prima linie din fisier\n")
    my_file.write("A doua linie din fisier\n")
    my_file.write("Ultima linie din fisier")

with open("test.txt", "r") as my_file:
    print(my_file.readline())
    print(my_file.readline())
    print(my_file.readline())
```

rezultat

```
Prima linie din fisier
A doua linie din fisier
Ultima linie din fisier
```

b) Fișiere CSV

Formatul CSV (Comma Separated Values – Valori Separate de Virgulă) este unul dintre cele mai cunoscute formate pentru importarea și exportarea datelor sub formă de tabel. Fiecare rând al fișierului CSV reprezintă un rând al tabelului, în timp ce valorile fiecărui rând sunt separate utilizând un separator (de obicei, virgula). ***Python*** oferă suport pentru fișierele CSV în librăria `csv` (inclusă în limbajul de bază ***Python*** – nu este necesară instalarea). Această librărie definește două tipuri de obiecte [1]:

- `reader` – utilizat pentru a citi datele dintr-un fișier CSV. Este iterabil, deci practic fișierul se poate citi linie cu linie prin iterarea acestui obiect.
- `writer` – utilizat pentru a scrie date într-un fișier CSV. Pune la dispoziție metoda `writerow` ce primește ca parametru un obiect iterabil și îl adaugă ca linie în fișierul CSV.

În exemplul următor, se poate vedea modul de definire al obiectelor `reader` și `writer`, precum și modul de scriere / citire a datelor.

main.py

```
import csv

x = [1, 7, 4, 5, 2, 3, 2, 2]
y = [round(elem ** (1/2)) for elem in x]

with open("test.csv", "w") as my_file:
    writer_object = csv.writer(my_file, delimiter=',')

    writer_object.writerow(x)
    writer_object.writerow(y)

with open("test.csv", "r") as my_file:
    reader_object = csv.reader(my_file, delimiter=',')

    for row in reader_object:
        print(row)
```

rezultat

```
['1', '7', '4', '5', '2', '3', '2', '2']
['1', '3', '2', '2', '1', '2', '1', '1']
```

c) Fișiere JSON

Formatul JSON (JavaScript Object Notation) este un format foarte popular de salvare a datelor, asemănător formatului în care poate fi definit un dicționar în ***Python***. Astfel, utilizând acest format se poate salva orice tip de dată din ***Python***. ***Python*** oferă suport pentru fișierele JSON în librăria `json` (inclusă în limbajul de bază ***Python*** – nu este necesară instalarea). Această librărie definește mai multe funcții, printre care `dump` (utilizată la scrierea unui fișier JSON) și `load` (utilizată la citirea unui fișier JSON) [1].

Exemplu de utilizare:

main.py

```
import json

studenti = [
    {
        "Nume": "Ion",
        "Prezente": 7,
        "Nota": 7.5
    }, {
        "Nume": "Vasile",
        "Prezente": 0,
        "Nota": 2.5
    }, {
        "Nume": "Gheorghe",
        "Prezente": 14,
        "Nota": 9.5
    }
]

with open("test.json", "w") as my_file:
    json.dump(studenti, my_file)

with open("test.json", "r") as my_file:
    bd_studenti = json.load(my_file)
    print(bd_studenti)
    print(bd_studenti[2]["Nume"])
```

rezultat

```
[{'Nume': 'Ion', 'Prezente': 7, 'Nota': 7.5}, {'Nume': 'Vasile', 'Prezente': 0, 'Nota': 2.5}, {'Nume': 'Gheorghe', 'Prezente': 14, 'Nota': 9.5}]

Gheorghe
```

F. Cerințe

1. Fie lista de șiruri de caractere: „8T”, „9IN”, „JT”, „QT”, „AR”. Definiți lista în ***Python*** și afișați-o pe ecran. Eliminați elementele „9IN” și „AR”, apoi inserați elementele „9T” și „10T” după elementul „8T”, folosind metodele specifice listelor.

2. Se dă următoarea listă: `lista = [10, 6, 3, 4, 1, 2, 5, 7, 9]`
   a) Afișați lista inițială.
   b) Verificați tipul de date.
   c) Adăugați un element la sfârșitul listei.
   d) Inserați elementul „8” la poziția 7.
   e) Determinați numărul de elemente din listă.
   f) Sortați lista.
   g) Extindeți lista cu o altă listă.
   h) Eliminați primul element din listă.
   i) Eliminați ultimul element din listă.
   j) Afișați lista finală.

3. Se dă următorul tuplu: `tuplu = (1, "xyz", 3.14, False, [4,6])`
   a) Afișați tuplul inițial.
   b) Afișați lungimea tuplului.
   c) Creați un nou tuplu care să conțină doar primele trei elemente.
   d) Testați modificarea valorii unui element (exemplu: `tuplu[0] = 10`).
   e) Convertiți tuplul într-o listă.

f) Afișați tuplul final.

4. Se dă următorul set: `set = {2, 4, 6, 8, 10}`
   a) Afișați setul inițial.
   b) Adăugați numărul 9 setului.
   c) Adăugați numărul 2 setului. Ce observați?
   d) Ștergeți un element din set folosind metoda `remove()`.
   e) Ștergeți un element din set folosind metoda pop().
   f) Actualizați setul cu `elemente_adiționale = {11, 12 ,13}`.
   g) Afișați setul final.

5. Definiți un dicționar, numit `student`, cu următoarele chei: nume, prenume, vârstă, an de facultate, universitate și alegeți valori pentru fiecare cheie.
   a) Afișați dicționarul inițial.
   b) Afișați lungimea dicționarului.
   c) Afișați numele de familie al studentului.
   d) Adăugați un element nou în dicționar, reprezentând țara în care studiază studentul.
   e) Actualizați anul de facultate.
   f) Ștergeți vârsta.
   g) Afișați dicționarul final.

6. Se dă următoarea listă: `lista_initiala = [ 27, 88, 46, 52 , 3, 1, 17, 15, 4, 2, 1, 0]`
   a) Afișați lista inițială.
   b) Utilizați list comprehension pentru a crea o listă cu elementele impare.
   c) Utilizați list comprehension pentru a crea o listă cu elementele pare.
   d) Afișați lista cu elementele impare și elementele pare.

7. Creați un fișier text numit „`lista_cumparaturi.txt`” cu următoarele cumpărături pe fiecare linie: Lapte, Carne, Unt, Cafea, Fructe.
   a) Citiți și afișați conținutul fișierului.
   b) Adăugați un element nou în listă cum ar fi Legume.
   c) Scrieți lista actualizată în fișierul „`lista_cumparaturi_acualizata.txt`”.
   d) Afișați un mesaj în consolă care să anunțe că lista este scrisă în fișier.

8. Să se definească următoarele funcții ***Python***:
   a) `valoare_maxima` – primește ca parametru o listă sau un tuplu și returnează valoarea maximă din acesta.
   b) `suma` – primește ca parametri două liste (`lista1`, `lista2`) și returnează o altă listă (`lista3`), însumând elementele din `lista1` și `lista2` element cu element. Dacă lungimea celor două liste primite ca parametru nu este egală, atunci se va returna o listă goală.
   c) `factorial` – primește ca parametru un număr întreg și returnează factorialul acestuia.
   d) `export_csv` – primește ca parametru o listă și un nume de fișier și adaugă lista, în format CSV, la sfârșitul fișierului.
   e) `rezistente_paralel` – primește ca parametru o listă reprezentând valorile unor rezistențe puse în paralel și returnează rezistența echivalentă:

$$R_e = \frac{1}{\frac{1}{R_1} + \frac{1}{R_2} + \cdots + \frac{1}{R_n}}$$

9. Definiți următoarele mulțimi matematice $A = \{1, 7, 2, 8, 19, 1, -1, -5, -7, 0\}$ și $B = \{0, 9, 1, -8, 7, 22, -5, -4, -3, 2\}$ ca seturi în ***Python***. Salvați cele două seturi într-un fișier CSV.
   Calculați și afișați:
   a) $A \cup B$
   b) $A \cap B$
   c) $A \backslash B$
   d) $B \backslash A$

10. Fie fișierul `config.json`. Încărcați conținutul fișierului într-un dicționar, afișați-l pe ecran, modificați valoarea cheii "`MAX_ERR`" din sub-dicționarul "`TESTING`" la 0.01 și actualizați fișierul cu noile modificări.

Lucrarea 3. Librăria NumPy

A. Obiectivele lucrării

1. Familiarizarea cu funcționalitatea și sintaxa librăriei ***NumPy***

B. Introducere

NumPy (prescurtare de la ***Numerical Python***) este o librărie ***Python*** care oferă posibilitatea de a defini și utiliza matrice multi-dimensionale, oferind de asemenea și o serie de funcții de nivel înalt pentru a lucra cu acestea (funcții pentru manipularea matricelor, funcții pentru generarea matricelor, funcții de algebră liniară etc.) [3].

După instalarea librăriei ***NumPy*** (așa cum s-a văzut în prima lucrare), pentru a fi folosită într-un program, aceasta trebuie importată, scriind la începutul programului instrucțiunea `import` urmată de numele librăriei (`numpy`). Funcțiile și constantele definite în această librărie pot fi accesate scriind numele librăriei (`numpy`), simbolul punct și apoi numele funcției sau constantei dorite. Totuși, pentru a economisi timp, putem să utilizăm un alias mai scurt pentru numele librăriei, completând instrucțiunea import cu un alt cuvânt cheie, `as`, urmat de alias-ul dorit. În mod uzual, pentru librăria ***NumPy*** se folosește alias-ul `np`:

main.py

```
import numpy as np
```

O matrice ***NumPy*** n-dimensională este o grilă de valori de același tip, putându-se inițializa utilizând funcția `np.array` căreia îi dăm ca parametru liste imbricate (ex.: o matrice bidimensională se poate inițializa cu o listă de liste).

Accesarea elementelor se face utilizând tot paranteze drepte între care se pune indexul / indecșii elementului, numărarea indecșilor începând de la 0 [3].

Exemplu:

main.py

```
import numpy as np

# matrice unidimensionala
X = np.array([3, 2, 1])
print(X)
print(X[1])
# matrice bidimensionala
Y = np.array([[3, 2, 1], [6, 5, 4]])
print(Y)
print(Y[0, 1])
```

rezultat

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

Un al doilea parametru care se poate da funcției `np.array` este tipul de date al elementelor. Dacă acesta lipsește, ***NumPy*** îi va stabili valoarea ca fiind tipul de date cu dimensiune minimă ce poate memora toate elementele date. Tipul de date al elementelor poate lua una dintre valorile [3]:

Tabelul 1 - Tipuri de date în NumPy

Data Type[1]	Descriere
?	Boolean
b	Signed byte
B	Unsigned byte
i	Signed integer
u	Unsigned integer
f	Floating point
c	Complex floating point
U	Unicode string

Pentru o matrice ***NumPy*** deja existentă, tipul de date se poate obține accesând proprietatea `dtype`. Utilizând metoda `astype`, valorile unei matrice deja existentă se pot pune într-o altă matrice de alt tip. Exemplu:

[1] tipurile `i`, `u`, `f` și `c` pot fi urmate și de numărul de byți utilizați la stocarea elementelor (ex: `i8` este unsigned integer pe 8 byți – int64)

main.py

```
import numpy as np

x = np.array([1, 2, 3])
print(f"{x} - {x.dtype}")

x = np.array([1, 2, 3], 'f8')
print(f"{x} - {x.dtype}")

y = x.astype('c8')
print(f"{y} - {y.dtype}")
```

rezultat

```
[1 2 3] - int32
[1. 2. 3.] - float64
[1.+0.j 2.+0.j 3.+0.j] - complex64
```

C. Generarea matricelor

a) Generarea matricelor unidimensionale

NumPy oferă trei funcții utile pentru generarea de secvențe de numere sub forma unor matrice unidimensionale [3]:

Tabelul 2 - Funcții pentru generarea secvențelor de numere

Funcția	Descriere
`np.arange(amin, amax, step)`	generează o matrice unidimensională ale cărei valori vor reprezenta o secvență de numere de la `amin` la `amax` cu pasul `step`. Dacă `step` lipsește atunci pasul va fi implicit 1. **Atenție**: elementul cu valoarea `amax` nu va fi inclus în secvență.
`np.linspace(amin, amax, N)`	generează o matrice unidimensională ale cărei valori vor reprezenta o secvență de `N` numere egal distanțate în intervalul `[amin, amax]`

`np.logspace(amin, amax, N)`	generează o matrice unidimensională ale cărei valori vor reprezenta o secvență de N numere egal distanțate pe scara logaritmică în intervalul $[10^{amin}, 10^{amax}]$

Exemple:

main.py

```
import numpy as np

print(np.arange(1, 11, 2))
print(np.linspace(0, 10, 3))
print(np.logspace(0, 3, 4))
```

rezultat

```
[1 3 5 7 9]
[0.  5. 10.]
[1.e+00 1.e+01 1.e+02 1.e+03]
```

b) Generarea matricelor multidimensionale

NumPy oferă o serie de funcții utile pentru generarea de matrice multidimensionale uzuale [3]:

Tabelul 3 - Funcții pentru generarea de matrice uzuale

Funcția	Descriere	Exemplu	Rezultat
`np.zeros(t)`	generează o matrice cu toate elementele având valoarea 0, ale cărei dimensiuni sunt specificate de tuplul `t`	`np.zeros((2,2))`	`[[0. 0.]` `[0. 0.]]`

`np.ones(t)`	generează o matrice cu toate elementele având valoarea 1, ale cărei dimensiuni sunt specificate de tuplul `t`	`np.ones((1,2))`	`[[1. 1.]]`

`np.full(t,z)`	generează o matrice cu toate elementele având valoarea `z`, ale cărei dimensiuni sunt specificate de tuplul `t`	`np.full((3,2),7)`	`[[7 7]`
			` [7 7]`
			` [7 7]]`

`np.eye(x)`	generează o matrice identitate cu `x` linii și `x` coloane	`np.eye(2)`	`[[1. 0.]` ` [0. 1.]]`
`np.diag(x)`	dacă `x` este unidimensional, va genera o matrice bidimensională cu valorile din `x` pe diagonala principală, și restul elementelor 0	`np.diag([3,2])`	`[[3. 0.]` ` [0. 2.]]`

D. Indexarea elementelor matricelor

Indexarea elementelor unei matrice unidimensionale ***NumPy*** se poate face în mai multe moduri, conform tabelului [3]:

Tabelul 4 - Moduri de indexare ale matricelor NumPy

Indexare	Descriere
`X[i]`	selectează elementul cu indexul `i` din matricea `X`
`X[i:j]`	selectează elementele cu indecșii de la `i` la `j - 1` din matricea `X`
`X[:i]`	selectează elementele cu indecșii de la `0` la `i - 1` din matricea `X`
`X[j:]`	selectează elementele cu indecșii de la `j` la ultimul din matricei `X`
`X[[i, j, k]]`	selectează elementele cu indecșii `i`, `j` și `k` din matricea `X`
`X[:]`	selectează toate elementele din matricea `X`

În cazul matricelor multidimensionale, indecșii aferenți fiecărei dimensiuni se vor pune între aceleași paranteze pătrate, separați de virgulă. Pentru indecșii fiecărei dimensiuni, se pot utiliza toate metodele prezentate în tabelul anterior. Spre exemplu, pentru o matrice bidimensională `X`, dacă dorim să selectăm elementele care se află pe primele 3 linii (toate coloanele), se va utiliza: `A[:3, :]`.

Pe lângă indexarea utilizând indecșii elementelor, ***NumPy*** mai permite și indexarea booleană, cu ajutorul căreia se pot selecta elementele unei matrice care satisfac o anumită condiție. Spre exemplu:

main.py

```
import numpy as np

A = np.array([[8, 7, 6, 1], [5, 4, 3, 7], [2, 1, 0, 4]])
print(A[A > 2])
```

rezultat

```
[8 7 6 5 4 3 7 4]
```

E. Operații matematice

a) Operații aritmetice

Operațiile aritmetice uzuale (adunare, scădere, înmulțire, împărțire și ridicare la putere) se pot efectua:

1. între două matrice ***NumPy*** – operația se va efectua element cu element; **atenție**: este necesar ca matricele să aibă aceleași dimensiuni, altfel se va returna o eroare.
2. între o matrice ***NumPy*** și un scalar – operația se va realiza între fiecare element al matricei și scalarul respectiv.

Exemple:

main.py

```
import numpy as np

A = np.array([[1,2],[3,4]])
B = np.array([[5,6],[7,8]])

print(A + B)  # adunare element cu element
print(A - B)  # scadere element cu element
print(A * B)  # inmultire element cu element
print(A / B)  # impartire element cu element
print(A ** B) # ridicare la patrat element cu element

print(A + 5)  # se adaugă valoarea 5 fiecarui element
print(A * 5)  # se multiplică cu 5 fiecare element
```

b) Funcții matematice

Întrucât funcțiile matematice disponibile în librăria ***math*** care au fost prezentate în prima lucrare se pot folosi doar pentru scalari, ***NumPy*** pune la dispoziție propriul set de funcții matematice care se aplică asupra tuturor elementelor unei matrice [3]:

Tabelul 5 - Funcții matematice din NumPy

Funcții pentru reprezentarea numerelor	
`np.ceil(X)`	Rotunjirea tuturor elementelor la cel mai mic întreg mai mare
`np.floor(X)`	Rotunjirea tuturor elementelor la cel mai mare întreg mai mic

`np.round(X, d)`	Rotunjirea tuturor elementelor matricei **`X`** la **`d`** zecimale
`np.absolute(X)`	Modulul tuturor elementelor matricei
Funcții exponențiale și logaritmice	
`np.exp(X)`	**e** la puterea fiecărui element din **`X`**
`np.log(X)`	Logaritmul natural al fiecărui element al matricei
`np.log2(X)`	Logaritmul în bază 2 al fiecărui element al matricei
`np.log10(X)`	Logaritmul în bază 10 al fiecărui element al matricei
`np.sqrt(X)`	Rădăcina pătrată a fiecărui element al matricei
Funcții trigonometrice	
`np.sin(X)`	Sinusul fiecărui element al matricei (în radiani)
`np.cos(X)`	Cosinusul fiecărui element al matricei (în radiani)
`np.tan(X)`	Tangenta fiecărui element al matricei (în radiani)
`np.arcsin(X)`	Arc sinusul în radiani al fiecărui element al matricei (rezultatul între $-\pi/2$ și $\pi/2$)
`np.arccos(X)`	Arc cosinusul în radiani al fiecărui element al matricei (rezultatul între 0 și π)
`np.arctan(X)`	Arc tangenta în radiani a fiecărui element al matricei (rezultatul între $-\pi/2$ și $\pi/2$)
`np.arctan2(B, A)`	Arc tangenta în radiani a fiecărui element al matricei **`B/A`** (rezultatul între $-\pi$ și π)
Funcții de conversie unghiulară	
`np.rad2deg(X)`	Convertește valorile matricei din radiani în grade
`np.deg2rad(X)`	Convertește valorile matricei din grade în radiani
Alte funcții	
`np.minimum(X, Y)`	Returnează o matrice calculând minimul valorilor din **`X`** și **`Y`** element cu element
`np.maximum(X, Y)`	Returnează o matrice calculând maximul valorilor din **`X`** și **`Y`** element cu element
`np.amin(X, axis)`	Dacă parametrul **`axis`** lipsește, va returna valoarea minimă din matrice. Altfel, va returna o matrice cu valorile minime pe axa **`axis`** a matricei (**`axis`** = 0 – minimul fiecărei coloane; **`axis`** = 1 – minimul fiecărui rând)
`np.amax(X, axis)`	Dacă parametrul **`axis`** lipsește va returna valoarea maxima din matrice. Altfel, va returna o matrice cu valorile maxime pe axa **`axis`** a matricei

`np.sum(X, axis)`	Dacă parametrul **`axis`** lipsește va returna suma tuturor elementelor matricei. Altfel, va returna o matrice cu suma elementelor de pe axa **`axis`** a matricei
`np.prod(X, axis)`	Dacă parametrul **`axis`** lipsește va returna produsul tuturor elementelor matricei. Altfel, va returna o matrice cu produsul elementelor de pe axa **`axis`** a matricei

Pe lângă funcțiile uzuale, ***NumPy*** oferă și două constante matematice de bază: `np.pi` (constanta matematică π) și `np.e` (numărul lui Euler).

Astfel, dacă dorim să calculăm cosinusul tuturor elementelor unei matrice unidimensionale ce conține 10 valori de la 0 la $\frac{\pi}{2}$, vom proceda astfel:

main.py

```
import numpy as np

A = np.linspace(0, np.pi / 2, 10)
print(np.sin(A))
```

rezultat

```
[0.         0.17364818 0.34202014 0.5        0.64278761
0.76604444
 0.8660254  0.93969262 0.98480775 1.        ]
```

c) Algebră liniară

Librăria ***NumPy*** permite utilizarea matricelor în rezolvarea problemelor de algebră liniară oferind o serie de funcționalități și funcții dedicate. Astfel, putem determina transpusa unei matrice accesând proprietatea `T` și putem înmulți două matrice în sens algebric, utilizând operatorul `@`.

Exemplu:

main.py

```
import numpy as np

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

```
print(X @ X.T) # X înmultit cu transpusa acestuia
```

rezultat

```
[[14 32]
 [32 77]]
```

Marea majoritate a funcțiilor dedicate algebrei liniare se află în sub-modulul `linalg`. Cele mai uzuale dintre acestea sunt descrise în tabelul următor [3]:

Tabelul 6 - Funcții uzuale de algebră liniara în NumPy

Funcția	**Descriere**
`np.linalg.det(X)`	Calculează determinantul matricei **X**
`np.linalg.inv(X)`	Calculează inversa matricei **X**
`np.linalg.matrix_rank(X)`	Calculează rangul matricei **X**
`np.linalg.svd(X)`	Descompunerea pe valori singulare a matricei **X**
`np.linalg.eig(X)`	Calculează valorile și vectorii proprii ai matricei **X**
`np.linalg.solve(A, B)`	Calculează soluțiile sistemelor de ecuații scrise în formă matriceală: $A \cdot x = B$

Exemplu de calcul al determinantului și al inversei unei matrice:

main.py

```
import numpy as np

X = np.array([[1,2],[3,4]])

print(np.linalg.det(X))
print(np.linalg.inv(X))
```

rezultat

```
-2.0000000000000004
[[-2.   1. ]
 [ 1.5 -0.5]]
```

Exemplu de rezolvare a unui sistem de ecuații $\begin{cases} 2 \cdot x_1 - 3 \cdot x_2 = 5 \\ -x_1 + 2 \cdot x_2 = 8 \end{cases}$ utilizând algebra liniară.

În primul rând, va trebui să scrie sistemul de ecuații în forma matriceală:

$\begin{bmatrix} 2 & -3 \\ -1 & 2 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 5 \\ 8 \end{bmatrix}$ și vom nota $A = \begin{bmatrix} 2 & -3 \\ -1 & 2 \end{bmatrix}$ și $B = \begin{bmatrix} 5 \\ 8 \end{bmatrix}$. Atunci:

main.py

```
import numpy as np

A = np.array([[2,-3],[-1,2]])
B = np.array([[5],[8]])
print(np.linalg.solve(A, B))
```

rezultat

```
[[34.]
 [21.]]
```

O altă funcție utilă definită de ***NumPy*** este funcția `np.roots(P)` care calculează rădăcinile polinomului ai cărui coeficienți sunt definiți de matricea unidimensională `P`. Astfel, pentru a calcula rădăcinile polinomului $x^2 - 5 \cdot x + 6$, matricea P va fi $[1, -5, 6]$:

main.py

```
import numpy as np

print(np.roots(np.array([1, -5, 6])))
```

Rezultat

```
[3. 2.]
```

F. Alte funcționalități utile

Pentru a determina dimensiunea unei matrice, se va accesa proprietatea `shape` a acesteia. Această proprietate este un tuplu ce specifică dimensiunile matricei. Funcția `np.reshape` ne permite să schimbăm forma unei matrice.

Exemplu:

main.py

```
import numpy as np

A = np.array([[1, 2, 3, 4, 5, 6], [7, 8, 9, 10, 11, 12]])
print(A)
print(A.shape)

B = np.reshape(A, (3, 4))
print(B)
print(B.shape)
```

Rezultat

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

Concatenarea a două sau mai multe matrice se va face cu funcțiile `np.hstack` (concatenarea pe orizontală a acestora) și `np.vstack` (concatenarea pe verticală a acestora). Aceste funcții vor primi ca parametru un tuplu ce conține matricele ce se doresc concatenate. Exemplu:

main.py

```
import numpy as np

X = np.array([1, 2])
Y = np.array([3, 4])

A = np.hstack((X, Y))
print(A)

B = np.vstack((X, Y))
print(B)
```

rezultat

```
[1 2 3 4]
[[1 2]
 [3 4]]
```

Salvarea matricelor ***NumPy*** într-un fișier CSV se face utilizând funcția `np.savetxt`, în timp ce încărcarea unei matrice ***NumPy*** dintr-un fișier CSV se face utilizând funcția `np.loadtxt`. Aceste două funcții nu implică deschiderea prealabilă a fișierului cu ajutorul structurii `with`. Exemplu de utilizare a acestor funcții:

main.py

```
import numpy as np

X = np.array([[1, 2, 3], [4, 5, 6]])

# sciere in fisier
np.savetxt("test.csv", X, delimiter = ",")

# citire din fisier
Y = np.loadtxt("test.csv", delimiter = ",")

print(Y)
```

rezultat

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

G. Cerințe

1. Folosind funcția `np.arange`, să se genereze și să se afișeze următoarele matrice:
   a) $[1 \quad 3 \quad 5 \quad 7 \quad \dots \quad 49]$
   b) $[12 \quad 11 \quad 10 \quad \dots \quad -10 \quad -11 \quad -12]$
   c) $\begin{bmatrix} 0 & \frac{1}{2} & \frac{2}{3} & \frac{3}{4} & \dots & \frac{59}{60} \end{bmatrix}$

2. Folosind funcția `np.linspace`, să se genereze și afișeze următoarele matrice:
   a) $[4 \quad 6 \quad 8 \quad 10]$
   b) $[-2 \quad -6 \quad -10 \quad -14 \quad -18]$
   c) $[12 \quad 10 \quad 8 \quad 6 \quad 4 \quad 2]$

3. Să se rezolve următorul sistem liniar $\begin{cases} 2 \cdot x - 3 \cdot y = 8 \\ 3 \cdot x - 2 \cdot y = -5 \end{cases}$ folosind ***NumPy***. Afișați rezultatele acestuia.

4. Să dau următoarele cerințe:
   a) Creați și afișați o matrice ***NumPy*** cu valori întregi.
   b) Utilizați proprietatea "`dtype`" pentru a afișa tipul de date al matricei.
   c) Creați și afișați o matrice ***NumPy*** cu valori în virgulă mobilă.
   d) Utilizați metoda „`astype`" pentru a crea o altă matrice cu aceleași valori, dar de tip întreg.

5. Folosind ***NumPy***, convertiți următoarea listă de valori numerice, `lista_1 = [15.56, 56.00, 52.36, 56, 58]` într-o matrice ***NumPy*** unidimensională.

6. Să se genereze o matrice cu dimensiunea 3x3 cu valori cuprinse între 5 și 14.

7. Fie matricele $A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$ și $B = \begin{pmatrix} 8 & 10 \\ 8 & 5 \\ 1 & 5 \end{pmatrix}$. Calculați și afișați:
   a) $A + 5,\ \ A - 5,\ \ A * 5, \frac{A}{5}, A + B^T,\ \ A - B^T,\ \ A * B^T,\ \ A/B^T$
   b) $A@B$, $(A@B)@(A@B)^{-1}$

8. Fie matricea $C = \begin{pmatrix} 2 & -4 & 1 \\ 6 & 3 & 2 \\ 3 & -2 & 4 \end{pmatrix}$. Să se calculeze și să se afișeze dimensiunea, transpusa, determinantul, inversa și rangul matricei C.

9. Să se calculeze și să se afișeze sinusul, cosinusul și tangenta elementelor matricei unidimensionale $y = [0° \quad 20° \quad 40° \quad ... \quad 180°]$.

10. Fie matricea $D = \begin{pmatrix} 6 & 5 & 4 & 3 & 2 & 1 \\ 1 & 2 & 3 & 4 & 5 & 6 \\ -6 & -5 & -4 & -3 & -2 & -1 \\ 4 & 5 & 6 & 7 & 8 & 9 \\ 8 & 7 & 6 & 5 & 4 & 3 \end{pmatrix}$

    a) Să se afișeze dimensiunea matricei.
    b) Să se extragă și să se afișeze submatricea E de dimensiune 3x3, ce constă din elementele situate pe ultimele 3 linii și primele 3 coloane ale matricei D.
    c) Să se extragă și să se afișeze submatricea F de dimensiune 3x3, ce constă din elementele situate pe primele 3 linii și ultimele 3 coloane ale matricei D.
    d) Să se concateneze matricele E și F sub forma unei matrice de dimensiune 3x6.
    e) Să se concateneze matricele E și F sub forma unei matrice de dimensiune 6x3.
    f) Să se redimensioneze matricea D din 5x6 într-o matrice 10x3 și să se salveze conținutul într-un fișier CSV.

11. Fie fișierul „`date.csv`”. Încărcați fișierul într-o matrice ***NumPy***.
    a) Să se afișeze dimensiunea matricei.
    b) Calculați și afișați minimul, maximul fiecărei coloane din matrice.
    c) Calculați și afișați minimul, maximul și suma fiecărei linii din matrice.

Lucrarea 4. Librăria Pandas

A. Obiectivele lucrării

1. Familiarizarea cu funcționalitatea și sintaxa librăriei ***Pandas***.
2. Analiza unui set de date real, utilizând funcționalitățile librăriei ***Pandas***.

B. Introducere

Pandas (prescurtare de la ***Python Data Analysis***) este o librărie ***Python*** utilizată pentru manipularea și analiza datelor. În principiu, ea oferă structuri de date și funcții pentru manipularea tabelelor și a seriilor de date, oferind posibilitatea de a analiza, explora, curăța și manipula cantități mari de date. Utilizând ***Pandas***, se pot trage concluzii pe baza analizei statistice a cantităților mari de date [4].

Pandas are la bază ***NumPy***, de la care adoptă mare parte a sintaxei, doar că, spre deosebire de ***NumPy*** care este construit astfel încât să lucreze cu matrice cu valori numerice de același tip (deci matrice omogenă), ***Pandas*** poate lucra cu date eterogene, adică în aceeași structură de date pot fi tipuri de date diferite.

După instalarea librăriei ***Pandas*** (așa cum s-a văzut în prima lucrare), pentru a fi folosită într-un program, aceasta trebuie importată scriind la începutul programului instrucțiunea `import` urmată de numele librăriei (`pandas`).

Totuși, pentru a economisi timp, putem să utilizăm un alias mai scurt pentru numele librăriei, în mod uzual, pentru librăria ***Pandas*** folosindu-se alias-ul `pd`:

main.py

```
import pandas as pd
```

Pe lângă structurile de date din limbajul de bază ***Python*** (liste, tupluri, seturi și dicționare), ***Pandas*** introduce două noi structuri [4]:

*Tabelul 1 – Structuri de date definite de **Pandas***

Structură de date	Descriere
Series	tablou unidimensional, etichetat, ce poate conține date de orice tip
DataFrame	tablou bidimensional (tabel), cu ambele axe (rânduri și coloane) etichetate.

C. Serii de date

Seriile de date (***Series***) sunt tablouri unidimensionale etichetate. Acestea pot conține orice tip de date, cu condiția ca toate valorile din serie să fie de același tip. Inițializarea seriilor se poate face utilizând funcția `Series` care poate primi unul sau mai mulți parametri:

1. `data` – obiect iterabil (lista, dicționar, matrice ***NumPy*** etc.) ce conține valorile propriu-zise ale seriei de date; dacă este de tip dicționar, cheile vor reprezenta etichetele valorilor.
2. `index` – vector de aceeași dimensiune cu `data` conținând etichetele; dacă lipsește, atunci seria de date va avea ca etichete valori de la 0 la numărul de elemente.
3. `dtype` – tipul de date al seriei specificat în același mod ca la matricele NumPy; dacă lipsește, tipul de date va fi moștenit de la `data`.

Exemplu de inițializare a seriilor de date utilizând o listă, fără a specifica etichetele:

main.py

```
import pandas as pd

A = pd.Series([7, 5, 3])
print(A)
```

rezultat

```
0    7
1    5
2    3
dtype: int64
```

Exemplu de inițializare a seriilor de date utilizând o listă, cu etichetele specificate:

main.py

```
import pandas as pd

A = pd.Series([7, 5, 3], ["a", "b", "c"])
print(A)
```

rezultat

```
a    7
b    5
c    3
dtype: int64
```

Exemplu de inițializare a seriilor de date utilizând un dicționar, fără a specifica etichetele:

main.py

```
import pandas as pd

d = {"a": 7, "b": 8, "c": 9}

A = pd.Series(d)
print(A)
```

rezultat

```
a    7
b    8
c    9
dtype: int64
```

Exemplu de inițializare a seriilor de date utilizând un dicționar, cu etichetele specificate:

main.py

```
import pandas as pd

d = {"a": 7, "b": 8, "c": 9}

A = pd.Series(d, ["c", "d", "e", "a", "b"])
print(A)
```

rezultat

```
c    9.0
d    NaN
e    NaN
a    7.0
b    8.0
dtype: float64
```

Indexarea elementelor seriilor de date se face în mod asemănător indexării dicționarelor (adică specificând etichetele între paranteze pătrate). Exemplu:

main.py

```
import pandas as pd

note = pd.Series({"Ion": 7, "Vasile": 8, "George": 9})
note["Ion"] = 2

print(note["Vasile"])
```

rezultat

```
8
```

Seriile de date acționează foarte asemănător cu matricele unidimensionale ***NumPy***, un obiect de tip ***Series*** fiind un argument valid pentru marea majoritate a funcțiilor definite de ***NumPy***. De asemenea, seriile de date pot fi convertite în matrice unidimensionale ***NumPy***, utilizând metoda `to_numpy()`:

main.py

```
import pandas as pd
import numpy as np

date1 = pd.Series([1, 2, 3, 4, 5, 6])
date2 = np.sqrt(date1)

print(date2)
print(date2.to_numpy())
```

rezultat

```
0    1.000000
1    1.414214
2    1.732051
3    2.000000
4    2.236068
```

```
5    2.449490
dtype: float64
[1.          1.41421356  1.73205081  2.          2.23606798
2.44948974]
```

Între două serii de date se pot aplica oricare dintre operațiile aritmetice de bază (adunare, scădere, înmulțire, împărțire, ridicare la putere), caz în care operațiile se vor realiza element cu element, cu mențiunea că cele două serii vor fi aliniate după etichete:

main.py

```
import pandas as pd

note_examen = pd.Series({"Ion":7, "Vasile":8, "George":9,
"Alin":2})
note_laborator = pd.Series({"Ion": 2, "George":10,
"Vasile":6})


media = (note_examen + note_laborator) / 2
print(media)
```

rezultat

```
Alin      NaN
George    9.5
Ion       4.5
Vasile    7.0
```

Pandas definește, pentru obiectele de tip ***Series***, o serie de metode pentru manipularea și analiza datelor [4]. Cele mai uzuale pot fi observate în tabelul următor:

Tabelul 2 - Metode pentru manipularea și analiza seriilor de date

Metode pentru manipularea seriilor de date	
`X.count()`	returnează numărul total de elemente diferite de ***`NaN`***/***`null`*** din serie
`X.drop(l)`	returnează o copie a seriei cu elementele ale căror etichete sunt specificate de lista `l` șterse
`X.drop_duplicates()`	returnează o copie a seriei cu elementele cu valori duplicate eliminate
`X.pop(i)`	șterge elementul cu eticheta `i` din serie

`X.astype(d)`	convertește seria la tipul de date specificat de `d`
Metode pentru analiza statistică a datelor	
`X.min()`	returnează valoarea minimă dintr-o serie
`X.max()`	returnează valoarea maximă dintr-o serie
`X.mean()`	returnează valoarea media dintr-o serie
`X.median()`	returnează mediana unei serii
`X.mode()`	returnează modul unei serii
`X.std()`	returnează deviația standard a unei serii
`X.describe()`	generează o statistică descriptivă a datelor dintr-o serie

D. DataFrame

DataFrame-ul este o structură de date bidimensională, cu ambele axe (rânduri și coloane) etichetate. Practic, obiectele de tip ***DataFrame*** pot fi văzute ca serii de serii. Acestea pot conține orice tip de date, cu condiția ca toate valorile de pe o coloană să fie de același tip.

Inițializarea **DataFrame**-urilor se poate face utilizând funcția `DataFrame` care poate primi unul sau mai mulți parametri [4]:

1. `data` – dicționar de iterabile (liste, serii de date etc.); cheile dicționarului vor reprezenta etichetele coloanelor, în timp ce valorile vor reprezenta valorile obiectului pe coloana respectivă.
2. `index` – vector conținând etichetele rândurilor. Dacă lipsește, atunci liniile vor avea ca etichete valori de la 0 la numărul de elemente.

Exemplu:

main.py

```
import pandas as pd

df = pd.DataFrame({
    "Nume": ["Ion", "Vasile", "Gheorghe"],
    "Nota Examen": [10, 4, 7],
    "Nota Laborator": [9, 8, 6]
    })

print(df)
```

rezultat

```
       Nume  Nota Examen  Nota Laborator
0       Ion            10               9
1    Vasile             4               8
2  Gheorghe             7               6
```

Totuși, pentru că ***Pandas*** este dedicat manipulării și analizei datelor de mari dimensiuni, cea mai comună metodă de inițializare a unui ***DataFrame*** este prin importarea dintr-o sursă externă (de obicei fișier CSV). Spre exemplu, dându-se un fișier CSV care conține informații despre emisiunile și filmele Netflix (https://www.kaggle.com/shivamb/netflix-shows), importarea acestuia se poate face astfel:

main.py

```
import pandas as pd

df = pd.read_csv('netflix_titles.csv')

print(df)
```

rezultat

```
     show_id  ...                                        description
0         s1  ...  As her father nears the end of his life, ...
1         s2  ...  After crossing paths at a party, a Cape T...
2         s3  ...  To protect his family from a powerful dru...
3         s4  ...  Feuds, flirtations and toilet talk go dow...
4         s5  ...  In a city of coaching centers known to tr...
...      ...  ...                                           ...
8802   s8803  ...  A political cartoonist, a crime reporter ...
8803   s8804  ...  While living alone in a spooky town, a yo...
8804   s8805  ...  Looking to survive in a world taken over ...
8805   s8806  ...  Dragged from civilian life, a former supe...
8806   s8807  ...  A scrappy but poor boy worms his way into...

[8807 rows x 12 columns]
```

Se poate observa că tabelul are 8807 rânduri și 12 coloane, dar afișarea se face trunchiat. De asemenea, implicit, prima linie din fișierul CSV acționează ca etichete pentru coloane, în timp ce rândurile au fost etichetate cu o secvență de numere de la 0 la 8806.

Pentru că afișarea se face trunchiat, se poate utiliza metoda `info()` [4], pentru a vedea ce coloane sunt disponibile și ce tip de date conțin ele.

main.py

```
import pandas as pd

df = pd.read_csv('netflix_titles.csv')

df.info()
```

rezultat

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8807 entries, 0 to 8806
Data columns (total 12 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   show_id       8807 non-null   object
 1   type          8807 non-null   object
 2   title         8807 non-null   object
 3   director      6173 non-null   object
 4   cast          7982 non-null   object
 5   country       7976 non-null   object
 6   date_added    8797 non-null   object
 7   release_year  8807 non-null   int64
 8   rating        8803 non-null   object
 9   duration      8804 non-null   object
 10  listed_in     8807 non-null   object
 11  description   8807 non-null   object
dtypes: int64(1), object(11)
memory usage: 825.8+ KB
```

Implicit, la importarea unui fișier CSV, etichetele rândurilor sunt generate ca o secvență de numere de la 0 la numărul total de rânduri. Totuși putem seta una dintre coloanele tabelului ca reprezentând etichetele rândurilor:

main.py

```
import pandas as pd
df = pd.read_csv('netflix_titles.csv')
df = df.set_index("show_id")
print(df)
```

rezultat

```
             type  ...                                    description
show_id            ...
s1          Movie  ...  As her father nears the end of his ...
s2        TV Show  ...  After crossing paths at a party, a ...
```

De asemenea, dacă în urma importării unui fișier CSV, tipul de date aferent fiecărei coloane nu a fost stabilit în mod corect, utilizând metoda `astype` căreia îi dăm ca parametru un dicționar ale cărui perechi cheie – valoare reprezintă perechi coloană – tip data putem seta tipurile de date corecte. Pe lângă tipurile clasice de date, se pot seta și alte tipuri, precum `datetime64[ns]` (care să țină minte o dată) sau `category` (pentru coloanele care specifică categorii).

main.py

```
import pandas as pd

df = pd.read_csv('netflix_titles.csv')
df.set_index("show_id", inplace=True)

df = df.astype({
    "type":"category",
    "release_year":"i4",
    "date_added":"datetime64[ns]"})
df.info()
```

rezultat

```
<class 'pandas.core.frame.DataFrame'>
Index: 8807 entries, s1 to s8807
Data columns (total 11 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   type          8807 non-null   category
 1   title         8807 non-null   object
 2   director      6173 non-null   object
 3   cast          7982 non-null   object
 4   country       7976 non-null   object
 5   date_added    8797 non-null   datetime64[ns]
 6   release_year  8807 non-null   int32
 7   rating        8803 non-null   object
 8   duration      8804 non-null   object
 9   listed_in     8807 non-null   object
 10  description   8807 non-null   object
dtypes: category(1), datetime64[ns](1), int32(1), object(8)
memory usage: 731.2+ KB
```

a) Indexarea DataFrame-urilor

Selectarea unei singure coloane se poate face scriind, după numele variabilei de tip ***DataFrame***, între paranteze pătrate eticheta coloanei respective, rezultând o serie ce conține doar datele de pe acea coloană. Exemplu:

main.py

```
import pandas as pd

df = pd.read_csv('netflix_titles.csv')
titluri = df["title"]

print(titluri)
```

rezultat

```
0         Dick Johnson Is Dead
1                Blood & Water
2                    Ganglands
3       Jailbirds New Orleans
4                 Kota Factory
                ...
8802                    Zodiac
8803              Zombie Dumb
8804               Zombieland
8805                    Zoom
8806                   Zubaan
Name: title, Length: 8807, dtype: object
```

Dacă între parantezele pătrate se dă o listă de etichete de coloane, va rezulta tot un ***DataFrame*** conținând doar coloanele specificate. Există mai multe modalități de a extrage elemente, rânduri, coloane sau subseturi, dar cele mai eficiente dintre ele includ utilizarea proprietăților `loc` (indexare pe baza etichetelor rândurilor și coloanelor) și `iloc` (indexare pe baza poziției rândurilor și coloanelor) [4]. Exemplu:

main.py

```
import pandas as pd

df = pd.read_csv('netflix_titles.csv')
df = df.set_index("show_id")

print(df.iloc[5, 1])
```

```
print(df.iloc[5:7, 1:3])
print(df.loc["s6", 'director'])
```

rezultat

```
Midnight Mass
                                    title
director
show_id
s6                          Midnight Mass                    Mike
Flanagan
s7    My Little Pony: A New Generation  Robert Cullen, José
Luis Ucha
Mike Flanagan
```

Dacă la utilizarea acestor proprietăți se folosește doar un index / o etichetă, atunci el va reprezenta rânduri, deci astfel putem extrage un rând dintr-un ***DataFrame***:

main.py

```
import pandas as pd

df = pd.read_csv('netflix_titles.csv')
print(df.iloc[0])
```

rezultat

```
show_id                                                      s1
type                                                      Movie
title                                      Dick Johnson Is Dead
director                                        Kirsten Johnson
cast                                                       NaN
country                                          United States
date_added                                  September 25, 2021
release_year                                              2020
rating                                                   PG-13
duration                                                 90 min
listed_in                                          Documentaries
description     As her father nears the end of his life, ...
Name: 0, dtype: object
```

b) Filtrarea datelor

O sarcină tipică în procesarea datelor este filtrarea acestora, adică extragerea anumitor înregistrări. ***Pandas*** oferă mai multe metode de a face acest lucru, una dintre ele fiind utilizarea metodei `query` care primește ca parametru un șir de caractere ce specifică expresia ce descrie filtrarea dorită [4]. Spre exemplu, dacă se dorește selectarea doar a filmelor/emisiunilor lansate după anul 2020:

main.py

```
import pandas as pd

df = pd.read_csv('netflix_titles.csv')

filme_noi = df.query("release_year > 2020")
print(filme_noi["title"])
```

rezultat

```
1                    Blood & Water
2                        Ganglands
3            Jailbirds New Orleans
4                     Kota Factory
                   ...
1696                  Polly Pocket
2920                 Love Is Blind
8437        The Netflix Afterparty
Name: title, Length: 592, dtype: object
```

Bineînțeles, se pot scrie expresii de filtrare mult mai complexe. Tabelul următor prezintă o serie de astfel de expresii:

Tabelul 3 – Exemple de expresii pentru filtrarea datelor

Expresie	Descriere
`release_year >= 2020`	filmele/emisiunile lansate începând cu anul 2020
`release_year == 2020 and type == 'Movie'`	filmele lansate în anul 2020
`release_year in (1999, 2007)`	filmele/emisiunile lansate fie în anul 1999 fie în anul 2007
`listed_in.str.contains('Documentaries')`	filmele/emisiunile care sunt de tip documentar

c) Analiza datelor

Bineînțeles, dacă se dorește analiza statistică a datele dintr-o coloană, se vor putea utiliza metodele prezentate anterior, specifice seriilor de date, după ce extragem coloana respectivă. Aceste metode se pot aplica și obiectelor de tip DataFrame, caz în care parametrul statistic dorit se va calcula pentru fiecare coloană ce conține un tip de date asupra căruia se poate aplica metoda (de exemplu: maximul se poate aplica pentru date numerice, dar și pentru șiruri de caractere, în timp ce media sau abaterea standard poate fi calculată doar pentru date numerice).

O metodă utilă la analiza statistică a datelor este metoda `groupby` care primește ca parametru o etichetă de coloană și care permite gruparea datelor și aplicarea metodelor de analiză statistică fiecărui grup. Spre exemplu, dacă se dorește să se afle câte emisiuni/filme au fost lansate în fiecare an, se vor extrage coloanele `release_year` și `title`, se va grupa după coloana `release_year` apoi se va aplica metoda `count`:

main.py

```
import pandas as pd

df = pd.read_csv('netflix_titles.csv')

print(df[["release_year",
"title"]].groupby("release_year").count())
```

rezultat

```
              title
release_year
1925              1
1942              2
1943              3
...             ...
2020            953
2021            592
[74 rows x 1 columns]
```

O altă metodă utilă pentru analiza datelor o reprezintă `corr`, care va calcula corelația fiecărei perechi de coloane [4]. Practic, această metodă este utilă pentru a cunoaște gradul de similaritate/dependență între oricare două coloane. Exemplu:

main.py

```
import pandas as pd
import numpy as np

df = pd.DataFrame({
    "x": np.arange(5, 10, 1),
    "y": np.sqrt(np.arange(5, 10, 1)),
    "z": [3, 2, 8, 1, 9]
})

print(df.corr())
```

rezultat

```
          x         y         z
x  1.000000  0.999067  0.491935
y  0.999067  1.000000  0.458418
z  0.491935  0.458418  1.000000
```

d) Alte metode pentru manipularea DataFrame-urilor

Pandas pune la dispoziție o serie de metode pentru manipularea ***DataFrame***-urilor. Acestea se aplică obiectelor de tip ***DataFrame***, dar nu le modifică, ci returnează un alt ***DataFrame*** cu modificările dorite. Cele mai uzuale dintre aceste metode sunt descrise în tabelul următor [4].

*Tabelul 4 – Exemple de metode de manipulare a **DataFrame**-urilor*

Metoda	Descriere
`X.drop(labels,axis)`	șterge rândurile/coloanele etichetate cu etichetele specificate de parametrul `labels`; parametrul `axis` specifică dacă e vorba de rânduri (0) sau coloane (1)
`X.dropna(axis)`	șterge rândurile sau coloanele din care lipsesc date
`X.drop_duplicates()`	șterge rândurile duplicate
`X.merge(Y, on)`	unește coloanele ***DataFrame***-ului `X` cu coloanele ***DataFrame***-ului `Y` aliniindu-le după coloana specificată de parametrul `on`. Dacă parametrul `on` lipsește, alinierea se va face după etichetele liniilor.

`X.rename(map, axis)`	redenumește etichetele rândurilor/coloanelor conform dicționarului `map`. Parametrul `axis` specifică dacă e vorba de rânduri sau coloane

Exemplu:

main.py

```
df1 = pd.DataFrame({
    "sensor_id": [1, 2, 4, 3, 2, 1, 1],
    "value": [22, 0.8, 28, 7, 0.6, 21, 20]
})

df2 = pd.DataFrame({
    "id_sensor": [1, 2, 3, 4],
    "sensor_name": ["Temp1", "Pres", "Temp2", "pH"]
})

df2 = df2.rename({"id_sensor": "sensor_id"}, axis = 1)

print(df1.merge(df2, on="sensor_id"))
```

rezultat

```
   sensor_id  value sensor_name
0          1   22.0       Temp1
1          1   21.0       Temp1
2          1   20.0       Temp1
3          2    0.8        Pres
4          2    0.6        Pres
5          4   28.0          pH
6          3    7.0       Temp2
```

E. Cerințe

Fie fișierul `date_climatice_2016.csv`, ce conține valorile minime, maxime și medii ale temperaturilor înregistrate în fiecare zi a anului 2016 de către cele 23 de stații esențiale [5]. Rezolvați următoarele cerințe:

1. Încărcați datele din fișierul CSV dat într-o variabilă de tip ***DataFrame*** (`df1`) și afișați atât datele, cât și informațiile despre coloane.

2. Ștergeți din `df1` coloanele `LAT`, `LON` și `R24`.

3. Corectați tipurile de date pentru coloanele `DATCLIM` (***datetime64[ns]***) și `CODST` (***category***) și reafișați informațiile despre coloane.

4. Încărcați în variabila `df2` de tip ***DataFrame*** fișierul `statii_meteo.csv`, ce conține ID-urile și numele stațiilor meteo și afișați atât datele, cât și informațiile despre coloane.

5. Observăm că coloanele aferente ID-urilor stațiilor din cele doua ***DataFrame***-uri au nume diferite. Redenumiți în `df2` coloana `ID` în `CODST` și coloana `Nume` în `NUME`.

6. Modificați `df1` prin a uni cu acest ***DataFrame*** coloanele din `df2` aliniindu-le pe coloana `CODST`.

7. Calculați și afișați din `df1`:

    a. temperatura medie înregistrată (media coloanei `TMED`).
    b. temperatura maximă înregistrată (maximul coloanei `TMAX`) și numele stației la care a fost ea înregistrată (**indiciu**: pentru numele stației va trebui să utilizați metoda `query`).
    c. temperatura minimă înregistrată (minimul coloanei `TMIN`) și numele stației la care a fost ea înregistrată.

8. Creați un nou obiect de tip ***DataFrame*** numit `date_statistice`, efectuând următorii pași:

    a. selectați din `df1` doar coloanele `NUME`, `ALT` și `TMIN`, grupați după coloana `NUME`, calculați minimul și puneți rezultatele în `date_statistice`.
    b. selectați din `df1` doar coloanele `NUME` și `TMED`, grupați după coloana `NUME`, calculați media și uniți `date_statistice` cu rezultatul.

c. selectați din `df1` doar coloanele `NUME` și `TMAX`, grupați după coloana `NUME`, calculați maximul și uniți `date_statistice` cu rezultatul.
d. afișați datele din `date_statistice`.

9. Studiați corelația temperaturilor minime, medii și maxime cu altitudinea (**indiciu**: aplicați metoda `corr` asupra obiectului `date_statistice`).

Lucrarea 5. Librăria Matplotlib

A. Obiectivele lucrării

1. Familiarizarea cu funcționalitatea și sintaxa librăriei ***Matplotlib.***

B. Introducere

Matplotlib este o librărie ***Python*** utilizată pentru generarea și afișarea graficelor. Această librărie conține un sub-modul numit ***pyplot*** ce oferă un mod de lucru și o sintaxă asemănătoare MATLAB-ului [6]. Întrucât vom folosi doar sub-modulul ***pyplot***, îl vom importa doar pe acesta putându-i da și un alias. În mod uzual, se folosește aliasul `plt`:

main.py

```
import matplotlib.pyplot as plt
```

C. Reprezentări grafice

Pentru reprezentări grafice ale datelor, ***Matplotlib*** pune la dispoziție o multitudine de funcții, dintre care, cea mai uzuală este funcția `plt.plot` ce are o sintaxă asemănătoare cu funcția `plot` din MATLAB. Astfel, această funcție primește ca parametri două matrice unidimensionale reprezentând valorile de pe abscisă, respectiv valorile de pe ordonată, și, opțional, un parametru de formatare de tip șir de caractere ce poate specifica atât culoarea liniei, cât și tipul acesteia și tipul de marcator, conform Tabelului 1.

După ce s-au adăugat toate elementele necesare pe grafic, pentru a putea vedea rezultatul, trebuie apelată funcția `plt.show()`. **Atenție**, codul scris după apelarea funcției `plt.show()` se va executa doar după închiderea graficului.

Tabelul 1 – Opțiuni de formatare a graficelor realizate cu `plt.plot` [7]

Culoare		Tip linie		Tip marcaj	
Simbol	**Descriere**	**Simbol**	**Descriere**	**Simbol**	**Descriere**
y	galben	-	continuă	+	plus
m	magenta	--	întreruptă	o	cerc
c	turcoaz	:	punctată	*	steluță
r	roșu	-.	linie-punct	.	punct
g	verde			x	x
b	albastru			s	pătrat
w	alb			d	romb
k	negru			^	triunghi ▲
				v	triunghi ▼
				>	triunghi ▶
				<	triunghi ◀
				p	pentagon
				h	hexagon

Exemplu:

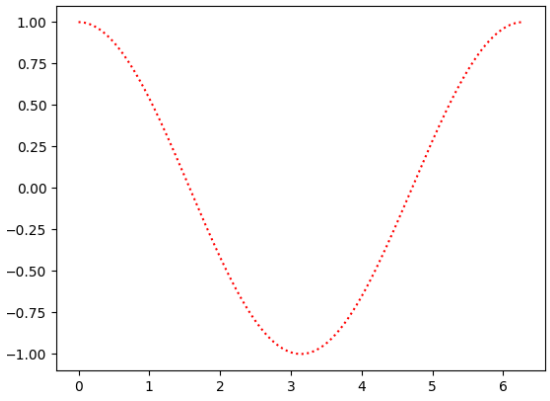
main.py

```
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(0, 2 * np.pi, 0.01)
y = np.cos(x)

plt.plot(x, y, "r:")
plt.show()
```

rezultat



Afișarea mai multor grafice, se poate face fie în ferestre diferite, fie în cadrul aceleiași ferestre. Pentru primul caz, este necesară apelarea funcției `plt.figure(n)` (unde `n` reprezintă numărul figurii / ferestrei) înainte de apelarea funcției `plt.plot`. În cel de-al doilea caz, pentru a afișa mai multe grafice în aceeași fereastră există trei posibilități [2]:

1. se poate apela funcția `plt.plot` de mai multe ori, pentru fiecare set de date ce se dorește reprezentat grafic; nu este necesară utilizarea unui echivalent pentru `hold on`.
2. se pot da mai mulți parametri funcției `plt.plot` [6]:
   a. `plt.plot(x1, y1, x2, y2, ...)`
   b. `plt.plot(x1, y1, f1, x2, y2, f2, ...)`
3. se poate utiliza funcția `plt.subplot(m, n, p)` ce va împărți fereastra într-o matrice de sub-ferestre, dispuse pe `m` rânduri și `n` coloane, și va selecta sub-fereastra cu numărul `p` pentru a afișa ulterior în ea. Numerotarea sub-ferestrelor se face la fel ca în MATLAB, începând cu numărul 1, de la stânga la dreapta și de sus în jos.

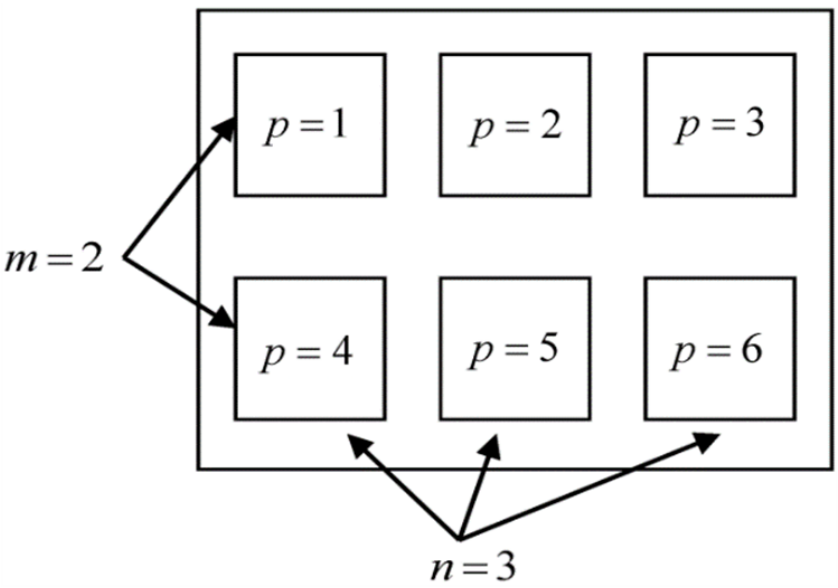


Figura 1 – Exemplu de împărțire și numerotare a sub-ferestrelor [7]

main.py

```
import numpy as np
import matplotlib.pyplot as plt

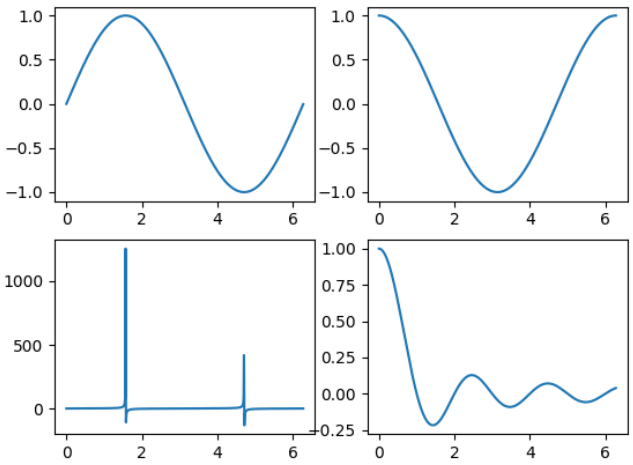
x = np.arange(0, 2 * np.pi, 0.01)

y1 = np.sin(x)
y2 = np.cos(x)
y3 = np.tan(x)
y4 = np.sinc(x)

plt.subplot(2, 2, 1)
plt.plot(x, y1)
plt.subplot(2, 2, 2)
plt.plot(x, y2)
plt.subplot(2, 2, 3)
plt.plot(x, y3)
plt.subplot(2, 2, 4)
plt.plot(x, y4)

plt.show()
```

rezultat



Asemănător mediului MATLAB, ***Matplotlib*** permite reprezentarea grafică și în coordonate logaritmice sau semilogaritmice:

Tabelul 2 – Funcții pentru reprezentarea în coordonate logaritmice și semilogaritmice

Funcția	Descriere
`plt.semilogx(x, y)`	grafic cu axa `x` logaritmică
`plt.semilogy(x, y)`	grafic cu axa `y` logaritmică
`plt.loglog(x, y)`	grafic cu ambele axe logaritmice

Matplotlib mai permite și reprezentarea grafică în coordonate polare utilizând funcția `plt.polar(theta, r)`, unde `theta` și `r` sunt matrice unidimensionale specificând unghiurile, respectiv distanțele [6].

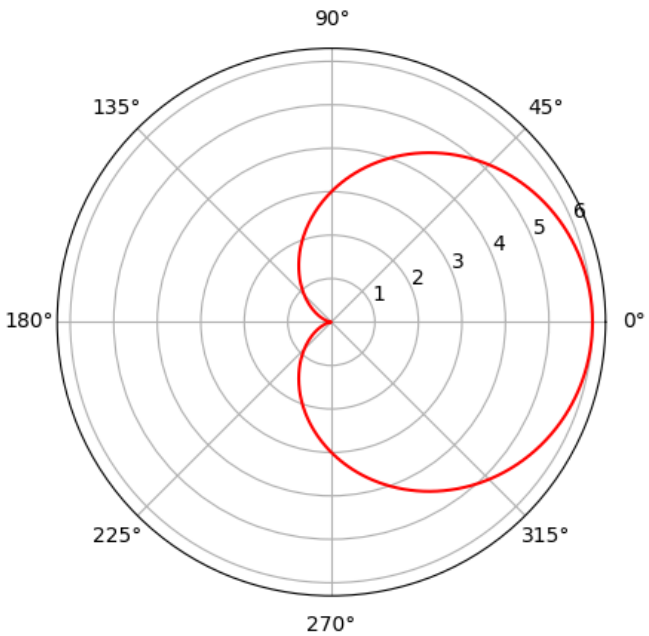
main.py

```
import numpy as np
import matplotlib.pyplot as plt

theta = np.arange(0, 2 * np.pi, 0.01)
r = 3 * (np.cos(theta) + 1)

plt.polar(theta ,r, 'r')
plt.show()
```

rezultat



Alte funcții specializate pentru reprezentări grafice:

Tabelul 3 – Funcții specializate pentru reprezentări grafice [6]

Funcția	**Descriere**
`plt.scatter(x, y)`	reprezentare grafică ca puncte
`plt.bar(x, y)`	reprezentare grafică cu bare
`plt.barh(y, x)`	reprezentare grafică cu bare orizontale
`plt.fill(x, y)`	reprezentare grafică ca poligon
`plt.stem(x, y)`	reprezentare grafică ca eșantioane
`plt.step(x, y)`	reprezentare grafică în trepte
`plt.pie(x, labels)`	reprezentare grafică tip „*pie chart*”
`plt.errorbar(x, y, err)`	reprezentare grafică cu erori

D. Adnotarea graficelor

Totuși, în inginerie, în marea majoritate a cazurilor este necesar ca pe grafic să apară mai multe detalii despre reprezentare, așa că ***Matplotlib*** implementează o serie de funcții pentru cosmetizarea graficului. Dintre aceste, cele mai importante sunt descrise în tabelul următor [6]:

Tabelul 4 – Opțiuni de cosmetizare a graficelor realizate cu `plt.plot` [7]

Funcția	**Descriere**
`plt.title(s)`	setează titlul graficului la valoarea șirului de caractere `s`
`plt.suptitle(s)`	setează titlul superior al graficului la valoarea șirului de caractere `s`
`plt.xlabel(s)`	setează eticheta axei `x` la valoarea șirului de caractere `s`
`plt.ylabel(s)`	setează eticheta axei `y` la valoarea șirului de caractere `s`
`plt.grid()`	adaugă grila pe grafic
`plt.xlim(xmin, xmax)`	setează intervalul de afișare al axei `x`
`plt.ylim(ymin, ymax)`	setează intervalul de afișare al axei `y`

`plt.legend(lista)`	adaugă o legendă, fiecărui grafic atribuindu-i-se o etichetă din lista primită ca parametru
`plt.text(x, y, s)`	adaugă textul `s` la coordonatele (`x`, `y`)
`plt.annotate(s, xy, xy2, arrowprops = {})`	adaugă textul `s` la coordonatele specificate de tuplul `xy2` și săgeată spre coordonatele date de tuplul `xy`

Pentru că este posibil ca atunci când adăugăm noi elemente graficului (etichete, titluri etc.) acestea să se suprapună cu graficul sau între ele, se poate apela funcția `plt.tight_layout()` pentru reașezarea automată a elementelor.

main.py

```
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(0, 2 * np.pi, 0.01)

y1 = np.sin(x)
y2 = np.cos(x)
y3 = np.tan(x)

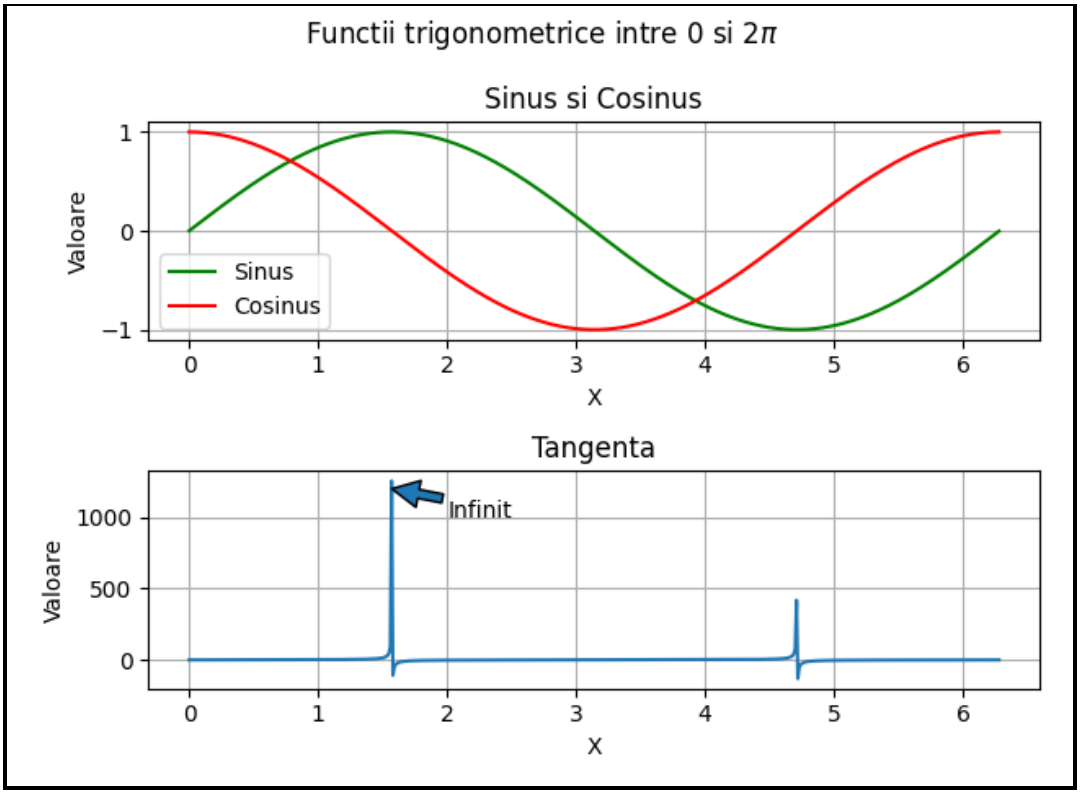
plt.subplot(2, 1, 1)
plt.plot(x, y1, 'g', x, y2, 'r')
plt.xlabel("X")
plt.ylabel("Valoare")
plt.title("Sinus si Cosinus")
plt.grid()
plt.legend(["Sinus", "Cosinus"])


plt.subplot(2, 1, 2)
plt.plot(x, y3)
plt.xlabel("X")
plt.ylabel("Valoare")
plt.title("Tangenta")
plt.annotate("Infinit", (np.pi / 2, 1200), (2, 1000),
arrowprops={})

plt.grid()

plt.suptitle("Functii trigonometrice intre 0 si 2$\pi$ ")
plt.tight_layout()
plt.show()
```

rezultat



Toate funcțiile pentru cosmetizarea graficelor ce primesc ca parametru un șir de caractere permit utilizare, în șirul de caractere, a sintaxei TeX între două simboluri `$`. La adresa https://matplotlib.org/stable/tutorials/text/mathtext.html se poate consulta o descriere detaliată a acestei sintaxe.

E. Afișarea imaginilor

Matplotlib implementează, în sub-modulul ***pyplot***, funcția `imread` ce primește ca parametru calea spre o imagine și returnează o matrice ***NumPy*** cu eșantioanele imaginii și funcția `imshow` ce primește ca parametru o astfel de matrice și o afișează grafic.

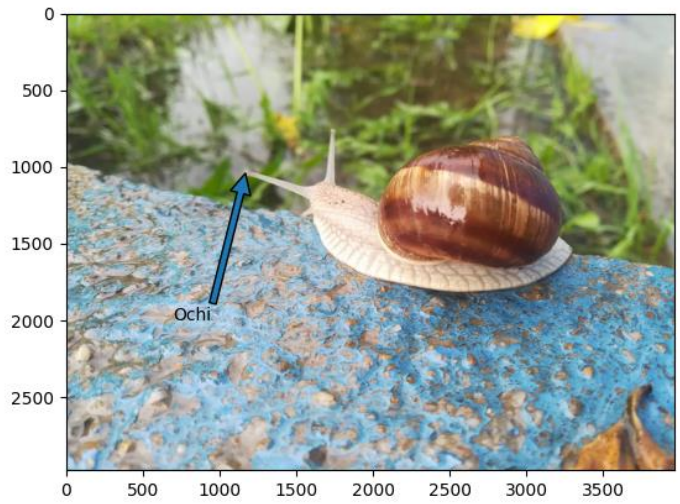
main.py

```
import numpy as np
import matplotlib.pyplot as plt

x = plt.imread("melc.jpg")
plt.imshow(x)
```

```
plt.annotate("Ochi", (1170, 1040), (700, 2000),
arrowprops={})
plt.show()
```

rezultat



F. Reprezentarea grafică a datelor statistice

Matplotlib oferă, în sub-modulul ***pyplot***, o serie de funcții pentru reprezentarea grafică a datelor statistice [6]. Una dintre aceste funcții este `plt.hist` care afișează histograma valorilor dintr-o matrice unidimensională. Histograma este un grafic ce reprezintă o interpretare statistică a unui set de valori prin indicarea numărului de valori care se află în anumite intervale de valori, numite clase. Astfel, histograma apare sub forma unui grafic cu bare vertical, fiecare bară reprezentând o clasă (un interval de valori reprezentat ca lățimea barei), iar înălțimea fiecărei bare reprezentând câte valori din întreg setul de date se află în clasa respectivă. Funcția `plt.hist` poate primi mai mulți parametri, dintre care:

- `x` – datele pentru care se dorește afișarea histogramei.
- `bins` – iterabil ce specifică limitele claselor (ex: lista [1, 3, 5] specifică două clase: clasa cu valori între 1 și 3 și clasa cu valori între 3 și 5); dacă lipsește, atunci ***Matplotlib*** va împărți intervalul de valori în 10 clase de dimensiune egală.

- `color` – culoarea barelor specificată ca în Tabelul 1; dacă lipsește, barele vor fi albastre.
- `edgecolor` – culoarea chenarelor barelor specificată ca în Tabelul 1; dacă lipsește, chenarul barelor va fi albastru.

Spre exemplu, dacă dorim să afișăm histograma filmelor și emisiunilor produse de Netflix anual după anul 2010, se va proceda astfel:

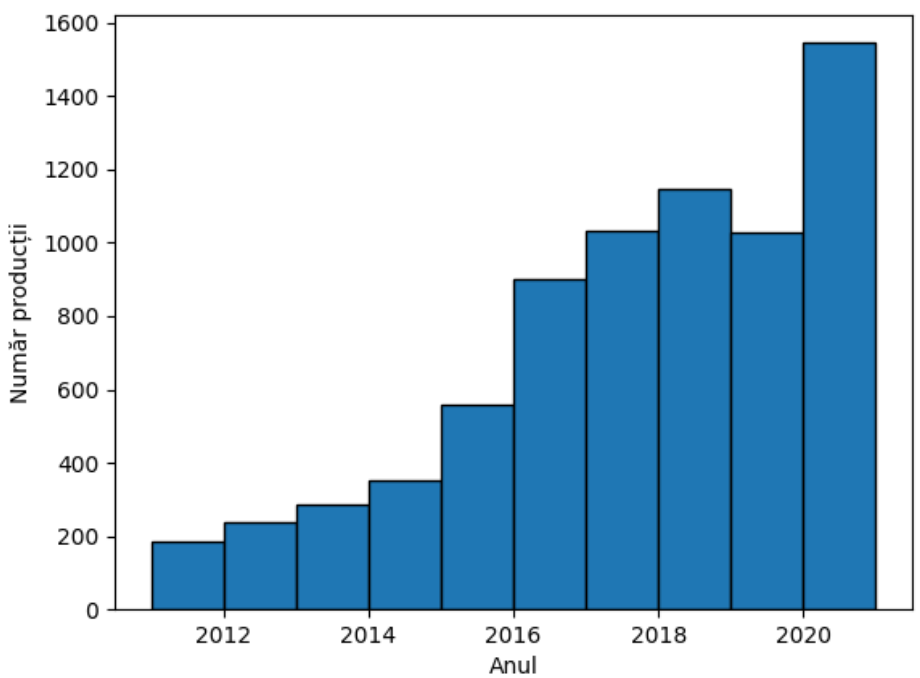
main.py

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

df = pd.read_csv("netflix_titles.csv").query("release_year > 2010")
edges = np.arange(df["release_year"].min(), df["release_year"].max()+1, 1)

plt.hist(df["release_year"], edges, edgecolor="k")
plt.xlabel("Anul")
plt.ylabel("Număr producții")
plt.show()
```

rezultat



O altă funcție utilă pentru reprezentarea datelor statistice este funcția `plt.boxplot`. Aceasta primește ca parametru o listă de iterabile și pentru fiecare iterabil, realizează, în aceeași figură, câte un grafic de tip ***boxplot***, grafic ce înglobează unul dintre cei mai importanți parametri statistici ai unei distribuții, oferind informații despre tendința centrală și forma distribuției. Figura următoare prezintă un astfel de grafic de tip ***boxplot*** în comparație cu o distribuție:

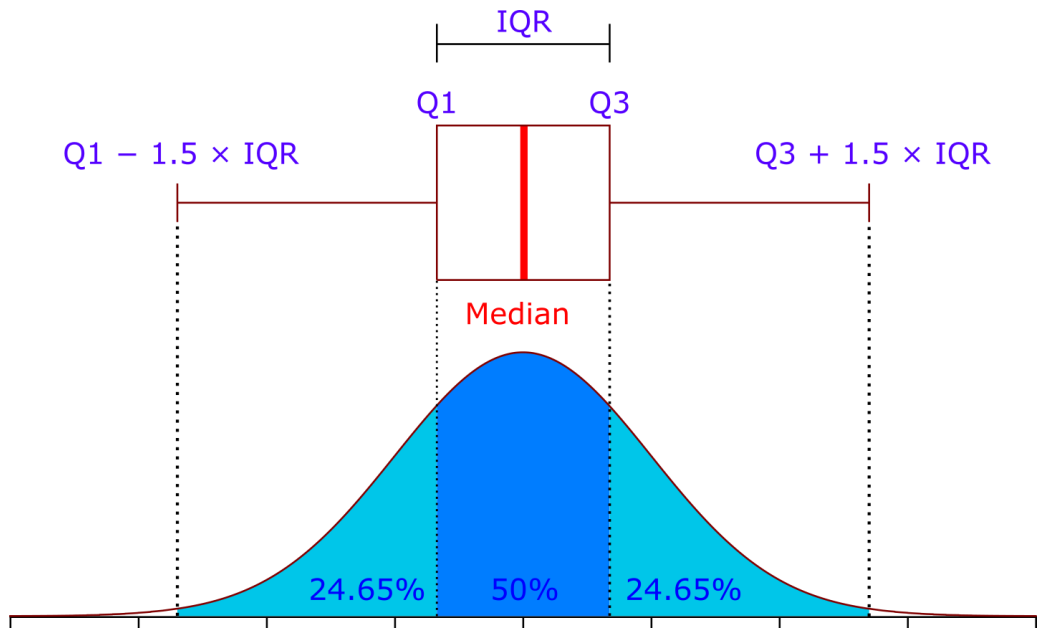


*Figura 2 – Exemplu **boxplot***

Exemplu:

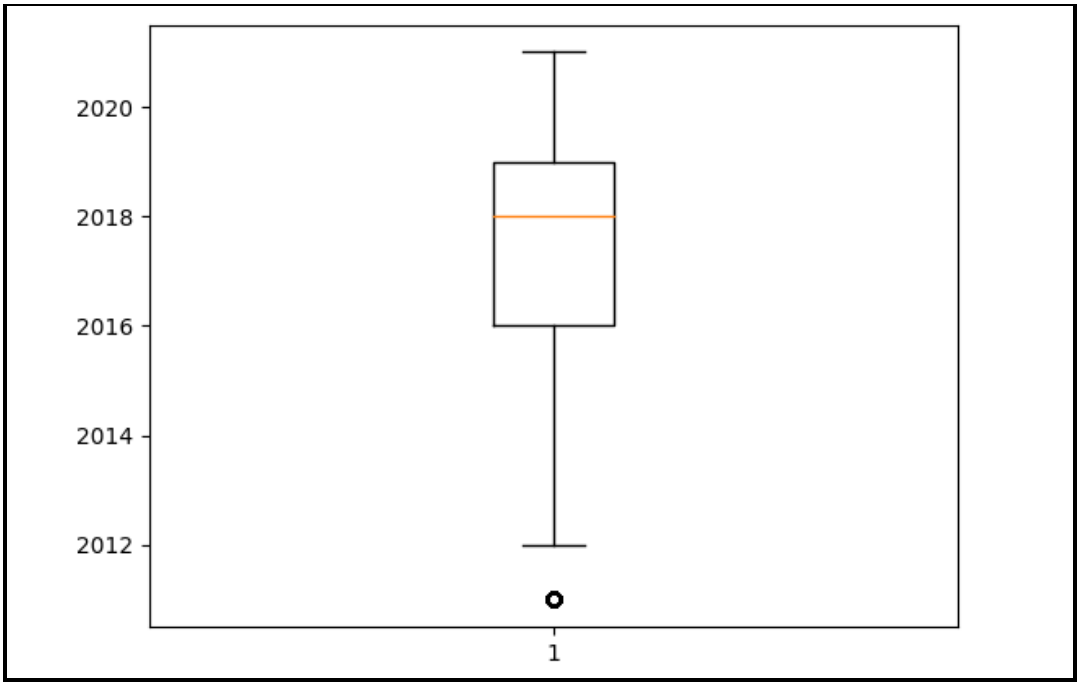
main.py

```
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_csv("netflix_titles.csv").query("release_year > 2010")

plt.boxplot(df["release_year"])
plt.show()
```

rezultat



Totuși, graficul de tip boxplot este util pentru a compara două sau mai multe distribuții. Astfel, dacă dorim să afișăm comparativ distribuția filmelor și distribuția emisiunilor produse de Netflix după anul 2010, va trebui să extragem într-o variabilă doar filmele și într-o altă variabilă doar emisiunile, apoi să creăm o listă cu cele două variabile pe care să o dăm funcției `plt.boxplot`. O variantă mai simplă ar fi utilizarea metodei `boxplot` pentru obiectul de tip ***DataFrame*** care adaugă funcției din ***Matplotlib*** posibilitatea grupării coloanelor dintr-un ***DataFrame***.

Exemplu:

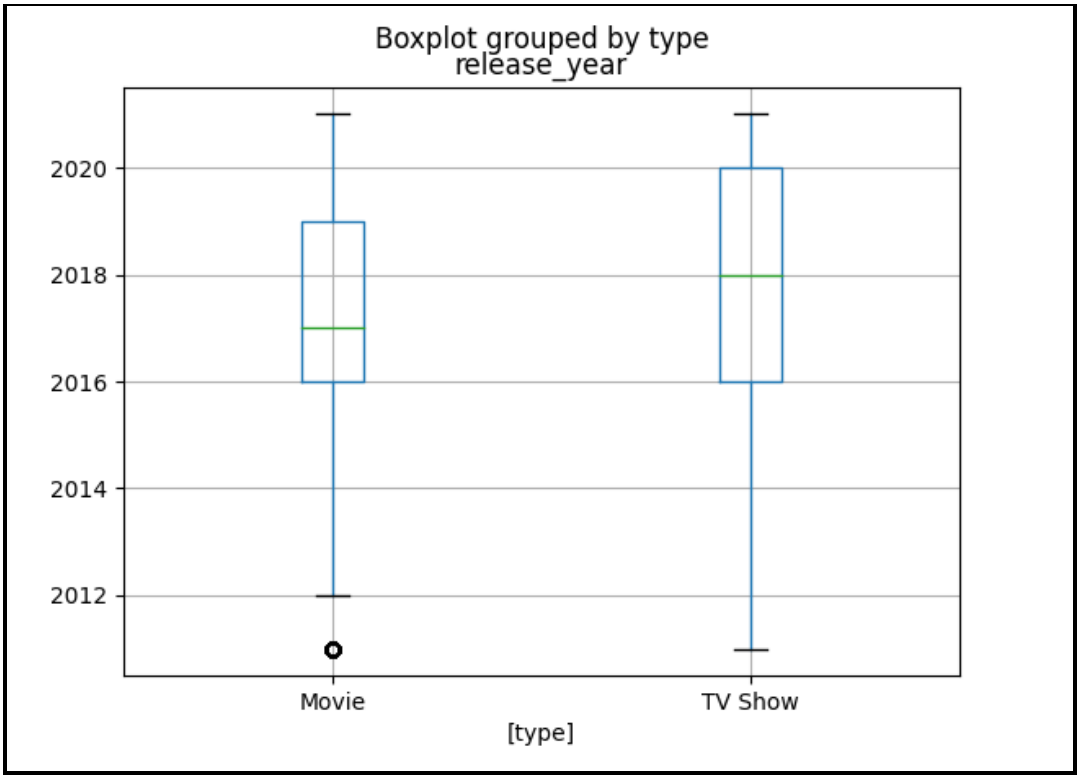
main.py

```
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_csv("netflix_titles.csv").query("release_year > 
2010")

df.boxplot(column=["release_year"], by=["type"])
plt.show()
```

rezultat



G. Cerințe

1. Pentru $x \in [-5 \quad 5]$, să se reprezinte pe același grafic (suprapuse) funcțiile:
   a) $y_1(x) = \frac{x+1}{x^2-2}$ cu linie roșie continuă
   b) $y_2(x) = x \cdot \log_2(x^2)$ cu linie verde întreruptă

2. Pentru $x \in [0 \quad 8 \cdot \pi]$, să se reprezinte în sub-ferestre diferite funcțiile:
   c) $y_3(x) = \sin(1.5 \cdot x)$
   d) $y_4(x) = \cos(x/2)$

3. Pornind de la ***DataFrame***-ul `df1` rezultat în urma rezolvării cerinței numărul 6 din lucrarea anterioară (Lucrarea 4), rezolvați:

a) Afișați grafic suprapus evoluția temperaturii minime, medie și maxime pentru stația „*Botoșani*”. Adăugați etichete axelor, titlu graficului și legendă. **Indiciu**: filtrați `df1` astfel încât să rămâneți doar cu valorile înregistrate la acea stație, apoi afișați grafic, pe rând, coloanele `TMIN`, `TMED` și `TMAX`, utilizând pentru axa `x` coloana `DATCLIM`.

b) Afișați grafic, sub formă de bare orizontale, temperatura minimă înregistrată de fiecare dintre stațiile meteo. Adăugați etichete axelor și titlu graficului. **Indiciu**: selectați din `df1` coloanele `CODST`, `NUME` și `TMIN`, grupați după coloana `CODST` și calculați minimul. Din rezultat utilizați coloanele `NUME` și `TMIN` pentru afișarea grafică.

c) Afișați un grafic de tip scatter cu altitudinea pe axa `x` și temperatura minimă înregistrată de fiecare stație pe axa `y`. Adăugați etichete axelor și titlu graficului. Adnotați fiecare punct al graficului cu numele stației. **Indiciu**: selectați din `df1` coloanele `CODST`, `NUME`, `ALT` și `TMIN`, grupați după coloana `CODST` și calculați minimul. Din rezultat utilizați coloanele `ALT` și `TMIN` pentru afișarea grafică. Treceți prin toate rândurile rezultatului și utilizați coloana `NUME` pentru adnotarea graficului.

d) Afișați grafic în 2 sub-ferestre histogramele temperaturilor medii și temperaturilor maxime înregistrate de stația „Vârfu Omu”. Adăugați etichete axelor și titlu graficului.

e) Afișați un grafic de tip boxplot pentru toate temperaturile medii înregistrate, grupate după numele stației meteorologice.

Lucrarea 6. Calcul simbolic. Librăria SymPy

A. Obiectivele lucrării

1. Familiarizarea cu funcționalitatea și sintaxa librăriei ***SymPy.***

B. Introducere

Utilizarea expresiilor matematice simbolice este uzuală în inginerie. Expresiile matematice simbolice sunt expresii ce sunt evaluate la nivel literar în raport cu una sau mai multe necunoscute. ***SymPy*** este o librărie ***Python*** utilizată pentru calcul simbolic ce definește o serie de funcționalități pentru crearea și lucrul cu obiecte simbolice. După instalarea acestei librării, aceasta se va importa dându-i și un alias, spre exemplu `sym`.

main.py

```
import sympy as sym
```

C. Definirea obiectelor simbolice

Atunci când vorbim despre obiecte simbolice ne referim fie la variabile simbolice, fie la expresii simbolice.

Definirea unei variabile simbolice se face utilizând funcția `sym.symbols` căreia îi dăm ca parametru un șir de caractere conținând numele variabilei simbolice [8].

main.py

```
import sympy as sym

x = sym.symbols('x')
print(x)
```

rezultat

```
x
```

Totuși, această funcție permite, cu un singur apel, definirea a mai multor variabile simbolice [8]. Exemple:

1. `x, y, z = sym.symbols('x y z')` – va defini 3 variabile simbolice `x`, `y` și `z`.
2. `x, y, z = sym.symbols('x:z')` – va defini 3 variabile simbolice `x`, `y` și `z`.
3. `a, b, c =s ym.symbols(':c')` – va defini 3 variabile simbolice `a`, `b` și `c`.
4. `x0, x1, x2 = sym.symbols('x:3')` – va defini 3 variabile simbolice `x0`, `x1` și `x2`.

O expresie simbolică se definește ca o expresie matematică ce conține variabile simbolice. Utilizarea operatorilor aritmetici uzuali (adunare, scădere, înmulțire, împărțire și ridicare la putere) cu variabile sau expresii simbolice este perfect validă, rezultatul fiind o expresie simbolică.

Exemplu:

main.py

```
import sympy as sym

x, y = sym.symbols('x y')

expr = 2 * x + y

print(expr)
print(expr - y)
print(expr * x)
```

rezultat

```
2*x + y
2*x
x*(2*x + y)
```

Firește, o expresie simbolică poate conține și funcții matematice specializate. Totuși, funcțiile definite de librăria ***math*** și de librăria ***NumPy*** sunt dedicate calculului numeric și nu pot primi ca parametru variabile sau expresii simbolice. Astfel, ***SymPy*** își definește propriile funcții și constante matematice cu aceeași denumire cu cele definite de librăria ***NumPy*** (exceptând funcția `abs` care devine `Abs`). Exemplu:

main.py

```
import sympy as sym

x = sym.symbols('x')
expr = sym.Abs(sym.exp(x) - 2 * sym.sin((x * sym.pi) / 6))
print(expr)
```

rezultat

```
Abs(exp(x) - 2*sin(pi*x/6))
```

O altă constantă matematică definită de SymPy este constanta `sym.oo` care simbolizează infinitul. Exemplu:

main.py

```
import sympy as sym

x = sym.symbols('x')

print(sym.oo)
print(x - sym.oo)
print(sym.oo / sym.oo)
```

rezultat

```
oo
x - oo
nan
```

SymPy oferă o serie de funcții pentru manipularea expresiilor simbolice, astfel încât acestea să aibă o formă convenabilă [8]. Cele mai uzuale dintre acestea sunt descrise în tabelul următor:

Tabelul 1 - Funcții pentru manipularea expresiilor

Funcția	**Descriere**
`sym.expand(expr)`	va rezolva parantezele expresiei și va simplifica termenii obținuți (exemplu: $(x+y)^2$ va deveni $x^2+2\cdot x\cdot y+y^2$)
`sym.simplify(expr)`	va încerca să simplifice cât se poate de mult o expresie; exemple: • $x^2+2\cdot x\cdot y+y^2$ va deveni $(x+y)^2$ • $\cos(x)\cdot\tan(x)$ va deveni $\sin(x)$ • $x^a\cdot x^b$ va deveni x^{a+b}

`sym.collect(expr, var)`	va colecta puterile comune ale termenului `var` din expresie; exemplu: $x^3 - x^2 \cdot z + 2 \cdot x^2 + x \cdot y + x - 3$, pentru `var = x` va deveni $x^3 + x^2 \cdot (2 - z) + x \cdot (y + 1) - 3$

O metodă utilă definită de ***SymPy*** pentru expresiile simbolice este metoda `subs(sym1,a)` ce va înlocui în expresie variabila simbolică `sym1` cu `a`, `a` putând fi fie o altă variabilă simbolică, fie o expresie simbolică, fie o valoare numerică.

Exemplu:

main.py

```
import sympy as sym

x, y = sym.symbols('x, y')

expr =  x ** 2 + y
print(expr)
print(expr.subs(x, x + 1))
print(expr.subs(x, 1))
```

rezultat

```
x**2 + y
y + (x + 1)**2
y + 1
```

O expresie simbolică poate fi transformată în funcție, utilizând funcția `sym.lambdify` ce primește ca parametri: lista de variabile simbolice ce vor reprezenta parametrii funcției rezultate și expresia simbolică. Astfel, expresia va putea fi evaluată pentru anumite valori ale variabilelor simbolice.

Exemplu:

main.py

```
import sympy as sym
import numpy as np

x, y = sym.symbols('x, y')

f1 = sym.lambdify([x, y], x ** 2 + x * y + y ** 2)
print(f1(1, 3))

f2 = sym.lambdify(x, 1 / (1 - sym.exp(-x)))
print(f2(np.arange(0.1, 1, 0.1)))
```

rezultat

```
13
[10.5083319  5.5166555  3.8582959  3.0332447  2.5414940
2.2163692
  1.9864338 1.81596622  1.68511775]
```

D. Rezolvarea ecuațiilor algebrice

Librăria ***SymPy*** permite rezolvarea ecuațiilor algebrice. Acestea pot avea una sau mai multe variabile simbolice, iar rezolvarea acestora presupune găsirea unei soluții pentru oricare variabilă simbolică în raport cu celelalte. Pentru a rezolva o ecuație algebrică se poate utiliza funcția `sym.solve` ce primește ca parametri: expresia simbolică ce descrie ecuația de forma `expresie = 0` și lista variabilelor simbolice a căror soluție se caută.

Spre exemplu, dacă dorim să aflăm valorile lui x ce satisfac ecuația $x^2 - y = 0$, vom proceda astfel:

main.py

```
import sympy as sym

x, y = sym.symbols('x, y')
print(sym.solve(x**2 - y, x))
```

rezultat

```
[-sqrt(y), sqrt(y)]
```

Această funcție poate fi folosită și pentru rezolvarea inecuațiilor, atât timp cât soluția acesteia are un interval clar definit și nu depinde de o altă variabilă simbolică. Exemplu:

main.py

```
import sympy as sym
x = sym.symbols('x')
print(sym.solve(x**2 - 1 > 0, x))
```

rezultat

```
((-oo < x) & (x < -1)) | ((1 < x) & (x < oo))
```

Funcția `sym.solve` poate primi ca prim parametru și o listă de ecuații, caz în care va căuta soluția ce satisface toate ecuațiile. Astfel, rezolvarea unui sistem de ecuații $\begin{cases} 2 \cdot x - 3 \cdot y = 5 \\ -x + 2 \cdot y = 8 \end{cases}$ se va face astfel:

main.py

```
import sympy as sym

x, y = sym.symbols('x, y')
print(sym.solve([2 * x - 3 * y - 5, -x + 2 * y - 8], [x, y]))
```

rezultat

```
{x: 34, y: 21}
```

În mod evident, lista poate conține atât ecuații, cât și inecuații. Spre exemplu, dacă se dorește determinarea lui $x \in [1, 5]$ pentru care $\sin(x) = \cos(x)$, se va proceda astfel:

main.py

```
import sympy as sym

x = sym.symbols('x')
print(sym.solve([sym.sin(x) - sym.cos(x), x >= 3, x <= 5],
x))
```

rezultat

```
(3 <= x) & (x <= 5) & (Eq(x, pi/4) | Eq(x, 5*pi/4))
```

E. Funcții de analiză matematică

Librăria ***SymPy*** permite rezolvarea problemelor de analiză matematică ce implică limite, derivate și integrale.

Calcularea limitelor în ***SymPy*** se face utilizând funcția `sym.limit(expr,var,val)` unde `expr` este expresia simbolică pentru care dorim să aflăm limita atunci când variabila `var` tinde la valoarea `val`.

Spre exemplu, dacă dorim să calculăm $\lim\limits_{x \to \infty} \frac{1}{x}$:

main.py

```
import sympy as sym

x = sym.symbols('x')

print(sym.limit(1 / x, x, sym.oo))
```

rezultat

```
0
```

Derivarea funcțiilor simbolice se face cu funcția `sym.diff(expr, var)` unde `expr` este expresia simbolică pentru care dorim să aflăm derivata în raport cu variabila `var`. Spre exemplu, pentru $f(x,y) = (2 \cdot x + y)^2$ calcularea derivatelor $\frac{\partial f}{\partial x}$, $\frac{\partial f}{\partial y}$ și $\frac{\partial^2 f}{\partial x \partial y}$ se face astfel:

main.py

```
import sympy as sym

x, y = sym.symbols('x, y')
f = (2 * x + y) ** 2

print(sym.diff(f, x))
print(sym.diff(f, y))
print(sym.diff(sym.diff(f, x), y))
```

rezultat

```
8*x + 4*y
4*x + 2*y
4
```

Dacă se dorește calcularea derivatei de ordin mai mare, se poate adăuga funcției un al treilea parametru ce specifică ordinul derivatei:

main.py

```
import sympy as sym

x, y = sym.symbols('x, y')
f = (2 * x + y) ** 2

print(sym.diff(f, x, 2))
```

rezultat

```
8
```

Integrarea funcțiilor simbolice se face cu funcția `sym.integrate(expr, var)` unde `expr` este expresia simbolică pe care dorim să o integrăm în raport cu variabila `var`. Spre exemplu, pentru $f(x,y) = (2 \cdot x + y)^2$ calcularea integralelor $\int f dx$ și $\int f dy$ se face astfel:

main.py

```
import sympy as sym

x, y = sym.symbols('x, y')
f = (2 * x + y) ** 2

print(sym.integrate(f, x))
print(sym.integrate(f, y))
```

rezultat

```
4*x**3/3 + 2*x**2*y + x*y**2
4*x**2*y + 2*x*y**2 + y**3/3
```

Bineînțeles, se pot calcula și integrale definite, modificând cel de-al doilea parametru într-un tuplu `(var, a, b)`, unde `var` este variabila în raport cu care se calculează integrala, iar a și b sunt limitele integralei. Astfel, pentru funcția anterioară se poate calcula $\int_a^b f dx$ și $\int_{-1}^{1} f dx$ astfel:

main.py

```
import sympy as sym

x, y, a, b = sym.symbols('x, y, a, b')
f = (2 * x + y) ** 2

print(sym.integrate(f, (x, a, b)))
print(sym.integrate(f, (x, -1, 1)))
```

rezultat

```
-4*a**3/3 - 2*a**2*y - a*y**2 + 4*b**3/3 + 2*b**2*y + b*y**2
2*y**2 + 8/3
```

Librăria ***SymPy*** oferă și posibilitatea rezolvării ecuațiilor diferențiale ordinare. Spre exemplu, dorim să rezolvăm ecuația diferențială

$$f(x) + \frac{d^2 f}{dx^2} = 0$$

Pentru asta, va trebui să definim `f` explicit ca funcție simbolică (`f=sym.symbols('f',cls=sym.Function)`), iar apoi să utilizăm funcția `sym.dsolve` în mod asemănător cu funcția `sym.solve`:

main.py

```
import sympy as sym

x = sym.symbols('x')
f = sym.symbols('f', cls=sym.Function)

print(sym.dsolve(f(x) + sym.diff(f(x), x, 2), f(x)))
```

rezultat

```
Eq(f(x), C1*sin(x) + C2*cos(x))
```

F. Cerințe

1. Definiți expresia simbolică $(x^3 - y^3) \cdot (1 - x^2 \cdot y^2) \cdot (1 - x \cdot y^2)$. Utilizați funcționalitatea ***SimPy*** pentru a desface parantezele.

2. Definiți expresia simbolică $8 \cdot x^3 \cdot y + 4 \cdot x^2 \cdot y^2 + 6 \cdot x \cdot y^4$. Utilizați funcționalitatea ***SimPy*** pentru a simplifica expresia.

3. Definiți expresiile simbolice $f = \cos x^2$, $g = \sin \sqrt{x}$, $h = e^{-\frac{x^2}{\pi}}$.
   a) calculați derivatele $\frac{df}{dx}, \frac{d^2f}{dx^2}, \frac{dg}{dx}, \frac{d^2g}{dx^2}, \frac{dh}{dx}, \frac{d^2h}{dx^2}$.
   b) rezolvați ecuația $\frac{df}{dx} = 0$.
   c) rezolvați ecuația $\frac{dg}{dx} = 0$.
   d) rezolvați ecuația $\frac{dh}{dx} = 0$.
   e) transformați expresia f în funcție ***Python***, utilizând metoda `lambdify`, și afișați grafic funcția pentru $x \in [-3, 3]$.
   f) transformați expresia g în funcție ***Python***, utilizând metoda `lambdify`, și afișați grafic funcția pentru $x \in [0, 100]$.
   g) transformați expresia h în funcție ***Python***, utilizând metoda `lambdify`, și afișați grafic funcția pentru $x \in [-5, 5]$.

4. Fie funcția $f(x) = 3x + a$. Determinați valoarea a, știind că $f(x) + f(1-x) = 7$. **Indiciu**: definiți $f(x)$ ca expresie simbolică, obțineți $f(1-x)$ prin substituirea lui x cu $1-x$ în expresie, iar apoi definiți expresia ce descrie ecuația ce trebuie rezolvată. Rezolvați ecuația pentru a.

5. Fie funcția $f(x) = \frac{1}{x-2} + \ln\frac{x-1}{x}$. Calculați $\frac{df}{dx}$ și $\lim_{x\to\infty} f(x)$.

6. Fie funcția $f(x) = \frac{x}{\sqrt{x^3+1}}$. Calculați:

   a) $\int f^2(x)dx$
   b) $\int_a^b f^2(x)dx$
   c) $\int_0^1 f^2(x)dx$
   d) $\lim_{n\to\infty} \int_0^n f^2(x)dx$

7. Calculați $\lim_{x\to 0} \frac{\sin x}{x}$

8. Rezolvați ecuația diferențială Bernoulli:

$$x \cdot \frac{df(x)}{dx} + f(x) - f(x)^2 = 0$$

Lucrarea 7. Calcul numeric. Librăria SciPy

A. Obiectivele lucrării

1. Familiarizarea cu funcționalitatea și sintaxa librăriei ***SciPy.***

B. Introducere

Atunci când problemele matematice (determinarea soluțiilor, optimizarea, integrarea etc.) sunt complexe, rezolvarea lor pe cale analitică nu este fezabilă. Pentru rezolvarea problemelor de acest tip se utilizează metode numerice. ***SciPy*** este o librărie ***Python*** ce oferă o multitudine de metode numerice de rezolvare a problemelor matematice de tip optimizări, integrări, interpolări, ecuații diferențiale, statistică, analiză frecvențială (Transformata Fourier), procesare de semnale etc. [9]. După instalarea acestei librării, aceasta se va importa și utiliza în mod asemănător altor librării.

main.py

```
import scipy
```

În continuare, se vor studia o serie de funcționalități de bază ale acestei librării, și anume: rezolvarea ecuațiilor, optimizarea, interpolarea, dar și integrarea numerică.

C. Rezolvarea ecuațiilor

SciPy permite determinarea rădăcinilor reale ale ecuațiilor liniare sau neliniare prin utilizarea funcției `root` din sub-modulul `optimize` [9]. Aceasta primește doi parametri: funcția f ce definește ecuația ce se dorește rezolvată, și un punct inițial x_0 din care se face căutarea lui $x \in \mathbb{R}\ a.\hat{\imath}.\ f(x) = 0$.

Pentru exemplificare, vom considera că se dorește aflarea unei rădăcini a ecuației

$$\cos x + e^{-2 \cdot x} = 0$$

pornind din punctul inițial $x_0 = 0$. Pentru acest lucru, se va defini în ***Python*** o funcție ce va implementa ecuația ce se dorește rezolvată, iar apoi, funcția implementată, împreună cu punctul inițial ales vor fi utilizați ca parametri pentru funcția `root`. Funcția va returna un obiect ce conține o serie de proprietăți. Dintre aceste proprietăți, proprietatea `x` va reprezenta soluția găsită.

main.py

```
import scipy
import numpy as np

def func1(x):
    return np.cos(x) + np.exp(-2 * x)

sol = scipy.optimize.root(func1, 0)
print(sol.x)
```

rezultat

```
[1.61070563]
```

Dacă ecuația are o singură rădăcină reală, atunci alegerea punctului inițial nu va avea efect asupra valorii calculate, însă poate avea efect asupra timpului de găsire a soluției. Dacă ecuația are mai multe rădăcini reale, atunci valoarea rezultată va depinde de punctul inițial ales.

Bineînțeles, funcția `root` permite rezolvarea ecuațiilor cu mai multe variabile, sau chiar a seturilor de ecuații. Spre exemplu, dorim să aflăm rădăcinile sistemului de ecuații neliniare: $\begin{cases} x \cdot \sin(y) = 3 \\ x \cdot y - x = 7 \end{cases}$ pornind din punctul inițial definit de $x_0 = 1$ și $y_0 = 1$.

main.py

```
import scipy
import numpy as np

def func2(x):
    return [
        x[0] * np.sin(x[1]) - 3,
        x[0] * x[1] - x[0]  - 7
    ]

sol = scipy.optimize.root(func2, [1, 1])
print(sol.x)
```

rezultat

```
[4.78279855 2.46357827]
```

D. Rezolvarea problemelor de optimizare

Rezolvarea problemelor de optimizare presupune determinarea valorii reale x pentru care valoarea funcției $f(x)$ este minimă / maximă. Funcțiile definite de ***SciPy*** permit determinarea minimului, și anume rezolvarea problemelor de tip $\min_x f(x)$. Dacă se dorește determinarea maximului unei funcții, va trebui să transformăm funcția într-o problemă de minimizare prin inversarea semnului funcției astfel: $\max_x f(x) = \min_x(-f(x))$. ***SciPy*** permite rezolvarea problemelor de minimizare prin utilizarea funcției `minimize` din sub-modulul `optimize` [9]. Aceasta primește doi parametri: funcția f, ce se dorește minimizată, și un punct inițial x_0 din care se face căutarea soluției.

Pentru exemplificare, se va considera problema construirii unei conducte de la o platformă petrolieră offshore, aflată la o distanță $H = 10\ km$ de linia țărmului, la o rafinărie aflată la o distanță $L = 50\ km$ de-a lungul țărmului. Costul de construcție al conductei este de $C_{ocean} = 20\ M\$/km$ pentru conducta construită sub apă și $C_{țărm} = 10\ M\$/km$ pentru conducta construită pe uscat. Conducta va urma o traiectorie dreaptă de la platformă la punctul de contact x aflat pe țărm între 0 și L, de unde va continua de-a lungul țărmului până la rafinărie. Se dorește determinarea punctului de contact x astfel încât costul de construire al conductei să fie minim.

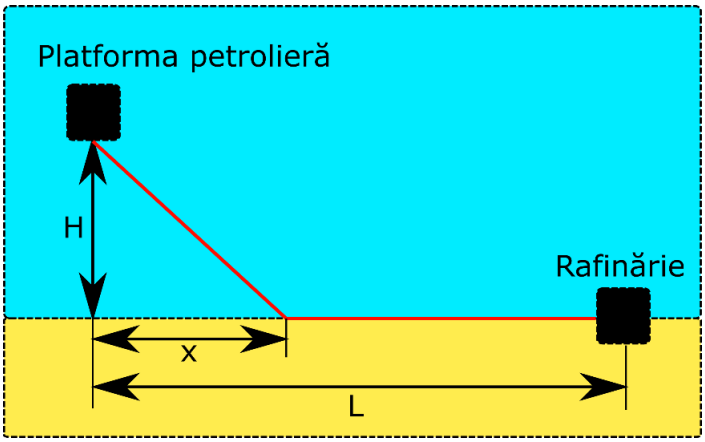


Figura 1 – Problema construirii unei conducte petroliere

Costul total de construcție al conductei va fi $f(x) = C_{ocean} \cdot \sqrt{x^2 + H^2} + C_{țărm} \cdot (L - x)$, iar problema ce trebuie rezolvată va fi $\min_x f(x)$. Rezolvarea acestei probleme în Python se face astfel:

main.py

```
import scipy
import numpy as np

def func3(x):
    H, L = 10, 50
    C1, C2 = 20, 10

    return C1 * np.sqrt(x ** 2 + H ** 2) + C2 * (L - x)

sol = scipy.optimize.minimize(func3, 0)
print(f"Costul minim {sol.fun} obținut pentru x =
{sol.x[0]}")
```

rezultat

```
Costul minim 673.205080756892 obținut pentru x =
5.773505257974372
```

Uneori, la rezolvarea problemelor de optimizare există anumite constrângeri în legătură cu parametrii funcției ce se dorește minimizată. Spre exemplu, dorim să construim un gard pentru a împrejmui un câmp dreptunghiular. Costul construirii părților laterale este de 100 RON/m, în timp ce costul construirii părții superioare este de 70 RON/m iar a părții inferioare de 20 RON/m, conform imaginii:

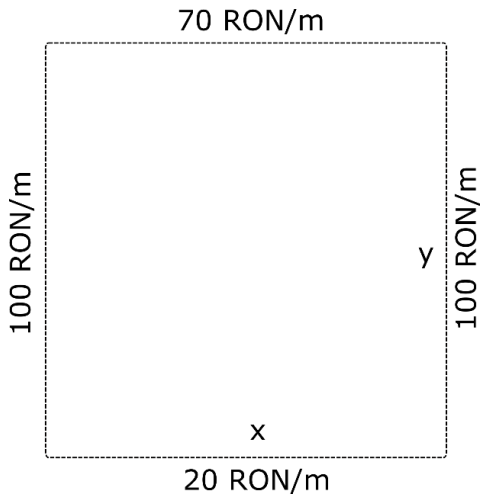


Figura 2 – Problema construirii unui gard

Se dorește determinarea ariei maxime ce poate fi împrejmuită cu un buget de 7000 RON. Evident, primul lucru pe care trebuie să-l facem este să transformăm problema în una de minimizare prin inversarea semnului $\max_{x,y} x \cdot y = \min_{x,y}(-x \cdot y)$. Totuși, în rezolvarea problemei, trebuie să ținem cont și de faptul că avem un buget maxim, în caz contrar soluția obținută va tinde la infinit. Astfel, va trebui adăugată o constrângere de forma: $100 \cdot y + 20 \cdot x + 100 \cdot y + 70 \cdot x = 90 \cdot x + 200 \cdot y \leq 7000$. Aceasta se poate scrie în formă matriceală, definind atât limita inferioară cât și cea superioară, astfel:

$$[0] \leq [90 \quad 200] \cdot \begin{bmatrix} x \\ y \end{bmatrix} \leq [7000]$$

Utilizând forma matriceală, constrângerea / constrângerile pot fi definite în ***SciPy*** utilizând funcția `LinearConstraint` din sub-modulul `optimize`, variabila rezultată putând fi dată ca parametru `contraints` al funcției `minimize`:

main.py

```
import scipy

def func4(x):
    return - x[0] * x[1]

constraint=scipy.optimize.LinearConstraint([90,
200],[0],[7000])
sol = scipy.optimize.minimize(func4, [1,
1],constraints=[constraint])
print(f"Aria maximă: {-sol.fun} ({sol.x[0]}, {sol.x[1]})")
```

rezultat

```
Aria maximă: 680.5556457883094 (38.8888876167074,
17.50000289275258)
```

E. Rezolvarea problemelor de interpolare

Rezolvarea problemelor de interpolare presupune determinarea valorilor unei funcții necunoscute, pe baza unui set de valori cunoscut apriori. Spre exemplu, fie funcția $f(x)$ necunoscută, pentru care avem disponibile doar valorile:

Tabelul 1 – Valori disponibile pentru funcția f(x)

x	f(x)
0	1.00
1	0.99
2	0.90
3	0.54
4	-0.20
5	-0.93
6	-0.65
7	0.67
8	0.67
9	-0.91
10	0.12

Astfel, observăm că nu știm valoarea funcției pentru toate valorile lui x (spre exemplu, nu știm valoarea lui $f(2.44)$). Pentru aceasta este necesară aplicarea unei metode de interpolare pentru a estima valoarea funcției în punctul dorit. În continuare, se vor studia două metode de interpolare:

1. Interpolarea liniară, unde valorile necunoscute ale funcției se determină prin unirea cu linii a tuturor valorilor cunoscute ale funcției. Pentru interpolare liniară, se va folosi funcția `interp` din librăria ***NumPy*** [3]. Aceasta primește 3 parametri: valoarea / valorile lui x pentru care se dorește estimarea lui $f(x)$, valorile lui x pentru care cunoaștem deja valorile lui $f(x)$ și valorile cunoscute ale lui $f(x)$.
2. Interpolarea polinomială pe porțiuni, unde valorile necunoscute ale funcției se determină utilizând o serie de ecuații polinomiale cubice ce trec prin punctele cunoscute. Pentru acest tip de interpolare se va folosi funcția `CubicSpline` din sub-modulul `interpolate` al librăriei ***SciPy*** [9]. Aceasta primește doi parametri: valorile lui x pentru care cunoaștem deja valorile lui $f(x)$ și valorile cunoscute ale lui $f(x)$ și returnează o funcție ce poate fi utilizată pentru a evalua alte puncte.

Exemplul următor estimează valoarea lui $f(2.44)$ pentru funcția $f(x)$ pentru care cunoaștem doar valorile definite de Tabelul 1.

main.py

```
import scipy
import numpy as np

# valorile cunoscute
xc = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
fc = [1.00,0.99,0.90,0.54,-0.20,-0.93,-0.65,0.67,0.67,-
0.91,0.12]

# estimare f(2.44)
fe1  = np.interp(2.44, xc, fc)
func = scipy.interpolate.CubicSpline(xc, fc)
fe2  = func(2.44)

print(f"Interpolare liniară: f(2.44) = {fe1}")
print(f"Interpolare polinomială: f(2.44) = {fe2}")
```

rezultat

```
Interpolare liniară: f(2.44) = 0.7416
Interpolare polinomială: f(2.44) = 0.7863397307216496
```

Bineînțeles, ambele metode se pot utiliza pentru a estima valoarea funcției în mai multe puncte, oferind posibilitatea de a reprezenta grafic o estimare a funcției:

main.py

```
import scipy
import numpy as np
import matplotlib.pyplot as plt

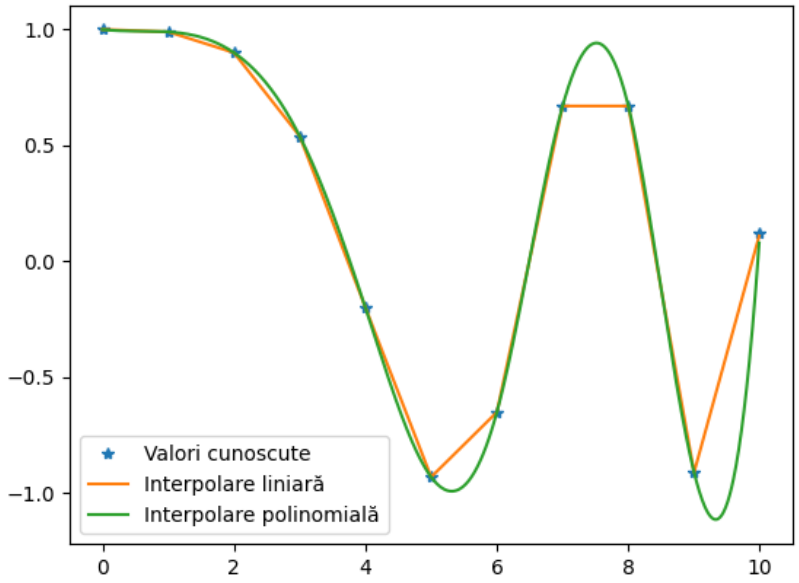
# valorile cunoscute
xc = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
fc = [1.00, 0.99, 0.90, 0.54, -0.20, -0.93,
      -0.65, 0.67, 0.67, -0.91, 0.12]

x = np.arange(0, 10, 0.01)

# estimare f(2.44)
fe1  = np.interp(x, xc, fc)
func = scipy.interpolate.CubicSpline(xc, fc)
fe2  = func(x)

plt.plot(xc, fc, '*')
plt.plot(x, fe1, x, fe2)
plt.legend(["Valori cunoscute",
            "Interpolare liniară",
            "Interpolare polinomială"])
plt.show()
```

rezultat



F. Integrarea numerică

Integrarea numerică presupune fie determinarea valorii unei integrale definite, fie rezolvarea unei ecuații / sistem de ecuații diferențiale. Pentru determinarea valorii unei integrale definite a unei funcții se va utiliza funcția `quad` din sub-modulul `integrate` al librăriei ***SciPy*** [9]. Această funcție primește ca parametri funcția ce se dorește integrată, precum și cele două limite ale integralei. Spre exemplu, se dorește calcularea $\int_0^1 f(x)dx$, unde $f(x) = \frac{x}{\sqrt{x^2+1}}$. În Python, se va proceda astfel:

main.py

```
import scipy
import numpy as np

def func5(x):
    return x / np.sqrt(x ** 2 + 1)

print(scipy.integrate.quad(func5, 0, 1))
```

rezultat

```
(0.41421356237309503, 4.598694340586185e-15)
```

Cele două valori rezultate reprezintă estimarea valorii integralei definite și eroarea maximă.

Așa cum a fost specificat mai devreme, este posibilă și rezolvarea numerică a ecuațiilor diferențiale de ordinul întâi de forma:

$$\begin{cases} \frac{dx}{dt} = f(t, x) \\ x(t_0) = x_0 \end{cases}$$

Rezolvarea se face utilizând, din același sub-modul, funcția `solve_ivp` ce implementează metoda *Runge-Kutta*. Aceasta primește ca parametri: funcția $f(t, x)$, un tuplu ce specifică limitele intervalului de timp și punctul inițial și va returna un obiect ce va avea printre proprietăți doi vectori, `t` și `y`, `y` reprezentând valorile obținute în urma integrării, la momentele de timp specificate de `t`.

Spre exemplu, considerând un rezervor cu suprafața de $A = 3\ m^2$ în care intră lichid cu un debit $Q_{in} = 2\ m^3/h$ și din care iese lichid cu un debit proporțional cu nivelul de lichid din acesta $Q_{out} = K \cdot h$, unde h reprezintă înălțimea nivelului de apă din rezervor în metri, iar $K = 0.5$ este o constantă. Dinamica nivelului de lichid din rezervor va fi dată de:

$$\frac{dh}{dt} = \frac{1}{A} \cdot (Q_{in} - Q_{out}) = \frac{1}{A} \cdot (Q_{in} - K \cdot h)$$

Se dorește determinarea evoluției nivelului de lichid din rezervor pentru următoarele 50 de ore, plecând din punctul inițial $h_0 = 0$ (rezervor gol).

main.py

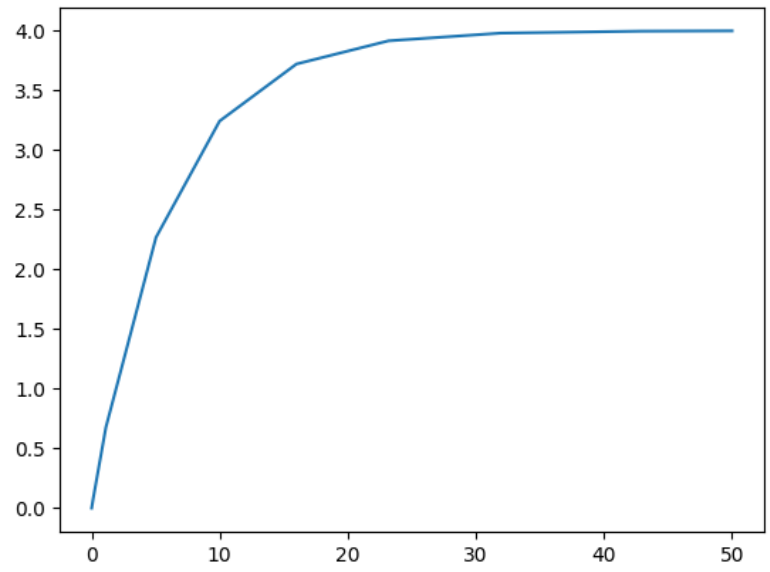
```
import scipy
import matplotlib.pyplot as plt

def func6(t, h):
    return 1 / 3 * (2 - 0.5 * h)

sol = scipy.integrate.solve_ivp(func6, (0, 50), [0])

plt.plot(sol.t, sol.y[0])
plt.show()
```

rezultat



Pentru ecuațiile diferențiale de ordin superior este necesară transformarea acestora într-un sistem de ecuații diferențiale de ordin întâi. Astfel, pentru ecuația diferențială de ordin n

$$\frac{d^n x}{dt^n} = f\left(t, x, \frac{dx}{dt}, \frac{d^2x}{dt^2}, \dots, \frac{d^{n-1}x}{dt^{n-1}}\right)$$

se vor nota:

$$z_i(t) = \frac{d^{i-1}x}{dt^{i-1}}$$

obținându-se sistemul de ecuații diferențiale de ordin întâi:

$$\begin{cases} \frac{dz_1}{dt} = z_2 \\ \frac{dz_2}{dt} = z_3 \\ \dots \\ \frac{dz_n}{dt} = f(t, z_1, z_2 \dots, z_n) \end{cases}$$

Spre exemplu, fie circuitul RLC din figura următoare:

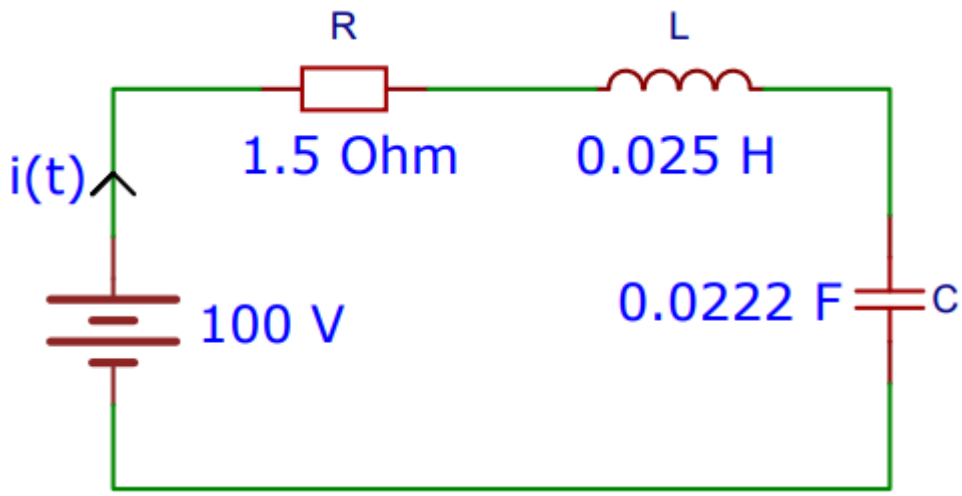


Figura 3 – Circuit RLC

Se dorește determinarea dinamicii curentului $i(t)$ prin circuit pentru 0.5 secunde după alimentarea acestuia. Se va porni de la cea de-a doua lege a lui Kirchhoff, și anume faptul că suma căderilor de tensiune pe rezistență, inductor și condensator este egală cu tensiunea de alimentare ($V_{in} = V_R + V_L + V_C$). Introducând în această ecuație formulele ce specifică căderile de tensiune în funcție de curent ($V_R = R \cdot i(t)$, $V_L = L \cdot \frac{di}{dt}$ și $V_C = \frac{1}{C} \cdot \int i(t)dt$) se va obține:

$$V_{in} = R \cdot i(t) + L \cdot \frac{di}{dt} + \frac{1}{C} \cdot \int i(t)dt$$

Astfel:

$$\frac{di}{dt} = \frac{1}{L} \cdot \left(V_{in} - R \cdot i(t) - \frac{1}{C} \cdot \int i(t)dt \right)$$

În continuare, se va transforma ecuația diferențială de ordinul doi, într-un sistem de ecuații diferențiale de ordinul întâi, făcând notațiile $z_1 = \int i(t)dt$ și $z_2 = i(t)$:

$$\begin{cases} \frac{dz_1}{dt} = z_2 \\ \frac{dz_2}{dt} = \frac{1}{L} \cdot \left(V_{in} - R \cdot z_2 - \frac{1}{C} \cdot z_1 \right) \end{cases}$$

Având sistemul de ecuații diferențiale de ordinul întâi, problema se poate rezolva în ***Python***:

main.py

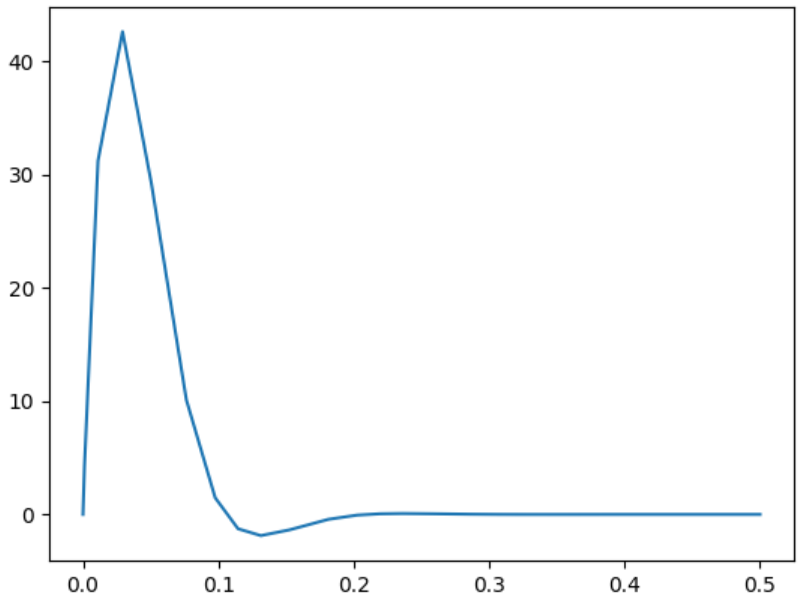
```
import matplotlib.pyplot as plt
import scipy

def func7(t, z):
    Vin, R, L, C = 100, 1.5, 0.025, 0.0222
    return [
        z[1],
        1 / L * (Vin - R * z[1] - 1 / C * z[0])
    ]

sol = scipy.integrate.solve_ivp(func7, (0, 0.5), [0, 0])

plt.plot(sol.t, sol.y[1])
plt.show()
```

rezultat



G. Cerințe

1. Rezolvați ecuațiile:
   a. $x - e^{-x} = 0$
   b. $2 \cdot x + \cos(x) = 0$

2. Rezolvați următoarele probleme de optimizare:
   a. Un producător a determinat că pentru a produce x componente zilnic, va avea un cost de producție $f(x) = 0.001 \cdot x^2 - 0.07 \cdot x + 84 + \frac{4000}{x}$. Determinați numărul optim de componente care ar minimiza costul zilnic de producție.

   b. Dintre toate dreptunghiurile cu perimetrul de 100, care are cea mai mare arie?

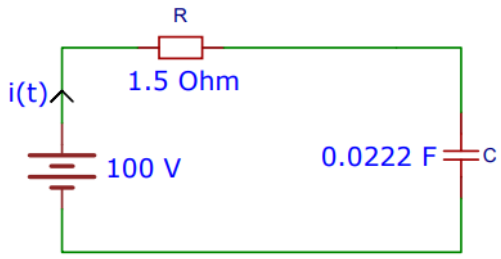
3. Fie următoarele valori cunoscute ale funcției $f(x)$:

x	f(x)
0	1.00
1	0.83
2	1.95
3	3.55
4	4.39
5	3.97
6	2.96
7	2.61
8	3.61
9	5.42
10	6.82

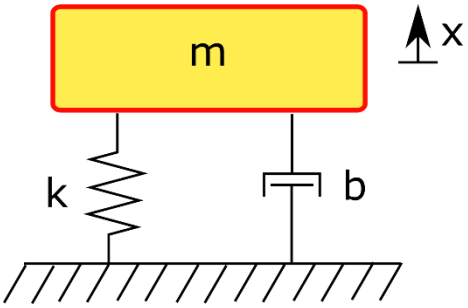
   a. Estimați valoarea lui $f(8.25)$ utilizând cele două metode de interpolare prezentate.
   b. Afișați grafic estimarea funcției $f(x)$ pentru $x \in [0, 10]$ cu pas de 0.01, utilizând cele două metode de interpolare prezentate. Comparați cu rezultatul afișării grafice a funcției $f_2(x) = 2 \cdot e^{x/10} - \cos(x) - \sin(x)$.

4. Calculați $\int_5^{10} \left(\frac{1}{x-2} + \ln\frac{x-1}{x}\right) dx$.

5. Se dă circuitul RC din figură. Să se determine grafic dinamica curentului la închiderea circuitului.



6. Fie sistemul mecanic compus dintr-un obiect de masă $m = 5\ kg$ susținut de o suprafață solidă printr-un arc cu constanta elastică $k = 800\ N/m$ și un amortizor cu un coeficient de amortizare $b = 20\ Ns/m$, conform imaginii de mai jos.



Determinați evoluțiile poziției și vitezei masei din poziția inițială $x_0 = 0.5\ m$ și viteza inițială $v_0 = 0\ m/s$ pe o perioadă de timp de 10 secunde, știind că dinamica sistemului este dată de ecuația diferențială de ordin doi:

$$m \cdot \frac{d^2x}{dt^2} + b \cdot \frac{dx}{dt} + k \cdot x(t) = 0$$

Lucrarea 8. Funcționalități avansate ale limbajului Python

A. Obiectivele lucrării

1. Familiarizarea cu o serie de funcționalități avansate ale limbajului ***Python***:
    a. operatorul ternar.
    b. operatorii `all` și `any`.
    c. decoratori.
    d. generatoare.
    e. funcții lambda.

B. Operatori avansați în Python

Operatorul ternar din ***Python*** este un operator condițional care permite evaluarea unei expresii pe baza unei condiții. Aceasta poate fi scrisă sub forma unei expresii condiționale cu trei operațiuni, cunoscută sub numele de operator ternar. Valoarea rezultată poate fi atribuită unei variabile sau utilizată direct într-o altă expresie. Structura generală a operatorului ternar este: `variabila = expresie1 if conditie else expresie2`, unde `conditie` este condiția ce trebuie verificată, `expresie1` este rezultatul operației dacă condiția este adevărată, iar `expresie2` este rezultatul operației dacă condiția este falsă [1].

Spre exemplu, având o variabilă `y`, se dorește să se atribuie variabilei `x` valoarea 1 dacă `y` este mai mare decât 5 și valoarea 0 în caz contrar. Firește, se poate utiliza structura condițională, dar operatorul ternar permite o scriere mai compactă:

main.py

```
# rezolvare utilizând structura condițională
if y > 5:
    x = 1
else:
    x = 0

# rezolvare utilizând operatorul ternar
x = 1 if y > 5 else 0
```

Aici, condiția este "`y > 5`", iar expresia evaluată dacă condiția este adevărată este "`1`", iar expresia evaluată dacă condiția este falsă este "`0`". Operatorul ternar este folosit frecvent în ***Python*** pentru a scrie un cod mai compact și mai eficient, în special în cazul în care o condiție este folosită pentru a decide între două valori posibile pentru o variabilă. De asemenea, poate fi folosit pentru a atribui valori într-un bloc de cod, astfel încât nu este nevoie să se scrie mai multe declarații de atribuire. În plus, operatorul ternar poate fi utilizat și cu mai multe condiții, cu ajutorul logicii booleene.

În general, operatorul ternar poate fi un instrument util pentru a simplifica și a face mai eficient codul în ***Python***, dar trebuie utilizat cu precauție pentru a nu afecta negativ ușurința de citire și de înțelegere a codului.

Operatorii `all` și `any` sunt operatori logici în Python care permit evaluarea unei liste de expresii și returnarea unui singur rezultat. Acești operatori sunt utilizați cu preponderență în construcții care necesită verificarea tuturor sau a uneia dintre expresii pentru a decide rezultatul final.

Operatorul `all` este utilizat pentru a verifica dacă toate expresiile dintr-o listă sunt evaluate ca adevărate. Dacă toate expresiile sunt adevărate, atunci rezultatul întregii expresii este `True`, altfel este `False`. Sintaxa operatorului `all` este următoarea: `all(iterabil)` unde `iterabil` este o listă, un tuplu sau o altă structură de date iterabilă ce conține expresiile ce trebuie verificate. De exemplu, presupunând o listă de numere, se dorește să se verifice dacă toate numerele din listă sunt mai mari decât 0. Firește, se poate utiliza o structură repetitivă, dar operatorul `all` permite o scriere mai compactă:

main.py

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

# rezolvare utilizând bucla for
result = True
for val in lista:
    if val < 0:
        result = False
# rezolvare utilizând operatorul all
result = all(val > 0 for val in lista)

print(result)
```

rezultat

```
True
```

În acest caz, expresia `val > 0` este evaluată pentru fiecare număr din listă. Deoarece toate numerele sunt mai mari decât 0, expresia `all(val > 0 for val in lista)` este adevărată, deci `result` va fi `True`.

Operatorul `any` este utilizat pentru a verifica dacă cel puțin una dintre expresiile dintr-o listă este evaluată ca adevărată. Dacă cel puțin una dintre expresii este adevărată, atunci rezultatul întregii expresii este `True`, altfel este `False`. Sintaxa operatorului `any` este următoarea: `any(iterabil)` unde `iterabil` este o listă, un tuplu sau o altă structură de date iterabilă ce conține expresiile ce trebuie verificate. De exemplu, presupunând o listă de numere, se dorește să se verifice dacă cel puțin un număr din listă este par. Firește, se poate utiliza o structură repetitivă, dar operatorul `any` permite o scriere mai compactă:

main.py

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

# rezolvare utilizând bucla for
result = False
for val in lista:
    if val % 2 == 0:
        result = True
        break

# rezolvare utilizând operatorul all
result = any(val % 2 == 0 for val in lista)
```

rezultat

```
True
```

În acest caz, expresia `val % 2 == 0` este evaluată pentru fiecare număr din listă. Deoarece cel puțin unul dintre numere este par, expresia `any(val % 2 == 0 for val in lista)` este adevărată, atunci și `result` va fi `True`.

C. Decoratori

Decoratorii sunt funcții ***Python*** care îmbunătățesc comportamentul altor funcții [1]. Această abordare se mai numește și "programarea orientată spre aspecte" sau "programarea modulară". În mod specific, decoratorii permit adăugarea de funcționalități suplimentare la funcții existente, fără a le modifica codul sursă.

Decoratorii sunt foarte flexibili și pot fi utilizați pentru o varietate de scopuri, inclusiv pentru a valida argumente, a înregistra datele de intrare/ieșire, a salva rezultatele într-un cache, a gestiona erorile și multe altele. Cu ajutorul decoratorilor, putem adăuga funcționalități suplimentare într-un mod modular și reutilizabil, fără a modifica codul sursă al funcțiilor originale.

Decoratorii sunt, în esență, funcții care primesc o altă funcție ca argument și returnează o altă funcție modificată. Aceasta poate fi o funcție nouă, care adaugă comportament suplimentar funcției originale, sau poate fi funcția originală îmbunătățită. După definirea decoratorului, acesta poate fi utilizat pentru a decora o funcție prin folosirea, pe linia anterioară definirii funcției, a caracterului `@` urmat de numele decoratorului. Atunci când apelăm o funcție ce este decorată, de fapt se va apela funcția decorator ce va primi ca parametru funcția decorată și va returna funcția îmbunătățită ce va fi apoi executată.

Exemplu:

main.py

```
def decoratorul_meu(func):
    def wrapper(*args, **kwargs):
        print("Facem ceva înainte de apelarea funcției")
        result = func(*args, **kwargs)
        print("Facem altceva după apelarea funcției")
        return result
    return wrapper

@decoratorul_meu
def func1(x):
    print(f"apel func1({x})")

@decoratorul_meu
def func2(x):
    print(f"apel func2({x})")

func1(12)
func2("Introducere Python")
```

rezultat

```
Facem ceva înainte de apelarea funcției
apel func1(12)
Facem altceva după apelarea funcției
Facem ceva înainte de apelarea funcției
apel func2(Introducere Python)
Facem altceva după apelarea funcției
```

Următorul exemplu arată un decorator ce poate fi folosit la măsurarea timpului necesar pentru a executa o funcție și utilizarea lui pentru a măsura timpul de execuție:

main.py

```
import time
import numpy as np
import scipy

def masoara_timp(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"{end_time - start_time} secunde")
        return result
    return wrapper

def func1(x):
    return np.sin(x) + np.exp(-2 * x)

@masoara_timp
def root(func, x0):
    return scipy.optimize.root(func, x0)
print(root(func1, 0).x)
```

rezultat

```
0.4717392921447754 secunde
[0.38457757]
```

Limbajul Python implementează o serie de decoratori predefiniți. Mare parte a acestora sunt utilizați în programarea orientată pe obiecte, așa cum se va vedea în lucrarea următoare. O parte dintre decoratorii din limbajul de bază se găsesc în librăria predefinită `functools`.

Dintre toți decoratorii definiți în această librărie, vom studia decoratorul `functools.lru_cache`. Acesta se utilizează pentru a cache-ui valorile funcțiilor decorate, astfel încât, dacă acea funcție se va apela de mai multe ori cu aceiași parametrii, prima dată, funcția va fi executată și valoarea salvată în cache, urmând ca la apeluri ulterioare funcția să nu se mai execute, ci să se utilizeze valoarea din cache.

Exemplu:

main.py

```
import functools

@functools.lru_cache
def factorial(n):
    print(f"Calcul {n}!")
    return n * factorial(n-1) if n > 0 else 1

print(factorial(5))
print()
print(factorial(10))
```

rezultat

```
Calcul 5!
Calcul 4!
Calcul 3!
Calcul 2!
Calcul 1!
Calcul 0!

120
Calcul 10!
Calcul 9!
Calcul 8!
Calcul 7!
Calcul 6!
3628800
```

În exemplu, este implementată o funcție recursivă pentru calculul factorialului unui număr. Primul apel al funcției (`factorial(5)`) va apela la rândul ei `factorial(4)` care va apela `factorial(3)` și așa mai departe, până la apelul `factorial(0)`. Valorile rezultate în urma celor 6 apeluri vor fi salvate în cache. Astfel, în momentul în care apelăm `factorial(10)` se va apela `factorial(9)` și așa mai departe, până la apelul `factorial(6)`. Acesta va încerca să apeleze `factorial(5)`, dar, întrucât a mai fost apelat o dată, se va utiliza valoarea cache-uită.

D. Generatori

În ***Python***, un generator este o funcție specială care poate fi utilizată pentru a produce secvențe de valori [1]. Oferă o alternativă mai eficientă decât crearea unei

liste complete de valori, deoarece generatorii calculează și returnează doar un singur element din secvență la un moment dat, în loc să construiască întreaga secvență în memorie. Un generator este o funcție care conține o instrucțiune `yield` ce va returna un iterator. În momentul execuției funcției, la întâlnirea instrucțiunii `yield` se va returna o valoare în iterator și se va opri temporar execuția funcției. La următorul apel al generatorului, execuția continuă de la ultima instrucțiune `yield` și, la întâlnirea următoarei instrucțiuni `yield`, se repetă procesul de returnare a unei valori și oprire a execuției. Rezultatul funcției de tip generator este un iterator ce poate fi folosit fie într-o structură `for`, fie se pot extrage valori, una câte una, cu funcția `next`.

Exemplu simplu de generator care produce numere pare:

main.py

```
def par():
    x = 0
    while True:
        yield x
        x += 2

valori = par()

print(next(valori))
print(next(valori))
print(next(valori))
```

rezultat

```
0
2
4
```

În acest exemplu, generatorul `par()` începe să numere de la 0 și produce numere pare la fiecare apel. Deoarece `par()` este o funcție care conține instrucțiunea `yield`, se comportă ca un iterator, permițând accesul la câte o valoare, la fiecare iterație.

Generatorii sunt utili atunci când avem de a face cu o secvență de valori, care ar putea fi prea mare pentru a fi stocată în memorie, sau atunci când dorim să generăm valori doar atunci când sunt necesare.

Avantaje ale utilizării generatorilor în ***Python***:

1. Consumă mai puțină memorie decât crearea unei liste complete de valori, fiind util atunci când avem de a face cu o secvență mare de valori.
2. Pot fi utilizați pentru a produce fluxuri infinite de valori, ceea ce nu ar fi posibil cu o listă sau un tuplu.
3. Sunt utilizați frecvent în Python pentru a crea iteratori personalizați.

Exemplu de generator care produce secvența de numere Fibonacci și iterarea acestuia într-o structură condițională `for`:

main.py

```
def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

for i in fibonacci():
    print(i)
    if i > 50:
        break
```

rezultat

```
0
1
1
2
3
5
8
13
21
34
55
```

E. Funcții lambda

Funcțiile lambda sunt funcții care pot fi definite într-o singură linie de cod în limbajul de programare ***Python*** [1]. Aceste funcții sunt numite "anonime" deoarece nu sunt definite folosind cuvântul cheie `def` și nu au un nume asociat. În schimb, sunt definite folosind cuvântul cheie `lambda` și se pot referi la ele prin intermediul unei variabile sau prin transmiterea lor ca argumente pentru alte funcții. Sintaxa

pentru definirea unei funcții lambda este `lambda argumente: expresie`, unde `argumente` reprezintă argumentele funcției, separate prin virgulă, iar `expresie` este o expresia ce dă valoarea returnată de funcție. O funcție lambda poate avea orice număr de argumente, dar trebuie să conțină o singură expresie, care este valoarea returnată de funcție.

Exemplu:

main.py

```
radical = lambda x, y: x ** (1 / y)
print(radical(125, 3)) # radical de ordin 3 din 125
```

rezultat

```
5.0
```

Funcțiile lambda sunt adesea utilizate în Python pentru a defini funcții *inline* pentru a efectua operații simple, cum ar fi: sortarea, filtrarea sau maparea elementelor dintr-un iterabil.

Astfel, funcția `sorted` se utilizează pentru a sorta lista primită ca parametru (spre exemplu în ordine crescătoare sau în ordine alfabetică). Totuși, câteodată, dorim să sortăm o listă în funcție de un alt criteriu. Pentru acest lucru, se va da un al doilea parametru funcției `sorted` (`key`), o funcție lambda, care mapează fiecare element al listei cu criteriu după care se dorește a se face sortarea. În exemplul următor, se sortează o listă de cuvinte în ordinea lungimii cuvintelor:

main.py

```
fructe = ["banana", "ananas", "para", "mar"]

# sortare în ordine alfabetică
print(sorted(fructe))

# sortare în orinea lungimii cuvintelor
print(sorted(fructe, key=lambda elem: len(elem)))
```

rezultat

```
['ananas', 'banana', 'mar', 'para']
['mar', 'para', 'banana', 'ananas']
```

Filtrarea elementelor unui iterabil se face utilizând funcția `filter` ce primește doi parametri: funcția lambda, ce definește condiția de filtrare a fiecărui element al iterabilului, și iterabilul în sine. Exemplu: pentru o listă conținând toate numerele de la 1 la 20, se dorește afișarea elementelor ce sunt pătrate perfecte:

main.py

```
import math

lista  = list(range(1, 21))
query =  lambda x: math.sqrt(x) == math.ceil(math.sqrt(x))
pp = filter(query ,lista)
print(list(pp))
```

rezultat

```
[1, 4, 9, 16]
```

Maparea elementelor unui iterabil presupune generarea unui alt iterabil prin aplicarea unei funcții asupra fiecărui element al iterabilului. Acest lucru se realizează cu funcția `map` ce primește ca parametri funcția ce se aplică elementelor iterabilului și iterabilul. Exemplu: pentru lista `[1, 7, 2, 4, 5]` se dorește calcularea pătratului elementelor pare și a cubului elementelor impare:

main.py

```
lista = [1, 7, 2, 4, 5]
rezultat = map(lambda x: x**2 if x%2==0 else x**3, lista)
print(list(rezultat))
```

rezultat

```
[1, 343, 4, 16, 125]
```

Funcțiile `filter` și `map` se pot aplica tuturor tipurilor de iterabile, inclusiv generatoarele. Astfel, plecând de la exemplul de generator care produce secvența de numere Fibonacci, utilizând funcția `filter`, se pot afișa doar numerele pare. Exemplu:

main.py

```
def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

fibonacci_par = filter(lambda x: x % 2 == 0, fibonacci())

for i in fibonacci_par:
    print(i)
    if i > 50:
        break
```

rezultat

`0`
`2`
`8`
`34`
`144`

F. Cerințe

1. Rezolvați, utilizând funcția `scipy.optimize.root`, ecuațiile:
   a) $e^{-x} - x = 0$
   b) $2 \cdot x + \sin(x) = -1$

   Dați funcției `scipy.optimize.root` ecuația sub formă de funcție lambda.

   Utilizați, pentru fiecare rezultat, **operatorul ternar** pentru a afișa dacă acesta este negativ sau pozitiv.

2. Determinați, utilizând operatorul **`all`**, dacă toate elementele listei `[15129, 2401, 5041, 784, 2146225]` sunt pătrate perfecte.

3. Determinați, utilizând funcția `np.roots`, rădăcinile polinoamelor:
   a) $x^2 - 5 \cdot x + 6$
   b) $x^6 - 14 \cdot x^5 + 52 \cdot x^4 + 22 \cdot x^3 - 329 \cdot x^2 + 136 \cdot x + 420$
   c) $x^4 - 16 \cdot x^3 + 86 \cdot x^2 - 176 \cdot x + 105$

   Pentru fiecare rezultat:

   i. determinați, utilizând operatorul **`all`**, dacă toate rădăcinile sunt pozitive.
   ii. determinați, utilizând operatorul **`all`**, dacă toate rădăcinile sunt impare.
   iii. determinați, utilizând operatorul **`any`**, dacă cel puțin una dintre rădăcini are valoarea 2.
   iv. determinați, utilizând operatorul **`any`**, dacă cel puțin una dintre rădăcini are valoarea 1 sau -1.

4. Se dă dicționarul morse:

```
dict_morse = {'A': '.-', 'B': '-...', 'C': '-.-.',
'D': '-..', 'E': '.', 'F': '..-.', 'G': '--.', 'H':
'....', 'I': '..', 'J': '.---', 'K': '-.-', 'L': '.-
..', 'M': '--', 'N': '-.', 'O': '---', 'P': '.--.',
'Q': '--.-', 'R': '.-.', 'S': '...', 'T': '-', 'U':
'..-', 'V': '...-', 'W': '.--', 'X': '-..-', 'Y': '-
.--', 'Z': '--..', '0': '-----', '1': '.----', '2':
'..---', '3': '...--', '4': '....-', '5': '.....',
'6': '-....', '7': '--...', '8': '---..', '9': '---
-.', ' ': '/'}.
```

Implementați în Python o funcție care să fie capabilă să decodifice un mesaj Morse. Testați funcția cu câteva exemple, afișând mesajul codificat și mesajul decodificat.

5. Scrieți un program în limbajul Python pentru gestionarea unei liste de cărți într-o librărie. Se dă următoarea listă de cărți:
   - Povestea unei minți, John Doe, 2005;
   - Explorarea lumii subacvatice, Jane Smith, 2010;
   - Arta programării, Donald Knuth, 1982;
   - Python Programming, Guido van Rossum, 2019;
   - Design Patterns, Erich Gamma, 1994;
   - O mie și una de nopți, Anonim, 900;
   - Romeo și Julieta, William Shakespeare, 1597;
   - Divina Comedie, Dante Alighieri, 1320;
   - Originea speciilor, Charles Darwin, 1859;
   - Amintiri din copilărie, Ion Creangă, 1881;

   Utilizați funcții lambda pentru a efectua operații specifice asupra listei de cărți date.
   a) Verificați dacă o carte are un anumit autor.
   b) Filtrați cărțile după anul publicării.
   c) Sortați cărțile după titlu.
   d) Sortați cărțile după autor.

6. Implementați un generator care să producă o secvență de numere prime, începând cu 2. Puteți verifica dacă un număr este prim, utilizând funcția **`isprime`** din librăria ***SimPy*** (ex: `sympy.isprime(5)`).
    a) Utilizați generatorul implementat pentru a crea un iterator ce va fi utilizat la afișarea tuturor numerelor prime până la 100.
    b) Creați un iterator prin filtrarea generatorului implementat, astfel încât să filtreze doar numerele prime ce se termină în cifra 1. Utilizați iteratorul pentru a afișa toate numerele prime ce se termină cu cifra 1 și sunt mai mici de 1000.
    c) Creați un iterator prin maparea generatorului implementat, astfel încât să se calculeze partea întreagă a logaritmului în bază 2 a fiecărui număr prim. Afișați primele 20 de elemente din iterator.

Lucrarea 9. Programare orientată pe obiect în Python

A. Obiectivele lucrării

1. Familiarizarea cu programarea orientată pe obiect în limbajul Python

B. Introducere

Programarea orientată pe obiect (OOP) este un stil de programare care se concentrează pe obiecte, ca fiind elemente de bază ale programelor. Aceste obiecte se definesc ca instanțe ale claselor, care reprezintă șabloane pentru crearea de obiecte. Fiecare obiect are proprietăți și comportamente definite de clasa respectivă. Noțiunea cea mai importantă în programarea orientată pe obiecte este cea de clasă, clasa fiind un șablon în interiorul căruia se definesc o serie de atribute comune ale obiectelor aparținând acelei clase. Aceste atribute pot fi fie proprietăți (variabile – ele specifică proprietăți ale obiectelor), fie metode (funcții – ele specifică comportamente ale obiectelor). Un obiect aparținând unei clase se mai numește și instanță a clasei respective.

Un aspect important al programării orientate pe obiect este încapsularea, care se referă la faptul că anumite proprietăți și metode ale unui obiect pot fi ascunse. OOP se concentrează, de asemenea, pe polimorfism, care se referă la faptul că obiectele pot fi de tipuri diferite, dar pot fi tratate într-un mod similar. Acest lucru permite crearea de funcții și metode care pot fi utilizate cu orice tip de obiect, atât timp cât respectă interfața comună definită de acestea.

În general, programarea orientată pe obiect are avantajele sale, printre care: reutilizarea codului, abstracția, modularitatea și securitatea. În plus, OOP permite reutilizarea codului prin intermediul moștenirii, unde o clasă derivată poate moșteni proprietățile și comportamentele unei clase de bază. Este utilizată în multe limbaje de programare, printre care și ***Python***.

Definirea unei clase în Python se face utilizând cuvântul cheie `class` urmat de numele clasei și simbolul două puncte, urmând ca, indentat, să se definească proprietățile și metodele. Acestea se declară ca orice variabile sau funcții, cu mențiunea că metodele se declară întotdeauna ca având un prim parametru `self` ce

este o variabilă ce va face referire întotdeauna la obiect însuși. Instanțierea unui obiect de tipul clasei definite se face scriind numele obiectului, urmat de simbolul egal, numele clasei și de un set de paranteze rotunde. După definirea unui obiect, putem accesa proprietățile și metodele acestuia scriind numele obiectului urmat de simbolul punct și de proprietatea / metoda dorită. Același lucru este valabil și dacă într-o metodă dorim să modificăm o proprietate sau să apelăm altă metodă, caz în care vom folosi, în loc de numele obiectului, parametrul `self`.

Spre exemplu, dorim să definim clasa **Persoana** având proprietatea **nume** și metoda **salut**:

main.py

```
class Persoana:
    # proprietate
    nume = "nedefinit"

    # metoda
    def salut(self):
        print(f"Salut, mă numesc {self.nume}")

# instanțiere
persoana1 = Persoana()
# accesare proprietate
persoana1.nume = "Gigel"
# apelare metoda
persoana1.salut()

# altă instanțiere
persoana2 = Persoana()
persoana2.salut()
```

rezultat

```
Salut, mă numesc Gigel
Salut, mă numesc nedefinit
```

Spre deosebire de alte limbaje de programare, în ***Python***, obiectele au o structură dinamică. Practic, putem adăuga noi proprietăți unui obiect în timpul execuției programului. Mai mult, putem schimba chiar definiția unei clase în timpul execuției programului.

Exemplu:

main.py

```
class Persoana:
    nume = "nedefinit"
    def salut(self):
        print(f"Salut, mă numesc {self.nume}")

# modificăm definiția clasei - va afecta toate instantele
Persoana.varsta = 20

persoana1 = Persoana()
print(persoana1.varsta)

# adaugam proprietatea inalțime
persoana1.inaltime = 1.73
print(persoana1.inaltime)
```

rezultat

```
20
1.73
```

C. Metode speciale

Metodele speciale în ***Python*** sunt metode cu nume special, nume ce începe și se termină cu două caractere underscore (_ _). Acestea sunt utilizate pentru a defini comportamente speciale ale obiectelor, cum ar fi: comportamentul în momentul instanțierii, în momentul afișării, operatori matematici între instanțe, operații de comparație între instanțe etc.

Definirea comportamentului unei clase în momentul instanțierii se face prin definirea metodei `__init__(self, ...)`, ce poate primi, pe lângă parametrul `self`, oricâți alți parametri. Aceasta este o metodă specială de inițializare a obiectelor și este apelată la fiecare instanțiere a clasei.

main.py

```
class Rezistenta:
    def __init__(self, nume, valoare):
        # adaugam noi proprietăți in interiorul __init__
        self.nume = nume
        self.valoare = valoare

R1 = Rezistenta("R1", 100)
print(f"Rezistența {R1.nume} are {R1.valoare} ohmi")
```

rezultat

```
Rezistența R1 are 100 ohmi
```

Dacă încercăm să dăm o instanță a unei clase ca parametru funcției print, putem observa că nu se va afișa nici o informație despre proprietățile obiectului.

main.py

```
R1 = Rezistenta("R1", 100)
print(R1)
```

rezultat

```
<__main__.Rezistenta object at 0x000001f3896794c0>
```

Dacă, totuși, dorim să afișăm informații specifice, va trebui să definim metoda specială `__str__(self)` ce va primi doar parametrul `self` și va returna un șir de caractere pe care dorim să-l afișam atunci când dăm obiectul ca parametru funcției `print`. Exemplu:

main.py

```
class Rezistenta:
    def __init__(self, nume, valoare):
        self.nume, self.valoare = nume, valoare

    def __str__(self):
        return f" Rezistența {self.nume} ({self.valoare}
ohmi)"

R1 = Rezistenta("R1", 100)
print(R1)
```

rezultat

```
Rezistența R1 (100 ohmi)
```

Mai există, de asemenea, o serie de metode speciale pentru a defini comportamentul obiectului atunci când încercăm să-l introducem într-o expresie, fie ea matematică sau comparație. Spre exemplu, dacă dorim să avem posibilitatea ca utilizând semnul matematic + între două instanțe ale clasei **Rezistenta** să obținem o a treia instanță prin punerea în serie a celor două rezistențe, vom defini metoda `__add__(self, x)` ce va returna rezultatul operației de adunare:

main.py

```
class Rezistenta:
    def __init__(self, nume, valoare):
        self.nume, self.valoare = nume, valoare

    def __str__(self):
        return f"{self.nume} ({self.valoare} ohmi)"

    def __add__(self, x): # self + x
        nume_rezultat = f" Rezistența SERIE({self.nume},
{x.nume})"
        valoare_rezultat = self.valoare + x.valoare
        return Rezistenta(nume_rezultat, valoare_rezultat)

R1 = Rezistenta("R1", 100)
R2 = Rezistenta("R2", 220)

print(R1 + R2)
```

rezultat

```
Rezistența SERIE(R1, R2) (320 ohmi)
```

Atenție, dacă valoarea cu care adunăm nu este tot de tip rezistență, exemplul de mai sus va da eroare. Se recomandă ca întotdeauna, în interiorul metodelor speciale ce definesc operatori, să se verifice tipul de dată al parametrului utilizând funcția `isinstance`:

main.py

```
class Rezistenta:
    ...
    def __add__(self, x):
        if not isinstance(x, Rezistenta):
            print("Nu se poate aduna !!!")
            return None

        nume_rezultat = f"Rezistența SERIE({self.nume},
{x.nume})"
        valoare_rezultat = self.valoare + x.valoare
        return Rezistenta(nume_rezultat, valoare_rezultat)

R1 = Rezistenta("R1", 100)
R2 = Rezistenta("R2", 220)

print(R1 + R2)
print(R1 + 7)
```

rezultat

```
Rezistența SERIE(R1, R2) (320 ohmi)
Nu se poate aduna !!!
None
```

Tabelul următor prezintă o serie de alți operatori ce pot fi definiți, împreună cu metodele speciale aferente.

Tabelul 1 – Metode speciale pentru definirea acțiunii operatorilor

Operatori matematici	
+	`__add__(self, x)`
-	`__sub__(self, x)`
*	`__mul__(self, x)`
/	`__truediv__(self, x)`
**	`__pow__(self, x)`
+=	`__iadd__(self, x)`
-=	`__isub__(self, x)`
*=	`__imul__(self, x)`
/=	`__itruediv__(self, x)`
**=	`__ipow__(self, x)`
Operatori de comparație	
==	`__eq__(self, x)`
!=	`__ne__(self, x)`
<	`__lt__(self, x)`
<=	`__le__(self, x)`
>	`__gt__(self, x)`
>=	`__ge__(self, x)`

Astfel, dacă dorim să utilizăm operatorul pentru a vedea dacă două instanțe ale clasei Rezistență au aceeași valoarea a rezistenței, se va proceda astfel:

main.py

```
class Rezistenta:
    ...
    def __eq__(self, x):
        return self.valoare == x.valoare

R1 = Rezistenta("R1", 100)
```

```
R2 = Rezistenta("R2", 220)
R3 = Rezistenta("R3", 100)

print(R1 == R2)
print(R1 == R3)
```

rezultat

```
False
True
```

D. Încapsulare

Încapsularea este un concept important în programarea orientată pe obiect, care se referă la restricționarea accesului la anumite proprietăți și metode din afara obiectului. În ***Python***, încapsularea este realizată prin utilizarea convenției de numire subliniată și prin utilizarea decoratoarelor.

În ***Python***, există trei niveluri de accesibilitate a proprietăților și metodelor într-o clasă:

1. public – proprietățile și metodele publice sunt accesibile din orice loc în codul sursă, inclusiv din afara clasei.
2. protected – proprietățile și metodele protejate sunt denumite cu un singur underscore la începutul numelui lor și pot fi accesate doar din interiorul clasei și din subclasele sale.
3. private – proprietățile și metodele private sunt denumite cu două underscore-uri la începutul numelui lor și pot fi accesate doar din interiorul clasei.

Încapsularea poate fi utilă pentru a proteja datele și pentru a preveni modificările accidentale din afara clasei. Astfel, se pot defini metode cu ajutorul cărora putem accesa proprietățile private în siguranță.

Spre exemplu, dorim să adăugăm clasei **Rezistență** o proprietate care să specifice curentul de alimentare. Totuși, se dorește ca valoarea acestei proprietăți să fi între 0 și 100. Astfel, pentru a împiedica asignarea de valori eronate proprietății, aceasta va fi făcută privată și va fi accesată prin intermediul unor metode speciale. În acest mod, vor fi necesare două metode: una pentru a citi valoarea și una pentru a modifica valoarea proprietății private.

main.py

```
class Rezistenta:
    def __init__(self, nume, valoare):
        self.nume, self.valoare = nume, valoare
        self.__tensiune = 0 # declarare proprietate privată

    def get_tensiune(self):
        return self.__tensiune

    def set_tensiune(self, valoare):
        if valoare >= 0 and valoare <= 100:
            self.__tensiune = valoare
        else:
            print("Tensiune Invalida")

R1 = Rezistenta("R1", 100)
print(R1.get_tensiune())
R1.set_tensiune(22)
print(R1.get_tensiune())
R1.set_tensiune(101)
print(R1.get_tensiune())
```

rezultat

```
0
22
Tensiune Invalida
22
```

Dacă denumim cele două metode cu același nume (***nume***) și decorăm metoda dedicată citirii proprietății private cu decoratorul `@property` și metoda dedicată modificării cu `@`***`nume`***`.setter`, atunci numele celor două metode va putea fi folosit ca proprietate a obiectului:

main.py

```
class Rezistenta:
    def __init__(self, nume, valoare):
        self.nume, self.valoare = nume, valoare

        # declarare proprietate privata
        self.__tensiune = 0

    @property
    def tensiune(self):
        return self.__tensiune
```

```
    @tensiune.setter
    def tensiune(self, valoare):
        if valoare >= 0 and valoare <= 100:
            self.__tensiune = valoare
        else:
            print("Tensiune Invalida")

R1 = Rezistenta("R1", 100)
print(R1.tensiune)
R1.tensiune = 22
print(R1.tensiune)
R1.tensiune = 101
print(R1.tensiune)
```

rezultat

```
0
22
Tensiune Invalida
22
```

E. Moștenire

Moștenirea este un mecanism prin care o clasă poate extinde sau împrumuta comportamentul unei alte clase numită clasă părinte sau clasă de bază. În ***Python***, această relație de moștenire este definită prin includerea numelui clasei părinte între paranteze rotunde după declarația clasei copil. Clasa copil va moșteni toate atributele și metodele din clasa părinte, dar își poate defini propriile atribute și metode sau chiar poate suprascrie metode definite de clasa părinte. Utilizând funcția `super()`, putem accesa metode din clasa părinte, chiar dacă au fost suprascrise. Această funcție este utilă și pentru a apela constructorul clasei părinte din constructorul clasei copil. Exemplu:

main.py

```
# clasa parinte
class Componenta:
    def __init__(self, nume, val, unit):
        # proprietati ce vor fi mostenite de copii
        self.nume,self.valoare,self.unitate = nume,val,unit

    # metoda ce va fi mostenita de copii
    def __str__(self):
        return f"{self.nume} ({self.valoare} {self.unitate})"
```

```
# clasa copil
class Rezistenta(Componenta):
    def __init__(self, nume, valoare):
        # apelare constructor clasa parinte
        super().__init__(nume, valoare, "ohmi")

    # metoda proprie copilului
    def __add__(self, x):
        if not isinstance(x, Rezistenta):
            print("Nu se poate aduna !!!")
            return None

        nume_rez = f"Rezistența SERIE({self.nume}, {x.nume})"
        valoare_rezultat = self.valoare + x.valoare
        return Rezistenta(nume_rez, valoare_rezultat)

# clasa copil
class Condensator(Componenta):
    def __init__(self, nume, valoare):
        super().__init__(nume, valoare, "F")

    def __add__(self, x):
        if not isinstance(x, Condensator):
            print("Nu se poate aduna !!!")
            return None

        nume_rez=f"Condensator SERIE({self.nume}, {x.nume})"
        valoare_rezultat = 1 / (1/self.valoare + 1/x.valoare)
        return Condensator(nume_rez, valoare_rezultat)

R1 = Rezistenta("R1", 330)
R2 = Rezistenta("R2", 220)
C1 = Condensator("C1", 0.001)
C2 = Condensator("C2", 0.002)

print(R1 + R2)
print(C1 + C2)
```

rezultat

```
Rezistența SERIE(R1, R2) (550 ohmi)
Condensator SERIE(C1, C2) (0.0006666666666666666 F)
```

În exemplul dat, clasa părinte este clasa **Componentă**. Aceasta definește proprietățile nume, valoare, unitate, dar și metoda specială `__str__`. Acestea vor fi moștenite de cele două clase copil: **Rezistență** și **Condensator**. Aceste două clase își definesc, în mod independent, metoda specială `__add__`.

F. Abstractizare

Abstractizarea este un concept care implică definirea unei clase cu anumite metode, dar fără a specifica implementarea lor. Aceste metode se numesc metode abstracte și reprezintă o modalitate de a defini comportamentul pe care trebuie să-l aibă o clasă, fără a se specifica cum anume se va realiza acest comportament. În ***Python***, putem crea clase abstracte prin utilizarea modulului **abc**. Mai precis, putem crea o clasă abstractă prin definirea unei clase care moștenește clasa `ABC` din modulul **abc** și prin utilizarea decoratorului `@abc.abstractmethod` pentru a marca metodele abstracte. Metodele abstracte nu sunt implementate, astfel, ele vor conține doar cuvântul cheie `pass`. În acest mod, clasele ce moștenesc clasa abstractă vor fi obligate să implementeze metodele abstracte, în caz contrar rezultând eroare.

main.py

```
import abc

class Componenta(abc.ABC):
    def __init__(self, nume, valoare, unitate):
        self.nume,self.valoare,self.unitate = nume, valoare,
unitate

    def __str__(self):
        return f"{self.nume} ({self.valoare} {self.unitate})"

    @abc.abstractmethod
    def __add__(self, other):
        pass

class Rezistenta(Componenta):
    def __init__(self, nume, valoare):
        super().__init__(nume, valoare, "ohmi")

    # implementare metodei abstracte
    def __add__(self, x):
        if not isinstance(x, Rezistenta):
            print("Nu se poate aduna !!!")
            return None
        nume_rez = f"Rezistența SERIE({self.nume}, {x.nume})"
        valoare_rezultat = self.valoare + x.valoare
        return Rezistenta(nume_rez, valoare_rezultat)

class Condensator(Componenta):
    def __init__(self, nume, valoare):
        super().__init__(nume, valoare, "F")
```

```
    def __add__(self, x):
        if not isinstance(x, Condensator):
            print("Nu se poate aduna !!!")
            return None

        nume_rez=f"Condensator SERIE({self.nume}, {x.nume})"
        valoare_rezultat = 1 / (1/self.valoare + 1/x.valoare)
        return Condensator(nume_rez, valoare_rezultat)
```

G. Cerințe

Implementați o clasă, numită **Punct**, ce conține trei proprietăți: **nume**, **x** și **y** (**x** și **y** fiind coordonatele punctului).

- Proprietățile **nume**, **x** și **y** se vor inițializa din parametrii constructorului.
- Implementați în clasa **Punct** o metodă specială care să permită afișarea punctului de forma `nume(x, y)`.

main.py

```
...
A = Punct("A", 1, 2)
print(A)
```

rezultat

```
A(1, 2)
```

- Implementați în clasa **Punct** o metodă specială care să definească operația de scădere a două puncte, rezultatul scăderii fiind un număr real reprezentând distanța dintre cele două puncte ($\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$).

main.py

```
...
A = Punct("A", 1, 2)
B = Punct("B", 5, 6)
print(A - B)
```

rezultat

```
5.656854249492381
```

Implementați o clasă, numită **Segment**, ce conține trei proprietăți: **nume**, **P1** și **P2** (**P1** și **P2** fiind cele două puncte ce definesc segmentul).

- **P1** și **P2** se vor inițializa din parametrii constructorului, în timp ce proprietate **nume** se va obține prin concatenarea numelor celor două puncte.
- Implementați în clasa **Segment** o metodă specială care să permită afișarea segmentului de forma `nume[P1(x, y), P2(x, y)]`.
- Implementați în clasa **Segment** o metodă numită **lungime** care să returneze lungimea segmentului. Decorați-o cu decoratorul `@property`, astfel încât să poată fi accesată ca proprietate.
- Implementați în clasa **Punct** o metodă specială care să definească operația de adunare a două puncte, rezultatul adunării fiind un segment definit de cele două puncte.

main.py

```
...
A = Punct("A", 1, 2)
B = Punct("B", 5, 6)
AB1 = Segment(A, B)
AB2 = A + B
print(AB1)
print(AB2)
print(AB2.lungime)
```

rezultat

```
AB[A(1, 2), B(5, 6)]
AB[A(1, 2), B(5, 6)]
5.656854249492381
```

Implementați o clasă, numită **Dreapta**, ce conține trei proprietăți: **a**, **b** și **c** care descriu ecuația dreptei de forma $a \cdot x + b \cdot y + c = 0$.

- Constructorul clasei va primi, ca parametru, un segment și va determina cei 3 parametri (**a**, **b** și **c**) ai dreptei. ($a = y_1 - y_2,\ b = x_1 - x_2,\ c = x_1 \cdot y_2 - x_2 \cdot y_1$).
- Implementați în clasa **Dreapta** o metodă specială care să permită afișarea dreptei de forma `(ax+by+c=0)`.
- Implementați în clasa **Dreapta** o metodă specială care să definească operatorul ==, rezultatul fiind **`True`** dacă cele două drepte sunt paralele (două drepte sunt paralele dacă $\frac{a_1}{b_1} = \frac{a_2}{b_2}$.

main.py

```
...
A = Punct("A", 1, 2)
B = Punct("B", 5, 6)
C = Punct("A", 2, 2)
D = Punct("B", 6, 6)

d1 = Dreapta(A + B)
d2 = Dreapta(C + D)
d3 = Dreapta(A + D)
print(d1)
print(d2)
print(d3)
print(d1 == d2)
print(d1 == d3)
```

rezultat

```
(-4x+-4y+-4=0)
(-4x+-4y+0=0)
(-4x+-5y+-6=0)
True
False
```

Implementați clasele **Punct3D**, **Segment3D** și **Dreapta3D** ca clase derivate din clasele **Punct**, **Segment** și **Dreaptă**. Adăugați noilor clase, pe lângă proprietățile moștenite de la clasele de bază, proprietățile necesare pentru a reprezenta instanțe geometrice în spațiul 3D (spre exemplu, clasa **Punct3D** va necesita, pe lângă proprietățile definite de clasa de bază **Punct**, și proprietatea **z**). Suprascrieți, în clasele copil, acolo unde este necesar, metodele definite de clasele de bază.

Lucrarea 10. Programare concurentă în Python

A. Obiectivele lucrării

1. Familiarizarea cu programarea concurentă în limbajul ***Python***
2. Familiarizarea cu funcționalitatea librăriei ***Threading***

B. Introducere

Programarea concurentă se referă la o abordare a programării în care se proiectează o aplicație astfel încât să poată gestiona mai multe sarcini simultan. Acest lucru se poate realiza în două moduri: prin utilizarea mai multor fire de execuție (thread-uri) în cadrul aceluiași proces sau prin utilizarea mai multor procese care să ruleze sarcinile simultan. Fiecare sarcină va fi executată separat de celelalte, în mod independent. Utilizarea programării concurente oferă o serie de avantaje, printre care creșterea eficienței și performanței aplicațiilor.

Spre exemplu, atunci când o aplicație implică o sarcină care necesită timp (cum ar fi scrierea sau citirea unui fișier, citirea unor eșantioane de la un proces, obținerea unor resurse din rețea etc.), utilizarea programării concurente permite rularea altor sarcini simultan cu rularea acesteia. În acest mod, nu se blochează întreaga aplicație până când se finalizează sarcina ce necesită timp, ducând la o reducere semnificativă a timpului de așteptare, deci la îmbunătățirea performanței aplicației.

Totuși, programarea concurentă aduce cu sine o serie de probleme de sincronizare și acces la resurse, probleme ce pot duce la conflicte și blocaje, ceea ce ar putea afecta performanța și stabilitatea aplicației implementate. Conflictele apar atunci când mai multe sarcini (thread-uri sau procese) încearcă să acceseze aceeași resursă (aceeași locație de memorie, același fișier etc.) în același timp. Blocajele apar atunci când o sarcină blochează accesul la o resursă, ceea ce duce la blocarea altor sarcini ce depind de acea resursă. Astfel, este important ca la programarea concurentă să se ia în considerare problemele de sincronizare și alocare a resurselor pentru a putea asigura o funcționare robustă, corectă și eficientă a aplicației.

În ***Python***, programarea concurentă se poate realiza cu ajutorul mai multor librării ce oferă o varietate de funcționalități, funcții și clase care să faciliteze acest lucru:

- ***Threading*** – permite rularea sarcinilor în fire de execuție separate.
- ***Multiprocessing*** – permite rularea sarcinilor în procese separate, nefăcând parte din obiectul de studiu al lucrării de față.
- ***Asyncio*** – permite programarea asincronă, nefăcând parte din obiectul de studiu al lucrării de față.

Librăria ***Threading*** din ***Python*** este o librărie standard (inclusă în pachetul de bază) ce permite crearea și gestionarea firelor de execuție (thread-uri). Această librărie este implementată astfel încât să ofere o interfață ușor de utilizat pentru pornirea și gestionarea ulterioară a firelor de execuție într-un mod sigur și eficient.

Crearea unui fir de execuție în ***Python*** se face prin importarea librăriei ***Threading*** și instanțierea clasei `Thread`. Aceasta se inițializează dându-i ca parametru `target` funcția sau metoda ce va reprezenta sarcina respectivului fir de execuție. Apoi, pentru a porni firul de execuție se va apela metoda `start` a instanței create. Astfel, firul de execuție va începe să ruleze funcția specificată. Exemplu de creare și pornire a unui fir de execuție:

main.py

```
import threading
import time

def func1():
    for i in range(0, 5):
        print("Salutări din firul de execuție\n", end='')
        time.sleep(0.01)

my_thread = threading.Thread(target=func1)
my_thread.start()

print("Firul principal\n", end='')
```

rezultat

```
Salutări din firul de execuție
Firul principal
Salutări din firul de execuție
Salutări din firul de execuție
Salutări din firul de execuție
Salutări din firul de execuție
```

Bineînțeles, se pot porni mai multe fire de execuție, creând mai multe instanțe ale clasei `Thread`. De asemenea, se pot da parametri funcției ce reprezintă sarcina ce se va executa în firul de execuție, utilizând la instanțierea clasei `Thread` parametrul `args`, reprezentând un tuplu ce conține parametrii ce vor fi dați funcției. Exemplu:

main.py

```
import threading
import time

def func1(p):
    for i in range(0, 5):
      print(f"Salutări din firul de execuție {p}\n", end='')
        time.sleep(0.01)

my_thread1 = threading.Thread(target = func1, args=(1,))
my_thread2 = threading.Thread(target = func1, args=(2,))
my_thread1.start()
my_thread2.start()
print("Firul principal\n", end='')
```

rezultat

```
Salutări din firul de execuție 1
Salutări din firul de execuție 2
Firul principal
Salutări din firul de execuție 1
Salutări din firul de execuție 2
Salutări din firul de execuție 2
Salutări din firul de execuție 1
Salutări din firul de execuție 1
Salutări din firul de execuție 2
Salutări din firul de execuție 1
Salutări din firul de execuție 2
```

Pe lângă metoda `start`, clasa `Thread` mai definește o serie de metode pentru a interacționa cu firul de execuție și a controla comportamentul acestuia. Printre cele mai uzuale metode se numără metoda `join()` ce se utilizează pentru a aștepta ca firul de execuție să se termine.

Exemplu:

main.py

```
import threading
import time

def func1(a):
    for i in range(0, 5):
        print(f"Salutări din firul de execuție {a}\n",end='')
        time.sleep(1) # asteptam o secunda intre afisari

my_thread1 = threading.Thread(target = func1, args=(1,))
my_thread1.start()

print("Firul principal - inainte de join\n", end='')
my_thread1.join()
print("Firul principal - după join\n", end='')
```

rezultat

```
Salutări din firul de execuție 1
Firul principal - inainte de join
Salutări din firul de execuție 1
Salutări din firul de execuție 1
Salutări din firul de execuție 1
Salutări din firul de execuție 1
Firul principal - după join
```

Metoda `join` poate primi și un parametru `timeout`, care, atunci când este specificat, definește timpul maxim de așteptare al unui fir de execuție. În plus, metoda `is_alive()` se poate utiliza pentru a verifica dacă firul de execuție este încă în execuție. Exemplu:

main.py

```
import threading
import time

def func1(a):
    for i in range(0, 5):
        print(f"Salutări din firul de execuție {a}\n",end='')
        time.sleep(1) # asteptam o secunda intre afisari

my_thread1 = threading.Thread(target = func1, args=(1,))
my_thread1.start()

print("Firul principal - inainte de join\n", end='')
my_thread1.join(timeout=2) # timeout 2 secunde
```

```
if my_thread1.is_alive():
    print("Firul principa - timeout wait\n", end='')

print("Firul principa - după join\n", end='')
```

rezultat

```
Salutări din firul de execuție 1
Firul principal - inainte de join
Salutări din firul de execuție 1
Salutări din firul de execuție 1
Firul principa - timeout wait
Firul principa - după join
Salutări din firul de execuție 1
Salutări din firul de execuție 1
```

Librăria ***Threading*** oferă și alte funcționalități pentru a gestiona firele de execuție, funcționalități precum: blocaje, semafoare sau evenimente. Aceste mecanisme permit controlul sincronizării și concurenței între fire de execuție.

C. Blocaje

Blocajele (lock-uri) sunt obiecte ce se pot utiliza pentru a sincroniza accesul la resursele partajate între firele de execuție.

O resursă partajată poate fi, de exemplu, o variabilă globală sau un fișier pe care mai multe fire de execuție încearcă, concomitent, să le acceseze. Dacă nu s-ar utiliza lock-uri, firele de execuție ar putea accesa aceeași resursă în același timp, rezultatul putând fi arbitrar.

În exemplul următor, se definește o funcție ce incrementează un counter de 5 ori. Funcția este executată de 10 ori în fire de execuție separate. Deși ar fi de așteptat ca la final valoarea counter-ului să fie 50, rezultatul obținut este arbitrar:

main.py

```
import threading
import time

counter = 0
```

```
def f2():
    global counter
    for i in range(0, 5):
        temp = counter
        time.sleep(0.001) # procesare ce dureaza 1ms
        counter = temp + 1

threads = [threading.Thread(target=f2) for i in range(0,10)]
for thread in threads:
    thread.start()
for thread in threads:
    thread.join()
print(counter)
```

rezultat

```
5
```

Librăria ***Threading*** oferă o clasă `Lock` ce poate fi utilizată pentru a implementa blocaje. După instanțiere, obiectul de tip `Lock` poate fi utilizat pentru a sincroniza accesul la resursele partajate. Astfel, porțiunile de cod ce accesează resurse partajate vor fi încadrate între apeluri ale metodelor `acquire()` și `release()` ale obiectului de tip `Lock`. În acest mod, aceste porțiuni de cod se vor executa atomic (fără a fi întrerupte de un alt fir de execuție). Astfel, doar un fir de execuție poate accesa blocul de cod în același timp, în timp ce celelalte fire de execuție care încearcă să acceseze blocul de cod sunt blocate.

main.py

```
import threading
import time

counter = 0
lock1 = threading.Lock()

def f2():
    global counter
    for i in range(0, 5):
        lock1.acquire()

        temp = counter
        time.sleep(0.001)  # procesare ce dureaza 1ms
        counter = temp + 1

        lock1.release()

threads = [threading.Thread(target=f2) for i in range(0, 10)]
```

```
for thread in threads:
    thread.start()
for thread in threads:
    thread.join()

print(counter)
```

rezultat

```
50
```

Este important să nu uităm să eliberăm accesul la resursă atunci când nu ne mai este necesar, adică să nu uităm să apelăm metoda `release()`. În caz contrar, resursa va rămâne blocată, nici un alt fir de execuție nemaiputând-o accesa. Lock-urile se mai pot utiliza și cu structura `with`, în acest mod nemaifiind nevoie de apelarea metodelor `acquire()` și `release()`.

main.py

```
import threading
import time

counter = 0
lock1 = threading.Lock()

def f2():
    global counter
    for i in range(0, 5):
        with lock1:
            temp = counter
            time.sleep(0.001)  # procesare ce dureaza 1ms
            counter = temp + 1

threads = [threading.Thread(target=f2) for i in range(0, 10)]

for thread in threads:
    thread.start()
for thread in threads:
    thread.join()

print(counter)
```

rezultat

```
50
```

D. Semafoare

Semafoarele, ca și lock-urile, sunt obiecte de sincronizare ce permit gestionarea într-un mod sigur a accesului la resursele partajate între diferite fire de execuție.

Totuși, spre deosebire de lock-urile ce permit unui singur fir de execuție să acceseze o resursă partajată la un moment dat, semafoarele permit accesul simultan la aceeași resursă a unui număr dat de fire de execuție. Acest lucru este util atunci când putem accesa simultan aceeași resursă, dar nu dorim să supraîncărcăm resursa (de exemplu, mai multe fire de execuție pot accesa o bază de date, dar dorim să limităm numărul de conexiuni cu baza de date la 5 – în acest caz vom folosi un semafor).

Modulul ***Threading*** oferă suport pentru semafoare prin intermediul clasei `Semaphore`. Constructorul acestei clase primește ca parametru numărul maxim de fire de execuție ce pot accesa resursa partajată în același timp. După definire, acesta se utilizează în același mod cu lock-urile (utilizând metodele `acquire()` și `release()`). Exemplu:

main.py

```
import threading
import time

s1 = threading.Semaphore(2) # maxim 2 conexiuni simultane

def func3(param):
    print(f"[{param}] Astept să mă conectez...\n", end='')
    s1.acquire()
    # conectare DB
    print(f"[{param}] M-am conectat\n", end='')
    time.sleep(1) # interogare baza de date etc.
    # deconectare DB
    s1.release()
    print(f"[{param}] M-am deconectat\n", end='')

thr=[threading.Thread(target=func3,args=(i,)) for i in
range(0,5)]
for thread in thr:
    thread.start()
for thread in thr:
    thread.join()
```

rezultat

```
[0] Astept să mă conectez...
[0] M-am conectat
[1] Astept să mă conectez...
[1] M-am conectat
[2] Astept să mă conectez...
[3] Astept să mă conectez...
[4] Astept să mă conectez...
[1] M-am deconectat
[0] M-am deconectat
[3] M-am conectat
[2] M-am conectat
[3] M-am deconectat
[2] M-am deconectat
[4] M-am conectat
[4] M-am deconectat
```

E. Evenimente

Evenimentele se utilizează pentru a coordona sincronizarea între mai multe fire de execuție, atunci când unul dintre fire trebuie să aștepte până când o anumită acțiune a fost finalizată de un alt fir de execuție. În ***Python***, obiectele de tip eveniment se inițializează utilizând clasa `Event` din librăria ***Threading***. După instanțiere, asupra obiectului se pot aplica metodele `set()` și `clear()` pentru a seta / reseta evenimentul și metoda `wait()` pentru a bloca un fir de execuție până la setarea evenimentului.

Spre exemplu, să zicem că avem un fir de execuție ce achiziționează date de la un sistem, și un fir de execuție ce procesează aceste date. În mod evident, firul de execuție ce procesează datele va trebui să aștepte ca firul de execuție ce achiziționează date să finalizeze achiziția:

main.py

```
import threading
import time

evt = threading.Event()
```

```
def achizitie():
    for i in range(0, 3):
        print(f"Achiziție: start\n", end='')
        time.sleep(1) # procedura achizitie date
        print(f"Achiziție: stop\n", end='')
        evt.set()

def procesare():
    for i in range(0, 3):
        evt.wait()
        evt.clear()
        print(f"Procesare: start\n", end='')
        time.sleep(0.5)  # procedura procesare date
        print(f"Procesare: stop\n", end='')
        evt.set()


th1 = threading.Thread(target=achizitie)
th2 = threading.Thread(target=procesare)

th1.start(), th2.start()
th1.join(), th2.join()
```

rezultat

```
Achiziție: înainte
Achiziție: dupa
Procesare: înainte
Achiziție: înainte
Procesare: dupa
Procesare: înainte
Achiziție: dupa
Achiziție: înainte
Procesare: dupa
Procesare: înainte
Procesare: dupa
Achiziție: dupa
```

F. Timere

Timerele permit crearea de fire de execuție ce așteaptă un anumit interval de timp înainte de a executa o funcție. Acest lucru este util atunci când se dorește efectuarea unei acțiuni după o anumită perioadă de timp. Crearea timerelor se face utilizând

clasa `Timer` din librăria ***Threading***. Aceasta se instanțiază primind ca parametri intervalul de timp pe care să-l aștepte și funcția pe care să o execute. După crearea obiectului de tip `Timer`, acesta se va porni prin utilizarea metodei `start()`. Exemplu:

main.py

```
import threading
import time

def func():
    print("Funcția din timer")

timer = threading.Timer(2, func)
timer.start()

print("Firul principal 1")
time.sleep(1)
print("Firul principal 2")
time.sleep(1)
print("Firul principal 3")
time.sleep(1)
```

rezultat

```
Firul principal 1
Firul principal 2
Funcția din timer
Firul principal 3
```

G. Cerințe

1. Scrieți un program în limbajul Python care simulează operațiunea de rezervare de locuri la cinema. Programul trebuie să gestioneze accesul concurent la locurile disponibile utilizând semafoare. Implementați:
    a) O clasă „Cinema” care să conțină un număr de 54 de locuri disponibile și 15 locuri care nu pot fi rezervate.
    b) Un mecanism de rezervare care să permită unui număr de 11 clienți să rezerve un număr de locuri alese la întâmplare, după cum urmează:
        - Clientul 1 rezervă un singur loc.
        - Clientul 2 rezervă 5 locuri.
        - Clientul 3 rezervă 2 locuri.

- Clientul 4 rezervă 3 locuri.
- Clientul 5 rezervă 3 locuri.
- Clientul 6 rezervă 9 locuri.
- Clientul 7 rezervă 2 locuri.
- Clientul 8 rezervă 6 locuri.
- Clientul 9 rezervă 7 locuri.
- Clientul 10 rezervă un singur loc.
- Clientul 11 încercă să rezerve un loc însă nu mai sunt locuri disponibile.

Cu următoarele mențiuni: timpul de așteptare între rezervări este aleatoriu între 0.2 și 2 secunde.

c) Afișați starea rezervărilor și numărul de locuri disponibile la finalul simulării.

2. Într-o gară A, există n linii feroviare ce fac legătura cu gara B. Există m trenuri care doresc să călătorească de la A la B. Fiecare tren trebuie să aștepte la intrarea în gara A, până când există o linie liberă pentru a călători spre gara B.

   Implementați o simulare în Python folosind semafoare pentru a gestiona trecerea trenurilor între gara A și gara B. Fiecare tren trebuie să îndeplinească următoarele condiții:

   a. Așteptare la intrarea în gara A:
      Trenurile care ajung trebuie să aștepte până când există o linie liberă la intrarea în gara A.
   b. Ocupare linie și plecare:
      Un tren poate ocupa o linie și să plece spre gara B dacă există o linie disponibilă.
   c. Durata aleatoare a călătoriei:
      Simulați trecerea efectivă de la A la B cu o durată aleatoare.
   d. Număr maxim de trenuri în gara A:
      Asigurați-vă că nu există mai multe trenuri în gara A decât numărul de linii disponibile (n).

Bibliografie

[1] „Python 3.9 Documentation”. https://docs.python.org/3.9/ (data accesării 24 februarie 2023).
[2] L. Diaconu, I. Vasiliev, și M. Barbu, *Procesarea datelor utilizând mediul Python. Lucrări practice*. Galați, România: Editura Fundației Universitare „Dunărea de Jos”, 2022.
[3] „NumPy user guide — NumPy v1.24 Manual”. https://numpy.org/doc/stable/user/index.html (data accesării 8 martie 2023).
[4] „Pandas documentation — Pandas 1.5.3 documentation”. https://pandas.pydata.org/docs/ (data accesării 15 martie 2023).
[5] „Date climatologice de la cele 23 de statii esentiale (flux RBSN) - data.gov.ro”. https://data.gov.ro/dataset/date-climatologice-de-la-cele-23-de-statii-esentiale-pentru-anul-2016 (data accesării 18 martie 2023).
[6] „Users guide — Matplotlib 3.7.1 documentation”. https://matplotlib.org/stable/users/index.html (data accesării 21 martie 2023).
[7] M. Barbu, G. A. Ifrim, A. Burlibașa, și L. Condrachi, *Informatică Aplicată. Lucrări Practice*. Galați, România: Editura Fundației Universitare „Dunărea de Jos”, 2019.
[8] „SymPy 1.11 documentation”. https://docs.sympy.org/latest/index.html (data accesării 30 martie 2023).
[9] „SciPy documentation”. https://docs.scipy.org/doc/scipy/ (data accesării 30 martie 2023).