UNIVERSITATEA TEHNICĂ A MOLDOVEI

BAZELE PROGRAMĂRII CALCULATOARELOR

în limbajul Python

Suport de curs

Chișinău
2025

UNIVERSITATEA TEHNICĂ A MOLDOVEI

FACULTATEA CALCULATOARE, INFORMATICĂ ȘI MICROELECTRONICĂ
DEPARTAMENTUL INFORMATICĂ ȘI INGINERIA SISTEMELOR

BAZELE PROGRAMĂRII CALCULATOARELOR

în limbajul Python

Suport de curs

EDITURA TEHNICA-UTM

2025

CZU 004.434:004.8(075.8)
C 93

Lucrarea a fost discutată și aprobată pentru editare la ședința Consiliului Facultății Calculatoare, Informatică și Microelectronică, proces-verbal nr. 6 din 23 iunie 2025.

Materialul educațional a fost elaborat în conformitate cu curriculumul disciplinei *Bazele programării calculatoarelor* și este destinat studenților Universității Tehnice a Moldovei care studiază la specialitățile în domeniile generale de studii: *021 – Arte; 041 – Științe economice; 042 – Drept; 071 – Inginerie și activități inginerești; 072 – Tehnologii de fabricare și prelucrare; 073 – Arhitectură și construcții; 101 – Servicii publice; 104 – Servicii de transport.*

Autori: dr., lect. univ. Rodica CUJBA
asist. univ. Olesea BOROZAN

Recenzent: dr., conf. univ. Mihail KULEV

DESCRIEREA CIP A CAMEREI NAȚIONALE A CĂRȚII DIN RM

Cujba, Rodica.

Bazele programării calculatoarelor în limbajul Python: Suport de curs / Rodica Cujba, Olesea Borozan; Universitatea Tehnică a Moldovei, Facultatea Calculatoare, Informatică și Microelectronică, Departamentul Informatică și Ingineria Sistemelor.

– Chișinău: Tehnica-UTM, 2025. – 194 p.: fig., tab. color.

Aut. indicați pe verso p. de tit. – Bibliogr.: p. 193-194 (23 tit.). – 50 ex.

ISBN 978-9975-64-557-7

CUPRINS

INTRODUCERE

Trăim într-o eră în care tehnologia modelează fiecare aspect al vieții noastre. De la aplicațiile mobile care ne organizează ziua, la algoritmii care ne recomandă filme sau chiar la sistemele de inteligență artificială care revoluționează medicina, programarea este forța invizibilă din spatele acestor inovații. Nu mai este doar un domeniu rezervat specialiștilor IT, ci o abilitate la fel de importantă ca scrisul sau matematica.

Lumea programării este vastă, iar limbajele de programare sunt instrumentele care permit să interacționăm cu tehnologia. Există multiple limbaje de programare unde fiecare limbaj are un scop specific: **C** și **C++** sunt utilizate pentru dezvoltarea software-ului performant, **Java** este esențial în aplicațiile enterprise, **JavaScript** domină web-ul, iar **R** și **SQL** sunt indispensabile analizei datelor. Printre acestea, **Python** s-a remarcat ca un limbaj extrem de versatil, ușor de învățat și utilizat într-o gamă largă de domenii, de la dezvoltarea web la inteligența artificială.

Acest material educațional a fost elaborat în conformitate cu curriculumul disciplinei *Bazele programării calculatoarelor*" și este destinat studenților Universității Tehnice a Moldovei care studiază la specialitățile din domeniile generale de studii: *021 – Arte; 041 – Ştiinţe economice; 042 – Drept; 071 – Inginerie şi activităţi inginereşti; 072 – Tehnologii de fabricare şi prelucrare; 073 – Arhitectură şi construcţii; 101 – Servicii publice; 104 – Servicii de transport*[1].

Fiecare capitol al materialului este însoțit cu câteva sarcini pentru studenți.

[1] Nomenclatorul domeniilor de formare profesională şi al specialităţilor în învățământul superior aprobat prin HG 482 din 28.06.2017.

1. INIȚIERE ÎN PYTHON

Obiective

La finalul acestui capitol studenții vor fi capabili:

- Să înțeleagă conceptele fundamentale ale limbajului de programare Python și ale modului în care acesta funcționează.
- Să utilizeze un mediu de dezvoltare integrat (IDLE, Jupyter etc.) pentru scrierea, rularea și testarea codului.
- Să recunoască avantajele și limitările limbajului Python în comparație cu alte limbaje de programare.
- Să descrie diferențele dintre compilarea și interpretarea codului și cum influențează acestea execuția programelor.
- Să scrie și să execute scripturi Python simple, utilizând corect indentarea și comentariile.

1.1. Limbajul de programare Python. Noțiuni generale

Limbajul de programare reprezintă un cadru formal compus din expresii și reguli valide, care permite formularea instrucțiunilor destinate unui computer. Asemenea unei limbi vorbite, limbajul de programare oferă o modalitate de a comunica idei și comenzi într-un mod comprehensibil pentru mașini.

Când dorim să executăm un program scris într-un limbaj de programare, există, în esență, două metode principale: **compilarea** și **interpretarea**.

1. **Compilarea** este un proces în care codul sursă, scris de programator, este tradus în întregime într-un cod mașină, într-un fișier executabil. Acest fișier poate fi apoi rulat direct de către sistemul de operare, oferind o performanță superioară în comparație cu metoda interpretării.

2. **Interpretarea**, pe de altă parte, implică traducerea și executarea codului sursă linie cu linie, pe măsură ce programul rulează. Aceasta permite o flexibilitate mai mare și o dezvoltare mai rapidă, dar poate duce la o viteză de execuție mai redusă.

Fiecare metodă are avantajele și dezavantajele sale, iar alegerea dintre compilare și interpretare depinde adesea de cerințele specifice ale proiectului și de mediul în care va fi utilizat programul. În acest fel, limbajul de programare devine un instrument-cheie pentru dezvoltarea software-ului, influențând modul în care gândim și construim aplicații pentru diverse platforme.

Python este un limbaj de programare extins și versatil, ideal atât pentru începători, cât și pentru programatorii experimentați. Unul dintre principalele avantaje ale limbajului Python este sintaxa sa clară și ușor de învățat, care permite dezvoltatorilor să scrie un cod concis și eficient. Python suportă multiple paradigme de programare, inclusiv programarea procedurală, funcțională și orientată pe obiecte, ceea ce oferă flexibilitate în abordarea diferitelor tipuri de proiecte. Biblioteca sa standard bogată și ecosistemul vast de module open-source permit realizarea rapidă a soluțiilor software.

Python este un limbaj de programare care a câștigat o popularitate majora în rândul dezvoltatorilor web. Multe site-uri web cunoscute au fost construite, folosind acest limbaj datorită eficienței sale și a bibliotecilor extensive disponibile. Mai jos, sunt prezentate câteva site-uri web populare care utilizează Python:

a. **YouTube** – platforma de partajare a videoclipurilor a fost dezvoltată în mare parte, folosind Python datorită capacităților sale de manipulare a datelor și a performanței în gestionarea utilizatorilor și a conținutului multimedia.
b. **Google** – cel mai mult utilizat motor de căutare din lume, dar și furnizor de alte servicii populare, nu este scris exclusiv în Python, însă Python este unul dintre limbajele de programare principale utilizate de Google încă de la începuturi. Python este folosit pentru prototipare rapidă, gestionarea infrastructurii și dezvoltarea unor servicii critice (ex.: crawling, indexing).

c. **Instagram** – această rețea socială populară folosește Python pentru backend-ul său, beneficiind de ușurința în utilizare și de rapiditatea cu care dezvoltatorii pot implementa noi funcționalități.
d. **Spotify** – serviciul de streaming muzical a optat pentru Python datorită abilității sale de a gestiona complexitatea și volumul mare de date generate de utilizatori, precum și pentru implementarea rapidă a noilor caracteristici.
e. **Dropbox** – Cloud storage-ul Dropbox a fost dezvoltat inițial folosind Python, deoarece permite o integrare ușoară cu diverse sisteme de operare și oferă un mediu stabil pentru gestionarea fișierelor.
f. **Pinterest** – platforma de descoperire a imaginilor folosește Python pentru a sprijini funcționalitățile complexe de căutare și clasare a imaginilor, facilitând o experiență de utilizator plăcută.

Figura 1.1. Site-uri web populare care utilizează Python

Aceste exemple arată clar puterea și flexibilitatea limbajului Python în dezvoltarea web. Fie că dorim să construim un site simplu sau o aplicație complexă, Python oferă uneltele necesare pentru a transforma ideile în realitate.

Ca orice limbaj de programare, Python are atât avantaje, cât și dezavantaje.

1.1.1. Avantajele Python

- Ușor de învățat și utilizat – sintaxa sa simplă și clară îl face ideal pentru începători.

- Biblioteci și ecosistem bogat – Python are numeroase biblioteci (NumPy, Pandas, TensorFlow, Django etc.) pentru diverse domenii, de la AI la web development.
- Portabilitate și compatibilitate – Python rulează pe diferite sisteme de operare – Windows, macOS și Linux, fără modificări majore ale codului.
- Comunitate mare și suport vast – este susținut de o comunitate activă, cu documentație bogată și forumuri de ajutor.
- Productivitate ridicată – Python permite dezvoltarea rapidă de aplicații datorită codului concis și expresiv.
- Flexibilitate ridicată – poate fi utilizat pentru scripting, machine learning, web development, automatizări etc.
- Integrare cu alte limbaje – Python poate interacționa cu C, C++, Java sau alte limbaje prin API-uri și biblioteci specifice.

1.1.2. Dezavantajele Python

- Viteză mai mică față de limbajele compilate – deoarece este interpretat, Python este mai lent decât C++ sau Java.
- Utilizare mare a memoriei – gestionarea automată a memoriei poate duce la un consum ridicat de resurse.
- Nu este ideal pentru aplicațiile mobile – Python nu este folosit pe scară largă în dezvoltarea de aplicații mobile, unde sunt preferate Java/Kotlin și Swift.
- Probleme de concurență – GIL (Global Interpreter Lock) limitează execuția multi-threading-ului real.
- Dificultăți în distribuirea aplicațiilor – crearea unui executabil independent pentru distribuirea aplicațiilor Python poate fi complicată.

1.1.3. Scurt istoric

Python a fost creat la sfârșitul anilor 1980 de către programatorul olandez Guido van Rossum, ca o succesiune a limbajului ABC. Scopul principal a fost să ofere un limbaj de

programare ușor de învățat, care să permită dezvoltarea rapidă a aplicațiilor.

Implementarea de referință a Python, cunoscută sub numele de CPython, este scrisă în limbajul C. Aceasta servește ca fundament pentru majoritatea versiunilor și distribuțiilor de Python disponibile astăzi, asigurând o compatibilitate și o performanță optimă. Au fost lansate mai multe versiuni, fiecare aducând îmbunătățiri semnificative, culminând cu popularitatea crescută pe care o are în prezent, datorită versatilității sale și a comunității active de dezvoltatori:

- ✈ **1991.** Van Rossum publică versiunea Python 0.9.0 pe alt.sources
- **1994. Python 1.0**, care include programare funcțională (lambda, map, filter, reduce)
- **2000. Python 2** introduce liste comprehensiuni și colectare automată a gunoiului (garbage collection)
- **2008. Python 3** rezolvă probleme fundamentale de design și nu este compatibil cu versiunile anterioare
- **2020. Python 2** ajunge la sfârșitul ciclului de viață, ultima versiune lansată fiind 2.7.18

1.2. Instalarea și analiza mediilor de dezvoltare Python

Mediul de dezvoltare integrat (IDE - Integrated Development Environment) este un set de instrumente software care reunește toate etapele necesare pentru crearea unui program într-o platformă unitară. Acesta oferă, de obicei, o interfață grafică prietenoasă, care facilitează munca programatorului.

Python poate fi utilizat într-o varietate de IDE-uri și editoare de cod, fiecare având avantaje specifice.

Cele mai populare medii de dezvoltare pentru Python:

1. **IDLE** (Integrated Development and Learning Environment – Mediul integrat de dezvoltare și învățare)
   - Avantaje:
     - Vine preinstalat cu Python.

    - Simplu și ușor de utilizat pentru testarea rapidă a codului.
  - Dezavantaje:
    - Lipsa funcționalităților avansate necesare în dezvoltarea complexă.

2. **PyCharm** (Dezvoltat de JetBrains)
  - Avantaje:
    - Oferă instrumente avansate de debugging, testare și refactorizare.
    - Integrare excelentă cu framework-uri precum Django și Flask.
    - Suport pentru controlul versiunilor (Git, SVN).
  - Dezavantaje:
    - Consumul mare de resurse, mai ales în versiunea Professional.
    - Varianta gratuită (Community) are funcționalități limitate.

3. **Jupyter Notebook** (utilizat des în AI și Data Science)
  - Avantaje:
    - Permite rularea codului pe celule, ideal pentru testare și vizualizare de date.
    - Suport pentru bibliotecile populare ca Pandas, NumPy, TensorFlow și altele.
    - Ușor de folosit în cloud (Google Colab).
  - Dezavantaje:
    - Mai puțin eficient pentru proiectele mari sau dezvoltarea software clasică.

4. **Spyder** (Popular în domeniul științific)
  - Avantaje:
    - Integrare nativă cu bibliotecile științifice (NumPy, SciPy, Matplotlib).
    - Interfață asemănătoare cu MATLAB.
    - Funcționalități de debugging și explorare de variabile.
  - Dezavantaje:
    - Nu este la fel de flexibil ca PyCharm pentru dezvoltarea generală.

1.2.1. Mediul integrat de dezvoltare și învățare IDLE

Pentru instalarea IDLE urmați pașii descriși în tabelul 1.1.

Tabelul 1.1. **Pașii de instalare IDLE pentru Python**

Nr.	***Pasul***	***Descriere***
1	Descărcarea Python	Pentru a începe să lucrăm cu Python, mai întâi trebuie să-l instalăm pe calculatorul personal. Python poate fi descărcat de pe site-ul oficial: https://www.python.org/downloads/.
2	Instalarea Python	Când fișierul a fost descărcat, îl executăm pentru a iniția procesul de instalare. Urmăm pașii asistați de instalator: - Ne asigurăm că bifăm opțiunea „Add Python to PATH" (Adaugă Python în variabilele de mediu) pentru a facilita executarea comenzilor Python din orice fereastră de terminal. - Continuăm instalarea standard, acceptând termenii și condițiile necesare.
3	Verificarea instalării	După finalizarea instalării, deschidem Command Prompt (Linia de comandă) pe Windows sau Terminal pe macOS/Linux. Tastăm următoarea comandă pentru a verifica dacă Python a fost instalat corect: ```**bash** **python–version** ``` Dacă instalarea a fost reușită, vom vedea versiunea Python instalată.

Ce va instala Python?

Pe lângă interpretorul Python (programul care execută codul, linie cu linie) și Mediul integrat de dezvoltare și învățare (IDLE) vor fi instalate și următoarele:

- **pip**: Managerul de pachete care permite să instalăm biblioteci și module externe necesare pentru dezvoltarea aplicațiilor.

- Documentația utilă ce ajuta la învățarea și la explorarea funcționalităților limbajului Python.
- Asocieri de fișiere astfel, încât fișierele cu extensia ".py" să fie deschise automat cu interpretorul Python când le accesam.

Python poate fi utilizat în două moduri distincte:

1. Modul interactiv (shell): în acest mod, utilizatorul poate interacționa direct cu interpretorul Python prin intermediul tastaturii. Când se introduce o comandă, aceasta este evaluată imediat, iar rezultatul este afișat pe loc. Această interactivitate ciclică permite utilizatorilor să experimenteze rapid și eficient cu diferite instrucțiuni Python, fiind ideală pentru testarea codului și învățare.
2. Modul script: în acest mod, programatorul poate scrie un set de comenzi Python într-un fișier denumit script. Când scriptul este salvat, acesta poate fi executat ulterior pentru a rula programul complet. Această abordare este mai potrivită pentru proiectele mai mari și complexe, oferind o modalitate organizată de a gestiona codul sursă.

1. Deschiderea IDLE

Primul pas în utilizarea IDLE este deschiderea mediului de dezvoltare. Acest mediu este de o importanță majoră pentru scrierea, testarea și rularea codului Python. Pentru a deschide IDLE, tastați în câmpul de căutare din sistemul de operare cuvântul IDLE. Faceți click pe aplicația **IDLE (Python <versiunea>)**:

Figura 1.2. Deschiderea IDLE

2. *Fereastra IDLE*

Când IDLE este deschis, vom observa o fereastră principală, numită **Shell**. În aceasta putem introduce comenzi Python direct și vom obține instantaneu feedback:

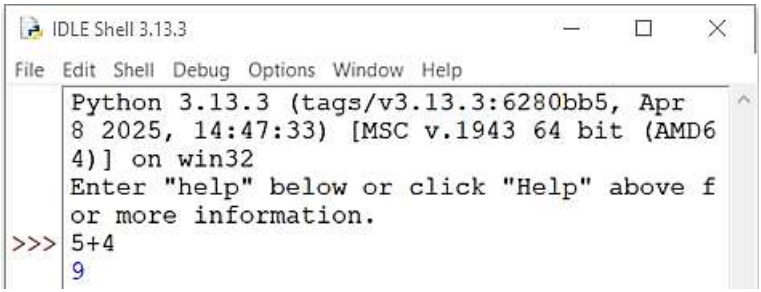

Figura 1.3. Fereastra IDLE Shell

3. *Scrierea scriptului*

Pentru a crea un script, apelăm meniul „File” și selectăm „New File”. Așa vom deschide o nouă fereastră de editare. Aici putem începe să scriem codul. Este indicat să organizăm codul într-o manieră clară, folosind comentarii pentru a explica ceea ce face fiecare parte a scriptului:

Figura 1.4. Scrierea scriptului în fișier

4. *Salvarea scriptului*

După ce am terminat de scris, nu uitam să salvăm scriptul! Apelăm la meniul „File” și alegem „Save” sau „Save As”. Ne vom asigura că alegem un nume descriptiv și îl vom salva cu extensia .py, pentru a fi recunoscut ca un script Python:

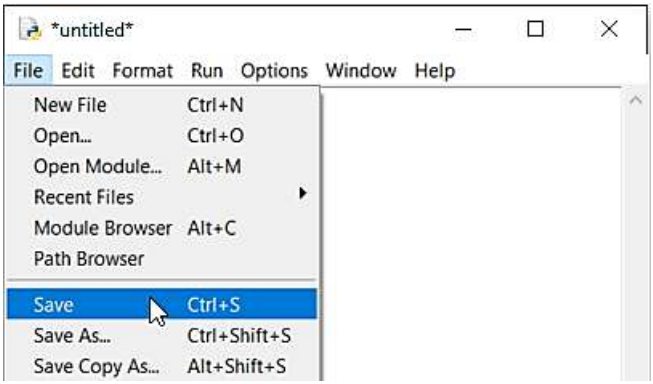

Figura 1.5. Salvarea scriptului în fișier

5. Executarea scriptului

Pentru a rula scriptul, ne vom asigura că este salvat, găsim meniul „Run” și selectăm „Run Module” sau apăsăm tasta F5. Vom vedea rezultatele execuției afișate în fereastra **Shell**:

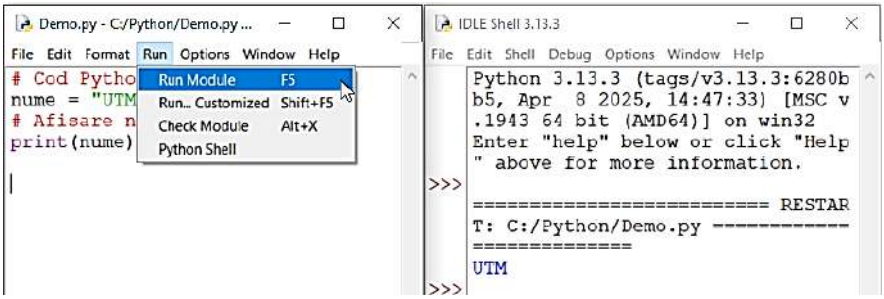

Figura 1.6. Executarea scriptului

Efectuând aceste etape, am învățat cum să deschidem IDLE, să scriem, să salvăm și să executăm un script Python. Aceste cunoștințe sunt importante pentru a învăța programarea și a dezvolta aplicații folosind Python.

1.2.2. Pip

pip – Python Install Packages (managerul de pachete Python), reprezintă un sistem eficient de gestionare a pachetelor, dezvoltat în limbajul Python, care facilitează instalarea și administrarea

pachetelor software. Acest instrument este important pentru programatori și dezvoltatori, deoarece permite integrarea rapidă a bibliotecilor și modulelor necesare în cadrul proiectelor lor.

Unul dintre avantajele de bază ale utilizării **pip** este interfața sa prietenoasă de linie de comandă. Datorită acesteia, instalarea pachetelor software Python devine o sarcină simplă, comparabilă cu emiterea unei comenzi. Acest lucru înseamnă că, indiferent de nivelul de experiență al utilizatorului, procesul de gestionare a pachetelor software este accesibil și eficient.

Prin simpla executare a comenzilor, utilizatorii pot instala, actualiza sau dezinstala pachete, economisind astfel timp prețios și eliminând complicațiile ce ar putea apărea în lipsa unor instrumente adecvate. De exemplu, pentru a instala un pachet, utilizatorul trebuie să deschidă linia de comandă (scriind în câmpul de căutare cmd) și să scrie:

```
pip install <nume_pachet>
```

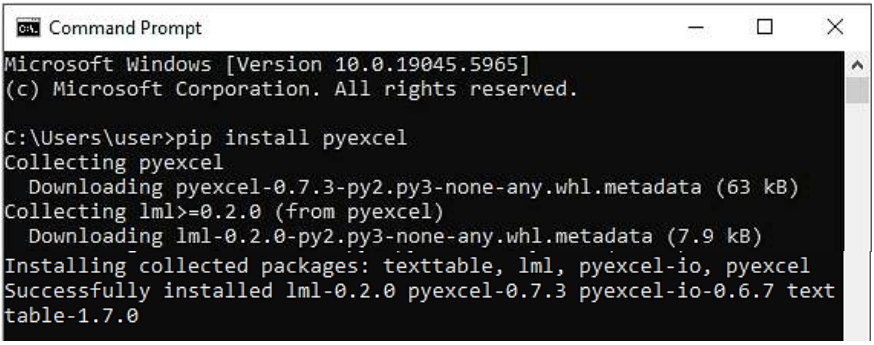

Figura 1.7. Managerul de pachete pip

Această comandă acționează ca un rezolvator de probleme, permițând dezvoltatorilor să se concentreze mai mult asupra codului și mai puțin asupra gestionării dependențelor. În concluzie, **pip** nu doar că simplifică procesul de gestionare a pachetelor, ci devine și un aliat de nădejde pentru toți cei care își doresc să dezvolte aplicații eficiente în Python.

1.2.3. Mediul de dezvoltare online Jupyter

Jupyter este un mediu interactiv de programare care permite dezvoltarea codului într-un mod eficient și colaborativ. Acesta este ideal pentru studii și experimente în domeniul științelor exacte, dat fiind faptul că suportă multiple limbaje de programare. Mediul de dezvoltare Jupiter poate fi descărcat la adresa: https://jupyter.org/. După descărcarea și instalarea mediului Jupiter, acesta se va deschide în browser, unde vom selecta un Notebook.

Pentru a demonstra puterea și simplitatea acestui mediu, mai jos expunem un scurt exemplu de cod Python:

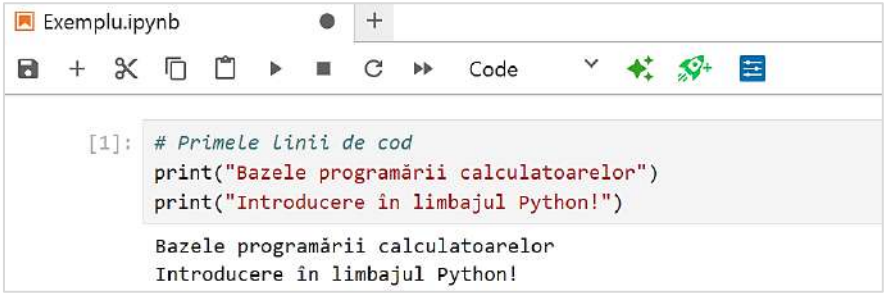

Figura 1.8. Exemplu de cod Python în Jupyter

Notă! În prezentul suport de curs exemplele de cod sunt realizate în mediul de dezvoltare Jupyter, iar rezultatul rulării scriptului este prezentat sub celula de cod.

Rularea unui program

După alegerea unui mediu de dezvoltare Python și instalarea acestuia, suntem gata să rulăm primul program. Deschidem mediul ales, scriem un cod simplu cum ar fi **print("Salut, lume!")** și observam rezultatul. Acesta este un pas important pentru orice programator: învățarea prin practică.

- **Interpretorul Python** – este un instrument foarte important pentru programatori, deoarece permite rularea și interpretarea programelor care sunt scrise în acest limbaj. Acesta transformă codul sursă scris de programatori în instrucțiuni pe care calculatorul le poate înțelege și executa. Interpretoarele sunt

utilitare și asigură o execuție rapidă a codului, ajutând programatorii să testeze și să depaneze aplicațiile mai eficient.

- **Bibliotecile Python** – sunt un alt aspect fundamental al acestui limbaj. Acestea sunt colecții de funcții, module și tipuri de date care extind capabilitățile Python-ului. Scrise de alți programatori, bibliotecile permit reutilizarea codului și simplificarea procesului de dezvoltare. Prin integrarea acestor biblioteci, programatorii pot economisi timp și pot beneficia de soluții deja testate pentru problemele lor.

1.3. Elementele de bază ale unui program Python

Când începem să învățam programarea în Python, este imprtant să înțelegem câteva concepte fundamentale care formează baza codului.

Iată cele mai importante elemente:

- **Comentariile**. Comentariile sunt părți ale codului care nu sunt executate de program, dar care oferă explicații sau note utile pentru cititor.
- **Identificatorii**. Identificatorii sunt segmente de caractere pe care le folosim pentru a denumi variabile, funcții, clase, module și alte entități. Un identificator trebuie să înceapă cu o literă (A-Z, a-z) sau cu un underscore (_) și poate fi urmat de litere, cifre (0-9) sau underscore. De exemplu, “var1”, “_valoare” și “functie_a” sunt identificatori valizi.
- **Literalii**. Literalii sunt reprezentări directe ale valorilor constante în cod. Acestea pot fi de mai multe tipuri, inclusiv:
  - literali numerici: cum ar fi `10`, `-5`, `3.14`;
  - literali de șiruri: texte incluse între apostrofuri sau ghilimele, precum `'Salut'` sau `"Python"`;
  - literali booleeni: valorile `True` și `False`.

1.3.1. Comentariile în Python

Comentariile sunt importante în procesul de programare, deoarece oferă claritate și context pentru codul scris. Acestea pot ajuta atât dezvoltatorul care scrie codul, cât și pe cei care îl vor citi ulterior.

În Python, comentariile pot fi de două tipuri:

1. **Comentarii pe o singură linie**: comentariile pe o singură linie sunt cele mai simple și sunt precedate de caracterul #. Tot ce urmează după # pe aceeași linie este considerat comentariu și nu este executat de interpretor:

```
# Acesta este un comentariu pe o singură linie
print("Hello, World!")  # Afișează mesajul "Hello, World!"
```

```
Hello, World!
```

2. **Comentarii pe mai multe linii (comentarii de bloc)**: acestea sunt delimitate de trei apostrofuri sau trei perechi de ghilimele (''' sau """) și pot să cuprindă multiple rânduri, până la un nou set de apostrofuri:

```
'''
Acesta este un comentariu pe mai multe linii.
Poți scrie orice text aici, iar Python îl va ignora.
'''
print("Hello, World!")
```

```
Hello, World!
```

1.3.2. Indentarea

Indentarea este o tehnică fundamentală în programarea Python, care se referă la alinierea și plasarea liniilor de cod astfel, încât să fie ușor de parcurs și de citit. Aceasta nu este doar o chestiune de estetică, ci joacă un rol crucial în structurarea codului.

În limbajul Python, nivelul de indentare are o semnificație sintactică profundă. Spre deosebire de alte limbaje de programare, care utilizează construcții precum "begin ... end" sau acoladele "{...}" pentru a delimita blocuri de cod, Python se bazează exclusiv pe indentare pentru a defini scopurile.

Indentarea corectă îmbunătățește nu doar vizibilitatea codului, ci și modul în care Python interpretează instrucțiunile. Fiecare bloc de cod (funcțiile, buclele, condițiile etc.) trebuie să fie indentat uniform pentru a distinge care instrucțiuni aparțin fiecărui bloc.

Importanța indentării:

- **Claritatea.** O indentare corectă face ca programul să fie mult mai ușor de înțeles, atât pentru dezvoltatorul care îl scrie, cât și pentru cei care îl citesc ulterior.
- **Structurarea codului.** Indentarea ajută la organizarea logică a instrucțiunilor dintr-un bloc de cod, ceea ce facilitează identificarea funcțiilor sau a instrucțiunilor care aparțin aceluiași grup. În cazul indentării greșite, interpretorul nu va putea executa corect scriptul.

Pentru a ilustra acest lucru, vom analiza următorul exemplu:

```
x = 5
if x > 0:
    print("Numărul e pozitiv.", "Această linie este indentată")
print("Această linie NU este indentată")
```

```
Numărul e pozitiv. Această linie este indentată
Această linie NU este indentată
```

În acest exemplu, toate instrucțiunile din cadrul blocului condițional **`if`** sunt indentate la același nivel. Aceasta denotă că ele fac parte din aceeași structură logică. În Python, standardul de indentare este de 4 spații pentru fiecare nivel de indentare. Acesta este recomandat de ghidul oficial de stil pentru Python. În majoritatea mediilor de dezvoltare, pentru indentarea codului este utilizată tasta Tab (tabulare). O tabulare reprezintă un nivel de indentare.

Dăm un exemplu simplu care ilustrează o eroare comună cauzată de indentarea incorectă:

```
# incorect
def salut():
print("Bun venit!")
```

```
  Cell In[6], line 3
    print("Bun venit!")
    ^
IndentationError: expected an indented block after functi
on definition on line 2
```

În acest caz, linia **`print("Bun venit!")`** nu este indentată corespunzător. Codul corectat ar trebui să arate astfel:

```
# corect
def salut():
    print("Bun venit!")
```

1.4. Operatorii aritmetici în Python

În programare, operatorii aritmetici sunt importanți pentru realizarea calculelor matematice. Acești operatori permit să efectuam diverse operațiuni asupra numerelor, fie ele întregi sau zecimale. Tabelul 1.2 include o prezentare detaliată a celor mai utilizați operatori aritmetici în Python:

Tabelul 1.2. Operatorii aritmetici

Simbol Python	***Operație***	***Descriere, exemplu***
`+`	Adunare	Utilizat pentru adăugarea valorilor. Exemplu: **`a + b`** va returna suma lui **`a`** și **`b`**.
`-`	Scădere	Folosit pentru a scădea o valoare din alta. Exemplu: **`a - b`** va returna diferența dintre **`a`** și **`b`**.
`*`	Înmulțire	Permite înmulțirea a două sau mai multe valori. Exemplu: **`a * b`** va returna produsul lui **`a`** și **`b`**.
`/`	Împărțire	Este folosit pentru a împărți o valoare la alta. Exemplu: **`a / b`** va returna câtul lui **`a`** împărțit la **`b`**. *Este important de menționat că în cazul în care* **`b`** *este* **`0`***, se va genera o eroare de divizare la zero.*
`**`	Ridicare la putere	Este folosit pentru a ridica un număr la o anumită putere. Este echivalent cu operația matematică de exponențiere. Exemplu: **`a ** b`** va returna valoarea lui **`a`** ridicată la puterea **`b`**.

Simbol Python	*Operaţie*	*Descriere, exemplu*
`%`	Restul împărțirii întregi	Returnează restul obținut după împărțirea a două numere. Acesta este adesea folosit pentru a verifica dacă un număr este par sau impar. Exemplu: **`a % b`** va returna restul împărțirii întregi lui **`a`** la **`b`**.
`//`	Împărțirea întreagă fără partea fracționară	Returnează câtul întreg al împărțirii a două valori, eliminând orice rest. Exemplu: **`a // b`** va returna partea întreagă a împărțirii lui **`a`** la **`b`**.

1.4.1. Ordinea (prioritatea) operațiunilor

Regulile de prioritate sunt următoarele:

1. Operațiunile aritmetice se execută de la stânga la dreapta (left-sided-binding). Excepție – operațiunea de ridicare la putere, care se execută de la dreapta la stânga (right-sided-binding).
2. Operațiile de înmulțire și împărțire sunt executate înaintea celor de adunare și scădere.
3. Dacă există paranteze, operațiile din interiorul acestora vor fi efectuate primele.

1.4.2. Matematica în Python

În continuare vom analiza câteva dintre cele mai comune operații matematice pe care le putem realiza utilizând Python.

1. Adunarea

Adunarea este una dintre cele mai fundamentale operații matematice. În Python, putem aduna două sau mai multe numere folosind operatorul “+”.

```
a = 5
b = 3
suma = a + b
print("Suma este:", suma)
```

```
Suma este: 8
```

2. *Scăderea*

Scăderea permite să găsim diferența dintre două numere. În Python, folosim operatorul "-" pentru a realiza această operație.

```
a = 10
b = 4
diferenta = a - b
print("Diferența este:", diferenta)
```

```
Diferența este: 6
```

3. *Înmulțirea*

Înmulțirea se face cu ajutorul operatorului "*". Este o operație de bază pe care o vom utiliza frecvent în matematică.

```
a = 6
b = 7
produs = a * b
print("Produsul este:", produs)
```

```
Produsul este: 42
```

4. *Împărțirea*

Pentru a împărți un număr la altul, utilizăm operatorul "/". Acesta va returna un număr zecimal (float).

```
a = 8
b = 2
diviziune = a / b
print("Rezultatul împărțirii este:", diviziune)
```

```
Rezultatul împărțirii este: 4.0
```

5. *Ridicarea la putere*

Pentru a ridica un număr la o putere, folosim operatorul "**". Aceasta este o operație utilizată frecvent în matematică și programare.

```
a = 2
putere = 3
rezultat = a ** putere
print("Rezultatul lui 2 la puterea 3 este:", rezultat)
```

```
Rezultatul lui 2 la puterea 3 este: 8
```

6. Restul împărțirii

Pentru a obține restul împărțirii a două numere, folosim operatorul "%". Aceasta este o operație foarte utilă în programare, mai ales când lucrăm cu numere întregi.

```
a = 10
b = 3
rest = a % b
print("Restul împărțirii lui 10 la 3 este:", rest)
```

```
Restul împărțirii lui 10 la 3 este: 1
```

7. Împărțirea întreagă

Operatorul "//" realizează o împărțire întreagă, adică returnează doar partea întreagă a rezultatului fără partea fracționară.

```
a = 10
b = 4
rezultat = a // b
print("Rezultatul impartirii intregi lui 10 la 4 este",
      rezultat)
```

```
Rezultatul impartirii intregi lui 10 la 4 este 2
```

1.4.3. Notația științifică

În Python, **notația științifică** este utilizată pentru a reprezenta numere foarte mari sau foarte mici într-un mod compact, folosind **notația exponențială**. Aceasta este utilă în domenii precum fizica, chimia, finanțele, analiza de date etc.

Un număr scris în notație științifică folosește litera **„e”** sau **„E”**, care reprezintă baza **10** ridicată la o anumită putere.

```
num1 = 3.5e4  # Echivalent cu 3.5 × 10^4 = 35000.0
num2 = 2.1E-3 # Echivalent cu 2.1 × 10^-3 = 0.0021

print(num1)  # 35000.0
print(num2)  # 0.0021
```

```
35000.0
0.0021
```

1.5. Cuvinte-cheie în Python

În Python, cuvintele-cheie (sau “keywords”) sunt cuvinte rezervate, care au semnificații speciale și nu pot fi utilizate pentru a denumi variabile, funcții, clase sau alte elemente în program. Ele sunt folosite pentru a exprima sintaxa și structura limbajului Python. Aceste cuvinte-cheie sunt rezervate și joacă un rol important în sintaxa Python, având funcții specifice în cadrul limbajului.

Pentru a obține lista cuvintelor-cheie din limbajul de programare Python, putem utiliza sistemul de ajutor incorporat, prin scrierea instrucțiunii **`help(keywords):`**

```
help('keywords')
```

```
Here is a list of the Python keywords.  Enter any keyword to get m
ore help.

False               class               from                or
None                continue            global              pass
True                def                 if                  raise
and                 del                 import              return
as                  elif                in                  try
assert              else                is                  while
async               except              lambda              with
await               finally             nonlocal            yield
break               for                 not
```

În concluzie, ca și alte limbaje de programare, Pyhon are atât avantaje, cât și dezavantaje. Comentariile și indentarea sunt elemente de bază, primele fiind importante pentru claritatea codului, iar cele din urmă delimitează blocurile de cod. Operatorii aritmetici sunt utilizați în orice aplicație care lucrează cu date numerice, iar cuvintele-cheie rezervate formează structura logică a limbajului, controlând fluxul programului, declarațiile și structurile din program.

Sarcini

1. **Evaluarea unei expresii aritmetice.** Scrieți un program care efectuează urmatoarele operații aritmetice:

$$\left(\frac{3+7}{2*3}\right)^2 + \left(\frac{1}{5+6^2}\right)$$

2. **Determinarea vitezei unui automobil.** Un vehicul se deplasează pe o distanță dată într-un anumit interval de timp. Cunoscând lungimea traseului (exprimată în kilometri) și durata deplasării (în ore), scrieți un program care calculează viteza medie de deplasare exprimată în metri pe secundă și afișează rezultatul pe ecran.
3. **Calculul perimetrului și al ariei unui triunghi.** Fie a, b și c trei valori reale, care reprezintă lungimile laturilor unui triunghi. Scrieți un program care calculează și afișează perimetrul și aria triunghiului, apoi afișează rezultatele într-un mod clar și bine structurat.

2. VARIABILE, ATRIBUIRI, TIPURI DE DATE, CONVERSIA DATELOR

Obiective

La finalul acestui capitol, studenții vor fi capabili:

- Să înțeleagă rolul și funcționarea variabilelor în programarea Python, precum și regulile de denumire și atribuirea valorilor acestora.
- Să identifice și să utilizeze corect tipurile de date fundamentale din Python.
- Să aplice operatorii de atribuire, relaționali și logici în scopul construirii de expresii și condiții valide în cadrul programelor.
- Să efectueze conversii explicite și implicite ale tipurilor de date, pentru a asigura compatibilitatea în operații și afișarea corectă a rezultatelor.
- Să manipuleze șiruri de caractere folosind metode și funcții specifice, incluzând concatenarea, repetarea, indexarea, slicing-ul și formatarea textului.
- Să utilizeze funcțiile predefinite `print()` și `input()` pentru a realiza interacțiuni de bază cu utilizatorul într-un program Python.

2.1. Variabile în Python

În cadrul programării, o variabilă poate fi considerată un element-cheie, reprezentând o zonă rezervată în memorie, în care putem stoca informații.

Variabilele permit să păstrăm valori temporare, deoarece informațiile stocate în ele folosesc memoria RAM (Memorie cu Acces Aleator), care, trebuie menționat, necesită energie electrică pentru a funcționa. Astfel, tot ceea ce se află în memoria RAM este temporar și va fi pierdut când aplicația se oprește sau computerul este închis.

În Python, procesul de creare a unei variabile devine activ când îi atribuim pentru prima dată o valoare. De exemplu, pentru a crea o variabilă numită "*universitate*" și a-i atribui valoarea "*Universitatea Tehnică a Moldovei*", vom scrie:

```
# declarăm o variabilă
universitate = "Universitatea Tehnică a Moldovei"
```

Aceasta linie de cod creează variabila **`universitate`** pe care o putem folosi ulterior în programul nostru.

Astfel, gestionarea variabilelor devine un instrument fundamental în procesul de programare, permițând să manipulăm datele și să dezvoltăm soluții eficiente a problemelor cu care ne confruntăm.

2.1.1. Caracteristicile variabilelor în programare

Fiecare variabilă are anumite caracteristici fundamentale, pe care le vom explica mai pe larg:

- **Numele**. Fiecare variabilă are un nume distinct, care o identifică. Numele trebuie să fie relevant și să reflecte rolul pe care variabila îl joacă în program.
- **Tipul**. Fiecare variabilă are un tip, care este determinat de mașina virtuală (MV). Tipul definește natura datelor stocate în variabilă și modul în care trebuie gestionate aceste date.
- **Spațiul de memorie**. Variabilele ocupă un singur spațiu de memorie, care este alocat în funcție de tipul variabilei. Acest spațiu este important pentru stocarea valorii variabilei și pentru gestionarea eficientă a resurselor de calcul.
- **Valoarea**. Fiecare variabilă are o valoare specifică asociată. Această valoare poate fi modificată pe parcursul execuției programului, iar înțelegerea modului în care se schimbă este foarte importantă pentru logica programului.

2.1.2. Numele variabilelor și reguli de sintaxă

Când lucrăm cu variabile în programare, trebuie să ținem cont de anumite reguli pentru a ne asigura că codul nostru este funcțional

și ușor de înțeles. Iată câteva dintre cele mai importante reguli referitoare la numele variabilelor:

- **Numele variabilei trebuie să fie unic**. Fiecare variabilă pe care o creăm trebuie să aibă un nume diferit pentru a evita confuziile. De exemplu, nu putem avea două variabile numite „vârsta” în același context, deoarece acest lucru ar duce la erori de rulare.
- **Începerea numelui**. Numele unei variabile trebuie să înceapă întotdeauna cu o literă sau cu caracterul subliniere „_”. Astfel, putem avea variabile precum „nume”, „_valoare”, dar nu putem începe cu un număr.
- **Caractere acceptate**. Numele variabilelor pot conține doar litere (de la A la Z atât majuscule, cât și minuscule), numere (0-9) și sublinierea „ _ ”. De exemplu, „nivel1”, „_student” și „data_nasterii” sunt toate valide.
- **Sensibilitatea la majuscule și minuscule**. Este foarte important să reținem că în Python numele variabilelor sunt sensibile la majuscule și minuscule (case sensitive). Aceasta înseamnă că „nume”, „nUme” și „NUME” sunt considerate trei variabile diferite. Prin urmare, este recomandat să fim consecvenți în utilizarea literelor mari și mici pentru a evita confuziile.

Reamintim că nu toate cuvintele pot fi folosite pentru e declara o variabilă, cuvintele-cheie rezervate (vezi § 1.5) nu pot fi utilizate.

2.2. Atribuirea în Python

2.2.1. Atribuirea simplă

În Python, atribuirea variabilelor se realizează prin intermediul operatorului egal „=”. Acest operator permite asocierea unei valori cu o variabilă, facilitând astfel manipularea ulterioară a acesteia în program.

```
# <variabila> = <valoare>
numar = 5
```

```
numar
```

```
5
```

Un aspect interesant al limbajului Python este capacitatea de a efectua atribuiri simple pentru mai multe variabile simultan. De exemplu, dacă dorim să atribuim aceeași valoare mai multor variabile, putem face acest lucru într-o singură linie de cod.

```
numar_1 = numar_2 = numar_3 = 5
print (numar_1, ";", numar_2, ";" ,numar_3)
```

```
5 ; 5 ; 5
```

2.2.2. Atribuirea multiplă

Python oferă conceptul de atribuire multiplă – posibilitatea de a asigna valorile unor variabile multiple într-o singură linie de cod, ceea ce îmbunătățește eficiența și claritatea programului. Variabilele și valorile sunt separate prin virgule. Iată un exemplu care ilustrează acest concept:

```
a, b, c = 1, 2, 3
print(a, b, c)
```

```
1 2 3
```

În acest exemplu, variabila **`a`** va lua valoarea **`1`**, variabila **`b`** va lua valoarea **`2`**, iar variabila **`c`** va lua valoarea **`3`**. Prin urmare, este posibil să atribuim mai multe variabile în același timp, ceea ce poate face codul mai concis și mai ușor de înțeles.

2.2.3. Atribuirea expresiilor

Expresiile în Python sunt evaluate automat, ceea ce înseamnă că limbajul interpretat „înțelege” și calculează rezultatul acestor

expresii fără a fi nevoie de instrucțiuni suplimentare. Acest proces permite să atribuim valorile calculate unor variabile, simplificând astfel gestionarea datelor în programele noastre.

De exemplu, putem utiliza expresii care implică variabile deja inițializate cu valori. Aceasta oferă flexibilitate în modurile în care putem manipula și utiliza datele:

```
# Inițializăm două variabile
a = 5
b = 3
# Atribuim o expresie care implică variabilele existente
c = a + b  # c va deveni 8
# Putem folosi expresii și în alte contexte
d = a * b  # d va deveni 15
print(d)
```

```
15
```

În Python, există o serie de operatori speciali de atribuire care vor ajuta să simplificăm scrierea codului și să facem operațiile matematice mai eficiente. Acești operatori nu doar că îndeplinesc atribuirea, ci și efectuează o operație aritmetică simultan. În tabelul de mai jos este o prezentare a fiecărui operator special de atribuire:

Tabelul 2.1. Operatori speciali de atribuire

Operator	***Exemplu***
+=	Expresia **x += 5** este echivalentă cu **x = x + 5**
-=	Expresia **y -= 3** este echivalentă cu **y = y - 3**
/=	Expresia **z /= 4** este echivalentă cu **z = z / 4**
***=**	Expresia **a *= 2** este echivalentă cu **a = a * 2**
****=**	Expresia **b **= 3** este echivalentă cu **b = b ** 3**
%=	Expresia **c %= 4** este echivalentă cu **c = c % 4**
//=	Expresia **d //= 2** este echivalentă cu **d = d // 2**

2.3. Tipuri de date în Python

În programare, un tip de date reprezintă caracteristica fundamentală a unei variabile, deoarece definirea corectă a tipului de date permite programului să știe cum să interpreteze și să manipuleze informațiile stocate.

Un aspect interesant al variabilelor este că, spre deosebire de multe alte limbaje de programare, în unele medii, inclusiv Python, acestea nu necesită o declarație strictă a tipului de date. Aceasta înseamnă că variabilele pot schimba tipul de date după ce au fost inițial setate. Această flexibilitate poate fi un avantaj în anumite situații, dar și o provocare, deoarece poate duce la erori neașteptate în execuția programului.

De asemenea, este important de reținut că, în funcție de tipul variabilei, operația de adunare va fi executată diferit. De exemplu, dacă adunăm două variabile de tip numeric, rezultatul va fi suma acestor numere. Însă, dacă încercăm să adunăm două șiruri de caractere (stringuri), rezultatul va fi concatenarea acestora, adică vor fi unite într-un singur șir. Prin urmare, este important să înțelegem diferitele tipuri de date pe care le putem utiliza. Aceste tipuri de date ajută să organizăm și să gestionăm informațiile în mod eficient.

Principalele tipuri de date în Python sunt reprezentate în figura 2.1.

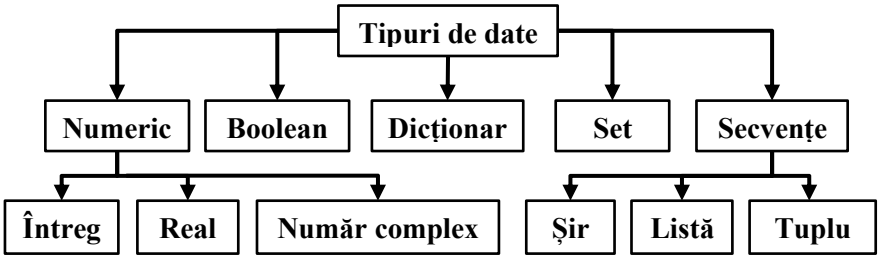

Figura 2.1. Principalele tipuri de date în Python

2.3.1. Numeric

În Python, toate valorile numerice pot fi clasificate în mai multe subtipuri.

Întreg (int)

Aceasta este o valoare numerică care nu conține zecimale. Este utilizată frecvent pentru a reprezenta numerele întregi pozitive, negative sau zero.

```
a = 10   # număr întreg pozitiv
b = -5   # număr întreg negativ
c = 0    # zero
print(type(a))
```

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

Real (`float`)

Acest tip de date cuprinde numerele raționale, adică numere care pot avea parte zecimală. Sunt utilizate când avem nevoie de precizie în calcule. Tipul **float** folosește precizia IEEE 754 double-precision.

```
x = 3.14
y = -0.001
z = 2.0  # tot float, chiar dacă pare întreg
print(type(x))
#Operațiile cu float pot duce la erori de precizie!
# Nu va da exact 0.3, ci 0.30000000000000004
print(0.1 + 0.2)
```

```
<class 'float'>
0.30000000000000004
```

Număr Complex (`complex`)

Acest tip de date este format dintr-o parte reală și o parte imaginară, reprezentate sub forma **`a + bj`**, unde **`j`** este unitatea imaginară **`sqrt(-1)`**. Este utilizat în matematică și inginerie, în special în analiza semnalelor și în calculul numeric.

```
c1 = 2 + 3j
c2 = 1 - 1j
print(c1.real)  # Partea reală: 2.0
print(c1.imag)  # Partea imaginară: 3.0
c3 = c1 + c2
print(c3)
```

```
2.0
3.0
(3+2j)
```

2.3.2. Boolean (bool)

Tipul de date logic sau boolean poate avea două valori posibile: **`True`** (adevărat) sau **`False`** (fals). Este deosebit de important în structurile de decizie. În Python, valorile booleene **`True`** și **`False`** sunt, de fapt, subclase ale tipului de date **`int`**. De aceea, **`True`** este echivalent cu **`1`**, iar **`False`** este echivalent cu **`0`.**

```
val1 = True
val2 = False
print(val1 , val2)
```

```
True False
```

```
#Booleanii sunt esențiali pentru structuri condiționale:
if val1:
    print("Este adevărat!")
else:
    print("Este fals!")
```

```
Este adevărat!
```

Operatori relaționali (de comparație)

Operatorii relaționali (tabelul 2.2) sunt utilizați pentru a face comparații între două valori. Rezultatul unei astfel de comparații este o valoare de tip boolean, adică poate fi **`True`** (adevărat) sau **`False`** (fals).

Tabelul 2.2. Operatori relaționali

Operator	*Exemplu*	*Semnificatia*
`>`	`a > b`	**a** mai mare ca **b**
`<`	`a < b`	**a** mai mic ca **b**
`>=`	`a >= b`	**a** mai mare sau egal cu **b**
`<=`	`a <= b`	**a** mai mic sau egal cu **b**
`!=`	`a != b`	**a** nu este egal cu **b**
`==`	`a == b`	**a** egal cu **b**

Regulile de prioritate ale operațiilor de comparație sunt următoarele:

1. Operațiile de comparație == și != au prioritate mai mare în comparație cu celelalte operații și sunt executate în primul rând.
2. Dacă avem paranteze, operațiile din interiorul acestora vor fi efectuate primele.

Prin utilizarea acestor operatori, putem dezvolta logica programelor, permițându-le să ia decizii bazate pe condiții specifice. Câteva exemple sunt prezentate mai jos:

```
print(1 == 1)
```

```
True
```

```
print(1 != 1)
```

```
False
```

```
print(5 > 10)
```

```
False
```

Operatori logici

Operatorii logici (**`not`**, **`and`** și **`or`**) sunt folosiți pentru a combina două sau mai multe expresii logice. Aceștia ajută la luarea unor decizii mai complexe în programare, returnând valori de tip boolean: **`True`** sau **`False`**.

În tabelul 2.3 sunt incluse operațiile logice.

Tabelul 2.3. Operațiile logice

A	B	not A	A and B	A or B
False	False	True	False	False
False	True	True	False	True
True	False	False	False	True
True	True	False	True	True

Dacă într-o expresie se întâlnesc mai mulți operatori logici, aceștia se execută în funcție de prioritate. Cea mai mare prioritate are operatorul logic **not**, apoi urmează **and** și cea mai mică prioritate are operatorul **or**:

```
print(not False and True or False)
```

```
True
```

Aici, evaluăm **not False** → devine **True**. Evaluăm **True and True** → rezultatul este **True**. Evaluăm **True or False** → rezultatul este **True**.

Deseori, operatorii logici sunt utilizați în combinație cu operatorii de comparație:

```
print(5 > 0 and 5 < 10)
```

```
True
```

```
print(5 < 0 or 5 == 5)
```

```
True
```

```
print((5 == 5) == (not False))
```

```
True
```

Importanța operatorilor logici

- Permit să luăm decizii complexe, bazate pe mai multe condiții în același timp.
- Folosim operatorii logici pentru a combina verificări multiple, astfel încât programele noastre să se comporte corect în funcție de diferite situații.
- Acești operatori ajută să controlăm mai bine fluxul unui program, luând decizii inteligente în funcție de mai mulți factori.
- Sunt importanți pentru că fără ei nu am putea verifica mai multe condiții în același timp, ceea ce ar limita funcționalitatea aplicațiilor.
- Operatorii logici adaugă flexibilitate codului – putem crea condiții mai precise și mai puternice, în funcție de ce ne dorim ca programul să facă.

2.3.3. Secvențe

În termeni generali, o secvență este o colecție ordonată de elemente. În majoritatea limbajelor de programare, secvențele includ șiruri de caractere, liste și tupluri.

***Șir* (`str`)**

Un șir este o secvență de caractere, folosită adesea pentru a reprezenta texte. În programare, șirurile sunt manipulate frecvent pentru diverse aplicații, cum ar fi procesarea textului. Detaliat șirurile sunt descrise în § 2.4.

***Listă* (`list`)**

O listă este o colecție ordonată de elemente care poate conține duplicate. Aceasta este una dintre cele mai des utilizate structuri de date datorită flexibilității sale. Listele sunt descrise detaliat în § 4.1.

***Tuplu* (`tuple`)**

Un tuplu este similar cu o listă, dar este imutabilă, ceea ce înseamnă că, odată creat, nu poate fi modificat. Aceasta este util pentru datele care nu ar trebui să se schimbe pe parcursul programului. Detaliat tuplurile sunt examinate în § 4.2.

2.3.4. *Dicționar (`dict`)*

Este o colecție de perechi cheie-valoare, unde fiecare cheie este unică. Dicționarele sunt utile pentru stocarea și accesarea rapidă a datelor. Mai multe detalii vezi în § 4.4.

2.3.5. *Set (`set`)*

Reprezintă o colecție de elemente unice, nesortate. Seturile sunt utile pentru operațiile matematice, cum ar fi intersecția și uniunea. Mai detaliat seturile sunt descrise în § 4.3.

2.3.6. *None*

În Python valoarea **`None`** reprezintă absența oricărei valori sau „nimic” și este utilizată pentru a indica faptul că o variabilă nu are încă asociată o valoare semnificativă.

2.4. Șir (`str`)

În Python, un șir (str) – de la *string* este o secvență de caractere (litere, cifre, simboluri, spații) delimitată de ghilimele (") sau apostrofuri ('). Șirurile de caractere sunt folosite adesea pentru a reprezenta texte, fiind manipulate frecvent pentru procesarea textului:

```
text1 = "Acesta este un șir de caractere"
text2 = 'Acesta este și el un șir'
```

Ghilimele triple (''' sau """) se folosesc pentru șiruri scrise pe mai multe linii:

```
mesaj3 = """Acesta este un mesaj
scris pe mai multe linii.
Respectă indentarea și spațiile."""
print(mesaj3)
```

```
Acesta este un mesaj
scris pe mai multe linii.
Respectă indentarea și spațiile.
```

2.4.1. *Operațiile asupra șirurilor*

Python oferă o gamă variată de operații ce pot fi realizate cu șirurile de caractere:

Concatenarea

În Python, putem uni două sau mai multe șiruri de caractere folosind operatorul "+". Acest proces se numește **concatenare** și rezultă într-un nou șir care conține caracterele celor două șiruri alipite.

```
sir1 = "Buna"
sir2 = " ziua!"
rezultat = sir1 + sir2
print(rezultat)
```

```
Buna ziua!
```

Pentru ca operatorul "+" să funcționeze corect, **toate variabilele trebuie să fie de tip *str***. Dacă încercăm să concatenăm un șir cu un alt tip de date (cum ar fi **int** sau **float**), Python va genera o eroare de tip **TypeError:**

```
varsta = 25
print("Vârsta mea este" + varsta) # Eroare!
```

```
---------------------------------------------------------------------
TypeError                                 Traceback (most recent
Cell In[100], line 2
      1 varsta = 25
----> 2 print("Vârsta mea este" + varsta)
```

Pentru a evita eroarea, putem converti numărul în șir folosind funcția **`str()`**:

```
varsta = 25
text = "Vârsta mea este " + str(varsta)
print(text)
```

```
Vârsta mea este 25
```

Repetarea (multiplicarea șirurilor)

Utilizând simbolul " * ", putem repeta un șir de caractere de mai multe ori:

```
repetare = "UTM ! " * 3
print(repetare)
```

```
UTM ! UTM ! UTM !
```

Accesarea elementelor dintr-un șir

Un șir este, în esență, un tablou unidimensional de caractere, unde fiecare caracter poate fi accesat folosind un **index**. Putem obține orice caracter dintr-un șir, utilizând indexul acestuia, specificându-l între paranteze pătrate [ ]. Indexul trebuie să fie un număr întreg.

Notă! Într-un șir, chiar și spațiile sunt considerate caractere și se indexează.

Indecșii încep de la stânga la dreapta de la `0` și merg până la `n-1`, unde `n` este numărul total de caractere. Dacă începem numărătoarea de la dreapta la stânga, vom avea indecși negativi: de la `-1` până la `-n`.

```
sir1 = "UTM Moldova"
print(sir1[0], end=",")  # Output: U
print(sir1[4], end=",")  # Output: M
print(sir1[-1], end=".")  # Output: a
```

```
U,M,a.
```

Tabelul 2.4. Indexarea elementelor în șir

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Șirul	U	T	M		M	o	l	d	o	v	a
Index	**-11**	**-10**	**-9**	**-8**	**-7**	**-6**	**-5**	**-4**	**-3**	**-2**	**-1**

În tabelul 2.4, primul caracter (U) are indexul 0 sau -11, iar ultimul caracter (a) are indexul 10 sau -1.

Slicing-ul

Utilizând indexarea, putem extrage o subsecvență dintr-un șir. Această tehnică se numește slicing, tăierea sau felierea.

Sintaxa de bază:

```
sir[start:stop:pas]
```

- **`start`** – indexul de început (inclusiv), implicit 0 (opțional);
- **`stop`** – indexul de sfârșit (exclusiv);
- **`pas`** – pasul cu care se face parcurgerea, implicit 1 (opțional).

De exemplu:

```
sir = "Facultate"
subsir_a = sir[0:6]
subsir_b = sir[3:-1]
subsir_c = sir[-1:-6:-2]
print(subsir_a)
print(subsir_b)
print(subsir_c)
```

```
Facult
ultat
eal
```

Putem omite indexul de început sau indexul de sfârșit, dar două puncte (:) trebuie indicate obligatoriu:

```
sir = "Facultate"
print(sir[:6])
print(sir[3:])
```

```
Facult
ultate
```

***Verificarea apartenenței unui subșir* (`in, not in`)**

În Python, operatorii **`in`** și **`not in`** sunt folosiți pentru a verifica dacă un element (un caracter, un subșir, un număr etc.) se regăsește într-o secvență (șir de caractere, listă, tuplu).

- **`in`** → verifică dacă un element **se află** într-o secvență;
- **`not in`** → verifică dacă un element **nu se află** într-o secvență.

Ambele operațiuni vor returna o valoare booleană: **`True`** sau **`False`**:

```
text = "Python este ușor de învățat."
print("Python" in text)
print("Java" not in text)
```

```
True
True
```

Reamintim că Python este **case-sensitive**, adică diferențiază literele mari de cele mici. Dacă dorim să facem căutarea fără a ține cont de majuscule, putem converti tot textul la litere mici, utilizând metoda **`lower()`** înainte de comparare:

```
text = "Python este ușor de învățat."
print("python" in text) # False
print("python" in text.lower()) # True
```

```
False
True
```

2.4.2. Funcții aplicate șirurilor

***Funcția* `len()`**

Funcția **`len()`** returnează lungimea unui șir (numărul de caractere care conține acesta):

```
sir1 = "UTM Moldova"
print(len(sir1))
```

```
11
```

***Funcția* `min()`**

Funcția **`min()`** returnează caracterul minim din șir, conform ordinii lexicografice. Este important de menționat că în codul ASCII, precum și Unicode, literele mari au valori numerice mai mici decât literele mici:

```
text = "Universitate"
caracter_minim = min(text)
print(caracter_minim)
```

```
U
```

***Funcția* `max()`**

Funcția **`max()`** funcționează similar cu **`min()`**, dar de data aceasta returnează caracterul maxim din șir:

```
text = "Universitate"
caracter_maxim = max(text)
print(caracter_maxim)
```

```
v
```

Aici, „v” este caracterul maxim, deoarece, așa cum s-a menționat mai sus, literele mici au valori numerice mai mari decât literele mari.

2.4.3. Metode aplicate șirurilor

Python oferă mai multe metode pentru a transforma un șir. Mai jos prezentăm exemple de conversie a șirului la majuscule și minuscule, capitalizarea primului caracter, inversarea majusculelor cu minuscule:

```
sir = "Python"
print(sir.upper()) # conversia șirului la majuscule
print(sir.lower()) # conversia șirului la minuscule
print(sir.capitalize()) # capitalizarea primului caracter
print(sir.swapcase()) # inversarea majusculelor cu minuscule
```

```
PYTHON
python
Python
pYTHON
```

Alte metode frecvent utilizate în procesarea șirurilor sunt incluse în tabelul 2.5.

Tabelul 2.5. Metode de procesare a șirurilor

Metodă	*Descriere*	*Exemplu*
`replace(old, new, count)`	Înlocuiește un subșir cu altul	`"Hello".replace("l", "L")` → `'HeLLo'`
`split(separator, maxsplit)`	Împarte un șir în listă	`"a,b,c".split(",")` → `['a', 'b', 'c']`
`join(iterable)`	Unește elementele unei liste într-un șir	`" - ".join(["a", "b"])` → `'a - b'`
`strip([chars])`	Elimină spațiile de la început și sfârșit	`" abc ".strip()` → `'abc'`
`lstrip([chars])`	Elimină spațiile de la început (din stânga)	`" abc ".lstrip()` → `'abc '`
`rstrip([chars])`	Elimină spațiile de la sfârșit (din dreapta)	`" abc ".rstrip()` → `' abc'`
`count(substring, start, end)`	Numără de câte ori apare un subșir	`"banana".count("na")` → `2`

Metodă	*Descriere*	*Exemplu*
`find(substring, start, end)`	Returnează poziția primei apariții a unui subșir (sau -1 dacă nu există)	`"banana".find("na")` `2`
`startswith(prefix, start, end)`	Verifică dacă un șir începe cu un anumit prefix	`"Python".startswith("Py")` `True`
`endswith(suffix, start, end)`	Verifică dacă un șir se termină cu un anumit sufix	`"script.py".endswith(".py")` `True`
`isdigit()`	Verifică dacă toate caracterele sunt cifre (0-9)	`"12345".isdigit()` `True`
`isnumeric()`	Verifică dacă șirul conține doar numere (inclusiv caractere numerice speciale)	`"⅔".isnumeric()` `True`
`isspace()`	Verifică dacă șirul conține doar spații	`" ".isspace()` `True`

2.4.4. Formatarea șirurilor de caractere

Formatarea șirurilor de caractere este o tehnică prin care putem insera variabile sau valori direct într-un șir de caractere, fără a folosi concatenarea (+). În Python, există mai multe moduri pentru formatarea textului, fiecare având avantaje și dezavantaje.

***Metoda* `f-string (f"...")`**

`f-string` (disponibilă din Python 3.6) este cea mai modernă și flexibilă metodă recomandată pentru formatarea textului. Folosește litera **`f`** înaintea ghilimelelor și permite includerea variabilelor direct în text, folosind acoladele **`{}`**:

```
nume = "Maria"
varsta = 25

mesaj = f"Salut, mă numesc {nume} și am {varsta} ani."
print(mesaj)
```

```
Salut, mă numesc Maria și am 25 ani.
```

***Metoda* `.format()`**

Această metodă permite inserarea variabilelor în șir, folosind acoladele `{}` și `.format()`:

```
nume = "Denis"
varsta = 30
mesaj = "Salut, mă numesc {} și am {} ani.".format(nume, varsta)
print(mesaj)
```

```
Salut, mă numesc Denis și am 30 ani.
```

Putem specifica poziția argumentelor unde numerele din acolade indică ordinea argumentelor din `.format()`:

```
mesaj = "Numele meu este {1} și am {0} ani.".format(10, "Vlad")
print(mesaj)
```

```
Numele meu este Vlad și am 10 ani.
```

Metodele `f-string` și `.format()` permit controlul precis asupra afișării numerelor zecimale:

```
pi = 3.14159265359
print("Valoarea lui Pi este {:.2f}".format(pi))
print(f"Valoarea lui Pi este {pi:.2f}")
```

```
Valoarea lui Pi este 3.14
Valoarea lui Pi este 3.14
```

2.5. Identificarea tipului de date în programare

Limbajul Python pune la dispoziție o funcție foarte utilă numită **`type()`** cu ajutorul căreia putem identifica tipul de date ale unei variabile.

Cu ajutorul acestei funcții putem valida tipul de date pe care îl avem, ceea ce este deosebit de important când facem operații matematice sau manipulări ale datelor:

```
x = 10
print(type(x))
```

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

```
y = 3.14
print(type(y))
```

```
<class 'float'>
```

```
z = "Salut, lume!"
print(type(z))
```

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

2.6. Conversia datelor

Conversia datelor reprezintă un proces important în programare, care permite să schimbăm tipul de date al unei variabile ori de câte ori este necesar. Acest lucru este util în diverse situații, cum ar fi când dorim să efectuam operații aritmetice sau să procesăm date în funcție de tipul acestora.

2.6.1. Conversia implicită (automată)

Python convertește automat dintr-un tip de date în altul când este necesar sau în funcție de rezultatul obținut. De exemplu, dacă vom adăuga un număr întreg (int) și un număr real (float), rezultatul va fi automat convertit la tipul de date real (float):

```
a = 5
b = 2.5
c = a + b
print(c, type(c))
```

```
7.5 <class 'float'>
```

2.6.2. Conversia prin utilizarea funcțiilor

Pentru a converti un tip de date, avem la dispoziție funcții preinstalate, fiecare având un nume specific asociat tipului de date dorit: **`int()`**, **`float()`**, **`str()`**, **`bool()`** ș.a.

Vom exemplifica acest proces:

1. Definim variabilă **`x`**, căreia îi atribuim o valoare de tip **`float`** (număr zecimal), folosind funcția **`float()`**.
2. Verificăm tipul de date al variabilei **`x`**, folosind funcția **`type()`** și afișăm rezultatul, folosind funcția **`print()`**:

```
x = 2.0
print(type(x))
```

```
<class 'float'>
```

3. Pentru a schimba tipul variabilei **`x`** din **`float`** în **`int`** (număr întreg), folosim funcția **`int()`**.
4. Din nou verificăm tipul de date al variabilei **`x:`**

```
x = int(x)
print(type(x))
```

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

Astfel, am realizat cu succes conversia variabilei **`x`** de la un tip de date **`float`** la un tip de date **`int`**. Această tehnică este foarte utilă când gestionăm date variate în programele noastre și dorim să ne asigurăm că operațiile efectuate sunt corecte și eficiente.

2.7. Funcțiile predefinite `print()` și `input()`

2.7.1. Funcția `print()`

Funcția **`print()`** este o funcție predefinită din limbajul Python, una dintre cele mai des utilizate funcții, permițând afișarea informațiilor pe ecran. Aceasta poate fi utilizată pentru a afișa text, valori numerice, rezultate ale unor calcule sau conținutul unor variabile.

Sintaxa de bază:

```
print(valoare1 [, valoare2, ..., sep=separator,
end=terminator, file=destinație, flush=valoare])
```

Parametrii funcției `print()`:

1. **`valori`** – parametrul principal al funcției, unde putem trece unul sau mai multe obiecte pe care dorim să le afișăm. De exemplu, putem folosi **`print("Salut, lume!")`** pentru a afișa un mesaj simplu.
2. **`sep`** – definește separatorul dintre valorile afișate. Implicit, separatorul este un spațiu `' '`, dar putem schimba acest lucru. De exemplu, **`print("A", "B", sep=", ")`** va produce ieșirea **`A, B`**.
3. **`end`** – stabilește caracterul ce va fi adăugat la sfârșitul liniei. Cu valoarea implicită **`'\n'`**, fiecare apel la **`print`**`()` va începe pe o linie nouă. Dacă dorim să afișăm pe aceeași linie mai multe apelări la print(), putem folosi **`end=" "`**.
4. **`file`** – permite să direcționăm ieșirea către un fișier sau un obiect care se comportă ca un fișier. Astfel, putem utiliza **`print("Mesaj", file=open("output.txt", "w"))`** pentru a scrie mesajul într-un fișier numit **`output.txt`**. Implicit valoarea lui **`file`** este setată ca **`sys.stdout`**.
5. **`flush`** – acesta este un parametru boolean care controlează dacă buffer-ul de ieșire va fi golit imediat sau nu. Dacă setăm **`flush=True`**, ieșirea va fi forțată să apară instantaneu pe consolă.

Prin utilizarea acestor parametri, putem adapta ieșirea funcției **`print()`** pentru a se potrivi nevoilor noastre specifice, ceea ce face programarea mai flexibilă și mai puternică.

Mai jos putem observa câteva exemple de utilizare a funcției **`print()`**:

```
#1. Afisarea unui text simplu
print("Bine ai venit la cursul de Python!")
```

```
Bine ai venit la cursul de Python!
```

```
#2. Afisarea mai mult elemente cu separator personalizat
print("Cursul", "de", "Python", sep="-")
```

```
Cursul-de-Python
```

```
#3. Schimbarea caracterului de sfarsit de linie
print("Cursul", end=" ")
print("Python", end="!")
```

```
Cursul Python!
```

2.7.2. Funcția `input()`

Funcția **`input()`** este folosită pentru a citi date introduse de utilizator de la tastatură.

Când apelăm funcția **`input()`**, aceasta creează o pauză în execuția programului actual, așteptând ca utilizatorul să introducă date de la tastatură. Când utilizatorul finalizează introducerea datelor și apasă tasta Enter, programul continuă să ruleze.

Notă! Funcția **`input()`** întotdeauna returnează un șir de caractere (**`str`**), chiar dacă utilizatorul introduce un număr.

Sintaxa de bază:

```
variabila = input(["mesaj_opțional"])
```

`mesaj_opțional` – un mesaj afișat utilizatorului înainte de introducerea valorii.

Mai jos sunt prezentate câteva exemple de utilizare a funcției **`input()`**:

```
#1. Citirea unui text introdus de utilizator
curs = input("Introduceti denumirea cursului")
print("Cursul introdus este:", curs)
```

```
Introduceti denumirea cursului
```

```
#2. Citirea unui numar introdus de utilizator
durata = int(input("Cât durează cursul?"))
print("Cursul durează", durata, "ore.")
```

```
Cât durează cursul?
```

```
#3. Citirea mai multor valori deodata
modul, nivel = input('''Introdu numele modulului
și nivelul acestuia''').split()
print("Modulul", modul, "este de nivel", nivel)
```

```
Introdu numele modulului
și nivelul acestuia
```

Folosind funcția **`input()`**, aplicațiile Python devin mai dinamice și receptive la acțiunile utilizatorului. Este un instrument indispensabil pentru interacțiunea cu utilizatorii, permițându-le să contribuie activ la execuția programului.

În concluzie, variabilele, atribuirea, tipurile și conversia de date formează baza manipulării informației într-un program Python. Înțelegerea acestor concepte este importantă pentru orice activitate de programare, de la cele mai simple scripturi până la aplicații complexe. În același timp, operatorii relaționali și logici sunt elemente fundamentale în programare și sunt foarte importanți în structurile de control, deoarece permit luarea deciziilor și controlul fluxului programului pe baza condițiilor.

Sarcini

1. **Multiplicarea șirului**. Realizați un program care citește de la tastatură un șir de caractere, apoi multiplică șirul de **n** ori (valoarea **n** trebuie citită de la tastatură).
2. **Conversia datelor.** Creați un program care cere utilizatorului 3 parametri introduși de la tastatură:
   a. Numele **`(string)`**
   b. Vârsta **`(int)`**
   c. Înălțimea **`(float)`**

   Ca rezultat, cu ajutorul *metodei* **`.format`** afișați o propoziție, de exemplu: *"Salut, mă numesc Ana, am 22 de ani și am 1.68 metri înălțime"*.
3. **Compararea lungimii a două șiruri de caractere**. Definiți un program în Python care primește două texte introduse de utilizator și determină care dintre ele este mai lung. Ca rezultat, programul va afișa atât dimensiunile celor două șiruri, cât și conținutul celui care are mai multe caractere. În cazul în care cele două șiruri au aceeași lungime, programul va afișa acest lucru cu un mesaj corespunzător.

3. STRUCTURI DE CONTROL

Obiective

La finalul acestui capitol, studenții vor fi capabili:

- Să înțeleagă rolul structurilor de control în definirea logicii unui program Python și modul în care acestea influențează fluxul execuției.
- Să utilizeze corect structurile condiționale if, if-else și if-elif-else, pentru a implementa decizii în cod în funcție de condițiile logice.
- Să implementeze structuri iterative while și for pentru a repeta secvențe de cod în mod controlat, cu sau fără ajutorul funcției range().
- Să aplice instrucțiunile de control break și continue în interiorul buclelor pentru a gestiona comportamentul execuției în mod flexibil și eficient.

3.1. Structuri condiționale (instrucțiunea if)

Structura **`if`** este o structură **decizională** (sau condițională) folosită pentru a evalua o condiție. Dacă această condiție este **adevărată (`True`)**, programul execută un bloc de cod asociat; dacă condiția este **falsă (`False`)**, acel bloc de cod nu este executat, în schimb, poate fi executat alt bloc de cod, dacă există.

Sintaxa de bază:

```
if <condiție>:
    <instrucțiuni>
```

- **`condiție`** – reprezintă o expresie care returnează o valoare booleană (adică True sau False).
- **`instrucțiuni`** – codul din interiorul blocului (indentat) va fi executat doar dacă condiția este adevărată (**`True`**).

Exemplu de o structură **`if`** simplă:

```
x = 5
if x > 3:
    print("x este mai mare decât 3")
```

```
x este mai mare decât 3
```

Condiția **`x > 3`** este evaluată. Deoarece **`x`** este 5, care este mai mare decât 3, condiția devine **adevărată**. Codul din interiorul blocului **`if`** (adică **`print("x este mai mare decât 3")`**) se va executa, iar programul va afișa: **`x este mai mare decât 3`**

3.1.1. Adăugarea unei ramuri `else`

Instrucțiunea **`if`** poate fi extinsă cu o ramură **`else`** pentru a specifica ce se întâmplă dacă condiția nu este îndeplinită (adică este falsă). Astfel, există două căi de execuție posibile: una în cazul în care condiția este adevărată și alta în cazul când este falsă.

```
x = 2
if x > 3:
    print("x este mai mare decât 3")
else:
    print("x nu este mai mare decât 3")
```

```
x nu este mai mare decât 3
```

În acest caz, condiția **`x > 3`** este **falsă** pentru că **`x`** este 2. Astfel, programul va executa codul din ramura **`else`** și va afișa:

`x nu este mai mare decât 3`.

3.1.2. Adăugarea unei ramuri `elif`

Pentru a evalua mai multe condiții în ordine, putem folosi instrucțiunea **`elif`** (abreviere pentru „else if”). Aceasta permite testarea unor condiții suplimentare în cazul în care condiția principală din **`if`** este falsă:

```
x = 10
if x > 15:
    print("x este mai mare decât 15")
elif x > 5:
    print("x este mai mare decât 5, dar mai mic sau egal cu 15")
else:
    print("x este mai mic sau egal cu 5")
```

Evaluarea expresiilor booleene

Orice expresie care returnează un tip de date boolean (**`True`** sau **`False`**) poate fi folosită într-o instrucțiune **`if`**. De exemplu, în loc să scriem explicit **`if x == True`**, putem folosi direct **`if x`**, iar pentru **`False`** putem scrie **`if not x`**:

```
x = True
if x:
    print("x este True")
```

```
x este True
```

```
x = False
if not x:
    print("x este False")
```

```
x este False
```

Operatori de comparație și logici utilizați cu instrucțiunea if

Instrucțiunea **`if`** este adesea folosită împreună cu operatorii de comparație și logici pentru a evalua condițiile.

Reamintim că în Python, **indentarea** este foarte importantă pentru a marca ce cod face parte din blocul **`if`** (sau **`elif / else`**). Indentarea incorectă va duce la erori de sintaxă.

Codul de mai jos verifică două condiții logice pentru a decide dacă să permită sau nu accesul în funcție de vârstă și permisiune:

```
age = 20
has_permission = True
if age >= 18 and has_permission:
    print("Acces permis!")
else:
    print("Acces refuzat!")
```

```
Acces permis!
```

3.2. Structuri iterative (bucle)

Buclele (sau structurile iterative) sunt folosite când dorim ca o secvență de instrucțiuni să fie executată de mai multe ori, până când o condiție specificată este îndeplinită sau până când avem o anumită limită. Acestea sunt utile când vrem să repetăm o acțiune pentru fiecare element dintr-o colecție (listă, tuplu, dicționar etc.) sau până când o condiție devine falsă.

Python oferă două comenzi principale pentru bucle:

- **`while`**
- **`for`**

3.2.1. Bucla `while`

Buclele **`while`** sunt folosite pentru a repeta un bloc de cod atât timp cât o condiție este adevărată**.** Executarea buclei se va opri când condiția devine falsă.

Sintaxa de bază:

```
while <condiție>:
    <instrucțiuni>
```

- **`condiție`** – reprezintă o expresie care returnează o valoare logică (**`True`** sau **`False`**).
- **`instrucțiuni`** – codul din interiorul blocului (indentat) va fi executat repetat, până când condiția devine falsă.

Exemplu de buclă **`while`**:

```
x = 0
while x < 5:
    print(x, end=" ")
    # Crește valoarea lui x cu 1
    x += 1
```

```
0 1 2 3 4
```

La început, **`x`** este 0. Condiția **`x < 5`** este **adevărată**, deci, programul va intra în buclă. Se va afișa valoarea lui **`x`**, care ulterior va fi incrementată cu 1 la fiecare iterație (**`x += 1`**). Bucla se va opri când **`x`** va deveni **`5`** și condiția **`x < 5`** va deveni **falsă**.

Bucla `while` *și instrucțiunea* `else`

În Python, bucla **`while`** poate fi însoțită de o instrucțiune **`else`**, care se execută **o singură dată**, **la finalul buclei**, când **condiția devine falsă** și bucla se încheie în mod normal.

```
i = 3
while i > 0:
    print(i)
    i -= 1
else:
    print("Sfârșit")
```

```
3
2
1
Sfârșit
```

Structura **`while-else`** este utilă pentru a semnala că bucla s-a încheiat în mod normal.

Bucla infinită

Este important să ne asigurăm că bucla **`while`** va ajunge întotdeauna într-un punct în care condiția devine **falsă**, altfel bucla va continua să ruleze la infinit. De exemplu, dacă uităm să incrementăm **`i`** în exemplul de mai jos, bucla nu s-ar opri niciodată.

```
# Buclă infinită
i = 3
while i > 0:
    print(i)
else:
    print("Sfârșit")
```

3.2.2. Bucla `for`

Buclele **`for`** sunt folosite pentru a itera peste o **secvență de elemente** (precum o listă, tuplu, dicționar sau chiar un șir de caractere). Ele sunt utile când vrem să aplicăm un bloc de cod asupra fiecărui element dintr-o colecție, fără a fi nevoie să ne preocupăm de numărul de iterații.

Sintaxa de bază:

```
for <element> in <secvența>:
    <instrucțiuni>
```

- **`element`** – reprezintă fiecare valoare din secvența respectivă (de exemplu, fiecare element dintr-o listă).
- **`secvența`** – este o colecție de valori asupra căreia dorim să iterăm (listă, tuplu, șir de caractere etc.).
- **`instrucțiuni`** - codul din interiorul blocului (indentat) se execută pentru fiecare element din secvență.

Bucla `for` și șirurile

În Python, putem folosi o buclă **`for`** pentru a itera prin fiecare caracter dintr-un șir de caractere (string). Fiecare literă din șir va fi procesată, una câte una:

```
text = "python"
for litera in text:
    print(litera, end = " ")
```

```
p y t h o n
```

Bucla **for** este foarte practică pentru a lucra cu șiruri de caractere. Fiecare caracter poate fi tratat individual, ceea ce este ideal pentru procesarea textelor.

***Bucla* `for` *și instrucțiunea* `else`**

În Python, putem adăuga un bloc **else** după o buclă **for**. Blocul **else** se execută doar dacă bucla **for** s-a încheiat în mod normal.

În exemplul următor, bucla **for** se termină normal, apoi se execută ramura **else:**

```
for numar in "1234":
    print(numar, end=",")
else:
    print("\nBucla s-a terminat normal.")
```

```
1,2,3,4,
Bucla s-a terminat normal.
```

***Bucla* `for` *și funcția* `range()`**

În Python, funcția **range()** este folosită pentru a genera o **succesiune de numere**, care pot fi parcurse cu o buclă **for**. Este utilă când dorim să repetăm o acțiune de un anumit număr de ori.

Sintaxa de bază:

```
for <element> in range(start, stop, pas):
    <instrucțiuni>
```

- **start** (opțional): de la ce număr începe (implicit 0)
- **stop**: unde se oprește (valoarea NU este inclusă!)
- **pas** (opțional): cu cât crește (sau scade) valoarea (implicit 1)

Câteva exemple de generare și afișare a unor liste în bucla **for** cu ajutorul funcției **range()** sunt prezentate mai jos:

```
# de la 0 la 4
for i in range(4):
    print(i)
```

```
0
1
2
3
```

```
# de la 3 la 7
for i in range(3, 7):
    print(i)
```

```
3
4
5
6
```

```
# de la 3 la 8 pasul 2
for i in range(3, 8, 2):
    print(i)
```

```
3
5
7
```

Avantajele funcției `range()` în bucle:

- Eficient pentru repetări controlate.
- Ușor de combinat cu liste, indecși sau condiții.
- Poate fi folosit pentru a accesa poziții (indecși) în liste sau șiruri.

3.3. Structuri de salt folosite în bucle

Structurile de salt sunt instrucțiuni folosite pentru a schimba fluxul normal de execuție a programului, sărind peste anumite instrucțiuni sau ieșind din bucle sau funcții.

3.3.1. Declarația `break`

Declarația **`break`** este folosită pentru a întrerupe imediat execuția unei bucle, chiar dacă condiția buclei nu a devenit încă falsă.

Este foarte utilă când dorim să oprim o buclă în mod controlat, când am găsit ceea ce am căutat, când apare o eroare, sau când o anumită condiție specială este îndeplinită.

Declarația **`break`** este folosită în bucle **`while`** și **`for`.**

În exemplul de mai jos execuția instrucțiunilor din bucla **`for`** se oprește la litera **„h”** datorită cuvântului-cheie **`break`**:

```
text = "python"
for litera in text:
    if litera == "h":
        break
    print(litera, end = " ")
```

```
p y t
```

3.3.2. Declarația `continue`

Declarația **`continue`** spune programului să sară peste restul codului din buclă pentru iterația curentă și să treacă direct la următoarea iterație din buclă. La fel ca și declarația **`break`**, **`continue`** se folosește în bucla **`while`** și **`for`**.

Când **`continue`** este întâlnit în interiorul unei bucle, restul codului din buclă NU se mai execută pentru acel pas. Execuția sare înapoi la verificarea condiției (în **`while`**) sau la pasul următor din secvența de iterare (în **`for`**).

În următorul exemplu litera „**o**” este o vocală, deci, când este întâlnită, sare peste funcția **`print()`**:

```
text = "python"
for litera in text:
    if litera in "aeiou":
        # Sare peste vocale
        continue
    print(litera, end=" ")
```

```
p y t h n
```

Diferența dintre **`break`** și **`continue`**:

- **`break`** – oprește complet bucla,
- **`continue`** – sare peste o iterație, dar bucla continuă.

În cadrul programării, structurile de control (**`if`**, **`for`**, **`while`**) joacă un rol important în modul în care sunt executate instrucțiunile și este respectată logica aplicațiilor create, oferind puterea deciziei și

a repetiției într-un program. Ele transformă codul simplu într-un cod logic, flexibil și capabil să rezolve probleme reale. Stăpânirea acestor instrumente fundamentale permite să decidem cum și când să executăm anumite secvențe de cod.

Sarcini

1. **Verificarea parității unui număr**. Definiți un program ce citește un număr întreg introdus de la tastatură și determină dacă acesta este par sau impar. În funcție de rezultat, programul va afișa un mesaj corespunzător, folosind o structură condițională **`if`** / **`else`**.
2. **Numărare până la o limită specificată.** Scrieți un program care citește de la tastatură un număr întreg pozitiv și folosește o buclă **`while`** pentru a afișa toate numerele de la 1 până la acel număr, inclusiv. Programul trebuie să verifice ca valoarea introdusă să fie strict pozitivă. Dacă utilizatorul introduce o valoare incorectă (zero sau negativă), se va cere reintroducerea acesteia până când condiția este îndeplinită.
3. **Afișarea numerelor pare dintr-un interval.** Un utilizator introduce două valori întregi de la tastatură, reprezentând limitele unui interval. Scrieți un program ce parcurge acest interval, folosind o buclă **`for`** și afișează doar numerele pare aflate între cele două capete (inclusiv, dacă se încadrează în condiție). Este important ca programul să gestioneze corect situațiile în care prima valoare este mai mare decât a doua.

4. COLECȚII DE DATE ÎN PYTHON

Obiective

La finalul acestui capitol, studenții vor fi capabili:

- Să înțeleagă conceptele fundamentale ale colecțiilor de date în Python, precum lista, tuplul, setul și dicționarul, și să explice diferențele dintre acestea în ceea ce privește organizarea, accesul și manipularea datelor.
- Să creeze și să utilizeze colecții de date în mod eficient, aplicând corect metodele și funcțiile specifice fiecărui tip de colecție pentru a rezolva probleme uzuale de programare.
- Să compare și să utilizeze adecvat tipurile de colecții în funcție de natura problemei și de caracteristici.
- Să rezolve sarcini practice de complexitate medie, care implică procesarea colecțiilor de date, analizând structura și manipulând datele pentru a obține rezultate relevante.

În Python, colecțiile de date sunt tipuri de date care pot conține mai multe valori, într-un mod organizat. Ele sunt utilizate pentru a stoca, accesa și manipula date într-un mod eficient.

Python oferă patru colecții de bază integrate:

- **Listă (`list`) -** este modificabilă, stochează valori duplicate, iar elementele pot fi accesate folosind indexarea**.**
- **Tuplu (`tuple`) -** este ordonat și de natură imuabilă, deși pot exista intrări duplicate într-un tuplu.
- **Set (`set`) -** este neordonat, nu este indexat și nu are intrări duplicate.
- **Dicționar (`dict`) -** are perechi cheie-valoare și este inerent volatil.

4.1. Listă (`list`)

O **listă** este o colecție ordonată, mutabilă, care permite stocarea și modificarea unui număr variabil de elemente.

4.1.1. Crearea listelor

În Python, listele sunt delimitate prin **paranteze pătrate** (`[]`), iar **elementele sunt separate prin virgule**. Acestea pot conține **valori de diferite tipuri de date**, inclusiv alte liste, fiind astfel extrem de versatile. De asemenea, o listă poate fi **creată fără a conține niciun element**, adică poate fi vidă.

```
list1 = [5, 3.14, True, "Hello", [False, 8.5, 'Hi']]
print(lst)
print(type(list1))
```

```
[5, 3.14, True, 'Hello', [False, 8.5, 'Hi']]
<class 'list'>
```

```
list2 = []
print(lst1)
print(type(list2))
```

```
[]
<class 'list'>
```

O listă poate fi inițializată și cu ajutorul funcției predefinite **`list()`**. Această funcție poate avea ca argument o colecție de date sau un șir. În cazul șirului, fiecare caracter va deveni element separat al listei:

```
list3 = list([1, 5.7, False, "Hello", {2: 5, 4: 5}])
print(list3)
```

```
[1, 5.7, False, 'Hello', {2: 5, 4: 5}]
```

```
list4 = list('Salut lume!')
print(list4)
```

```
['S', 'a', 'l', 'u', 't', ' ', 'l', 'u', 'm', 'e', '!']
```

Pentru a genera elementele listei, putem folosi funcția **`range()`**. Cu ajutorul funcției **`list()`** putem crea și o listă vidă:

```
list5 = list(range(8))
print(list5)
```

```
[0, 1, 2, 3, 4, 5, 6, 7]
```

```
list6 = list()
print(list6)
```

```
[]
```

4.1.2. Operații aplicate listelor

Concatenarea listelor

Python permite concatenarea a două sau mai multe liste, iar acest proces se realizează foarte simplu, folosind operatorul **"+"**.

Prin aplicarea operatorului "+", combinăm aceste două liste într-una singură, rezultând o **nouă listă** ce include toate elementele celor două liste inițiale:

```
list1 = ['unu', 'doi']
list2 = ['trei', 'patru']
lista_concatenata = list1 + list2
print(lista_concatenata)
```

```
['unu', 'doi', 'trei', 'patru']
```

Repetarea listelor

Operatorul de repetare a unei liste, de **n** ori, este "*". În cazul repetării este obligatoriu a folosi o lista și un număr întreg (multiplicatorul):

```
list1 = ['unu', 'doi', 'trei']
list2 = list1 * 2
print(list2)
```

```
['unu', 'doi', 'trei', 'unu', 'doi', 'trei']
```

Compararea listelor

Pe lângă diverse operații pe liste, Python permite și compararea acestora. Pentru acest scop, putem folosi operatorii de comparație deja cunoscuți: == (egal), != (diferit), < (mai mic), <= (mai mic sau egal), > (mai mare) și >= (mai mare sau egal). Reamintim că rezultatul comparării este o valoarea logică.

```
list_a = [1, 2, 3]
list_b = [1, 2, 3]
print(list_a == list_b)
```

```
True
```

```
print([1, 2, 4] > [1, 2, 3])
```

```
True
```

```
print([1, 2] == ["unu", "doi"])
```

```
False
```

Verificarea apartenenței unui element al listei

Pentru a verifica dacă un element face parte dintr-o listă sau nu, folosim cuvintele-cheie **`in`** sau **`not in`**.

În primul exemplu, folosind **`in`**, verificăm dacă șirul **`'doi'`** se află în lista **`['unu', 'doi', 'trei']`**, iar rezultatul este **`True`**, deoarece șirul există în listă. În exemplul doi verificăm dacă 6 nu se află în lista **`[1, 2, 3, 4, 5]`** și rezultatul este din nou **`True`**:

```
a = 'doi'
print(a in ['unu', 'doi', 'trei'])
```

```
True
```

```
print(6 not in [1, 2, 3, 4, 5])
```

```
True
```

Copierea listelor

La fel ca și în cazul altor tipuri de variabile sau obiecte, o listă poate fi copiată în Python, iar acest lucru se poate face în două moduri:

1. Prin referință.
2. Prin valoare.

Copierea prin referință înseamnă că atunci când atribuim o listă unei alte liste, a doua listă nu este o copie independentă a listei, ci doar o referință către lista originală. Astfel, orice modificare făcută pe copia listei va afecta și lista originală.

De exemplu, avem **`lista_a`** căreia îi atribuim valorile 1, 2, 3. Apoi copiem **`lista_a`** în **`lista_b`**. În acest caz, **`lista_b`** va conține doar o **referință** către **`lista_a`**, iar dacă adăugăm un element în **`lista_b`**, modificarea va fi vizibilă și în **`lista_a`**. Deși **`lista_a`** nu pare modificată direct, ambele liste vor conține acum aceleași elemente:

```
list_a = [1, 2, 3]
list_b = list_a
list_b += [4]
print(list_a)
print(list_b)
```

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

Copierea prin valoare. Pentru a evita situația de modificare a listei originale la modificarea copiei, utilizăm funcția **`list()`**. Astfel, va fi creată o listă nouă cu o referință nouă în memorie, în care copiem elementele din **`lista_a`** în **`lista_b`.**

```
list_a = [1, 2, 3]
list_b = list(list_a)
list_b += [4]
print(list_a)
print(list_b)
```

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

4.1.3. Indexarea și accesarea elementelor din liste

Asemănător cu șirurile de caractere, și în cazul listelor putem accesa orice element, utilizând un index specific plasat între paranteze pătrate. Numărătoarea pozițiilor începe de la zero, de la stânga la dreapta și nu de la dreapta la stânga**.**

De exemplu, dacă avem o listă formată din elementele **`'unu'`**, **`'doi'`**, **`'trei'`**, **`'patru'`** și **`'cinci'`**, fiecărui element îi corespunde o **poziție numerică,** începând de la 0. Astfel, **`'unu'`** se află la poziția 0, **`'doi'`** la poziția 1 și așa mai departe. Similar funcționează indecșii negativi, dar în direcție opusă. Pentru a selecta ultimul element din listă **(`'cinci'`)**, putem folosi indexul -1:

```
list_1 = ['unu', 'doi', 'trei', 'patru', 'cinci']
print(list_1[0], list_1[1], list_1[2])
print(list_1[-2], list_1[-1])
```

```
unu doi trei
patru cinci
```

Tabelul 4.1. Indexarea elementelor în listă

Index	0	1	2	3	4
Element	**`'unu'`**	**`'doi'`**	**`'trei'`**	**`'patru'`**	**`'cinci'`**
Index	-5	-4	-3	-2	-1

Dacă încercăm să accesăm un element aflat **în afara limitelor listei**, adică la un index care nu există, Python va genera o eroare de tip **`list index out of range`**.

Prin urmare, trebuie să fim atenți la **lungimea listei** când lucrăm cu indecși, pentru a evita astfel de erori:

```
list_1 = ['unu', 'doi', 'trei', 'patru', 'cinci']
print(list_1[6])
```

```
---------------------------------------------------------------
-----------------
IndexError                                Traceback (most
recent call last)
Cell In[17], line 2
      1 list_1 = ['unu', 'doi', 'trei', 'patru', 'cinci']
----> 2 print(list_1[6])

IndexError: list index out of range
```

În plus, între parantezele pătrate putem efectua și **operații matematice**, care vor returna un **index** calculat:

```
list_1 = ['unu', 'doi', 'trei', 'patru', 'cinci']
print(list_1[-1+1])
```

```
unu
```

```
print(list_1[1-2])
```

```
cinci
```

```
print(list_1[2**2])
```

```
cinci
```

Slicing

Pentru a selecta mai multe elemente dintr-o listă (o sublistă sau „felie"), folosim tehnica **slicing** (vezi § 2.4.1).

Reamintim sintaxa utilizată pentru extragerea unei subliste: **`secvența[start:stop:step]`**, unde specificăm **start** (indexul de început, inclusiv, opțional), **stop** (indexul de sfârșit, exclusiv) și **step** (pasul de selecție, opțional):

```
lista = ["unu", "doi", "trei", 4, "cinci"]
# începe de la indexul 2 inclusiv până la 5, exclusiv
print(lista[2:5])
```

```
['trei', 4, 'cinci']
```

Ca și în cazul șirurilor, putem omite oricare din acești parametri, păstrând cel puțin 2 puncte (:), care separă **start** de **stop**:

```
# începe de la indexul 0 până la 3 exclusiv
print(lista[:3])
```

```
['unu', 'doi', 'trei']
```

```
# începe de la indexul 3 până la final
print(lista[3:])
```

```
[4, 'cinci']
```

Inversarea elementelor într-o listă

O altă opțiune utilă este **inversarea listei**, sau mai exact, inversarea valorilor acesteia.

Pentru a realiza acest lucru, în sintaxa **slice**, indicăm **pasul `-1`**, iar **start** și **stop** le omitem:

```
list_1 = ['unu', 'doi', 'trei', 'patru', 'cinci']
# inversarea listei
print(list_1[::-1])
```

```
['cinci', 'patru', 'trei', 'doi', 'unu']
```

Putem extrage și o „felie” în ordine inversă, utilizând tehnica **slicing** și valoarea negativă pentru **pas**:

```
list_1 = ['unu', 'doi', 'trei', 'patru', 'cinci']
# inversarea listei
print(list_1[2::-1])
```

```
['trei', 'doi', 'unu']
```

Modificarea elementelor listei

Pentru modificarea valorii unui singur element din listă, din nou folosim indexarea:

```
lista = ["unu", "doi", "trei", 4, "cinci"]
lista[3] = "patru"
print(lista)
```

```
['unu', 'doi', 'trei', 'patru', 'cinci']
```

Pentru a modifica mai multe elemente dintr-o listă, folosim tehnica **slicing**:

```
lista = ['unu', 'doi', 'trei', 4, 'cinci']
lista[0:1] = [1]
print(lista)
```

```
[1, 'doi', 'trei', 4, 'cinci']
```

```
lista = ['unu', 'doi', 'trei', 4, 'cinci']
lista[1:4] = [2]
print(lista)
```

```
['unu', 2, 'cinci']
```

Eliminarea unui element sau secvențe din listă, ștergerea listei

Operatorul **`del`** este utilizat pentru a elimina un element sau o secvență de elemente din listă. Pentru a șterge un element, vom indica indexul elementului. Pentru a șterge o „felie” dintr-o listă, aplicăm tehnica **slicing**:

```
lista = ["unu", "doi", "trei", "patru", "cinci"]
del lista[0]
print(lista)
```

```
['doi', 'trei', 'patru', 'cinci']
```

```
del lista[2:4]
print(lista)
```

```
['doi', 'trei']
```

Operatorul **`del`** nu returnează nicio valoare, ci efectuează direct modificarea listei.

Pentru a șterge lista din memoria calculatorului, nu doar elementele acesteia, după operatorul **`del`** vom indica numele listei. La încercarea afișării listei șterse vom obține eroare:

```
del lista
print(lista)
```

```
---------------------------------------------------------------
NameError                                 Traceback (most
Cell In[39], line 1
----> 1 del lista
      2 print(lista)

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

4.1.4. Funcții aplicate listelor

Python oferă un set de funcții încorporate pentru a efectua diverse operații asupra listelor. Aceste funcții permit manipularea și analizarea eficientă a datelor stocate în liste. Tabelul 4.2 conține fiecare funcție, descrierea și exemplele de utilizare.

Tabelul 4.2. Funcții utile aplicate listelor

Funcție	*Descriere*	*Exemplu*
`len()`	Returnează numărul de elemente dintr-o secvență iterabilă	`lista = [1, 2, 3]` `print(len(lista))` `3`
`sorted()`	Returnează o copie (o listă nouă) sortată a listei.	`lista = [3, 1, 4, 2]` `sorted_lista = sorted(lista)` `print(sorted_lista)` `[1, 2, 3, 4]`
`min()`	Returnează valoarea minimă din listă, dacă lista conține elemente comparabile.	`lista = [3, 1, 4, 2]` `print(min(lista))` `1`
`max()`	Returnează valoarea maximă din listă, dacă lista conține elemente comparabile.	`lista = [3, 1, 4, 2]` `print(max(lista))` `4`
`sum()`	Returnează suma tuturor elementelor din listă (pentru liste de numere).	`lista = [3, 1, 4, 2]` `print(sum(lista))` `10`

4.1.5. Metode aplicate listelor

Python dispune de un set de metode predefinite pentru a efectua operații asupra listelor. O parte dintre metodele utile aplicate listelor este inclusă în tabelul 4.3.

Tabelul 4.3. Metode utile aplicate listelor

Metoda	*Descriere*	*Exemplu*
`append()`	Adaugă un element la sfârșitul listei.	`lst = [3, 1, 4, 2]` `lst.append(5)` `print(lst)` `[3, 1, 4, 2, 5]`

Metoda	*Descriere*	*Exemplu*
`clear()`	Elimină toate elementele din listă.	`lst.clear()` `print(lst)` `[]`
`copy()`	Returnează o copie a listei.	`lst = [3, 1, 4, 2]` `new_lst = lst.copy()` `print(new_lst)` `[3, 1, 4, 2]`
`count()`	Returnează numărul de elemente cu valoarea specificată.	`lst = [3, 1, 3, 4]` `c = lst.count(3)` `print(c)` `2`
`extend()`	Adăugă elementele unei liste (sau orice iterabil), la sfârșitul listei curente.	`lst = [3, 1, 4]` `lst.extend([5, 1])` `print(lst)` `[3, 1, 4, 5, 1]`
`index()`	Returnează indexul primului element cu valoarea specificată.	`lst = [3, 5, 4, 5]` `print(lst.index(5))` `1`
`insert()`	Adaugă la poziția specificată un element nou.	`lst = [3, 1, 4]` `lst.insert(2, 5)` `print(lst)` `[3, 1, 5, 4]`
`pop()`	Îndepărtează elementul din poziția specificată și-l returnează.	`lst = [3, 1, 5, 4]` `el = lst.pop(3)` `print(lst, el)` `[3, 1, 5] 4`

Metoda	*Descriere*	*Exemplu*
remove()	Îndepărtează elementul cu valoarea specificată.	`lst = [3, 1, 5, 4]` `lst.remove(5)` `print(lst)` `[3, 1, 4]`
reverse()	Inversează ordinea elementelor în listă.	`lst = [3, 1, 5, 4]` `lst.reverse()` `print(lst)` `[4, 5, 1, 3]`

4.1.6. Parcurgerea listelor în bucle

Parcurgerea unei liste se poate realiza folosind bucla **for**, pe care deja o cunoaștem. De asemenea, putem utiliza și bucla **while** pentru a parcurge o listă. Totuși, bucla **for** este mai indicată în acest caz, deoarece este mai eficientă și mai simplă pentru acest tip de operațiune.

```
lista = ['unu', 'doi', 'trei']
for element in lista:
    print (element, end=" ")
```

```
unu doi trei
```

```
for i in range(len(lista)):
    print(i, " - ", lista[i])
```

```
0  -  unu
1  -  doi
2  -  trei
```

În primul exemplu, avem o listă cu valorile **'unu', 'doi', 'trei'**, iar cu ajutorul buclei **for**, parcurgem fiecare element al listei și îl afișăm.

În al doilea exemplu, folosim aceeași listă, dar de această dată utilizăm funcția **`range()`** pentru a obține indexul fiecărui element. Astfel, folosind indexul **`i`**, accesăm și afișăm elementele listei pe baza valorii lor.

4.2. Tuplu (`tuple`)

Un alt tip de colecție despre care vom discuta este **tuplul**. Tuplul reprezintă o colecție de date **imutabilă**, ceea ce înseamnă că nu poate fi modificată după ce a fost creată, așa cum am procedat cu listele, care sunt mutabile. Tuplurile sunt folosite pentru a stoca mai multe elemente într-o singură variabilă separate prin virgulă.

4.2.1. Crearea tuplurilor

Deși, pentru liste folosim paranteze pătrate, în cazul tuplurilor utilizăm parantezele rotunde:

```
tuplu = ('unu', 2, 'trei')
print(type(tuplu))
```

```
<class 'tuple'>
```

Totodată, un tuplu poate fi definit chiar și fără paranteze, doar prin utilizarea virgulelor, așa cum observăm în exemplul de mai jos, iar variabila va fi tot de tip tuplu:

```
tuplu = 'unu', 2, 'trei'
print(type(tuplu))
```

```
<class 'tuple'>
```

Pentru a crea un tuplu, putem folosi și funcția predefinită în Python, **`tuple()`**, care primește ca argument o colecție de date iterabilă. În acest caz, obligatoriu utilizăm parantezele rotunde duble:

```
tpl  = tuple((2, 3, 4))
print(type(tpl))
```

```
<class 'tuple'>
```

Cu ajutorul funcției **`tuple()`** putem crea și o colecție vidă:

```
tpl  = tuple()
print(type(tpl))
```

```
<class 'tuple'>
```

Tupluri imbricate

Tuplurile pot fi **impricate**, adică pot conține și alte tupluri sau liste. Astfel, în acest caz, nu vom întâmpina nici o eroare la construcția tuplului.

```
date_tpl = ('UTM', 1, ['a', 'b'], False, ('x', 'y'))
print(type(date_tpl))
print(date_tpl)
```

```
<class 'tuple'>
('UTM', 1, ['a', 'b'], False, ('x', 'y'))
```

4.2.2. Packing și Unpacking

Packing și **Unpacking** sunt concepte importante în Python care se referă la manipularea valorilor din colecții, cum ar fi listele sau tuplurile, în scopul de a le atribui direct variabilei.

1. Packing (împachetare)

Packing se referă la procesul de **împachetare a mai multor valori într-o colecție**, cum ar fi un tuplu sau o listă. Acesta este procesul prin care valorile sunt grupate într-o singură unitate. **Packing** este util când dorim să grupăm mai multe valori într-o singură entitate (exemplu: tuplu, listă).

```
x_tpl = 1, 2, 3
print(x_tpl)
```

```
(1, 2, 3)
```

2. Unpacking (despachetare)

Unpacking este procesul invers, adică **despachetarea valorilor dintr-o colecție** și atribuirea acestora unor variabile separate. Aceasta permite accesul direct la valorile din colecție și le poate atribui variabilelor individuale. **Unpacking** este util când vrem să descompunem o colecție în valori individuale, pe care le putem manipula sau utiliza separat.

```
x_tpl = (1, 2, 3)
a, b, c = x_tpl
print(a, b, c)
```

```
1 2 3
```

Python permite chiar și **unpacking** pentru colecții de dimensiuni diferite, inclusiv liste sau tupluri mai mari.

Folosim simbolul asterisc (**`*`**) pentru a capta valorile rămase dintr-o colecție și a le stoca într-o listă. Acest lucru este util când vrem să despachetăm o colecție, dar să păstrăm valorile care nu sunt direct atribuite unor variabile.

În exemplul de mai jos, variabila **`a`** ia prima valoare, variabila **`c`** ia ultima valoare, iar restul valorilor sunt **împachetate** în lista **`b`**:

```
a, *b, c = 1, 2, 3, 4, 5
print(a)  # Output: 1
print(b)  # Output: [2, 3, 4]
print(c)  # Output: 5
```

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

4.2.3. Operații aplicate tuplurilor

În Python, tuplurile pot fi **comparate** între ele. Compararea se face în mod similar cu listele, adică fiecare element al unui tuplu este comparat cu elementul corespunzător din celălalt tuplu.

Două tupluri sunt considerate **egale** dacă îndeplinesc următoarele condiții:

- **Au aceeași dimensiune**: adică numărul de elemente din cele două tupluri este același.
- **Au aceleași valori pe aceleași poziții**: fiecare element din primul tuplu trebuie să fie egal cu elementul corespunzător din cel de-al doilea tuplu, în aceeași ordine.

```
print((3,4,5) == (3,4,5))
print(('a','b','c') == ('a','b','c'))
print(('a','c','b') > ('a','b','c'))
print((1,1,1) < (1,1,2))
```

```
True
True
True
True
```

Verificarea apartenenței elementelor în tuplu

Putem verifica dacă un anumit element face parte dintr-un tuplu folosind operatorii **`in`** și **`not in`**. Reamintim că caracterele în Python sunt sensitive la majuscule și minuscule (case-sensitive).

```
tuplu = ('UTM','USM','ASEM')
print('UTM' in tuplu)
print('utm' in tuplu)
print('utm' not in tuplu)
```

```
True
False
True
```

Extragerea elementelor din tuplu

Elementele unui tuplu, la fel ca și în cazul listelor și șirurilor, pot fi accesate folosind indecși, iar aceștia încep de la 0 (de la stânga la dreapta) sau -1 (de la dreapta la stânga).

```
tuplu = ('UTM','USM','ASEM')
print(tuplu[0])
print(tuplu[1])
print(tuplu[-1])
```

```
UTM
USM
ASEM
```

Tabelul 4.4. Indexarea elementelor în tuplu

Index	0	1	2
Element	**UTM**	**USM**	**ASEM**
Index	-3	-2	-1

Imutabilitatea tuplurilor

Un aspect important de menționat este că un tuplu nu poate fi modificat după ce a fost creat. Dacă încercăm să schimbăm o valoare sau să modificăm un element la o anumită poziție din tuplu, vom obține o eroare. Acest comportament se datorează faptului că tuplurile sunt **imutabile**. Exemplul următor demonstrează acest lucru:

```
tuplu = ('UTM','USM','ASEM')
tuplu[0] = 1
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent
Input In [48], in <cell line: 2>()
      1 tuplu = ('UTM','USM','ASEM')
----> 2 tuplu[0] = 1

TypeError: 'tuple' object does not support item assignment
```

4.2.4. Funcții și metode aplicate tuplurilor

În Python, tuplurile beneficiază de un șir de funcții și metode utile care facilitează manipularea și operarea cu acestea. Deși tuplurile sunt imutabile (nu pot fi modificate după ce au fost create), există mai multe funcții predefinite care pot fi folosite pentru a le manipula sau analiza. În tabelul 4.5 sunt incluse câteva dintre cel mai des utilizate funcții și metode pentru tupluri.

Tabelul 4.5. Funcții și metode utile aplicate tuplurilor

Funcție/ metodă	*Descriere*	*Exemplu*
`len()`	Returnează numărul de elemente din tuplu.	`tuplu = (1, 2, 3, 4, 5)` `print(len(tuplu))` `5`
`min()`	Returnează valoarea minimă din tuplu (pentru elemente comparabile)	`tuplu = (5, 2, 9, 1, 8)` `print(min(tuplu))` `1`
`max()`	Returnează valoarea maximă din tuplu (pentru elemente comparabile)	`tuplu = (5, 2, 9, 1, 8)` `print(max(tuplu))` `9`
`sum()`	Returnează suma tuturor elementelor din tuplu (pentru elemente numerice)	`tuplu = (1, 2, 3, 4)` `print(sum(tuplu))` `10`
`count()`	Returnează numărul de apariții al unui element în tuplu	`tuplu = (1, 2, 2, 3, 2, 4)` `print(tuplu.count(2))` `3`
`index()`	Returnează primul index al unui element într-un tuplu	`tuplu = (1, 2, 3, 4, 5)` `print(tuplu.index(3))` `2`

Ștergerea unui tuplu

Pentru ștergerea unui tuplu din memoria calculatorului, utilizăm operatorul **`del`** și numele tuplului. La încercarea de afișare a tuplului șters obținem eroare:

```
tuplu = (1, 2, 3, 4)
del tuplu
print(tuplu)
```

```
---------------------------------------------------------------------
NameError                                 Traceback (most recent
Cell In[69], line 3
      1 tuplu = (1, 2, 3, 4)
      2 del tuplu
----> 3 print(tuplu)

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

4.3. Set (mulțime)

Un set sau o mulțime este un tip de obiect care stochează o colecție de date. Seturile au câteva trăsături importante:

- Toate elementele dintr-un set sunt **unice**, adică nu pot exista două elemente cu aceeași valoare.
- Seturile sunt structuri **neordonate**, ceea ce înseamnă că elementele pot fi plasate în orice ordine și indexarea nu poate fi aplicată elementelor unui set.
- Elementele unui set pot fi de **tipuri diferite**.

4.3.1. Crearea seturilor

Pentru a crea o colecție de date de tip set, putem utiliza fie funcția **`set()`**, fie acoladele **`{}`**.

În primul exemplu, creăm un obiect folosind funcția **`set()`** căreia îi atribuim un șir de caractere ca parametru. Funcția **`set()`** acceptă ca parametru o colecție de date. În al doilea exemplu utilizăm acoladele pentru crearea unui set:

```
# cu ajutorul funției set()
set_a = set("Universitate")
print(type(set_a))
print(set_a)
```

```
<class 'set'>
{'a', 'U', 'i', 'r', 'v', 's', 'e', 't', 'n'}
```

```
# cu ajutorul acoladelor
set_b = {1, 6, 2.5, 4, 3.0}
print(set_b)
```

```
{1, 2.5, 3.0, 4, 6}
```

Pentru că **seturile conțin doar valori unice**, dacă se încearcă adăugarea unui element existent în set, acesta nu va fi adăugat din nou, iar setul va rămâne neschimbat:

```
setul = set(['unu','doi','trei','unu','trei'])
print(setul)
```

```
{'doi', 'trei', 'unu'}
```

Deoarece seturile sunt structuri neordonate, nu există o ordine specifică a elementelor. Prin urmare, elementele acestora **nu pot fi accesate pe bază de index**. Din același motiv, seturile nu acceptă operații de slicing:

```
setul = set(['unu','doi','trei','unu','trei'])
print(setul[0])
```

```
---------------------------------------------------------------
TypeError                                 Traceback (most
Input In [8], in <cell line: 3>()
      1 # doar valori unice
      2 setul = set(['unu','doi','trei','unu','trei'])
----> 3 print(setul[0])

TypeError: 'set' object is not subscriptable
```

4.3.2. *Accesarea elementelor din set în bucla* `for`

Pentru a accesa elementele unui set, folosim bucla **`for`**. Deoarece seturile sunt colecții neordonate, ordinea în care sunt accesate valorile poate fi diferită la fiecare execuție a buclei. Astfel, putem itera prin set pentru a obține acces la fiecare element, fără a avea nevoie de un index specific.

```
setul = {'unu','doi','trei','unu','trei'}
for x in setul:
    print(x)
```

```
doi
trei
unu
```

4.3.3. *Operații aplicate seturilor*

Seturile permit o serie de operații pe mulțimi de date, care sunt utile pentru rezolvarea diverselor probleme. Aceste operații includ: reuniune, intersecție și diferență. Metoda **`union()`** este utilizată pentru reuniune, **`intersection()`** pentru intersecție, iar **`difference()`** pentru a calcula diferența dintre seturi. Toate cele trei metode returnează un set ce conține rezultatul operației respective:

```
setul_a = {1,2,3,4}
setul_b = {5,6,7}
print(setul_a.union(setul_b))
```

```
{1, 2, 3, 4, 5, 6, 7}
```

```
setul_a = {1,2,3,4}
setul_b = {4,5,3,7}
print(setul_a.intersection(setul_b))
```

```
{3, 4}
```

```
setul_a = {1,2,3,4}
setul_b = {4,5,3,7}
print(setul_a.difference(setul_b))
print(setul_b.difference(setul_a))
```

```
{1, 2}
{5, 7}
```

4.3.4. Metode aplicate seturilor

Adăugarea elementelor în set

Pentru a adăuga elemente noi într-un set, putem utiliza metodele:

- **`add(element)`** pentru a adăuga un singur element.
- **`update(element1, element2, …)`** pentru a adăuga mai multe elemente simultan.

```
# adăugăm un element nou
setul = {'unu', 'doi', 'trei', 'unu', 'trei'}
setul.add('patru')
print(setul)
```

```
{'trei', 'unu', 'patru', 'doi'}
```

```
# adăugăm o colecție de elemente
setul = {'unu', 'doi', 'trei', 'unu', 'trei'}
setul.update({'patru', 'cinci'})
print(setul)
```

```
{'trei', 'cinci', 'unu', 'patru', 'doi'}
```

Ștergerea elementelor din set

Pentru a elimina un element dintr-un set, putem utiliza metoda **`pop()`**. Aceasta va șterge un element aleatoriu din set și va returna valoarea elementului eliminat, astfel încât putem verifica elementul care a fost efectiv șters:

```
# stergem elementul aleatoriu
setul = {'unu','doi','trei','unu','trei'}
print(setul.pop())
print(setul)
```

```
doi
{'trei', 'unu'}
```

O altă metodă disponibilă pentru eliminare este **`discard()`**. Această metodă ia o valoare ca parametru și va șterge elementul respectiv dacă există în set. Dacă valoarea nu se găsește, nu va face nimic și nu se va genera vreo eroare.

```
# ștergem elementul după valoare
setul = {'unu', 'doi', 'trei', 'unu', 'trei'}
setul.discard('unu')
print(setul)
```

```
{'doi', 'trei'}
```

O altă metodă pentru a elimina un element dintr-un set este **`remove()`**. Aceasta funcționează similar cu **`discard()`**, luând ca parametru valoarea care trebuie eliminată, dar în cazul în care elementul nu există în set, **`remove()`** va genera o eroare, spre deosebire de **`discard()`**, care nu face nimic în această situație.

```
# ștergem un element inexistent
setul = {'unu', 'doi', 'trei', 'unu', 'trei'}
setul.remove('patru')
```

```
---------------------------------------------------------------------
-----------
KeyError                                  Traceback (most recent
call last)
Cell In[5], line 3
      1 # ștergem un element inexistent
      2 setul = {'unu', 'doi', 'trei', 'unu', 'trei'}
----> 3 setul.remove('patru')
```

Ștergerea unui set

Pentru ștergerea unui set din memoria calculatorului, utilizăm operatorul **`del`**. La încercarea de afișare a setului șters obținem eroare:

```
setul = {'unu', 'doi', 'trei', 'unu', 'trei'}
del setul
print(setul)
```

```
---------------------------------------------------------------
NameError                                 Traceback (most recer
Cell In[77], line 3
      1 setul = {'unu', 'doi', 'trei', 'unu', 'trei'}
      2 del setul
----> 3 print(setul)

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

4.4. Dicționar (`dictionary`)

Un alt tip de colecție de date în Python este dicționarul. Un dicționar este o structură de date **mutabilă**, care stochează informații într-un format **`cheie:valoare`**. Caracteristici importante ale dicționarelor sunt:

- Cheile sunt **unice** și nu pot fi duplicate.
- Dicționarele sunt **mutabile**, însă **cheile** trebuie să fie de tipuri **imutabile** (numere, șiruri de caractere, tupluri sau chiar valori logice).
- Valorile pot fi **de orice tip** de date.
- Este posibil să avem chei și valori de tipuri diferite într-un dicționar.
- Elementele din dicționar (**`cheie:valoare`**) pot fi accesate, utilizând cheia asociată.

4.4.1. Crearea dicționarelor

Un dicționar poate fi inițializat folosind acolade **`{}`** sau prin utilizarea funcției **`dict()`**. Ambele metode permit definirea unui

dicționar, dar diferența constă în modul de sintaxă. Cu acolade, cheile și valorile sunt plasate direct în interior, iar cu **`dict()`**, putem specifica perechile **`cheie:valoare`** fie prin argumente, fie printr-o listă de tupluri:

```
dictionar = {'nume':'Mocanu', 'prenume':'Mihai',
             'virsta':35}
print(dictionar)
```

```
{'nume': 'Mocanu', 'prenume': 'Mihai', 'virsta': 35}
```

```
dictionar = dict(nume = 'Mocanu', prenume = "Mihai",
                 virsta = 35)
print(dictionar)
```

```
{'nume': 'Mocanu', 'prenume': 'Mihai', 'virsta': 35}
```

Funcția **`dict.fromkeys(seq, value)`** creează un dicționar, folosind elementele din secvența **`seq`** ca chei, iar valorile asociate fiecărei chei vor fi setate la **`value`**. Dacă nu este specificată nicio valoare pentru **`value`**, atunci valoarea implicită va fi **`None`**. Această funcție este utilă când dorim să generăm un dicționar cu chei predefinite și valori implicite.

```
dict_a = dict.fromkeys(['a', 'b', 'c'])
print(dict_a)
```

```
{'a': None, 'b': None, 'c': None}
```

```
dict_b = dict.fromkeys(['a', 'b', 'c'], [1])
print(dict_b)
```

```
{'a': [1], 'b': [1], 'c': [1]}
```

Valorile din elementele unui dicționar pot fi de orice tip de date, cum ar fi numere, șiruri de caractere, liste, tupluri, seturi sau chiar alte dicționare sau valori logice. Aceasta oferă o mare flexibilitate în stocarea datelor asociate fiecărei chei.

De asemenea, este posibil să asociem mai multe valori cu o singură cheie. O modalitate de a face acest lucru este prin utilizarea unui tip de colecție (de exemplu, o listă sau un tuplu) ca valoare a cheii respective. Astfel, o cheie poate avea un număr variabil de valori asociate.

```
dictionar = {"Student":"Mihai",
             1:10,
             "lista": [1, 2, 3],
             'dictionar':{"unu": 1, "doi":2},
             ("tuplu",):10
             }
print(dictionar)
```

```
{'Student': 'Mihai', 1: 10, 'lista': [1, 2, 3], 'dictionar':
{'unu': 1, 'doi': 2}, ('tuplu',): 10}
```

4.4.2. Accesarea elementelor

Accesul la elemente într-un dicționar se realizează prin utilizarea cheilor. Elementele unui dicționar pot apărea într-o ordine diferită față de cea în care au fost adăugate. Pentru a obține valoarea asociată unei chei, utilizăm sintaxa **`dictionar[cheie]`** sau aplicând metoda **`get()`**:

```
dictionar = {'unu':1, 'doi':2, 'trei':3, 'patru':4}
print(dictionar['unu'])
print(dictionar.get('unu'))
```

```
1
1
```

4.4.3. Adăugarea elementelor

Pentru a adăuga un element, trebuie să creăm o nouă pereche **`cheie:valoare`**. Adăugarea de elemente într-un dicționar în Python se face prin utilizarea parantezelor pătrate și a semnului de atribuire. Pentru că dicționarele sunt colecții de perechi **`cheie:valoare`**, adăugarea unui nou element se face prin specificarea unei chei noi și atribuirea valorii corespunzătoare:

```
dictionar = {'unu':1, 'doi':2, 'trei':3, 'patru':4}
dictionar['cinci'] = 5
print(dictionar)
```

```
{'unu': 1, 'doi': 2, 'trei': 3, 'patru': 4, 'cinci': 5}
```

Verificarea apartenenței elementelor unui dicționar

Pentru a verifica dacă o cheie există sau nu într-un dicționar, putem folosi operatorii **`in`** sau **`not in`**. În exemplul de mai jos avem un dicționar definit unde verificăm dacă cheia **`'patru'`** este prezentă în dicționar. Folosind operatorul **`in`**, obținem rezultatul **`True`**. În schimb, pentru cheia **`'cinci'`**, rezultatul va fi **`False`**, deoarece aceasta nu există în dicționar. Dacă utilizăm operatorul **`not in`**, acesta va returna **`True`**, deoarece reprezintă negarea operatorului **`in`**.

```
dictionar = {'unu':1, 'doi':2, 'trei':3, 'patru':4}
print('patru' in dictionar)
print('cinci' in dictionar)
print('cinci' not in dictionar)
```

```
True
False
True
```

4.4.4. Modificarea elementelor în dicționar

Modificarea elementelor dintr-un dicționar în Python se realizează cu ajutorul parantezelor pătrate **`[]`** și a semnului de atribuire. Mai exact, se indică cheia elementului pe care dorim să-l modificăm, după care îi atribuim o nouă valoare:

```
dictionar = {'unu':1, 'doi':2, 'trei':3, 'patru':4}
dictionar['doi'] = 22
print(dictionar)
```

```
{'unu': 1, 'doi': 22, 'trei': 3, 'patru': 4}
```

4.4.5. *Funcții și metode aplicate dicționarelor*

La fel ca în cazul listelor, tuplurilor și seturilor, putem obține informații utile și din dicționare folosind funcțiile **`len()`**, **`min()`** și **`max()`**. Totuși, aceste funcții se aplică doar cheilor dicționarului, nu și valorilor. Astfel, funcția **`len()`** va returna numărul de elemente (perechi) din dicționar, funcția **`max()`** va returna cheia cu valoarea maximă, iar funcția **`min()`** va returna cheia cu valoarea minimă:

```
dictionar = {2:0,
             1:0,
             8:0,
             3:0,
             1:0
            }
print("Lungimea:", len(dictionar))
print("Maxim:",max(dictionar))
print("Minim:",min(dictionar))
```

```
Lungimea: 4
Maxim: 8
Minim: 1
```

Crearea copiei unui dicționar

Pentru a crea o copie a unui dicționar în Python, folosim metoda **`copy()`**. Această metodă returnează o **copie superficială** (*shallow copy*) a dicționarului, adică se creează un nou dicționar care conține aceleași chei și valori ca și originalul, dar este stocat într-o locație diferită în memorie:

```
dictionar = {'unu':1, 'doi':2, 'trei':3, 'patru':4}
dictionar_nou = dictionar.copy()
print(dictionar_nou)
```

```
{'unu': 1, 'doi': 2, 'trei': 3, 'patru': 4}
```

Vizualizarea conținutului unui dicționar

În Python, conținutul dintr-un dicționar poate fi vizualizat în mai multe moduri, în funcție de ce rezultat dorim să obținem.

Pentru vizualizarea tuturor cheilor dintr-un dicționar este utilizată metoda **`dict.keys()`**. Pentru returnarea valorilor se folosește metoda **`dict.values()`**. Atât cheile, cât și valorile vor fi returnate în formă de elemente a unei listei. Pentru vizualizarea perechilor din dicționar, vom utiliza metoda **`dict.items()`**, care returnează o listă ce conține perechi de tipul tupluri (cheie, valoare):

```
dictionar = {'unu':1,
             'doi':2,
             'trei':3
            }
print(dictionar.keys())
print(dictionar.values())
print(dictionar.items())
```

```
dict_keys(['unu', 'doi', 'trei'])
dict_values([1, 2, 3])
dict_items([('unu', 1), ('doi', 2), ('trei', 3)])
```

Eliminarea elementelor dintr-un dicționar

Eliminarea elementelor dintr-un dicționar se poate realiza folosind mai multe metode.

Pentru a șterge un element, este suficient să folosim cuvântul-cheie **`del`**, urmat de numele dicționarului și cheia elementului:

```
dictionar = {'unu':1, 'doi':2, 'trei':3, 'patru':4}
del dictionar['doi']
print(dictionar)
```

```
{'unu': 1, 'trei': 3, 'patru': 4}
```

Această metodă formează eroare dacă cheia nu există.

Pentru ștergerea tuturor elementelor dintr-un dicționar, putem apela metoda **`clear()`**. Drept rezultat vom obține un dicționar vid:

```
dictionar = {'unu':1, 'doi':2, 'trei':3, 'patru':4}
dictionar.clear()
print(dictionar)
```

```
{}
```

Metoda **`dict.pop()`** elimină și returnează valoarea asociată cheii specificate:

```
dictionar = {'unu':1, 'doi':2, 'trei':3, 'patru':4}
a = dictionar.pop('patru')
print(dictionar, a)
```

```
{'unu': 1, 'doi': 2, 'trei': 3} 4
```

Metoda **`dict.pop()`** returnează eroare dacă cheia nu există și dacă nu este implicit specificată una.

Metoda **`dict.popitem()`** elimină **ultimul** element adăugat (doar din Python 3.7+). Returnează perechea (cheie, valoare):

```
dictionar = {'unu':1, 'doi':2, 'trei':3, 'patru':4}
a = dictionar.popitem()
print(dictionar, a)
```

```
{'unu': 1, 'doi': 2, 'trei': 3} ('patru', 4)
```

Pentru eliminarea tuturor elementelor din dicționar putem utiliza metoda **`dict.clear()`**. După aplicarea acestei metode, dicționarul devine vid:

```
dictionar = {'unu':1, 'doi':2, 'trei':3, 'patru':4}
dictionar.clear()
print(dictionar)
```

```
{}
```

Ștergerea unui dicționar

Similar cu celelalte colecții de date, pentru ștergerea unui dicționar utilizăm operatorul **`del`**. La încercarea de afișare a dicționarului șters obținem eroare:

```
dictionar = {'unu':1, 'doi':2, 'trei':3, 'patru':4}
del dictionar
print(dictionar)
```

```
---------------------------------------------------------------
---------------
NameError                                 Traceback (most
ecent call last)
Cell In[13], line 3
      1 dictionar = {'unu':1, 'doi':2, 'trei':3, 'patru':4
      2 del dictionar
----> 3 print(dictionar)

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

În concluzie, colecțiile de date din Python sunt instrumente puternice și flexibile pentru lucrul cu grupuri de informații. Înțelegerea diferențelor dintre ele și folosirea lor corectă reprezintă un pas important spre scrierea de cod eficient, organizat și scalabil.

Sarcini

1. **Analiza unei liste de numere.** Având o listă de numere întregi introdusă de utilizator, afișați informații relevante despre această listă: câte elemente conține, suma tuturor numerelor și valoarea maximă.
2. **Identificarea cuvintelor distincte.** Scrieți un program care acceptă unele cuvinte specifice. Tipăriți numărul de cuvinte distincte și numărul de apariții pentru fiecare cuvânt distinct, în funcție de aspectul lor.

3. **Gestionarea unui dicționar de note ale studenților.** Se consideră un dicționar în care cheia este numele unui student (șir de caractere), iar valoarea este o listă cu notele obținute la mai multe examene (numere întregi).

   Pornind de la acest dicționar, determinați și afișați pentru fiecare student:

   - media notelor;
   - numărul examenelor susținute;
   - dacă media este mai mare sau egală cu 5 (promovat) sau mai mică (respins).

   La final, afișați numele studentului cu cea mai mare medie și valoarea acesteia.

5. FIȘIERE, PRELUCRAREA FIȘIERELOR

Obiective

La finalul acestui capitol, studenții vor fi capabili:

- Să înțeleagă conceptul de fișier și rolul prelucrării fișierelor în programare.
- Să aplice tehnici eficiente de deschidere, citire, scriere și închidere a fișierelor, folosind metodele și modurile de acces specifice, în contextul diferitelor cerințe de procesare.
- Să aplice tehnici eficiente de citire și scriere condiționată a datelor, inclusiv parcurgerea fișierelor linie cu linie și scrierea formatată, în scopul prelucrării datelor mari fără a suprasolicita memoria.

Până acum, în procesul de scriere a codului în Python, au fost folosite metode simple de introducere și afișare a datelor. Datele de intrare, introduse de utilizator de la tastatura, erau preluate utilizând funcția **`input()`**, iar datele de ieșire se afișau în consolă prin utilizarea funcției **`print()`**. Aceste funcții de introducere și afișare a datelor sunt bune, dacă e vorba de un volum mic de date. Dar închipuiți-vă că vrem să calculăm suma a 150 de numere și introducem greșit al 140 număr. Începem din nou? Și dacă greșim și a doua, a treia oară? Dar dacă este necesar să procesăm un volum mare de date în fiecare zi?

Soluția este să citim datele dintr-un fișier stocat pe computer sau să le înregistrăm într-un fișier. Procesul de citire și înregistrare a datelor într-un fișier se numește **file input/output** (file I/O).

Un **fișier** reprezintă un set de date, stocate pe un dispozitiv de stocare, cum ar fi un hard disc, SSD, USB sau alt mediu de stocare. Datele din fișiere sunt stocate în forma unei secvențe de biți și sunt structurate într-un anumit mod și sub un nume comun – numele fișierului (**filename**).

În Python, fișierele pot fi clasificate în funcție de conținut:

1. Fișiere textuale, în care se stochează secvențe de caractere Unicode. Formatele de fișiere mai des întâlnite sunt: .txt, .html, .rtf, .csv, .dat ș.a.
2. Fișiere binare, în care sunt stocate secvențe de biți în formă codificată, utilizând doar două simboluri: zero (0) și unu (1). De regulă, acestea conțin imagini, audio, video, sau date comprimate și sunt stocate în fișiere cu extensia .bin.

Fișierele create în Python sunt manipulate de sistemul de fișiere (file system) al computerului. Python utilizează funcții specifice ale sistemului de operare al computerului pentru importul, modificarea și salvarea fișierelor.

Pentru că Python a fost proiectat pentru a procesa în primul rând fișiere textuale, acesta oferă modalități pentru deschiderea fișierului, executarea operațiunilor (citire sau/și scriere) și închiderea fișierului. Procesul de citire a unui fișier textual are loc în câteva etape:

1. Deschiderea fișierului de pe HDD, SDD, Memory Stick sau alt mediu de stocare a informației.
2. Conținutul fișierului este scris în memoria operativă (RAM), unde sunt efectuate operațiile de citire/scriere.
3. Închiderea fișierului de pe mediul de stocare.

Notă! Modificările datelor din fișier se fac pe o copie stocată în memoria RAM, nu în fișierul original din mediul de stocare.

5.1. Deschiderea fișierului

Pentru deschiderea unui fișier în Python este utilizată funcția încorporată (build-in function) **`open()`**. Din punct de vedere tehnic, Python creează un nou obiect pe baza fișierului deschis, pe care îl procesează.

Sintaxa de bază:

```
my_file = open(file_name[,access_mode])
```

- **`my_file`** – variabila-indicator la obiectul de clasa *file* (file, handler sau file object).
- **`file_name`** (*obligatoriu*) – numele fișierului pe care vrem să-l deschidem. Se scrie în formă de șir de caractere, cuprinse în ghilimele.

- **`access_mode`** (*opțional*) – modul de acces la fișier, care la fel se scrie în formă de șir de caractere. Nu este argument obligatoriu. Acesta poate fi: pentru citire (**`'r'`**), pentru scriere (**`'w'`**), pentru adăugare (**`'a'`**). Dacă modul de acces nu este indicat, implicit este aplicat regimul de citire (**`'r'`**). Modurile posibile de acces la fișier sunt incluse în yabelul 5.1.

Tabelul 5.1. Lista modurilor de acces la fișier

Mod de acces	***Descriere***
`r`	Deschide fișierul pentru citirea conținutului. Indicatorul de fișier este plasat la începutul fișierului. Dacă fișierul nu există, afișează eroare. Acesta este modul implicit.
`r+`	Deschide fișierul pentru citirea și scrierea conținutului. Dacă fișierul nu există, afișează eroare.
`rb`	Deschide fișierul pentru citirea conținutului în format binar. Indicatorul de fișier este plasat la începutul fișierului. Dacă fișierul nu există, afișează eroare.
`rb+`	Deschide fișierul atât pentru citirea, cât și scrierea conținutului în format binar.
`w`	Deschide fișierul pentru scrierea conținutului. Suprascrie fișierul dacă acesta există. Va crea fișier nou, dacă nu-l găsește cu numele indicat.
`w+`	Deschide fișierul atât pentru scrierea, cât și citirea conținutului. Suprascrie fișierul dacă acesta există. Va crea fișier nou, dacă nu-l găsește cu numele indicat.
`wb`	Deschide fișierul pentru scrierea conținutului în format binar. Suprascrie fișierul dacă acesta există. Suprascrie fișierul dacă fișierul există. Va crea fișier nou, dacă nu-l găsește cu numele indicat.
`wb+`	Deschide fișierul atât pentru scrierea, cât și citirea conținutului în format binar. Suprascrie fișierul dacă fișierul există. Va crea fișier nou, dacă nu-l găsește cu numele indicat.

Mod de acces	*Descriere*
`a`	Deschide fișierul pentru adăugarea conținutului. Indicatorul de fișier se află la sfârșitul fișierului. Va crea fișier nou pentru scriere, dacă nu găsește fișierul cu numele indicat.
`a+`	Deschide fișierul atât pentru adăugarea, cât și scrierea conținutului. Indicatorul de fișier se află la sfârșitul fișierului. Va crea fișier nou pentru scriere, dacă nu găsește fișierul cu numele indicat.
`ab`	Deschide fișierul pentru adăugarea conținutului în format binar. Indicatorul de fișier se află la sfârșitul fișierului. Va crea fișier nou pentru scriere, dacă nu găsește fișierul cu numele indicat.
`ab+`	Deschide fișierul atât pentru adăugarea, cât și scrierea conținutului în format binar. Indicatorul de fișier se află la sfârșitul fișierului. Va crea fișier nou pentru scriere, dacă nu găsește fișierul cu numele indicat.
`x`	Se va crea un fișier cu numele specificat. Va afișa eroare, dacă fișierul cu numele indicat deja există.

Pentru exemplificare, creăm fișierul textual **`zen.txt`** în care vom include primele 5 aforisme din „The Zen of Python” – setul de principii care ghidează designul limbajului de programare Python, scris de Tim Peters, un contribuitor important în comunitatea Python (Figura 5.1):

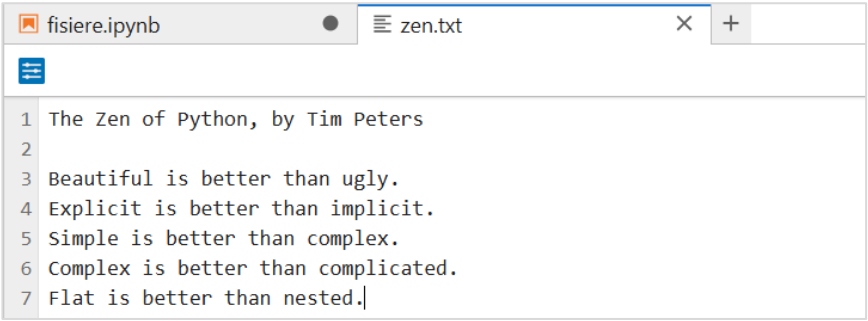

Figura 5.1. Conținutul fișierului zen.txt

Vom urmări un exemplu de deschidere a acestui fișier în modul de citire și de afișare a conținutului acestuia, folosind funcția **`print(*)`**:

```
# Deschidem fișierul
my_file = open('zen.txt', 'r')
```

```
# Afișăm întregul conținut din my_file
print(*my_file)
```

```
The Zen of Python, by Tim Peters

 Beautiful is better than ugly.
 Explicit is better than implicit.
 Simple is better than complex.
 Complex is better than complicated.
 Flat is better than nested.
```

Dacă vom omite asteriscul din **`print()`**, Python va returna descrierea obiectului de tip fișier:

```
# Afișăm descrierea obiectului my_file
print(my_file)
```

```
<_io.TextIOWrapper name='zen.txt' mode='r' encoding='UTF-8'>
```

Din momentul deschiderii fișierului și creării în baza lui a unui nou obiect, putem afla un șir de informații despre acesta, aplicând metode specifice (tabelul 5.2).

Tabelul 5.2. Metode aplicate fișierelor deschise

Metoda	***Descrierea***
`file.closed`	Returnează `True` dacă fișierul a fost închis
`file.mode`	Returnează modul de acces aplicat fișierului deschis
`file.name`	Returnează numele fișierului

Următorul cod afișează rezultatul aplicării metodelor descrise în tabelul 5.2:

```
# Verificăm dacă fișierul este închis
print(my_file.closed)
```

```
False
```

```
# Verificăm modul de acces aplicat fișierului deschis
print(my_file.mode)
```

```
r
```

```
# Afișăm numele fișierului deschis
print(my_file.name)
```

```
zen.txt
```

În locul numelui fișierului sau modului de acces putem folosi și variabile, dacă acestora le-au fost atribuite valori corespunzătoare:

```
file_name = 'zen.txt'
access_mode = 'r+'
f_file = open(file_name, access_mode)
```

```
print(*f_file)
```

```
The Zen of Python, by Tim Peters

 Beautiful is better than ugly.
 Explicit is better than implicit.
 Simple is better than complex.
 Complex is better than complicated.
 Flat is better than nested.
```

Dacă fișierul nu există, deschiderea acestuia va eșua. Interpretorul va afișa mesajul de eroare:

```
# Deschidem fișierul
my_file = open('zzen.txt', 'r')
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call la
Cell In[10], line 2
      1 # Deschidem fișierul
----> 2 my_file = open('zzen.txt', 'r')
```

5.2. Închiderea fișierului

După deschiderea fișierului și efectuarea operațiunilor necesare (citire, scriere), acesta trebuie închis pentru a elibera resursele utilizate și a șterge „gunoiul" creat după deschiderea fișierului.

În Python există următoarele metode de închidere a fișierului deschis:

1) Utilizând metoda **`close()`**:

```
# Deschidem fișierul
my_file = open('zen.txt', 'r')
# Închidem fișierul
my_file.close()
# Verificăm dacă fișierul este închis
print('Fisierul este inchis -', my_file.closed)
```

```
Fisierul este inchis - True
```

După închiderea fișierului acesta nu poate fi utilizat până când nu va fi deschis din nou.

2) Utilizând construcția **`try/finally`** (vezi § 7), care asigură că, dacă o operațiune aplicată fișierului va provoca excepție după deschiderea acestuia, fișierul va fi închis automat:

```
# Deschidem fișierul
my_file = open('zen.txt', 'r')
try:
    print(*my_file)  # Afișarea conținutului fișierului
finally:
    my_file.close()
# Verificăm dacă fișierul este închis
print('Fisierul este inchis -', my_file.closed)
```

```
The Zen of Python, by Tim Peters

 Beautiful is better than ugly.
 Explicit is better than implicit.
 Simple is better than complicated.
 Flat is better than nested.
Fisierul este inchis - True
```

Fără această construcție, programul se va termina anormal.

3) Utilizând instrucțiunea **`with`**, care simplifică gestionarea excepțiilor prin încapsularea operațiunilor inițiale și a sarcinilor de închidere. În acest caz, metoda **`close`** nu este necesară, deoarece **`with`** va închide automat fișierul:

```
with open('zen.txt') as my_file:
    # Operațiuni aplicate fișierului
    print(*my_file)

# Verificăm dacă fișierul este închis
print('Fisierul este inchis -', my_file.closed)
```

```
The Zen of Python, by Tim Peters

 Beautiful is better than ugly.
 Explicit is better than implicit.
 Simple is better than complex.
 Complex is better than complicated.
 Flat is better than nested.
Fisierul este inchis - True
```

5.3. Citirea conținutului din fișier

În Python putem folosi mai multe metode pentru a citi conținutul unui fișier.

1) Metoda **`print(*my_file)`** este metoda pe care deja am folosit-o mai sus.

Când folosim funcția **`print(*my_file)`** pentru a afișa conținutul fișierului, Python utilizează un mecanism de dezambalare (unpacking) care permite iterarea peste fișier linie cu linie. Cu alte cuvinte, fiecare linie este citită pe rând și trimisă către funcția **`print()`** pentru a fi afișată. În acel moment, doar o linie din conținut este în memoria operativă. Astfel, memoria este utilizată mai eficient pentru fișierele mari, însă dacă aveți un fișier cu mii de linii, dar vreți să afișați doar o parte din acestea, această metodă nu este cea mai reușită.

2) Metoda **`read()`**

Dacă fișierul deschis are un volum mic comparativ cu memoria operativă a calculatorului, putem citi întregul text din fișier, folosind metoda **`read()`** aplicată variabilei-indicator la fișier.

În exemplul de mai jos, întregul conținut din fișierul **`zen.txt`** este citit direct în variabila **`text`**. Putem folosi tăierea șirului (string slicing) pentru a citi primele 32 de caractere:

```
# Deschidem fișierul
my_file = open('zen.txt', 'r')
# În variabila text citim conținutul fișierului
text = my_file.read()
# Afișăm lungimea textului
print(len(text))
# Afișăm primele 32 de caractere
print(text[:32])
# Închidem fișierul
my_file.close()
```

```
193
The Zen of Python, by Tim Peters
```

Dacă textul nu este voluminos, afișăm tot conținutul citit în variabila **`text`**:

```
# Deschidem fișierul
my_file = open('zen.txt', 'r')
# În variabila text citim conținutul fișierului
text = my_file.read()
# Afișăm textul
print(text)
# Închidem fișierul
my_file.close()
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
```

Când un fișier este citit în acest mod, toate caracterele, inclusiv caracterele de linie nouă, formează un singur șir lung de caractere atribuit variabilei **`text`**. E bine la aplicarea metodei **`read()`** să stocăm conținutul fișierului în variabilă, pentru că fiecare apel de citire epuizează resursa:

```
# Deschidem fișierul
my_file = open('zen.txt', 'r')
# Afișăm lungimea textului
print(len(my_file.read()))
# Afișăm repetat lungimea de caractere
print(len(my_file.read())) # resursă epuizată
# Închidem fișierul
my_file.close()
```

```
193
0
```

Putem limita numărul de caractere citite din fișier, indicând numărul de caractere în interiorul parantezelor: **`read([size])`**.

Dacă vrem să citim în continuare același sau alt număr de caractere, vom repeta instrucțiunea **`read([size])`**:

```
# Deschidem fișierul
my_file = open('zen.txt', 'r')
# Afișăm primele 20 de caractere din fișier
print(my_file.read(20))
# Afișăm următoarele 20 de caractere din fișier
print(my_file.read(20))
# Închidem fișierul
my_file.close()
```

```
The Zen of Python, b
y Tim Peters

Beauti
```

3) Metoda **`readline()`**

Această metodă citește o singură linie din fișier la fiecare apel și menține poziția de citire, astfel încât la următorul apel să returneze următoarea linie:

```
# Deschidem fișierul
my_file = open('zen.txt', 'r')
# Afișăm prima linie din fișier
print(my_file.readline())
# Afișăm următoarea linie
print(my_file.readline()) # este linie goală în fișier
# Afișăm a treia linie
print(my_file.readline())
# Închidem fișierul
my_file.close()
```

```
The Zen of Python, by Tim Peters
```

La fel ca și în cazul metodei **`read()`**, aplicând metoda **`readline()`** putem indica numărul de caractere care vor fi citite:

```
# Deschidem fișierul
my_file = open('zen.txt', 'r')
# Afișăm 10 caractere din linie
print(my_file.readline(10))
# Închidem fișierul
my_file.close()
```

```
The Zen of
```

4) Metoda **`readlines()`**

Pentru a afișa toate liniile din fișier, putem folosi metoda **`readlines()`**. În acest caz, conținutul va fi extras în formă de listă, în care fiecare element este un șir. Fiecare șir (element al listei) va finisa cu caracterul de linie nouă (**`\n`**). Metoda **`readlines()`** este utilă când vrem să procesăm toate liniile simultan, având flexibilitatea de a le accesa după nevoie, dar este mai potrivită pentru fișierele de dimensiuni mai mici, deoarece încarcă totul în memorie.

```
# Deschidem fișierul
my_file = open('zen.txt', 'r')
# Afișăm întregul conținut
print(my_file.readlines())
# Închidem fișierul
my_file.close()
```

```
['The Zen of Python, by Tim Peters\n', '\n', 'Beautiful is
better than ugly.\n', 'Explicit is better than implici
t.\n', 'Simple is better than complicated.\n', 'Flat is bet
ter than nested.']
```

5) Folosirea indicatorului la obiectul fișier în bucla **`for`**

O metodă de citire a conținutului din fișier linie cu linie constă în folosirea indicatorului la obiectul fișier ca secvență în bucla **`for`**:

```
my_file = open('zen.txt', 'r')
for line in my_file:
    print(line)
```

```
The Zen of Python, by Tim Peters



Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.
```

Pseudocodul pentru această buclă **`for`** ar fi: „Pentru fiecare linie din fișierul reprezentat de variabila-indicator la fișier, afișează linia în consolă”.

Când fișierul este citit folosind o buclă `for` în acest mod, Python separă datele din fișier în linii distincte, folosind caracterul de linie nouă **`(\n)`**. După fiecare linie citită, Python include caracterul de

linie nouă ca ultim caracter din variabila **`line`** pentru fiecare iterație a buclei **`for`**.

Deoarece bucla **`for`** citește linie cu linie pe rând, Python poate citi eficient liniile din fișiere foarte mari și nu rămâne fără memorie operativă pentru a stoca datele. Exemplul de mai sus poate fi utilizat pentru fișiere de orice volum, pentru că folosește foarte putină memorie, deoarece fiecare linie este citită, apoi ștearsă din memoria operativă.

6) Metoda de citire linie cu linie în bucla **`for`**

Pentru a afișa conținutul fișierului linie cu linie și fără caracterul de linie nouă, se recomandă folosirea buclei **`for`, în care iterativ se va afișa fiecare linie** și a metodei **`strip()`**, care elimină spațiile și caracterul de linie nouă de la sfârșitul liniei.

```
my_file = open('zen.txt', 'r')
# În variabila lines citim toate liniile in formă de listă de șiruri
lines = my_file.readlines()
# În buclă afișăm fiecare șir (element al listei) din rând nou
for line in lines:
    print(line.strip())

my_file.close()
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
```

5.4. Scrierea conținutului în fișier

În Python, scrierea se face doar după deschiderea fișierului, indicând unul din modurile de acces descrise în tabelul 5.1, care permit scrierea: (**`w`**, **`a`** sau **`x`**). Dacă fișierul indicat deja există, deschiderea acestuia în modul de scriere (w) va șterge tot conținutul vechi din fișier și va scrie un conținut nou. Fiți atenți! Dacă un fișier cu numele indicat nu există, acesta va fi creat.

Există două metode de scriere în fișier: metoda **`write()`** și **`writelines()`**.

1) Metoda **`write()`**

Metoda **`write()`** permite scrierea unui singur șir de caractere. Dacă avem nevoie să scriem mai multe rânduri, folosim această metodă pentru fiecare rând. Trebuie să ne asigurăm că gestionăm corect sfârșitul rândurilor pe măsură ce le scriem în fișier, adăugând în mod explicit caracterul de nouă linie (**`\n`**) când dorim să încheiem o linie. Instrucțiunea print adaugă automat o nouă linie, dar metoda de scriere nu adaugă automat linia nouă. După scrierea textului, fișierul deschis trebuie obligatoriu închis, astfel ne asigurăm că ultimul bit de date este scris fizic pe disc, astfel încât să nu se piardă dacă apare o pană de curent:

```
# Creăm un nou fișier zzen.txt
new_file = open('zzen.txt', 'w')
new_file.write('The Zen of Python, by Tim Peters\n')
# Adăugăm o linie goală
new_file.write('\n')
new_file.write(' Beautiful is better than ugly.')
new_file.close()
```

Dacă la deschiderea fișierului vom indica modul de adăugare (**`a`**), va fi adăugată o nouă linie de text fără a șterge conținutul existent.

```
# Deschidem fișierul cu modul de acces 'a'
my_file = open('zen.txt', 'a')
# Adăugăm un rând în plus la conținutul din fișier
my_file.write('\nSparse is better than dense.\n')
# Închidem fișierul
my_file.close()
```

Dacă nu adăugăm în fața liniei semnul de nouă linie, textul va fi scris în continuarea ultimei linii din fișier.

2) Metoda **`writelines()`**

În Python metoda writelines() este folosită pentru a scrie o listă de șiruri de caractere (mai multe linii de text dintr-o dată). Ca și în

cazul metodei **`write()`**, la sfârșitul fiecărei linii vom adăuga în mod explicit caracterul de nouă linie (**`\n`**):

```
# Creăm o listă cu două șiruri pentru a fi adăugate în fișier
linii = ['Readability counts.\n',
         "Special cases aren't special enough to break the rules.\n"]
# Deschidem fișierul cu modul de acces 'a'
my_file = open('zen.txt', 'a')
# Adăugăm liniile la sfârșitul conținutului existent din fișier
my_file.writelines(linii)
# Închidem fișierul pentru a salva modificările
my_file.close()
```

Verificăm rezultatul final obținut:

```
# Deschidem fișierul pentru a vedea liniile adăugate
my_file = open('zen.txt', 'r')
print(*my_file)
```

```
The Zen of Python, by Tim Peters

 Beautiful is better than ugly.
 Explicit is better than implicit.
 Simple is better than complicated.
 Flat is better than nested.
 Sparse is better than dense.
 Readability counts.
 Special cases aren't special enough to break the rules.
```

5.5. Redenumirea fișierelor

Funcția **`rename()`** este utilizată pentru redenumirea fișierelor în Python. Pentru că redenumirea fișierelor este efectuată la nivel de sistem de operare, mai întâi trebuie să importăm modulul **`os`**. Funcția **`rename()`** transmite două argumente: numele fișierului actual, numele nou al fișierului:

```
import os
os.rename('zen.txt', 'zenn.txt')
```

5.6. Lucrul cu fișierele în format CSV

Volume uriașe de date sunt gestionate pe scară largă în timpul utilizării site-urilor de socializare, comerț electronic, sănătate și genetică, aplicații IoT etc. Stocarea și partajarea datelor reprezintă principala cerere în mediul de lucru industrial de astăzi.

Fișierul CSV (comma-separated values) este un format de fișier simplu, dar universal care organizează datele tabelare într-o formă structurată. Fișierele CSV sunt gestionate cu ușurință, inclusiv pentru transferul de informații între aplicații (foi de calcul, SGBD, agende etc.). Din păcate, formatul fișierului nu are un standard strict definit, așa că există unele diferențe subtile între fișierele CSV generate de diferite aplicații.

Diferența dintre fișiere CSV și XLS(X) este prezentată în tabelul 5.3.

Tabelul 5.3. Diferența dintre formatele CSV și XLS(X)

Formatul CSV	***Formatul Excel***
Formatul CSV este un format de text simplu cu o serie de valori separate prin virgule.	Excel este un fișier binar care conține informații despre toate foile de calcul dintr-un fișier, inclusiv atât conținutul, cât și formatarea.
CSV poate fi deschis cu orice editor de text din Windows, cum ar fi Notepad, MS Excel etc.	Fișierele XLS pot fi citite doar de aplicații care au fost special scrise pentru a citi formatul lor și pot fi scrise doar în același mod.
CSV este un format pentru salvarea informațiilor tabelare într-un fișier text delimitat cu extensia .csv.	Excel este o foaie de calcul care salvează fișiere în propriul format .xls sau .xlsx.
Importul fișierelor CSV poate fi mult mai rapid și, de asemenea, consumă mai puțină memorie.	Excel consumă mai multă memorie la importul datelor.

Conținutul unui fișier CSV este păstrat în formă de tabel, unde coloanele sunt divizate prin virgule sau alte simboluri (; , \t, ș.a.).

În interior, conținutul unui fișier arată apoximativ așa:

Name, Age, City
Ann, 35, Brussels
John, 30, Bologna

Există două metode de lucru cu fișierele CSV în Python.

1) Utilizarea modulului încorporat CSV.

Modulul CSV oferă posibilitatea de a prelucra fișierele CSV similar cu cele de tip TXT. Ca și orice alt modul, pentru a putea lucra cu funcțiile acestuia, modulul trebuie importat. Vom crea fișierul „stud.csv" în care vom adăuga date despre trei persoane. Vom utiliza instrucțiunea **`with`** pentru deschiderea/crearea fișierului. Pentru scrierea datelor linie cu linie într-un fișier CSV putem utiliza metoda **`writer.writerow()`**:

```
import csv
with open('stud.csv', mode='w', newline='',
          encoding='utf-8') as file:
    writer = csv.writer(file)
    writer.writerow(["Name", "Age", "City"])
    writer.writerow(["Ann", 35, "Brussels"])
    writer.writerow(["John", 30, "Bologna"])
    writer.writerow(["Maria", 29, "Budapest"])
```

Reamintim că utilizând instrucțiunea **`with`**, Python va închide fișierul **`stud.csv`** automat. Vom afișa conținutul fișierului creat utilizând funcția **`print()`**:

```
m_file = open('stud.csv', 'r+')
print(*m_file)
m_file.close()
```

```
Name,Age,City
 Ann,35,Brussels
 John,30,Bologna
 Maria,29,Budapest
```

Vom afișa conținutul fișierului creat, utilizând funcția **`reader()`** din modulul **`csv`**:

```
with open('stud.csv',  newline='', encoding='utf-8') as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)
```

```
['Name', 'Age', 'City']
['Ann', '35', 'Brussels']
['John', '30', 'Bologna']
['Maria', '29', 'Budapest']
```

2) Utilizarea bibliotecii **Pandas** (vezi § 8.4.2)

Lista metodelor care pot fi aplicate fișierelor în Python este inclusă în tabelul 5.4.

Tabelul 5.4. Metode aplicate fișierelor în Python

Metoda	*Descrierea*
`file.close()`	închide un fișier deschis
`file.fileno()`	returnează un descriptor de fișier întreg
`file.flush`	șterge buffer-ul intern
`file.isatty()`	returnează True dacă fișierul este atașat la terminal
`file.next()`	returnează următoarea linie a fișierului
`file.read(n)`	citește primele n caractere ale unui fișier
`file.readline()`	citește o linie dintr-un șir de caractere sau fișier
`file.readlines()`	citește și returnează o listă cu toate liniile dintr-un fișier
`file.seek(offset [, whence])`	stabilește poziția curentă în fișier
`file.seekable()`	verifică dacă fișierul acceptă acces aleatoriu. Returnează True, dacă da
`file.tell()`	returnează poziția curentă în fișier
`file.truncate(n)`	reduce dimensiunea unui fișier. Dacă se specifică n, fișierul este trunchiat la n octeți, dacă nu - la poziția curentă
`file.write(str)`	adaugă șirul str la fișier
`file.writelines (sequence)`	adaugă o secvență de linii la un fișier

În concluzie, fișierele permit stocarea și accesarea permanentă a datelor, dincolo de rularea temporară a programului. Prin citirea și scrierea fișierelor, programele Python pot interacționa cu date externe, cum ar fi texte, tabele sau configurații. Stăpânirea lucrului cu fișiere este importantă pentru automatizarea sarcinilor, salvarea rezultatelor și prelucrarea eficientă a informațiilor.

Sarcini

1. **Citirea unui fișier și afișarea conținutului.** Scrieți un program care deschide un fișier text numit text.txt și afișați conținutul său linie cu linie.
2. **Numărarea cuvintelor dintr-un fișier**. Scrieți un program care deschide un fișier și numără câte cuvinte conține. Afișați rezultatul sub forma: fișierul conține X cuvinte.
3. **Scrierea unui fișier nou**. Scrieți un program care cere utilizatorului să introducă 5 propoziții de la tastatură, apoi le salvează într-un fișier propozitii.txt, fiecare propoziție pe un rând nou.

6. FUNCȚII

Obiective

La finalul acestui capitol, studenții vor fi capabili:

- Să înțeleagă rolul și structura funcțiilor în Python, recunoscând importanța modularității, reutilizării codului și separării logicii programului în subprograme clar definite.
- Să definească și să apeleze funcții personalizate, folosind sintaxa specifică și să aplice corect apelurile de funcții în diverse contexte.
- Să distingă între variabilele locale și globale, să înțeleagă comportamentul acestora în cadrul funcțiilor.
- Să utilizeze diverse tipuri de argumente într-o funcție, inclusiv argumente poziționale, cuvinte-cheie, implicite și de lungime variabilă, adaptând apelul funcției la diferite cerințe.
- Să scrie și să utilizeze funcții imbricate, în scopul încapsulării codului și al limitării accesului la logică internă, dezvoltând programe mai clare și mai sigure.
- Să creeze și să aplice funcții lambda pentru sarcinile simple, precum și să le utilizeze eficient ca argumente în funcțiile de ordin superior precum map(), filter(), sorted() și reduce().

Funcțiile sunt unul dintre conceptele fundamentale ale programării. Funcțiile organizează mai eficient codul, facilitează modularitatea și reutilizarea codului, permițând programatorilor să împartă eficient sarcinile complexe în subsarcini mai simple.

6.1. Definirea și apelarea funcției

În Python, o funcție este definită folosind cuvântul-cheie **`def`**, după care urmează numele funcției și paranteze rotunde. După paranteze se pun două puncte, iar corpul funcției începe din rând nou cu indentare.

Sintaxa de bază de definire a funcției:

```
def <numele_funcției>():
    corpul funcției
```

Numele funcțiilor sunt create după aceleași reguli, ca și numele variabilelor, dar nu trebuie să coincidă cu numele acestora, și viceversa.

Pentru executarea codului din funcție, aceasta trebuie apelată. Funcția se apelează, dacă scriem numele funcției și adăugăm paranteze rotunde.

Sintaxa de bază de apelare a funcției:

```
<numele_funcției>()
```

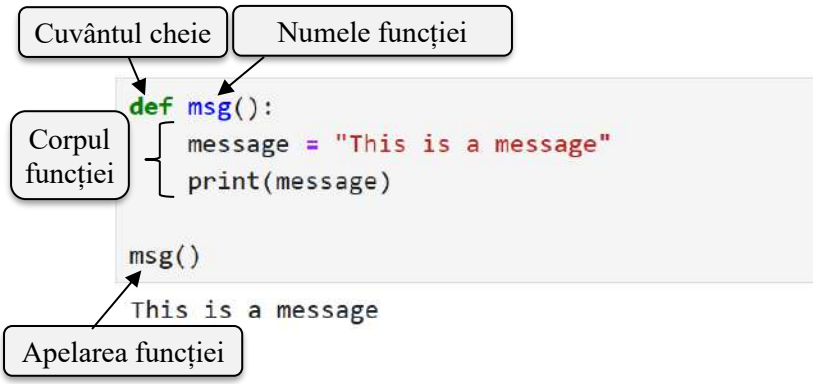

Figura 6.1. Definirea și apelarea funcției în Python

Notă! Funcția trebuie să fie definită până la apelarea acesteia.

6.2. Parametrii și argumentele funcției

Parametrii funcției se indică în paranteze la definirea funcției.

Argumentele funcției sunt variabilele transmise funcției la apelarea acesteia. Argumentele permit funcției să preia datele de intrare pentru prelucrare:

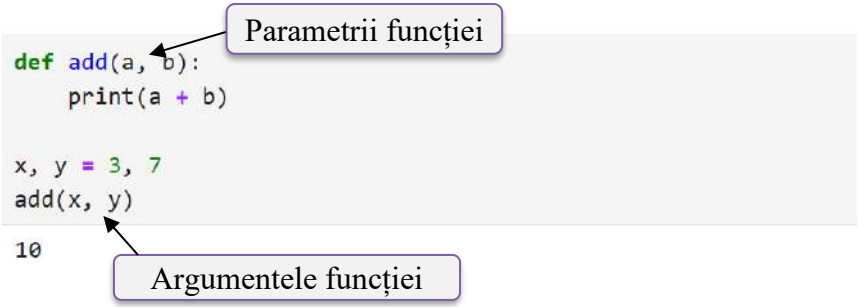

Figura 6.2. Parametrii și argumentele funcției

6.3. Returnarea rezultatului execuției funcției

Corpul unei funcții conține aproape întotdeauna un operator de returnare a rezultatului – cuvântul-cheie **`return`**. Operatorul **`return`** poate apărea oriunde în corpul funcției. Când interpretorul întâlnește acest operator, rezultatul execuției este returnat, iar codul de mai departe din funcție este ignorat:

```
# Definim funcția m_max
def m_max(x, y):
    # Verificăm care din două variabile este mai mare și o returnăm
    if x > y:
        return x
    else:
        return y
# Variabilei max_v atribuim rezultatul returnat
max_v = m_max(3, 5)
print(max_v)
# Afișăm rezultatul execuției funcției
print(m_max(8, 5))
```

```
5
8
```

Dacă după operatorul **`return`** nu urmează nici-o expresie, funcția returnează obiectul **`None`**. Chiar dacă funcția nu returnează nimic, este o bună practică să fie utilizat în mod explicit operatorul **`return:`**

```
def prod_2_nums(a, b):
    print(a * b)
    return

prod_2_nums(3, 4)
print(prod_2_nums(2, 3))
```

```
12
6
None
```

Obiectul **None** reprezintă absența oricărei valori sau „nimic” și nu poate fi utilizată în careva expresie pentru că va returna eroare:

```
print(None + 2)
```

```
---------------------------------------------------------------
------------------
TypeError                                 Traceback (most
recent call last)
Cell In[16], line 1
----> 1 print(None + 2)
```

Există două cazuri în care **None** este util:

1) Când este atribuit unei variabile sau ca valoare returnată dintr-o funcție;
2) Când este comparat cu o variabilă.

În exemplul de mai jos variabila var_value este comparată cu None și dacă rezultatul este True, interpretorul afișează mesajul corespunzător:

```
var_value = None
if var_value is None:
    print('No value for var_value')
```

```
No value for var_value
```

6.4. Tipuri de argumente ale funcției

În Python, funcțiile pot primi mai multe tipuri de argumente, oferind flexibilitate.

1) Argumente poziționale (positional arguments)

Dacă la definirea funcției a fost specificat un anumit număr de parametri și ordinea acestora, atunci funcția trebuie apelată cu același număr de argumente și în aceeași ordine stabilită:

```
# Funcția de conversie a timpului în secunde
def hour_to_sec(hour, min, sec):
    return hour * 360 + min * 60 + sec
# Apelăm funcția, numărul și ordinea argumentelor contează
print(hour_to_sec(2, 10, 32))
```

```
1352
```

Dacă ordinea argumentelor transmise va fi indicată incorect, rezultatul returnat va fi greșit.

2) Argumente cuvinte-cheie (keyword arguments)

În Python, argumentele cuvinte-cheie sunt utilizate la apelarea funcției și obligatoriu vor avea același nume ca și la definirea funcției. În acest caz, putem specifica o ordine arbitrară a argumentelor:

```
# Funcția de conversie a timpului în secunde
def hour_to_sec(hour, min, sec):
    return hour * 360 + min * 60 + sec
# Apelăm funcția, ordinea argumentelor poate fi alta
print(hour_to_sec(min = 10, sec = 32, hour = 2))
```

```
1352
```

3) Argumente implicite (default arguments)

Valorile argumentelor implicite sunt stabilite la definirea funcției. În acest caz, la apelarea funcției putem omite aceste argumente:

```
# Funcția descrie doi parametri, 'age' având valoarea implicită 25
def pers(name, age = 25):
    print(f"{name} - {age} years old")
# Transmitem funcției 2 argumente
pers('Tom', 19)
# Transmitem funcției 1 argument
pers('John')
```

```
Tom - 19 years old
John - 25 years old
```

În cazul utilizării argumentelor implicite, trebuie să fim atenți la ordinea argumentelor. Dacă ordinea argumentelor la apelare se va schimba, vom obține rezultat incorect.

4) Argumente de lungime variabilă/arbitrară (variable-length arguments)

Există situații, când nu cunoaștem dinainte câte argumente va trebui să accepte funcția. În astfel de cazuri vom folosi argumente de lungime variabilă. Python oferă două variante de utilizare a argumentelor de lungime arbitrară:

a) *args (argumente variabile non-cheie)

Acestea sunt specificate printr-un nume de variabilă arbitrară precedat de un asterisc (*) și captează argumente poziționale suplimentare în formă de tuplu:

```
def print_args(*args):
    print(args)
print_args('Tom')
print_args('John', 32)
print_args('Gabi', 25, 'Valencia')
```

```
('Tom',)
('John', 32)
('Gabi', 25, 'Valencia')
```

b) **kwargs (argumente variabile cuvinte-cheie)

Argumentele suplimentare sunt specificate prin precedarea de două asteriscuri (**) și sunt captate în formă de cuvinte-cheie ca dicționar:

```
def profile(**details):
    for key, value in details.items():
        print(f"{key}:{value}")
profile(name = "Maria", city = "Bucharest", age = 25)
```

```
name:Maria
city:Bucharest
age:25
```

6.5. Variabile globale și locale

În Python, în funcție de unde sunt definite și unde sunt accesate, variabilele pot fi globale sau locale.

6.5.1. Variabile globale

Variabilele declarate în afara unei funcții, dar accesibile în cadrul acesteia, se numesc **globale**:

```
def f():
    print(a)
# Definim variabila in afara funcției
a = 10
f()
```

```
10
```

Aici variabila **`a`** este definită în afara funcției **`f()`**. Funcția afișează această valoare, chiar dacă aceasta este inițializată după ce funcția a fost definită, **dar înainte de a fi apelată**.

6.5.2. Variabile locale

Dacă inițializăm o variabilă în interiorul funcției, aceasta nu este „vizibilă” în afara funcției. Variabilele inițializate în interiorul funcției se numesc **locale**:

```
def f1():
# Definim variabila in interiorul funcției
    b = 5
    print(b)

f1()
print(b)
```

```
5
---------------------------------------------------------------------
NameError                                 Traceback (most recent
Cell In[14], line 7
      4     print(b)
      6 f1()
----> 7 print(b)

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

După ieșirea din funcția **`f1()`**, variabila **`b`** a devenit indisponibilă.

6.5.3. Protejarea variabilelor locale

Pentru a proteja variabila locală, modificăm valoarea variabilei globale în interiorul funcției:

```
def f2():
    c = 1
    print(c)
c = 0
f2()
print(c)
```

```
1
0
```

Observăm că s-au afișat valorile 1 și 0. Chiar dacă valoarea variabilei **`c`** s-a schimbat în interiorul funcției **`f2()`**, în afara funcției valoarea lui **`c`** rămâne aceeași – 0 . Acest lucru se face pentru a proteja variabilele globale împotriva modificării accidentale din interiorul unei funcții.

6.5.4. „Conflictul" variabilelor

În unele cazuri apar „situații de conflict" între variabilele locale și globale:

```
def f3():
    print(d)
    if False:
        d = 0
d = 1
f3()
```

```
---------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent
Cell In[15], line 6
      4         d = 0
      5 d = 1
----> 6 f3()
```

În funcția `f3()`, variabila `d` devine locală. Deoarece funcția conține o comandă care modifică variabila `d`, chiar dacă nu este executată niciodată, interpretorul nu poate urmări acest lucru și returnează eroare.

6.5.5. Cuvântul-cheie global

Pentru ca o funcție să poată schimba valoarea unei variabile globale, o declarăm în interiorul funcției ca globală, utilizând cuvântul-cheie global:

```
def f3():
    global d
    d = 5
    print(d)
    if False:
        d = 1
f3()
print(d)
```

```
5
5
```

Variabila **d** este declarată ca globală. Modificarea acesteia în interiorul funcției duce la disponibilitatea acesteia și din exterior.

6.6. Funcții și liste

Listele pot fi argumente ale funcțiilor:

```
def sumList(lst):
    s = 0
    for elem in lst:
        s += elem
    return s
# În calitate de argument transmitem o listă
print(sumList([3, 7, 2]))
```

```
12
```

Însă dacă vom încerca în loc de listă să transmitem în funcția sumList() un număr (întreg sau zecimal), vom obține eroare:

```
def sumList(lst):
    s = 0
    for elem in lst:
        s += elem
    return s
print(sumList(4)) # ca argument transmitem un numar
```

```
---------------------------------------------------------------
-----------------
TypeError                                 Traceback (most
recent call last)
Cell In[19], line 6
      4         s += elem
      5     return s
----> 6 print(sumList(4))
```

Funcțiile pot returna liste în bucla **for**, utilizând funcția **range()** și metoda **insert()**:

```
def generateList(n):
    newList = []
    for elem in range(0, n):
        newList.insert(0, elem)
    return newList

print(generateList(5))
```

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

6.7. Funcții imbricate

Funcțiile imbricate sunt funcțiile definite în interiorul altor funcții. Acestea sunt utilizate în diverse situații. De exemplu, ele pot servi la încapsularea codului care nu ar trebui să fie accesibil din exterior:

```
# Definim funcția externă
def outer_func(x):
    # Definim funcția internă
    def inner_func(y):
        return x * y
    return inner_func(x)

result = outer_func(5)
print(result)
```

```
25
```

Funcția externă returnează rezultatul apelării funcției interne. Funcția internă nu este accesibilă din exterior, ceea ce asigură încapsularea logicii.

Notă! Funcțiile imbricate au acces la variabilele funcției externe.

6.8. Funcții lambda

În Python funcțiile **lambda** sunt funcții scurte, fără nume (anonime).

Sintaxa de bază:

```
lambda <argumente>: <expresie>
```

```
double = lambda x: x * 2
print(double(5))
```

```
10
```

Aici **x** este argument, iar **x * 2** – expresie ce calculează și returnează rezultatul.

Echivalentul acestei funcții:

```
def double(x):
    return x * 2

print(double(5))
```

```
10
```

Un alt exemplu al funcției **lambda** cu trei argumente:

```
calc = lambda x, y, z: x / y + z
v = calc(6, 3, 2)
print(v)
```

```
4.0
```

În funcțiile **lambda** deseori în calitate de expresie sunt folosiți **operatori ternari** (operatori cu trei operanzi):

```
maximum = lambda x, y: x if x > y else y
print(maximum(8, 5))
```

```
8
```

6.8.1. Utilizarea funcției lambda în calitate de argument

Deseori funcțiile **lambda** sunt folosite ca argumente pentru funcțiile de ordin superior, aplicabile secvențelor iterabile: **`map()`**, **`filter()`**, **`sorted()`**, **`reduce()`** ș.a.

1) Funcția `map()`

Funcția **`map()`** funcționează ca un conveier, aplicând o anumită acțiune fiecărui element al secvenței.

Sintaxa de bază:

```
map(function, iterable[, iterable2, ...])
```

- function este funcția lambda de transformare care se aplică fiecărui element al iterabilului;
- iterable este obiectul iterabil căruia i se aplică funcția de transformare;
- [, iterable2, …] este un parametru opțional care trece obiecte iterabile suplimentare. Dacă treceți două liste, funcția va prelua primul element din prima listă și primul element din a doua listă, apoi al doilea element și așa mai departe;

```
nums = [1, 2, 3, 4]
squares = list(map(lambda x: x ** 2, nums))

print(squares)
```

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

2) Funcția `filter()`

Funcția **`filter()`** filtrează elementele unei secvențe în funcție de anumite criterii.

Sintaxa de bază:

```
filter(function|None, iterable)
```

- function este funcția care conține condiții pentru filtrarea valorilor de intrare. Returnează True sau False. Dacă treceți None, va elimina toate elementele, cu excepția celor care returnează True în mod implicit;

- iterable este obiectul iterabil ale cărui elemente sunt testate pentru conformitatea cu o condiție. Fiecare apel de funcție folosește unul dintre obiectele secvenței pentru testare;

```
nums = [1, 2, 3, 4, 5]
odd = list(filter(lambda x: x % 2, nums))
print(odd)
```

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

3) Funcția `sorted()`

Funcția **`sorted()`** returnează o nouă listă sortată obținută din obiectul iterabil.

Sintaxa de bază:

`sorted(iterable [, key = None] [, reverse = False])`

- iterable este obiectul iterabil ale cărui elemente vor fi sortate;
- [key] este un parametru opțional. Dacă specificați o cheie, sortarea va fi efectuată în funcție de cheie, care este definită prin funcția lambda;
- [reverse] este un parametru opțional. În mod implicit, sortarea se face în ordine crescătoare. Pentru a sorta în ordine descrescătoare, trebuie să specificați reverse = True.

```
pairs = [(1, 'a'), (3, 'b'), (2, 'd'), (4, 'c')]
# Sortăm perechile după indexul 0
sorted_pairs_0 = sorted(pairs, key = lambda x: x[0])
# Sortăm perechile după indexul 1
sorted_pairs_1 = sorted(pairs, key = lambda x: x[1])
print(sorted_pairs_0)
print(sorted_pairs_1)
```

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

4) Funcția `reduce()`

Funcția **`reduce()`** se aplică elementelor (perechi) ale unei secvențe, reducându-le la o singură valoare.

Sintaxa de bază:

```
reduce(function, iterable [, initializer])
```

- function preia două argumente și returnează o valoare;
- iterable este obiectul iterabil ale cărui elemente vor fi reduse la o singură valoare;
- [initializer] este parametru opțional care specifică valoarea inițială pentru a începe procesul de acumulare.

Pentru că funcția **`reduce()`** nu este o funcție încorporată, dar face parte din modulul **`functools`**, aceasta trebuie mai întâi importată:

```
from functools import reduce
nums = [1, 2, 3, 4]
prod = reduce(lambda x, y: x * y, nums)
print(prod)
```

```
24
```

Elementele din listă sunt înmulțite pe rând: 1 * 2, apoi rezultatul obținut (2) devine primul argument înmulțit cu 3. În continuare rezultatul obținut (6) este înmulțit cu 4.

6.9. Funcții recursive

Recursivitatea este o tehnică prin care o funcție se autoapelează.

Funcțiile recursive constau din 2 părți:

1. Cazul de bază este condiția în care funcția încetează să se mai apeleze și returnează rezultatul.
2. Cazul recursiv este condiția în care o funcție se autoinvocă pentru a rezolva o subproblemă mai mică.

Un exemplu de funcție recursivă de calculare a factorialului este prezentat mai jos:

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)

print(factorial(5))
```

```
120
```

Pentru exemplul menționat, cazul de bază este: dacă n == 1, atunci n! = 1; cazul recursiv: dacă n > 1, atunci n! = n * (n-1)!

Un exemplu de funcție recursivă de generare a șirului Fibonacci este prezentat mai jos:

```
def fib(x):
    if x <= 0:
        return 0
    elif x == 1:
        return 1
    else:
        return fib(x - 1) + fib(x - 2)
# Afișăm șirul Fibanacci în buclă
for i in range(10):
    print(f"{fib(i)}", end =" ")
```

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

Aici cazul de bază este: dacă x <= 0, se returnează 0; dacă x = 1, se returnează 1; cazul recursiv: funcția se autoapelează, micșorând valorile argumentelor (x – 1, x – 2), până când x = 1 (cazul de bază).

Funcțiile recursive sunt foarte utile, dar pot avea unele riscuri dacă nu sunt gestionate corect. Enumerăm cele mai importante riscuri:

1. **Depășirea limitei de recursivitate**. Python are o limită implicită de adâncime a recursivității (aprox. 1000 apeluri). Dacă funcția nu atinge cazul de bază, programul poate intra

într-o buclă infinită recursivă, ceea ce duce la eroarea "RecursionError: maximum recursion depth exceeded".

2. **Consum mare de memorie (stack overflow)**. La fiecare apel recursiv se creează un nou cadru de stivă (stack frame). Multe apeluri pot duce la consum mare de memorie, în special la algoritmi prost optimizați.
3. **Performanță scăzută**. O implementare naivă a unei funcții recursive poate fi ineficientă, deoarece recalculează aceleași valori de mai multe ori.

Prin urmare, când se dezvoltă o funcție recursivă este necesar, în primul rând, să se formalizeze condițiile de terminare a recursiunii și să se prevadă cazul finalizării acesteia.

În concluzie, funcțiile sunt foarte importante în scrierea de cod eficient, organizat și reutilizabil, iar programele complexe devin gestionabile prin utilizarea de funcții care colaborează între ele.

Sarcini

1. **Funcție de calcul al mediei aritmetice**. Scrieți o funcție `media_aritmetica()` care primește o listă de numere și returnează media aritmetică a acestora. Testați funcția cu diferite tipuri de obiecte iterabile (listă, tuplu, set).

2. **Funcție de verificare a numărului prim**. Scrieți o funcție `este_prim(n)` care primește un număr întreg și returnează `True` dacă este prim și `False` în caz contrar.

3. **Funcție pentru formatarea unui text**. Scrieți o funcție `formateaza_text(text, majuscule=False)` care primește un șir de caractere și returnează:
   - textul cu toate caracterele mari dacă majuscule=True;
   - textul cu prima literă mare în caz contrar.

7. ERORI ȘI EXCEPȚII ÎN PYTHON

Obiective

La finalul acestui capitol, studenții vor fi capabili:

- Să înțeleagă conceptul de excepție și importanța gestionării acestora în scrierea de programe robuste.
- Să identifice tipurile comune de erori care pot apărea în timpul execuției unui program Python.
- Să implementeze tratarea diferențiată a excepțiilor, folosind blocuri multiple except pentru a trata diferite tipuri de erori.
- Să exploreze resurse oficiale Python pentru a înțelege gama completă a excepțiilor disponibile și utilizarea lor.

Ce se va întâmpla dacă codul scris în Python conține erori? Dar dacă utilizatorul introduce din greșeală sau intenționat valori nepermise? Pentru astfel de cazuri Python oferă un mecanism încorporat de gestionare a comportamentului unui program în timpul rulării, chiar dacă au loc erori importante sau așa-numite excepții.

Sintaxa de bază:

```
try:
    <Instrucțiun care vor rula dacă nu sunt
    erori/excepții>
except:
    <Instrucțiuni care vor rula în cazul
    excepției>
```

Vom analiza un exemplu în care solicităm utilizatorului să introducă 2 numere, al doilea fiind egal cu 0, și îi returnăm rezultatul împărțirii acestor numere. Pentru că al doilea număr introdus este 0, apare eroarea "**ZeroDivisionError**" (Eroare de divizare la zero).

```
num_1 = float(input('Introdu primul numar'))
num_2 = float(input('Introdu al doilea numar'))
print('Rezultatul impartirii a doua numere este', num_1 / num_2)
```

```
Introdu primul numar 5
Introdu al doilea numar 0
---------------------------------------------------------------------
-----------
ZeroDivisionError                         Traceback (most recent
call last)
Cell In[6], line 3
      1 num_1 = float(input('Introdu primul numar'))
      2 num_2 = float(input('Introdu al doilea numar'))
----> 3 print('Rezultatul impartirii a doua numere este', num_1
/ num_2)
```

Încercăm să introducem litere în loc de numere:

```
num_1 = float(input('Introdu primul numar'))
num_2 = float(input('Introdu al doilea numar'))
print('Rezultatul impartirii a doua numere este', num_1 / num_2)
```

```
Introdu primul numar a
---------------------------------------------------------------------
-----------
ValueError                                Traceback (most recent
call last)
Cell In[7], line 1
----> 1 num_1 = float(input('Introdu primul numar'))
      2 num_2 = float(input('Introdu al doilea numar'))
      3 print('Rezultatul impartirii a doua numere este', num_1
/ num_2)
```

Din nou avem eroare, dar de această dată "**ValueError**" (valoare eronată).

Pentru a evita afișarea erorilor în cazul introducerii valorilor nepermise, Python oferă structura **`try-except`**.

În exemplul de mai jos, în cazul apariției oricărei erori, se va afișa cuvântul „Excepție!":

```
try:
    num_1 = float(input("Introdu primul numar"))
    num_2 = float(input("Introdu al doilea numar"))
    print('Rezultatul impartirii a doua numere este',
          num_1 / num_2)
except:
    print('Exceptie!')
```

```
Introdu primul numar 5
Introdu al doilea numar 0
Exceptie!
```

Fiind un limbaj flexibil, Python oferă posibilitatea de gestionare a erorilor în funcție de tipul acestora. Vom analiza gestionarea erorilor de mai sus. Vom folosi o buclă **`while`** pentru a continua rularea programului până când vor fi introduse valori valide:

```
while True:
    try:
        num_1 = float(input("Introdu primul numar"))
        num_2 = float(input("Introdu al doilea numar"))
        print('Rezultatul impartirii a doua numere este',
              num_1 / num_2)
        break
    except ZeroDivisionError:
        print('Nu putem imparti la 0, mai incearca')
    except ValueError:
        print('Valoare nevalida, mai incearca')
    except:
        print('S-a produs alta eroare, mai incearca')
```

```
Introdu primul numar 5
Introdu al doilea numar 0
Nu putem imparti la 0, mai incearca
Introdu primul numar A
Valoare nevalida, mai incearca
Introdu primul numar 5
Introdu al doilea numar 2
Rezultatul impartirii a doua numere este 2.5
```

Observăm că am folosit operatorul **except** de 3 ori. În primul **except** am specificat eroarea de divizare la zero. În al doilea **except** am specificat eroarea de valoare eronată, iar al treilea **except** include toate celelalte erori.

Lista cu toate excepțiile/tipurile de erori pentru Python versiunea 3.15 poate fi găsită pe adresa https://docs.python.org/3.15/library/exceptions.html#Exception.

Ca și structura condițională **if** sau buclele **while** și **for**, structura **try-except** poate avea o ramură **else**. Blocul din **else** se va executa după **try** dacă nu au fost întâmpinate careva erori:

```
while True:
    try:
        num_1 = float(input("Introdu primul numar"))
        num_2 = float(input("Introdu al doilea numar"))
        print('Rezultatul impartirii a doua numere este',
              num_1 / num_2)
    except:
        print('S-a produs o eroare, mai incearca')
    else:
        print('Ai reusit!')
        break
```

```
Introdu primul numar 5
Introdu al doilea numar 2
Rezultatul impartirii a doua numere este 2.5
Ai reusit!
```

Putem observa că acum operatorul **break** a fost mutat în blocul **else**. Acest lucru a fost făcut pentru a exclude oprirea programului în cazul introducerii valorilor valide. Dar dacă vom exclude cu totul operatorul **break**, vom avea o buclă infinită.

Următoarea ramură opțională a structurii **try-except-else** formează operatorul **finally**.

Blocul din **finally** va fi executat în orice caz, fie după **try-else**, fie după **except:**

```
while True:
    try:
        num_1 = float(input("Introdu primul numar"))
        num_2 = float(input("Introdu al doilea numar"))
        print('Rezultatul impartirii a doua numere este',
              num_1 / num_2)
    except:
        print('S-a produs o eroare, mai incearca')
    else:
        print('Ai reusit!')
        break
    finally:
        print('Acest mesaj se afiseaza intotdeauna')
```

```
Introdu primul numar 5
Introdu al doilea numar 0
S-a produs o eroare, mai incearca
Acest mesaj se afiseaza intotdeauna
Introdu primul numar 5
Introdu al doilea numar 2
Rezultatul impartirii a doua numere este 2.5
Ai reusit!
Acest mesaj se afiseaza intotdeauna
```

Operatorul **`finally`** este necesar pentru a închide fișiere, pentru eliberarea resurselor (memorie, conexiuni la baze de date, rețele etc.) sau pentru curățenia finală într-un program (logging, notificări, confirmări etc.).

În concluzie, gestionarea erorilor și excepțiilor în Python permite prevenirea blocării programului și oferă utilizatorului mesaje clare în caz de probleme.

Prin utilizarea blocurilor try-except, codul devine mai robust, mai sigur și mai ușor de întreținut, chiar și în situații neașteptate.

Sarcini

1. **Conversie sigură la întreg**. Scrieți o funcție `citeste_int(val)` care încearcă să convertească valoarea primită în număr întreg. Gestionați excepția dacă valoarea nu este convertibilă și returnează None.
2. **Calculator simplu cu excepții**. Scrieți o funcție `calculator(a, b, operatie)` care efectuează operații de bază (+, -, *, /) și gestionați erori precum: a) împărțirea la zero; b) operație invalidă.

8. MODULE ȘI PACHETE

Obiective

La finalul acestui capitol, studenții vor fi capabili:

- Să explice conceptul de module și pachete și rolul acestora în organizarea, reutilizarea și scalarea codului în Python.
- Să identifice și să utilizeze diferite modalități de importare a modulelor și pachetelor.
- Să deosebească tipurile de module: încorporate, terțe și personalizate.
- Să utilizeze eficient modulele standard importante în rezolvarea problemelor specifice.
- Să implementeze programe care folosesc biblioteci externe populare, cum ar fi NumPy, Pandas și Matplotlib, pentru calculele numerice, manipularea datelor și vizualizarea grafică.
- Să aplice metode de organizare modulară a codului în proiecte Python, folosind bune practici de structurare cu module și pachete.

Una dintre caracteristicile-cheie ale limbajului Python este flexibilitatea și puterea sa de a organiza codul prin module și pachete. Aceste instrumente oferă mijloace eficiente de structurare și gestionare a spațiilor de nume, făcând codul mai accesibil, ușor de întreținut și scalabil.

8.1. Module

Pe măsură ce programele cresc în dimensiune, apare necesitatea împărțirii lor în mai multe fișiere pentru a ușura întreținerea. Fișierele sunt împachetate în module, apoi, la necesitate, încărcate în programe.

Organizarea codului în module permite reutilizarea acestuia în diferite părți ale unui program fără a fi nevoie să fie duplicat.

Modulele din Python sunt organizate ierarhic, ca directoarele. În același timp, orice program conține un fișier principal (de obicei main()), la care se pot „conecta” fișierele (modulele principale).

Așadar, un modulul Python este un fișier cu extensia .py care conține definiții și instrucțiuni Python. Un modul poate include funcții, clase și variabile, precum și un cod executabil.
Numele modulului corespunde cu numele fișierului (figura 8.1).

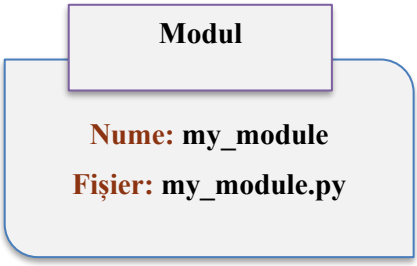

Figura 8.1. Modulul în Python

Modulele îndeplinesc cel puțin trei funcții importante:

- reutilizarea codului;
- gestionarea spațiului de adrese;
- globalizarea serviciilor și datelor.

8.1.1. Importarea unui modul

Pentru a putea utiliza funcțiile, variabilele, clasele definite în module, acestea trebuie importate. Importul modulului în alt fișier sau modul se efectuează folosind cuvântul-cheie **`import`**. Există câteva moduri de import al unui modul/pachet:

1) Folosim cuvântul `import` și numele modulului:

```
import <nume_modul>
```

Să importăm modulul încorporat **`math`**, care oferă funcții matematice. Pentru apelarea unei funcții din modul vom indica numele modulului și numele funcției, divizându-le prin punct:

```
# Importăm modulul math (oferă funcții matematice)
import math
x, y = 2, 3
# Apelăm funcția pow
print(math.pow(x, y))
# Apelăm funcția sqrt
print(math.sqrt(9))
```

```
8.0
3.0
```

2) **Folosim cuvântul `import`, numele modulului, cuvântul `as` și pseudonimul acestuia:**

`import <nume_modul> as <pseudonim>`

În acest caz, la apelarea funcțiilor în loc de numele modulului vom folosi pseudonimul:

```
# Importăm modulul math sub pseudonimul mt
import math as mt
x, y = 2, 2
# Apelăm funcția pow
print(mt.pow(x, y))
# Apelăm funcția sqrt
print(mt.sqrt(25))
```

```
4.0
5.0
```

Dacă după importul modulului cu indicarea pseudonimului vom încerca să indicăm numele deplin al modulului, vom obține eroarea **`NameError`:**

```
# Importăm modulul math cu pseudonimul mt
import math as mt
# Apelăm funcția sqrt folosind numele, nu pseudonimul
print(math.sqrt(25))
```

```
---------------------------------------------------------------
-----------------
NameError                                 Traceback (most
recent call last)
```

3) **Specificăm funcția sau funcțiile necesare din modul, utilizând cuvântul `from`:**

`from <nume_modul> import <func1>[, <func2>][, <func3>]`…

În acest caz, la apelarea funcțiilor nu mai indicăm numele modulului:

```
# Importăm din modulul math funcțiile pow și sqrt
from math import pow, sqrt
x, y = 3, 2
# Apelăm funcția pow
print(pow(x, y))
# Apelăm funcția sqrt
print(sqrt(16))
```

```
9.0
4.0
```

4) **Importăm toate funcțiile din modul, utilizând asteriscul (*) după cuvântul `import`:**

`from <nume_modul> import` *

În acest caz, toate obiectele din modul vor fi importate:

```
# Importăm toate obiectele din modulul math
from math import *
x, y = 4, 2
# Apelăm funcția pow și sqrt
print(pow(x, y), sqrt(9))
```

```
16.0 3.0
```

Totuși, utilizarea acestei metode nu este recomandată din câteva motive:

- La utilizarea metodei **`import *`**, toate funcțiile, clasele și variabilele din modulul respectiv sunt importate direct în spațiul de nume al scriptului. Dacă două module conțin variabile sau funcții cu același nume, ultimul importat va suprascrie pe cel anterior, fără avertisment.

- Când este utilizată metoda **`import *`**, nu este clar de unde provine o funcție sau o variabilă. Acestea trebuie căutate în fișiere pentru a înțelege sursa, ceea ce îngreunează depanarea și înțelegerea codului.

8.2. Pachete

Dacă începeți să împărțiți codul unui proiect destul de mare în module, este posibil să doriți să grupați rapid mai multe module care au o temă similară. Sau este posibil să doriți să scoateți unele module din proiect, astfel încât să poată fi utilizate în alte proiecte. Pachetele vin în ajutor, fiind folosite pentru a grupa modulele.

Un pachet este un set de module interconectate, concepute pentru a rezolva probleme dintr-o anumită clasă dintr-un anumit domeniu. Altfel spus, un pachet este o modalitate de organizare și structurare a modulelor.

Un pachet este un folder care conține module și alte pachete, precum și fișierul necesar **`__init__.py`**, care este responsabil de inițializarea pachetului. Numele pachetului este numele folderului. (figura 8.2).

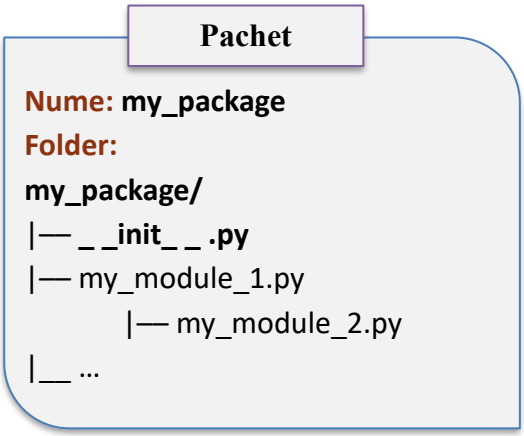

Figura. 8.2. Pachetul în Python

8.2.1. Importarea unui pachet

Pentru importarea unui pachet sunt utilizate metodele aplicate modulelor. La importarea unui pachet, putem naviga între module, folosind punctul:

Importare: **`import <pachet>`**

Apelare: **`<pachet>.<modul>.<functie>`**

Sau:

Importare: **`from <pachet> import <modul1>`**

Apelare: **`<modul1>.<functie>`**

Notă! O **bibliotecă** este o colecție de module și pachete care oferă funcționalități suplimentare în Python.

Clasificarea modulelor

Pentru o dezvoltare convenabilă și eficientă în Python, a fost creat un număr imens de module și pachete. Toate pot fi împărțite în 3 grupuri, în funcție de cine creează și unde este stocat codul unui anumit modul:

- module încorporate / biblioteci standard (build-in modules / standard libraries);
- module/biblioteci terțe (third party modules/libraries);
- module personalizate (custom/user-defined modules).

8.3. Module încorporate

Modulele încorporate sunt modulele care se instalează implicit odată cu instalarea Python.

Pentru a vizualiza lista obiectelor (claselor, funcțiilor, variabilelor) incluse în modul, după importarea acestuia apelăm funcția **`dir(nume_modul)`**.

Dacă nu știm ce face un obiect din modul, putem obține informații despre acesta cu ajutorul funcției **`help(nume_modul.obiect)`**.

```
# Importăm modulul math
import math
# Afișăm clasele, variabilele, funcțiile modulului math
dir(math)
```

```
['__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'acos',
```

```
help(math.acos)
```

```
Help on built-in function acos in module math:

acos(x, /)
    Return the arc cosine (measured in radians) of x.

    The result is between 0 and pi.
```

Modulele încorporate în Python joacă un rol important, oferind funcționalități de bază fără a necesita instalare suplimentară.

Vom analiza câteva module / bibliteci standard.

8.3.1. Modulul math

Modulul **`math`** oferă un set de funcții și constante pentru operațiunile matematice avansate, precum operațiile trigonometrice, logaritmice și de calcul al puterilor.

Reamintim că pentru a utiliza elementele modulului, acesta trebuie mai întâi importat:

```
import math
```

Constantele utile, principalele funcții și funcțiile speciale din **`math`** sunt incluse în tabelele 8.1–8.3).

Tabelul 8.1. Constante utile din modulul math

Constanta, descrierea	*Exemplu*
`math.pi` Valoarea lui π - pi	`math.pi` `3.141592653589793`
`math.e` Baza logaritmului natural	`math.e` `2.718281828459045`
`math.tau` 2 * pi	`math.tau` `6.283185307179586`
`math.inf` Valoarea infinită pozitivă — o valoare mai mare decât orice alt număr real	`math.inf` `inf` `10 < math.inf` `True`
`-math.inf` Valoarea infinită negativă — o valoare mai mică decât orice alt număr real	`-math.inf` `-inf` `10 > -math.inf` `True`
`math.nan` Not-a-Number (NaN) - o valoare numerică invalidă sau nedefinită	`math.nan` `nan`

Tabelul 8.2. Principalele funcții ale modulului math

Funcția, descrierea	*Exemplu*
`math.pow(x, y)` Ridică x la puterea y.	`math.pow(2, 3)` `8.0`
`math.sqrt(x)` Rădăcina pătrată a lui x.	`math.sqrt(25)` `5.0`
`math.exp(x)` Exponențialul lui x (e^x).	`math.exp(2)` `7.38905609893065`
`math.log(x)` Logaritm natural (bază e).	`math.log(10)` `2.302585092994046`
`math.log10(x)` Logaritm în baza 10.	`math.log10(1000)` `3.0`
`math.sin(x)` Sinusul unghiului x (în radiani).	`math.sin(math.pi / 2)` `1.0`
`math.cos(x)` Cosinusul unghiului x (în radiani).	`math.cos(0)` `1.0`
`math.tan(x)` Tangenta unghiului x (în radiani).	`math.tan(1)` `1.5574077246549023`
`math.degrees(x)` Convertește radianii în grade.	`math.degrees(math.pi)` `180.0`
`math.radians(x)` Convertește gradele în radiani.	`math.radians(180)` `3.141592653589793`

Funcția, descrierea	*Exemplu*
`math.ceil(x)` Rotunjire în sus (ceil - tavan).	`math.ceil(4.3)` `5`
`math.floor(x)` Rotunjire în jos (floor - podea).	`math.floor(4.8)` `4`
`math.fabs(x)` Modulul valorii absolute (ca valoare pozitivă).	`math.fabs(-7.25)` `7.25`
`math.factorial(x)` Factorialul unui număr întreg.	`math.factorial(5)` `120`

Tabelul 8.3. **Funcții speciale din modulul math**

Funcția, descrierea	*Exemplu*
`math.isfinite(x)` Verifică dacă un număr este finit	`math.isfinite(10)` `True`
`math.isinf(x)` Verifică dacă un număr este infinit	`math.isinf(10)` `False`
`math.isnan(x)` Verifică dacă un număr este NaN	`math.isnan(10)` `False`
`math.gcd(x, y)` Cel mai mare divizor comun	`math.gcd(48, 18)` `6`
`math.lcm(x, y)` Cel mai mic multiplu comun	`math.lcm(4, 5)` `20`

Aceste funcții și constante sunt fundamentale pentru calculele științifice, inginerie, analiză de date și multe alte domenii.

8.3.2. *Modulul random*

Modulul random este folosit pentru a genera valori aleatorii (pseudoaleatorii), foarte utile în jocuri, simulări, alegeri întâmplătoare, teste etc.

Înainte de a folosi modulul random, îl vom importa:

```
import random
```

În tabelele 8.4–8.6 sunt incluse principalele funcții din modulului **`random`**.

Tabelul 8.4. Principalele funcții de generare de numere aleatorii ale modulului `random`

Funcția, descrierea	*Exemplu*
`random.random()` Returnează un număr float aleatoriu în Intervalul [0.0, 1.0)	`random.random()` `0.033971566310019075`
`random.uniform(a, b)` intervalul [a, b] Returnează un float aleatoriu în	`random.uniform(2, 10)` `6.487119328274833`
`random.randint(a, b)`	`random.randint(2, 10)` `4`
`random.randrange([start,] stop[, step])`	`random.randrange(2, 10, 2)` `6`

Tabelul 8.5. Principalele funcții de selecție și manipulare a secvențelor modulului `random`

Funcția	*Exemplu*
`random.choice(seq)` Returnează un element aleatoriu dintr-o secvență	`a = [2, 4, 1, 6, 8]` `random.choice(a)` `4`

Funcția	*Exemplu*
`random.sample(population, k)` Returnează o listă cu k elemente unice aleatorii	`print(random.sample(range(100), 3))` `[89, 97, 59]`
`random.shuffle(x)` Amestecă elementele unei liste in-place	`a = [2, 4, 1, 6, 8]` `random.shuffle(a)` `print(a)` `[8, 2, 6, 1, 4]`

Tabelul 8.6. Principalele funcții de control al generatorului modulului `random`

Funcția, descrierea	*Exemplu*
`random.seed(a=None, version=2)` Inițializează generatorul cu o valoare specifică	`random.seed(42)` `print(random.randint(1, 100))` `random.seed(42)` `print(random.randint(1, 100))` `82` `82`
`random.getstate()` Returnează starea internă a generatorului **`random.setstate(state)`** Restabilește starea generatorului	`random.seed(123)` `print(random.random())` `state = random.getstate()` `print(random.random())` `random.setstate(state)` `print(random.random())` `0.052363598850944326` `0.08718667752263232` `0.08718667752263232`

Modulul **`random`** este important pentru orice aplicație care implică întâmplarea – jocuri, teste, simulări sau alegeri automate.

8.3.3. Modulul datetime

Modulul **`datetime`** din Python este unul dintre cele mai importante când trebuie să lucrăm cu **data și ora** și oferă clase și funcții pentru reprezentarea și orei actuale, pentru calculul diferențelor de timp, pentru conversii între formate de dată/oră și altele.

Înainte de a-l utiliza, vom importa modulul:

```
import datetime
```

Principalele constante și funcții ale modulului datetime sunt incluse în tabelele 8.7-8.8.

Tabelul 8.7. Constante utile din modulul `datetime`

Constanta, descrierea	*Exemplu*
`datetime.min` Returnează cea mai mică dată posibilă	`datetime.datetime.min` `datetime.datetime(1, 1, 1, 0, 0)`
`datetime.max` Returnează cea mai mare dată posibilă	`datetime.datetime.max` `datetime.datetime(9999, 12, 31, 23, 59, 59, 999999)`

Tabelul 8.8. Principalele funcții din `datetime`

Funcția, descrierea	*Exemplu*
Returnează data și ora actuală	`d_n = datetime.datetime.now()` `print(d_n)` `2025-05-11 15:42:16.247178`
`datetime.date.today()` Returnează data și ora curentă	`d_t = datetime.datetime.today()` `print(d_t)` `2025-05-11 15:41:48.954052`

Funcția, descrierea	*Exemplu*
`datetime.strptime()` Transformă un string într-un obiect datetime	`d = "2025-05-08"` `f = "%Y-%m-%d"` `data = datetime.datetime.strptime(d, f)` `print(data)` `2025-05-08 00:00:00`
`datetime.strftime()` Transformă un obiect datetime într-un string formatat	`new_f = "%d.%m.%Y"` `sir = data.strftime(new_f)` `print(sir)` `08.05.2025`

Modulul **`datetime`** este modulul principal pentru lucrul cu **timpul și data** în Python, oferind funcționalități precise și flexibile de măsurare a timpului, comparații, calcule și conversii alte datei și timpului.

8.3.4. Modulul time

Modulul time din Python este folosit pentru lucrul cu **timpul și temporizările**: obținerea orei curente, pauze (delay), măsurarea duratei etc. Este util în dezvoltarea de cronometre, alarme, animații, simulări și jocuri, benchmark-uri și profilare de cod.

Ca și celelalte module, vom importa modulul **`time`** înainte de utilizare:

```
import time
```

Principale constante și funcții ale modulului time sunt incluse în tabelele 8.9-8.10.

Tabelul 8.9. Constante utile din modulul time

Funcția, descrierea	*Exemplu*
`time.timezone` diferența (în secunde) între UTC și fusul orar local	`time.timezone` `0`

Funcția, descrierea	*Exemplu*
`time.daylight` 1 dacă este activ ora de vară (DST), altfel 0	`time.daylight` `0`
`time.altzone` offset-ul pentru ora de vară	`time.altzone` `0`

Tabelul 8.10. Principalele funcții din modulul `time`

Funcția, descrierea	*Exemplu*
`time.time()` Returnează timpul curent în **secunde** de la "epoca Unix" (1 ianuarie 1970)	`time.time()` `1751049131.709554`
`time.sleep()` Pauză în execuția programului pentru numărul dat de secunde	`time.sleep(10)`
`time.ctime()` Convertește un timestamp în dată și oră citibilă	`time.ctime(1746978448)` `'Sun May 11 15:47:28 2025'`
`time.localtime([secunde])` Returnează un **struct_time** (dată și oră) locală	`t = time.localtime(1746978448)` `print(t)` `time.struct_time(tm_year=2025, tm_mon=5, tm_mday=11, tm_hour=15, tm_min=47, tm_sec=28, tm_wday=6, tm_yday=131, tm_isdst=0)`
`time.gmtime()` Returnează timpul în UTC	`time.gmtime()` `time.struct_time(tm_year=2025, tm_mon=5, tm_mday=27, tm_hour=21, tm_min=0, tm_sec=0, tm_wday=1, tm_yday=147, tm_isdst=0)`

Funcția, descrierea	*Exemplu*
`time.strftime(format, t)` Formatează data/ora în șir (string) (opusul lui strptime)	`new_f = "%d.%m.%Y"` `sir = time.strftime(new_f)` `print(sir)` `11.05.2025`
`time.strptime(string, format)` Transformă un șir în obiect struct_time	`f = "%Y-%m-%d"` `d_t = "2025-05-11"` `tm = time.strptime(d_t, f)` `print(tm)` `time.struct_time(tm_year=2025, tm_mon=5, tm_mday=11, tm_hour=0, tm_min=0, tm_sec=0, tm_wday=6, tm_yday=131, tm_isdst=-1)`

Modulul **`time`** conține funcționalități utile pentru gestionarea timpului și a pauzelor în execuția programului și poate fi combinat cu alte module (**`datetime`**, **`sched`**) pentru gestionări mai avansate ale timpului.

8.4. Module/biblioteci terțe

În Python o colecție de module și pachete predefinite, care oferă funcționalități specifice, formează o **bibliotecă**.

Modulele terțe (sau third-party modules/libraries) sunt biblioteci Python dezvoltate de comunitate sau companii, care nu fac parte din distribuția standard a limbajului. Ele extind funcționalitatea Python pentru domenii specifice, cum ar fi:

- analiza datelor (pandas, numpy);
- vizualizare (matplotlib, seaborn) etc.;
- învățare automată (scikit-learn, tensorflow);
- dezvoltare web (Django, Flask).

Aceste biblioteci se instalează separat, de obicei cu ajutorul managerului de pachete **pip** care se instalează odată cu instalarea Python și sunt găzduite pe PyPI.

Instalarea bibliotecilor terțe

În § 1.2.2 s-a discutat despre managerul de pachete **pip**. Reamintim, dacă utilizați IDLE pe calculatorul dvs., pentru instalarea unei biblioteci terțe în SO Windows, vom deschide linia de comandă (cmd) din câmpul de căutare și vom scrie:

```
pip install <nume_modul>
```

După instalarea cu succes a bibliotecii, aceasta poate fi importată în program prin una din metodele descrise în §8.1.1 și ulterior utilizată.

În distribuția Anaconda, care include și Jupyter Notebook, sunt preinstalate numeroase module terțe utilizate pentru analiza de date, știința datelor și învățarea automată. Aceste biblioteci sunt instalate implicit în mediul de bază (base environment) și sunt accesibile direct în Jupyter Notebook.

Vom analiza 3 cele mai populare biblioteci: NumPy, Pandas, Matplotlib.

8.4.1. Biblioteca NumPy

NumPy (de la Numeric Python) este o bibliotecă fundamentală pentru lucrul cu tablourile multidimensionale și oferă calcule matematice, științifice, inginerești și de știința datelor în Python. Aceasta oferă suport nu doar pentru procesarea tablourilor, dar și o gamă largă de funcții matematice pentru efectuarea de operații asupra acestora. Tipul de date oferit de pachetul NymPy este **`ndarray`** (n-dimensional array).

Tabloul (array) este structura de date centrală a bibliotecii **NumPy**. Tablourile din **NumPy** diferă de listele și tuplurile obișnuite prin faptul că trebuie să conțină doar elemente de același tip. Această restricție permite o accelerare de 50-100 de ori a calculelor, precum și evitarea erorilor inutile de conversie a tipurilor și de gestionare a tipurilor.

După instalarea bibliotecii (**`pip install numpy`**) și importarea acesteia, putem afla ce clase, funcții, constante conține aceasta, folosind funcția **`dir()`**:

```
# Importăm biblioteca NumPy cu pseudonimul np
import numpy as np
# Afișăm elementele bibliotecii NumPy
print(dir(np))
```

```
['ALLOW_THREADS', 'AxisError', 'BUFSIZE', 'CLIP', 'ComplexW
arning', 'DataSource', 'ERR_CALL', 'ERR_DEFAULT', 'ERR_IGNO
RE', 'ERR_LOG', 'ERR_PRINT', 'ERR_RAISE', 'ERR_WARN', 'FLOA
TING_POINT_SUPPORT', 'FPE_DIVIDEBYZERO', 'FPE_INVALID', 'FP
E_OVERFLOW', 'FPE_UNDERFLOW', 'False_', 'Inf', 'Infinity',
'MAXDIMS', 'MAY_SHARE_BOUNDS', 'MAY_SHARE_EXACT', 'ModuleDe
precationWarning', 'NAN', 'NINF', 'NZERO', 'NaN', 'PINF',
```

Crearea tablourilor

Pentru crearea unui tablou utilizăm atributul **`array.`**

Un **tablou unidimensional** sau un **vector** (1D array) în NumPy are structura unei liste de elemente. Exemplu de creare a două tablouri unidimensionale este dat mai jos:

```
arr = np.array([1, 2, 3, 4])
print(arr)
arr1 = np.array([26])
print(arr1)
```

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

Un **tablou bidimensional** sau o **matrice** (2D array) în **NumPy** are structura:

[nr_de_rânduri, nr_de_coloane]

- **Rânduri** → liniile din fiecare matrice.
- **Coloane** → elementele pe orizontală din fiecare rând.

Prin urmare, pentru crearea unui tablou bidimensional, vom arăta ca parametru o **listă de liste**, unde fiecare listă interioară reprezintă un rând al matricei:

```
# Creăm o matrice 2D 2x4
arr2 = np.array([[1, 2, 3, 4], [5, 6, 7, 8]] )
print(arr2)
```

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

```
# Creăm o matrice 2D 3x4
c = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]] )
print(c)
```

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

Un **tablou tridimensional** (3D array) în NumPy are structura: [nr_de_blocuri, nr_de_rânduri, nr_de_coloane]

- **Blocuri** (sau „planuri") → colecții de matrice 2D stivuite una peste alta.
- **Rânduri** → liniile din fiecare matrice.
- **Coloane** → elementele pe orizontală din fiecare rând.

Vom **crea un tablou tridimensional:**

```
# Creăm o matrice 3D 2x2x3
d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])
print(d)
```

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

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

Notă! O matrice 4D reprezintă 2 matrice 3D; o matrice 5D reprezintă 2 matrice 4D etc.

Python oferă mai multe metode rapide de creare a tablourilor.

La crearea unui tablou putem utiliza funcția **`range()`** pentru a genera o secvență de elemente sau **`random.rand()`** pentru a genera valori aleatorii dintr-un interval specificat:

```
c = np.array(range(0, 10, 2))
print(c)
```

```
[0 2 4 6 8]
```

```
d = np.random.rand(2, 3)
print(d)
```

```
[[0.71329581 0.07760149 0.8145895 ]
 [0.46663645 0.89280661 0.25035714]]
```

Biblioteca NumPy conține un șir de atribute, care oferă posibilitatea de generare a tablourilor cu valori implicite:

- **`zeros()`** – creează un tablou completat cu valori de 0;
- **`ones()`** – completează un tablou creat cu valori de 1;
- **`full()`** – permite specificarea unei valori concrete.

Câteva exemple de creare a tablourilor cu valori implicite sunt expuse mai jos:

```
# Creăm un vector din zerouri
print(np.zeros(3))
```

```
[0. 0. 0.]
```

```
#Creăm o matrice din valori de 1
print(np.ones((3, 4)))
```

```
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
```

```
#Creăm un tablou 3D din valori de 4
print(np.full((2, 2, 3), 4))
```

```
[[[4 4 4]
  [4 4 4]]

 [[4 4 4]
  [4 4 4]]]
```

Aflarea informației despre tablouri

Atributul **`ndim`** returnează numărul de dimensiuni ale tabloului:

```
a = np.array([42])
b = np.array([1, 2, 3, 4, 5])
c = np.array([[1, 2, 3], [4, 5, 6]])
d = np.array([[[1, 2, 3], [4, 5, 6]],
              [[7, 8, 9], [10, 11, 12]]])
print('Dimensiunea tabloului "a" este', a.ndim)
print('Dimensiunea tabloului "b" este', b.ndim)
print('Dimensiunea tabloului "c" este', c.ndim)
print('Dimensiunea tabloului "d" este', d.ndim)
```

```
Dimensiunea tabloului "a" este 1
Dimensiunea tabloului "b" este 1
Dimensiunea tabloului "c" este 2
Dimensiunea tabloului "d" este 3
```

Atributul **`size`** returnează numărul de elemente ale tabloului:

```
print('Numărul de elemente în "a" este', a.size)
print('Numărul de elemente în "b" este', b.size)
print('Numărul de elemente în "c" este', c.size)
print('Numărul de elemente în "d" este', d.size)
```

```
Numărul de elemente în "a" este 1
Numărul de elemente în "b" este 5
Numărul de elemente în "c" este 6
Numărul de elemente în "d" este 12
```

Atributul **`shape`** returnează dimensiunile tabloului:

```
print('Dimensiunea lui "a" este', a.shape)
print('Dimensiunea lui "b" este', b.shape)
print('Dimensiunea lui "c" este', c.shape)
print('Dimensiunea lui "d" este', d.shape)
```

```
Dimensiunea lui "a" este (1,)
Dimensiunea lui "b" este (5,)
Dimensiunea lui "c" este (2, 3)
Dimensiunea lui "d" este (2, 2, 3)
```

Putem afla câți octeți ocupă un element în memorie, folosind atributul **`itemsize`** sau întregul tablou, utilizând atributul **`nbytes`**:

```
print(a.itemsize, c.itemsize)
print(a.nbytes, b.nbytes)
print(c.nbytes, d.nbytes)
```

```
8 8
8 40
48 96
```

Accesarea elementelor sau a unui interval din tablou

Accesarea elementelor se face prin indicarea pozițiilor (indecșilor): [bloc, rând, coloană] pentru tablouri 3D; [rând, coloană] pentru matrice; [coloana] pentru vector. Ca și în cazul secvențelor, indexarea începe cu 0.

În exemplul de mai jos avem o matrice. Prin urmare, vom indica indecșii rândului și ai coloanei elementului:

```
arr1 = np.array([[1, 2, 3, 4, 5, 6],
                 [7, 8, 9, 10, 11, 12]])
print(arr1[0, 2])
print(arr1[1, 1])
```

```
3
8
```

În cazul unui tablou 3D, indicăm indecșii blocului, ai rândului și ai coloanei:

```
arr2 = np.array([[[1, 2, 3], [4, 5, 6]],
                 [[7, 8, 9], [10, 11, 12]]])
print(arr2[0, 1, 2])
print(arr2[1, 1, 2])
```

```
6
12
```

Pentru a accesa un interval, vom folosi începutul și sfârșitul intervalului, similar cum o făceam în șiruri de caractere, liste sau tupluri:

```
arr3 = np.array([[1, 2, 3, 4, 5],
                 [6, 7, 8, 9, 10]])
print(arr3[1, 1:4])
print(arr3[0:2, 2])
print(arr3[0:2, 1:4])
```

```
[7 8 9]
[3 8]
[[2 3 4]
 [7 8 9]]
```

Vectorizarea

Vectorizarea este un concept important care face codul Python mult mai rapid și eficient. Operațiile asupra tablourilor se fac vectorial, adică sunt aplicate operații pe întregul tablou, ceea ce duce la un cod mai concis și o performanță mai mare.

De exemplu, putem executa operații aritmetice cu valorile din tablouri, indicând doar numele tabloului și operația:

```
arr1 = np.array([1, 2, 3, 4])
arr2 = np.array([5, 6, 7, 8])
print(arr1 + 10)
print(arr1 + arr2)
print(arr1 * 2)
print(arr2 ** 2)
```

```
[11 12 13 14]
[ 6  8 10 12]
[2 4 6 8]
[25 36 49 64]
```

Funcții matematice și statistice

NumPy include un șir de funcții statistice utile: calcularea valorii medii (**`mean()`**), abaterii standard (**`std()`**), a valorilor maximale (**`max()`**) și minimale (**`min()`**) și altele.

```
arr = np.array([1, 2, 3, 4, 5])
print(arr.mean())
print(arr.std())
```

```
3.0
1.4142135623730951
```

Modulul NumPy este compatibil cu biblioteci ca Pandas, Matplotlib, SciPy, TensorFlow, ceea ce îl face indispensabil în proiectele complexe

În concluzie, NumPy este baza calculelor numerice în Python, oferind performanță, claritate și flexibilitate pentru prelucrarea numerică la scară largă.

Mai multe informații despre biblioteca NumPy pot fi găsite pe https://numpy.org.

8.4.2. Biblioteca Pandas

Biblioteca Pandas este o bibliotecă foarte populară în Python, utilizată pentru manipularea și analiza datelor tabelare.

După instalarea bibliotecii (**`pip install pandas`**), importăm biblioteca și putem afla ce clase, funcții, constante conține aceasta, utilizând funcția **`dir()`**:

```
# Importăm biblioteca Pandas cu pseudonimul pd
import pandas as pd
# Afișăm elementele bibliotecii Pandas
print(dir(pd))
```

```
['ArrowDtype', 'BooleanDtype', 'Categorical', 'CategoricalDtyp
dex', 'DataFrame', 'DateOffset', 'DatetimeIndex', 'DatetimeTZD
e', 'ExcelWriter', 'Flags', 'Float32Dtype', 'Float64Dtype', 'G
e', 'Index', 'IndexSlice', 'Int16Dtype', 'Int32Dtype', 'Int64D
e', 'Interval', 'IntervalDtype', 'IntervalIndex', 'MultiIndex'
```

Principalele structuri de date

1. Seriile de date (Series)

Seriile de date sunt structuri de date unidimensionale, asemănătoare cu o listă sau un vector, dar cu etichete (index) specificat pentru fiecare element. Pentru crearea unei serii de date este utilizată funcția **`Series()`**:

```
serie = pd.Series([11, 12, 13, 14],
                  index = ['a','b','c','d'])
print(serie)
```

```
a    11
b    12
c    13
d    14
dtype: int64
```

Acum putem accesa un element al seriei folosind etichetarea:

```
print(serie['c'])
```

```
13
```

2. DataFrame

DataFrame este o structură de date bidimensională (formată din rânduri și coloane), similară cu un tabel Excel. DataFrame-urile pot fi create din dicționare de listă, din liste de dicționare sau citite din fișierele CSV, EXCEL, SQL etc. Să creăm un DataFrame dintr-un dicționar de liste:

```
data = {'Nume': ['Ion', 'Maria', 'Vasile'],
        'Virsta': [30, 26, 28],
        'Oras': ['Chisinau', 'Balti', 'Comrat']}
df = pd.DataFrame(data)
print(df)
```

```
     Nume  Virsta      Oras
0     Ion      30  Chisinau
1   Maria      26     Balti
2  Vasile      28    Comrat
```

Citirea și scrierea fișierelor

Biblioteca Pandas suprotă multiple formate de fișiere. Modurile de citire și scriere pentru diferite tipuri de fișiere sunt incluse în tabelele 8.11-8.12.

Tabelul 8.11. Citirea fișierelor în Pandas

Tip fișier	*Citire*
`CSV`	`pd.read_csv('fisier.csv')`
`EXCEL`	`pd.read_excel('fisier.xlsx')`
`SQL`	`pd.read_sql(query, conexiune)`
`JSON`	`pd.read_json('fisier.json')`

Tabelul 8.12. Scrierea fișierelor în Pandas

Tip fișier	*Scriere*
`CSV`	`df.to_csv('fisier.csv', index=False)`
`EXCEL`	`df.to_excel('fisier.xlsx', index=False)`
`SQL`	`df.to_sql('tabel', conexiune)`
`JSON`	`df.to_json('fisier.json')`

Vom examina un exemplu de lucru cu un fișier EXCEL (citirea și scrierea datelor). Presupunem că avem un fișier cu numele sales_data.xlsx compus din 7 rânduri (primul rând conține titlurile coloanelor) și 4 coloane (figura 8.3).

	A	B	C	D	E
1	**Region**	**Product**	**Number**	**Price_per_unit**	
2	North	Coffee	10	100	
3	South	Tea	20	80	
4	West	Juice	15	90	
5	East	Water	25	50	
6	Centre	Cola	18	60	
7	North	Coffee	12	100	

Figura 8.3. Conținutul fișierului sales_data.xlsx

Pentru citirea fișierului folosim funcția **`read.excel()`**:

```
# Citirea fișierului 'sales_data.xlsx'
df = pd.read_excel('sales_data.xlsx')
print(df)
```

```
   Region Product  Number  Price_per_unit
0   North  Coffee      10             100
1   South     Tea      20              80
2    West   Juice      15              90
3    East   Water      25              50
4  Centre    Cola      18              60
5   North  Coffee      12             100
```

Pentru a adăuga încă un rând, creăm mai întâi rândul nou în formă de dicționar și adăugăm rândul nou prin concatenare. În final, scriem și salvăm datele în fișier:

```
# Creăm un dicționar cu date pentru un rând nou
new_row = {'Region': 'Centre', 'Product':'Milk',
           'Number':30, 'Price_per_unit':70}
# Convertim în DataFrame
new_row_df = pd.DataFrame([new_row])
# Adăugăm rândul în tabel
df = pd.concat([df, new_row_df])
# Salvăm în fișier fără index
df.to_excel('sales_data.xlsx', index = False)
# Afișăm rezultatul
print(df)
```

```
   Region Product  Number  Price_per_unit
0   North  Coffee      10             100
1   South     Tea      20              80
2    West   Juice      15              90
3    East   Water      25              50
4  Centre    Cola      18              60
5   North  Coffee      12             100
0  Centre    Milk      30              70
```

Operațiile de bază cu DateFrame-uri

Biblioteca Pandas oferă un șir de operații pentru manipularea cu DataFrame-uri. Exemplele de mai jos sunt bazate pe DataFrame citit din fișierul „sales_data.xlsx”.

Afișăm primele 5 rânduri din DataFrame, folosind metoda **`head()`**:

```
print(df.head())
```

```
   Region Product  Number  Price_per_unit
0   North  Coffee      10             100
1   South     Tea      20              80
2    West   Juice      15              90
3    East   Water      25              50
4  Centre    Cola      18              60
```

Pentru a afla informații despre numărul de rânduri, coloane, tipurile datelor din DataFrame aplicăm metoda **`info()`**:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6 entries, 0 to 5
Data columns (total 4 columns):
 #   Column          Non-Null Count  Dtype 
---  ------          --------------  ----- 
 0   Region          6 non-null      object
 1   Product         6 non-null      object
 2   Number          6 non-null      int64 
 3   Price_per_unit  6 non-null      int64 
dtypes: int64(2), object(2)
memory usage: 324.0+ bytes
```

Putem afișa datele dintr-o anumită coloană din DataFrame, precum și tipul datelor din coloană, indicând titlul coloanei ca index, similar cum accesăm valorile dintr-un dicționar indicând cheia.

Afișăm titlurile coloanelor din DataFrame, utilizând atributul **`columns`**:

```
print(df['Product'])
```

```
0    Coffee
1       Tea
2     Juice
3     Water
4      Cola
5    Coffee
Name: Product, dtype: object
```

```
print(df.columns)
```

```
Index(['Region', 'Product', 'Number',
'Price_per_unit'], dtype='object')
```

Pentru adăugarea unei coloane noi în DataFrame, vom indica numele coloanei ca index și datele adăugate ca listă:

```
df['Sold'] = [3, 5, 2, 4, 9, 7, 4]
print(df.head())
```

```
   Region Product  Number  Price_per_unit  Sold
0   North  Coffee      10             100     3
1   South     Tea      20              80     5
2    West   Juice      15              90     2
3    East   Water      25              50     4
4  Centre    Cola      18              60     9
```

Pentru accesarea datelor din DataFrame sunt folosite metodele **`loc[]`** și **`iloc[]`**, dar acestea funcționează diferit. Metoda **`iloc[]`** accesează după etichetă, iar **`iloc[]`** – după poziție:

```
print(df.loc[1, 'Product'])
```

```
Tea
```

```
print(df.iloc[0, 0])
```

```
North
```

```
print(df.iloc[0])
```

```
Region              North
Product            Coffee
Number                 10
Price_per_unit        100
Sold                    3
Name: 0, dtype: object
```

Modulul **Pandas** oferă posibilități de filtrare a datelor din anumite coloane ale DataFrame-ului, folosind operatori relaționali:

```
df[df['Number'] > 20]
```

	Region	Product	Number	Price_per_unit	Sold
3	East	Water	25	50	4
0	Centre	Milk	30	70	4

Pentru returnarea valorilor unice dintr-o coloană concretă, utilizăm funcția **`unique()`**:

```
df['Region'].unique()
```

```
array(['North', 'South', 'West', 'East', 'Centre'],
dtype=object)
```

Dacă avem nevoie să ștergem un anumit rând sau coloană din DataFrame, putem folosi funcția **`drop()`** în care vom indica eticheta,

axa (0 – rând, 1 – coloană) și tipul de modificare – inplace (False – returnează un nou DataFrame, True – modifică DataFrame-ul direct):

```
df.drop('Sold', axis=1, inplace=True)
print(df)
```

```
     Region Product  Number  Price_per_unit
0     North  Coffee      10             100
1     South     Tea      20              80
2      West   Juice      15              90
3      East   Water      25              50
```

Statistici și agregare

Biblioteca Pandas permite afișarea rapidă a statisticilor și agregărilor. Pentru afișarea statisticilor de bază (doar pentru coloane numerice), este utilizată funcția **`describe()`**:

Pentru a afla media aritmetică, mediana, abaterea standard pentru valorile dintr-o coloană specifică, vom indica coloana și vom utiliza funcțiile **`mean()`**, **`median()`**, **`std()`**.

```
print(df['Number'].mean())
print(df['Number'].median())
print(df['Number'].std())
```

```
18.571428571428573
18.0
7.114706432386907
```

Biblioteca Pandas este foarte flexibilă și se integrează foarte bine cu multe alte biblioteci Python pentru analiza datelor, vizualizare și machine learning: NumPy, Matplotlib, Seaborn, SQAlchemy și altele.

În concluzie, Pandas facilitează încărcarea, curățarea, transformarea, filtrarea și analiza datelor tabelare, ceea ce face din Python un instrument puternic pentru lucrul cu datele.

Mai multe informații despre biblioteca Pandas pot fi găsite pe https://pandas.pydata.org/.

8.4.3. Biblioteca Matplotlib

Matplotlib este una dintre cele mai populare biblioteci utilizate pentru vizualizarea datelor în Python. Aceasta permite generarea de grafice statice, interactive și animate, fiind ideală pentru analiza datelor și reprezentarea acestora într-un mod clar și biblioteci informativ.

Dacă lucrați în IDLE, nu uitați să instalați biblioteca din linia de comandă (**`pip install matplotlib`**). După aceasta o importăm în mediul de programare. De obicei, se importă submodulul **PyPplot** din **Matplotlib**, care oferă funcții pentru crearea și personalizarea graficelor. La nevoie, afișăm lista elementelor din submodul:

```
# Importăm submodulul pyplot din matplotlib
import matplotlib.pyplot as plt
# Afișăm elementele submodulului
print(dir(plt))
```

```
['Annotation', 'Arrow', 'Artist', 'AutoLocator', 'Axes', 'B
utton', 'Circle', 'Enum', 'ExitStack', 'Figure', 'FigureBas
e', 'FigureCanvasBase', 'FigureManagerBase', 'FixedFormatte
r', 'FixedLocator', 'FormatStrFormatter', 'Formatter', 'Fun
cFormatter', 'GridSpec', 'IndexLocator', 'Line2D', 'LinearL
ocator', 'Locator', 'LogFormatter', 'LogFormatterExponent',
```

Grafice suportate de Matplotlib

Matplotlib oferă o gamă largă de funcții pentru generarea diferitor tipuri de grafice (tabelul 8.12).

Tabelul 8.13. Tipuri de grafice suportate de Matplotlib

Funcție, tip grafic	*Descriere*
`plt.plot()` Linie (Line Plot)	Grafice de evoluție în timp sau tendințe
`plt.bar()` Bare (Bar Plot)	Comparații între categorii
`plt.hist()` Histograme (Histogram)	Distribuția frecvenței datelor
`plt.scatter()` Puncte (Scatter Plot)	Relația între două variabile

Funcție, tip grafic	*Descriere*
`plt.pie()` Plăcintă (Pie Chart)	Reprezentarea proporțională a părților
`plt.boxplot()` Cutie și mustăți (Box Plot)	Analiza distribuției și detectarea valorilor anormale
`plt.imshow()` Hartă termică (Heatmap)	Vizualizare în formă de hartă termică

Line plot

```
# Date de intrare pentru grafic
x = [1, 2, 3, 4, 5]
y = [10, 14, 12, 15, 20]
# Crearea graficului
plt.plot(x,y, marker = 'o', linestyle = '-', color = 'blue',
         label ='Vânzări', linewidth = 2.0)
plt.title("Evoluția vânzărilor", fontsize = 14, color = 'red')
plt.xlabel("Luna")
plt.ylabel('Vânzări (mii)')
plt.legend()
plt.grid(True, linestyle = ":")
plt.show()
```

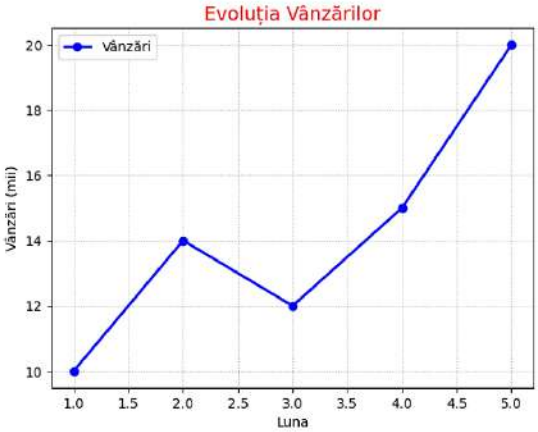

Figura 8.4. Grafic cu linie desenat de biblioteca Matplotlib

Vom explica codul de mai sus:

- **`plt.plot()`** este funcția principală care desenează linia pe grafic cu următoarele argumente:
  - **`x, y`** preiau valorile din listele x și y (perechi) pentru axele corespunzătoare;
  - **`marker`** este marker-ul pentru fiecare punct din grafic: **`'o'`** reprezintă un cerc, **`'s'`** – pătrat, **`'^'`** – triunghi, **`'+'`** – plus, **`'x'`** - ×;
  - **`linestyle`** descrie stilul liniei: **`'-'`** este linie continuă, **`'--'`** este linie întreruptă, **`':'`** este linie punctată;
  - **`color`** setează culoarea liniei care poate fi nume de culoare (**`'blue'`**, **`'red'`**) sau hex (**`'#0000FF'`**);
  - **`linewidth`** specifică grosimea liniei;
  - **`label`** este eticheta pentru linia de legendă care apare doar dacă se apelează **`plt.legend().`**
- **`plt.title()`** adaugă titlul la grafic. Putem ajusta stilul folosind argumente suplimentare:
  - **`fontsize`** setează mărimea textului;
  - **`color`** setează culoarea titlului care poate fi nume de culoare sau hex.
- **`plt.xlabel()`** / **`plt.ylabel()`** afișează eticheta pentru axa X / Y. Putem ajusta stilul folosind argumentele:
  - **`fontsize`** setează mărimea textului;
  - **`color`** setează culoarea titlului care poate fi nume de culoare sau hex.
- **`plt.legend()`** activează legenda pe grafic. Dacă a fost specificat un label la **`plt.plot()`**, acesta va apărea în legendă. Poate avea un argument care specifică poziția legendei:
  - **`loc`** poate fi egal cu **`'upper left'`**, **`'upper right'`**, **`'lower left'`**, etc.
- **`plt.grid(True/False)`** activează grila pe fundalul graficului care ajută la o mai bună vizualizare a valorilor pe axele X și Y. Grila poate fi personalizată prin argumente:
  - **`linestyle`** descrie stilul liniei: '-', '--' sau ':'

- **`color`** setează culoarea liniei care poate fi nume de culoare sau hex;
- **`linewidth`** specifică grosimea liniei.

- **`plt.show()`** activează efectiv graficul. Fără acest apel graficul nu va fi vizibil. Dacă descriem mai multe grafice într-un singur script, **`plt.show()`** le va afișa pe toate odată.

Bar Plot

```
products = ['Cafea', 'Ceai', 'Suc', 'Apă']
sales = [100, 150, 90, 130]

plt.bar(products, sales, color='orange')
plt.title('Vânzări pe Produse')
plt.xlabel('Produs')
plt.ylabel('Număr Vândut')
plt.show()
```

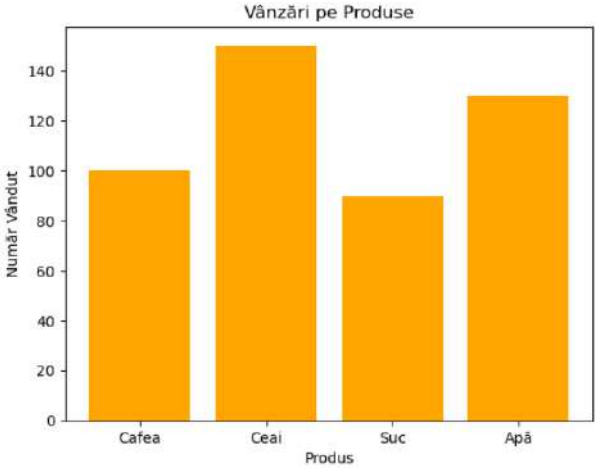

Figura 8.5. Grafic cu bare desenat de biblioteca Matplotlib

- **`plt.bar()`** este funcția care desenează graficul cu bare și poate avea următoarele argumente:
  - **`x`** reprezintă valorile de pe axa x (categorii, etichete etc.);

- **`height`** reprezintă valorile de pe axa y (înălțimea fiecărei bare);
- **`color`** stabilește culoarea barelor care poate fi nume de culoare sau hex;
- **`width`** setează lățimea fiecărei bare (implicit este 0.8);
- **`edgecolor`** stabilește culoarea conturului fiecărei bare;
- **`label`** reprezintă eticheta pentru legendă (dacă se folosește plt.legend());
- **`align`** poziționează barele ('center' sau 'edge').

Putem adăuga funcția **`plt.grid`**() sau argumente suplimentare pentru funcțiile **`plt.title()`**, **`plt.xlabel()`**, **`plt.ylabel()`**, descrise la **Line Plot**.

Histograma

```
# Importăm numpy cu pseudonimul np
import numpy as np
# Generăm date random
data = np.random.randn(1000)

plt.hist(data, bins=20, color='purple', edgecolor='black')
plt.title('Distribuție aleatorie')
plt.show()
```

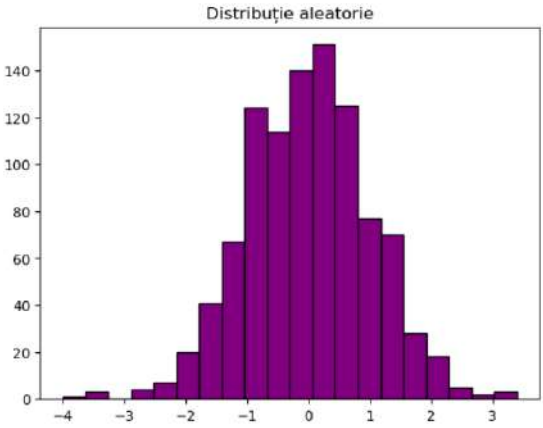

Figura 8.6. Grafic de tip Histogramă desenat de biblioteca Matplotlib

- **`plt.hist()`** este funcția de bază de generare a graficului de tip histogramă care reprezintă distribuția frecvenței datelor și poate avea următoarele argumente:
  - **`data`** reprezintă lista sau array-ul cu valorile de analizat;
  - **`bins`** este numărul de coloane (bin-uri) în histogramă (implicit este 10);
  - **`color`** setează culoarea fiecărui bin;
  - **`edgecolor`** stabilește culoarea marginilor bin-urilor;
  - **`alpha`** setează transparența bin-urilor (valoare între 0 și 1);
  - **`density`** stabilește densitatea. Dacă este setat la True, normalizează datele (arată probabilități în loc de frecvențe).

La dorință, adăugăm funcția **`plt.grid`**() sau argumente suplimentare pentru funcțiile **`plt.title()`**, **`plt.xlabel()`**, **`plt.ylabel()`**, descrise la **Line Plot**.

Scatter Plot

```
x = [5, 10, 15, 20, 25]
y = [10, 20, 15, 25, 30]
plt.scatter(x, y)
plt.show()
```

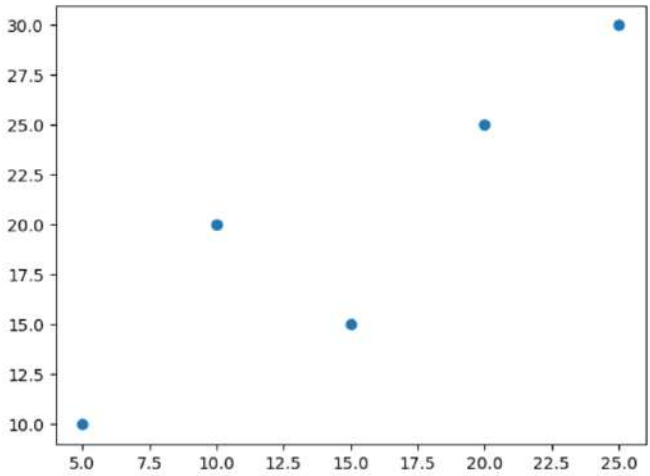

Figura 8.7. Grafic Scatter Plot desenat de biblioteca Matplotlib

- **`plt.scatter()`** este funcția care generează graficul de puncte ce reprezintă relația dintre două seturi de date și poate avea următoarele argumente:
  - **`x`** definește valorile pe axa X;
  - **`y`** definește valorile pe axa Y;
  - **`color`** setează culoarea punctelor;
  - **`marker`** reprezintă simbolul pentru fiecare punct ('o', '^', 's', '+', 'x', etc.);
  - **`s`** setează dimensiunea punctelor;
  - **`alpha`** determină transparența punctelor (valoare între 0 și 1);
  - **`label`** este eticheta pentru legendă (dacă se folosește plt.legend()).

La necesitate, adăugăm funcția **`plt.grid`**() sau argumente suplimentare pentru funcțiile **`plt.title()`**, **`plt.xlabel()`**, **`plt.ylabel()`**, descrise la **Line Plot**.

Pie chart

```
y = [35, 25, 25, 15]
pie_labels=['Fruits', 'Bananas', 'Cherries', 'Dates']
plt.pie(y, labels = pie_labels)
plt.show()
```

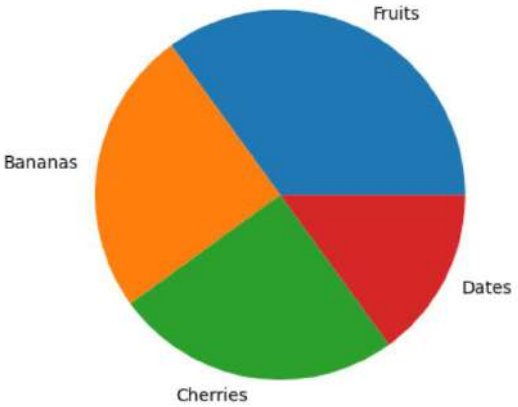

Figura 8.8. Grafic de tip Pie chart desenat de biblioteca Matplotlib

`plt.pie()` este funcția care generează graficul de tip plăcintă, reprezintă proporțiile valorilor într-un cerc și poate avea următoarele argumente:

- **`sizes`** sunt valorile pentru fiecare felie din plăcintă**;**
- **`labels`** reprezintă etichetele pentru fiecare categorie (feliile din plăcintă);
- **`colors`** setează culorile pentru fiecare felie (['red', 'blue', 'green']);
- **`autopct`** stabilește formatul procentajului afișat ('%1.1f%%' pentru un singur decimal);
- **`startangle`** indică unghiul de start pentru prima felie (de exemplu, 90 pentru ora 12);
- **`shadow`** afișează o umbră în spatele plăcintei dacă este True;
- **`explode`** adaugă separația dintre feliile plăcintei (de exemplu, (0, 0.1, 0, 0)).

La necesitate, adăugăm argumente suplimentare pentru funcțiile **`plt.title()`**, **`plt.labels()`.**

Subgrafice

Biblioteca **Matplotlib** oferă posibilitatea de creare a mai multor grafice într-o singură figură. În acest caz, pentru fiecare grafic vom apela funcția **`subplot(nr_rând, nr_coloana, index):`**

```
import matplotlib.pyplot as plt
import numpy as np
x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])
plt.subplot(1, 2, 1)
plt.plot(x, y)
ax_x, ax_y = ['A', 'B', 'C', 'D'], [5, 3, 7, 4]
plt.subplot(1, 2, 2)
plt.bar(ax_x, ax_y, color = 'green')
plt.show()
```

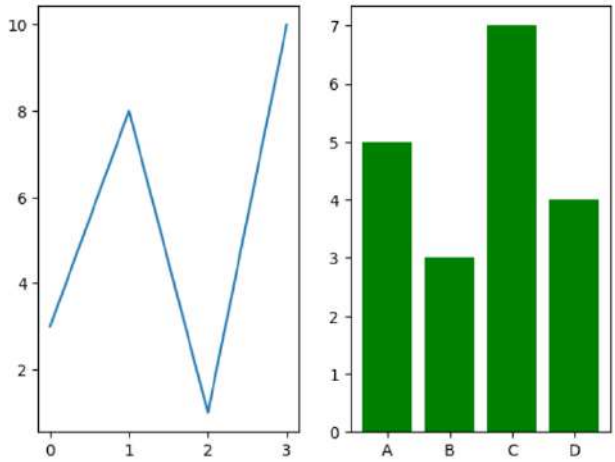

Figura 8.9. Crearea subgraficelor

Biblioteca Matplotlib se integrează perfect cu Pandas pentru a reprezenta direct coloane dintr-un DataFrame și NumPy pentru a genera date de intrare.

În concluzie, Matplotlib este o bibliotecă foarte puternică pentru vizualizarea datelor. Este flexibilă, personalizată și se integrează perfect cu alte biblioteci precum Pandas, NumPu și altele.

Mai multe informații despre biblioteca Matplotlib pot fi găsite pe https://matplotlib.org/.

8.5. Module personalizate

Pentru a crea propriul modul în Python, salvați codul într-un fișier cu extensia .py. Acum este disponibil în orice alt fișier.
De exemplu, creăm fișierul **`module_1.py`** și scriem în el funcția **`hello()`** care afișează mesajul „Salut de la modulul 1”.

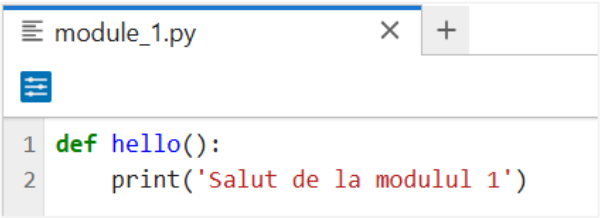

Figura 8.10. Crearea unui modul

Acum putem folosi modulul pe care tocmai l-am creat. Importăm **`module_1`** și apelăm funcția **`hello()`**. După executarea codului obținem mesajul din module_1.

```
from module_1 import hello

hello()
```

```
Salut de la modulul 1
```

Notă! Modulul creat (module_1.py) și fișierul în care importăm modulul și apelăm funcția **`hello()`** trebuie să se afle în același folder.

În concluzie, modulele și pachetele Python oferă instrumente puternice pentru organizarea codului, îmbunătățirea structurii, utilizabilității și scalabilității acestuia. Înțelegerea și utilizarea corectă a acestor instrumente este importantă pentru dezvoltarea de software eficient și ușor de întreținut.

Sarcini

1. **Utilizarea modulelor încorporate**. Importați modulul **`random`** pentru a crea un miniprogram care:
   - generează o listă de 10 numere aleatorii între 1 și 100;
   - sortează lista în ordine descrescătoare;
   - selectează aleatoriu un număr din lista creată și sortată.

2. **Importarea**. Luați drept bază un program scris de dvs. care conține o singură funcție. Salvați această funcție într-un fișier separat. Apelați funcția din fișierul salvat în fișierul cu programul principal, utilizând metodele de import:

```
import <nume_modul>
import <nume_modul> as <alias>
from <nume_modul> import <nume_funcție>
from <nume_modul> import <nume_funcție> as <alias>
from <nume_modul> import *
```

3. **Crearea unui modul propriu**. Creați un fișier Python numit **`matematica.py`** care să conțină următoarele funcții:
   - adunare(a, b);
   - scadere(a, b);
   - inmultire(a, b);
   - impartire(a, b).

   Creați apoi un alt fișier main.py în care:
   - importați modulul matematica;
   - apelați fiecare funcție cu date alese de tine;
   - afișați rezultatele

9. PROGRAMAREA ORIENTATĂ PE OBIECTE

Obiective

La finalul acestui capitol, studenții vor fi capabili:

- Să explice conceptele fundamentale ale programării orientate pe obiecte (clasă, obiect, atribut, metodă, instanță).
- Să dezvolte și să utilizeze clase personalizate în Python, incluzând metode de inițializare și atribute.
- Să aplice principiile POO precum: moștenirea, încapsularea, polimorfismul și abstractizarea în cadrul programelor proprii.
- Să protejeze datele interne ale obiectelor prin utilizarea atributelor private și controlul accesului la acestea.
- Să modelizeze concepte din lumea reală în cod, dezvoltând aplicații mai clare, mai modulare și mai ușor de întreținut.

Programarea orientată pe obiecte (POO) este considerată una dintre cele mai eficiente metodologii pentru crearea de produse software. În POO sunt create clase care descriu obiecte și situații din lumea reală, apoi sunt create obiecte pe baza acestor clase. Când este creată o clasă, este definit un comportament comun pentru o întreagă categorie de obiecte.

Când sunt create obiecte specifice pe baza acestor clase, fiecare obiect este înzestrat automat cu un comportament comun; apoi fiecare obiect poate fi înzestrat cu caracteristici unice la alegere. Este interesant cât de bine POO modelează situațiile din lumea reală.

POO oferă instrumente puternice pentru gestionarea complexității programelor mari. Python acceptă POO și oferă toate instrumentele necesare pentru lucrul cu clase și obiecte. În acest capitol vom analiza în detaliu modul în care Python implementează conceptele POO, inclusiv crearea de clase, obiecte, moștenirea, încapsularea și polimorfismul.

POO în Python este construit, ținând cont de următoarele principii:

1. Toate datele din el sunt reprezentate de obiecte.
2. Un program poate fi compus ca un set de obiecte care interacționează și care trimit mesaje între ele.
3. Fiecare obiect are propria parte de memorie și poate fi format din alte obiecte.
4. Fiecare obiect are un tip.
5. Toate obiectele de același tip pot primi aceleași mesaje (și pot efectua aceleași acțiuni).

Un program scris cu ajutorul instrumentelor POO reprezintă un set de obiecte care interacționează.

9.1. Clase, obiecte, atribute, metode

9.1.1. Definirea și utilizarea unei clase

Clasele permit modelarea aproape a orice. O clasă este un tip special de date care definește modul de construire a unui anumit tip de obiect, comportamentul (metode) și starea acestuia (atribute). Clasa stochează, de asemenea, unele elemente de date care sunt partajate de toate instanțele acestei clase. O clasă servește drept șablon pentru crearea de obiecte (instanțe ale clasei).

Vom începe prin a scrie o clasă simplă de tip Cat care reprezintă o pisică. Nu o pisică anume, ci o pisică în general. Ce știm despre pisici? Au un nume, au o vârstă. De asemenea, știm că pisicile prind șoareci, se joacă, zgârie. Aceste două informații (nume și vârsta) și trei comportamente (prinde șoareci, se joacă, zgârie) vor fi incluse în clasa **`Cat`**, deoarece sunt comune majorității pisicilor. Clasa îi spune lui Python cum să creeze un obiect care reprezintă o pisică. Când clasa este scrisă, o folosim pentru a crea instanțe, fiecare reprezentând o singură pisică.

Crearea clasei

Un exemplu de clasă este prezentat mai jos:

```
class Cat(): # Definim clasa Cat
    # Model simplu al unei pisici care are un nume și o vârstă
    def __init__(self, cat_name, cat_age):
        # Inițializăm atributele name și age
        self.name = cat_name
        self.age = cat_age

    def get_age(self): # Metoda Pisica are 'age' ani
        return f"{self.name} is {self.age} years old"
    def catch_mice(self):   # Metoda Pisica prinde șoareci
        return f"{self.name} is catching mice"
    def play(self):  # Metoda Pisica se joacă
        return f"{self.name} is playing"
    def scratch(self):  # Metoda Pisica zgârie
        return f"{self.name} is scratching"
```

Vom analiza codul de mai sus. În Python, o clasă este definită folosind cuvântul-cheie **`class`**. După cuvântul-cheie **`class`**, urmează numele clasei **`Cat`**. Prin convenție, numele care încep cu majusculă în Python denotă clase. Parantezele din definiția clasei sunt goale, deoarece clasa este creată de la zero. După închiderea parantezelor punem două puncte (:).

Funcțiile definite în cadrul clasei se numesc **metode**. Tot ce am învățat despre funcții (Cap. 6) se aplică și metodelor. Singura diferență practică constă în modul în care sunt apelate metodele.

Metoda __init__()

Clasa **`Cat`** este definită cu un constructor **`__init__()`**. Metoda **`__init__()`** este o metodă specială care automat este executată la crearea unei instanțe în baza clasei create. Numele metodei începe și se termină cu două semne de subliniere; acest model previne conflictul dintre numele metodelor standard Python și metodele clasei create. Fiți atenți: două sublinieri trebuie să apară pe fiecare parte a funcției **`__init__()`**. Dacă omitem câte o subliniere pe fiecare parte, metoda nu va fi apelată automat când clasa este utilizată, ceea ce poate duce la erori subtile.

În exemplul nostru, metoda **__init__** inițializează un obiect nou cu trei parametri: **self**, **cat_name**, **cat_age**. Parametrul **self** se include obligatoriu la definirea metodei și trebuie să preceadă toți ceilalți parametri, deoarece data viitoare, la apelarea **__init__()** (pentru a crea o nouă instanță **Cat**), Python transmite automat argumentul **self**. Ori de câte ori apelăm o metodă asociată cu o clasă, **self** (referința la instanță) este transmisă automat. Aceasta oferă instanței specifice acces la atributele și metodele clasei. La crearea unei instanțe **Cat**, Python apelează metoda **__init__()** a clasei **Cat**. Transmitem metodei **Cat()** un nume și o vârstă ca argumente; valoarea lui **self** este transmisă automat, deci, nu este nevoie să o transmitem repetat. De fiecare dată când creăm o instanță a clasei **Cat**, trebuie doar să furnizăm valori pentru ultimele două argumente, **cat_name**, și **cat_age**.

Inițializarea atributelor

Fiecare dintre cele două variabile definite în metoda __init__ conține prefixul **self** (**self.name**, **self.age**). Orice variabilă cu prefixul **self** este disponibilă pentru fiecare metodă din clasă și putem accesa aceste variabile în fiecare instanță creată în baza clasei. Construcția **self.name = name** preia valoarea stocată în parametrul **cat_name** și o stochează în variabila **name**, care este apoi asociată cu instanța creată. Procesul se repetă și cu **self.age** = **cat_age**. Variabilele accesate prin instanțe se numesc **atribute**.

Clasa **Cat** definește, de asemenea, patru metode: **get_age()**, **catch_mice()**, **play()** și **scratch()**. Deoarece aceste metode nu necesită informații suplimentare, decât nume sau vârstă, ele sunt definite cu un singur parametru, **self**. Instanțele create ulterior vor putea apela aceste metode. Deocamdată, metodele definite se limitează la simpla afișare a unui mesaj. Cu toate acestea, conceptul poate fi extins cu ușurință la aplicații practice: dacă această clasă ar face parte dintr-un joc pe calculator, aceste metode ar putea conține cu ușurință un cod pentru a crea o animație a pisicii care prinde un șoarece, se joacă sau zgârie.

9.1.2. Crearea obiectelor

Fiecare obiect aparține unei anumite clase (tip), care specifică comportamentul obiectelor create pe baza sa. Un obiect creat pe baza unei anumite clase se numește instanță a clasei.

Obiectele sunt create prin apelarea clasei ca funcție, transmițându-i argumente pe care le primește metoda **`__init__`**. Creăm o instanță a clasei **`Cat`**:

```
my_cat = Cat("Molly", 2)
```

În linia de cod de mai sus solicităm crearea unei instanțe a unei pisici pe nume "**`Molly`**" cu vârsta de **`2`** ani. Când Python procesează această linie, apelează metoda **`__init__()`** a clasei **`Cat`** cu argumentele "**`Molly`**" și **`2`**. Metoda **`__init__()`** creează o instanță care reprezintă o anumită pisică și setează atributele nume și vârstă la valorile transmise. Python returnează apoi instanța care reprezintă pisica. Această instanță este stocată în variabila **`my_cat`**. Ne amintim convențiile de denumire: un nume care începe cu literă mare (de exemplu, **`Cat`**) este în general considerat a reprezenta o clasă, în timp ce un nume scris cu literă mică (de exemplu, **`my_cat`**) este considerat a reprezenta o singură instanță a clasei.

Pentru a accesa atributele unei instanțe, folosim notația „punct”. Notația punct este utilizată frecvent în Python. Această sintaxă arată cum Python caută valorile atributelor.

În prima linie de cod din exemplul de mai jos, accesăm valoarea atributului **`name`** al instanței **`my_cat`** (**`my_cat.name`**). În acest caz, Python accesează instanța **`my_cat`** și caută atributul **`name`** asociat cu instanța creată. Acesta este același atribut care a fost numit **`self.name`** în clasa **`Cat`**. Similar se prelucrează atributul **`age`**:

```
print(f"My cat's name is {my_cat.name}.")
print(f"{my_cat.name} is {my_cat.age} years old.")
```

```
My cat's name is Molly.
Molly is 2 years old.
```

După crearea unei instanțe a clasei **`Cat`** putem aplica notația „punct” pentru apelarea oricărei metode definite în clasa **`Cat.`** Pentru a apela o metodă, indicăm instanța (în cazul nostru **`my_cat`**) și metoda apelată, divizându-le prin punct. În timpul procesării metodei **`my_cat.get_age()`**, Python caută metoda **`get_age()`** în clasa **`Cat`** și execută codul acesteia.

Celelalte metode sunt interpretate în mod similar. Acum instanța acționează conform instrucțiunilor:

```
print(my_cat.get_age())
print(my_cat.catch_mice())
print(my_cat.play())
print(my_cat.scratch())
```

```
Molly is 2 years old
Molly is catching mice
Molly is playing
Molly is scratching
```

În baza unei clase pot fi create atâtea instanțe, de câte e nevoie. Creăm alte trei instanțe ale clasei **`Cat`**. Afișăm atributele **`name`** ale obiectelor și aplicăm metoda **`get_age()`** instanțelor. Ca și instanța **`my_cat`**, celelalte trei instanțe create (**`your_cat`**, **`his_cat`**, **`her_cat`**) pot îndeplini acțiunile din setul general de metode:

```
your_cat = Cat('Alfie', 4)
his_cat = Cat('Poppy', 5)
her_cat = Cat('Oreo', 7)
print(f"Your cat's name is {your_cat.name}.",
      your_cat.get_age())
print(f"His cat's name is {his_cat.name}.",
      his_cat.get_age())
print(f"Her cat's name is {her_cat.name}.",
      her_cat.get_age())
print(f"{your_cat.catch_mice()}, {his_cat.play()}",
      f"\n and {her_cat.scratch()}")
```

```
Your cat's name is Alfie. Alfie is 4 years old
His cat's name is Poppy. Poppy is 5 years old
Her cat's name is Oreo. Oreo is 7 years old
Alfie is catching mice, Poppy is playing
 and Oreo is scratching
```

Chiar dacă altei pisici i se va atribui același nume și aceeași vârstă, Python va crea în continuare o instanță separată a clasei **`Cat`**. Puteți crea oricâte instanțe doriți ale aceleiași clase, atât timp cât instanțele sunt stocate în variabile cu nume diferite sau ocupă poziții diferite într-o listă sau dicționar.

9.2. Moștenirea

Nu trebuie neapărat să o luați de la capăt când scrieți o clasă nouă. Dacă clasa pe care o scrieți este o versiune specializată a unei clase scrise anterior, puteți utiliza **moștenirea**. O clasă care moștenește de la alta primește automat toate atributele și metodele primei clase. Clasa originală se numește părinte, clasa de bază sau superclasă, iar noua clasă se numește clasa-copil sau subclasă. Subclasa moștenește atributele și metodele clasei-părinte, dar poate, de asemenea, să definească propriile atribute și metode.

Când definim o clasă nouă bazată pe o clasă existentă, trebuie să apelăm metoda **`__init__()`** a clasei-părinte. Aceasta inițializează orice atribute definite în metoda **`__init__()`** a clasei-părinte și face ca aceste atribute să fie disponibile clasei-copil.

Notă! La crearea unei clase-copil, clasa-părinte trebuie să facă parte din fișierul curent, iar definirea clasei-părinte trebuie să preceadă definirea clasei-copil din fișier.

Vom defini clasa-copil **`PersianCat`** în baza clasei **`Cat`**, care are aceleași caracteristici și acționează la fel, dar mai descrie suplimentar atributul **`fur_length`** și metoda nouă **`groom()`**:

```
# Definirea clasei copil, indicăm clasa părinte în paranteze
class PersianCat(Cat):
    def __init__(self, name, age, fur_length = "long"):
        super().__init__(name, age)
        self.fur_length = fur_length

    def groom(self):
        return f"{self.name} with {self.fur_length} fur"
        "is being groomed."
```

La definirea clasei-copil, în parantezele rotunde care urmează după numele clasei-copil, indicăm numele clasei-părinte (**Cat**). Metoda **__init__** obține informația necesară pentru crearea instanței **Cat**.

Funcția **super()** din a treia linie este o funcție specială, care permite apelarea metodei **__init__** din clasa-părinte **Cat**. Drept rezultat, orice instanță a clasei-copil **PersianCat** obține acces la toate atributele și metodele clasei-părinte. Numele funcției **super()** provine de la superclasă.

Suplimentar, în clasa copil am inițializat atributul **fur_length** și am definit metoda **groom()**. Pentru a verifica dacă moștenirea funcționează, vom crea un obiect al clasei-copil – **pers** și alături de metoda **groom()** aplicăm instanței **pers** metoda din clasa-părinte **get_age()**:

```
pers = PersianCat("Fluffy", 3)
print(pers.groom())
print(pers.get_age())
```

```
Fluffy with long fur is being groomed.
Fluffy is 3 years old
```

Ne-am convins că moștenirea funcționează.

9.3. Polimorfismul

Orice metodă a clasei-părinte care face altceva decât ceea ce este necesar în situația simulată poate fi suprascrisă.

Pentru a face acest lucru, o metodă cu același nume ca și metoda clasei-părinte este definită în clasa-copil. Python ignoră metoda părinte și acordă atenție doar metodei definite în clasa-copil.

De exemplu, clasa **`Cat`** are definită metoda **`catch_mice()`**. În același timp, pisicile de rasă persiană sunt cunoscute pentru un temperament mai calm și preferă confortul și relaxarea în locul activităților de vânătoare. În acest caz, vom suprascrie metoda **`catch_mice()`** pentru instanțele subclasei **`PersianCat`**:

```
class PersianCat(Cat):
    def __init__(self, name, age, fur_length = "long"):
        super().__init__(name, age)
        self.fur_length = fur_length
    def groom(self):
        return f"{self.name} with {self.fur_length} fur"
        "is being groomed."
    def catch_mice(self):
        return f"{self.name} is friend with mice"
```

Acum, la crearea unei instanțe în baza subclasei **`PersianCat`**, metoda **`catch_mice()`** aplicată instanței returnează alt rezultat decât obiectele clasei-părinte **`Cat`**:

```
pers = PersianCat("Fluffy", 3)
print(pers.catch_mice())
```

```
Fluffy is friend with mice
```

Această tehnică se numește polimorfism. Polimorfismul permite folosirea aceleiași metode din clasa-părinte, dar cu comportament diferit în subclase.

9.4. Abstractizarea

Abstractizarea presupune definirea unei clase abstracte prin metode care nu sunt implementate direct, ci trebuie să fie suprascrise de subclase. Abstractizarea datelor poate fi definită ca fiind ascunderea tuturor datelor/proceselor irelevante dintr-o clasă pentru a reduce complexitatea și a crește eficiența programului.

De regulă, clasele reflectă anumite obiecte din viața reală, dar uneori trebuie să lucrăm cu entități care nu au o întruchipare concretă. De exemplu, entitatea **felină** este un animal din familia Felidae, care include foarte multe specii: tigru, leu, pisică, jaguar, leopard, puma, ghepard etc. Prin urmare, felina este un animal abstract și pentru definirea entității **felină**, avem nevoie de o clasă abstractă.

O **clasă abstractă** este o clasă care nu permite crearea unor instanțe în baza acesteia direct. Ea este destinată să fie moștenită de alte clase și definește un **șablon** pentru metodele pe care acestea trebuie să le implementeze.

În Python, toate instrumentele destinate creării claselor abstracte sunt definite în modulul special **`abc`**, care trebuie conectat suplimentar în aplicație. Componentele-cheie ale acestui modul sunt clasa **`ABC`** și adnotarea **`@abstractmethod`**.

Clasa **`ABC`** simplifică crearea unei clase abstracte, iar toate clasele abstracte definite moștenesc din această clasă, iar adnotarea **`@abstractmethod`** este destinată creării unei metode abstracte. Clasele abstracte sunt definite ca și clasele normale, cu excepția faptului că ele moștenesc din clasa **`ABC`** din modulul **`abc`**.

Să definim o clasă abstractă **`Feline`** din care moștenesc toate pisicile și subclasa **`Cat`**. Observăm că metodele sunt descrise în subclasa **`Cat`**, deoarece **`Feline`** este o clasă abstractă și metodele ei trebuie implementate în subclase:

```
from abc import ABC, abstractmethod
class Feline(ABC):
    @abstractmethod
    def make_sound(self):
        pass
    @abstractmethod
    def sleep(self):
        pass

class Cat(Feline):
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def make_sound(self):
        return "Meow!"
    def sleep(self):
        return f"{self.name} is sleeping"
```

Acum putem crea o instanță a clasei **`Cat`** și să apelăm metodele definite în clasa abstractă **`Feline`**, dar descrise în subclasa **`Cat`**:

```
my_cat = Cat("Fluffy", 3)
print(my_cat.make_sound())
print(my_cat.sleep())
```

```
Fluffy says Meow!
Fluffy is sleeping
```

9.5. Încapsularea

În mod implicit, atributele din clase sunt publice, ceea ce înseamnă că, din punct de vedere al programului, putem prelua un atribut al obiectului și îl putem modifica.

De exemplu, dacă pentru instanța **`my_cat`** vom schimba valorile atributelor **`name`** și **`age`**, numele **`Fluffy`** și vârsta de **`3`** ani nu mai sunt asociate cu **`my_cat`**:

```
my_cat.name = "Tom"
my_cat.age = -3
print(my_cat.sleep())
print(f"{my_cat.name} is {my_cat.age} years old")
```

```
Tom is sleeping
Tom is -3 years old
```

Această capacitate poate duce la situații nedorite, în special când valoarea unui atribut a fost modificată întâmplător sau poate chiar intenționat.

Încapsularea presupune protejarea atributelor interne ale unei clase pentru a nu fi accesibile direct din exterior. Se folosesc variabile **private**, prefixate cu două semne de subliniere (__). Vom rescrie codul din clasa Cat(Feline) prezentat mai sus, făcând ambele atribute **`name`** și **`age`** private:

```
class Cat(Feline):
    def __init__(self, name, age):
        self.__name = name # atribut privat
        self.__age = age # atribut privat
    def make_sound(self):
        return f"{self.__name} says Meow!"
    def sleep(self):
        return f"{self.__name} is sleeping"
    def get_age(self):
        return f"{self.__name} is {self.__age} years old"
```

Acum încercăm să modificăm numele și vârsta pisicii, însă la apelarea metodelor observăm că valorile **`Fluffy`** și **`3`** nu s-au schimbat în **`Tom`** și **`-3`**.

Astfel, încapsularea protejează atributele și metodele unei clase de la modificări intenționate sau accidentale.

```
my_cat.name = "Tom"
my_cat.age = -3
print(my_cat.sleep())
print(my_cat.get_age())
```

```
Fluffy is sleeping
Fluffy is 3 years old
```

În concluzie, programarea orientată pe obiecte reprezintă o paradigmă fundamentală în dezvoltarea aplicațiilor moderne, oferind instrumente puternice pentru modelarea conceptelor din lumea reală prin intermediul claselor și obiectelor. Principiile fundamentale ale POO – abstractizarea, încapsularea, moștenirea și polimorfismul – permit crearea de cod modular, reutilizabil și ușor de întreținut, facilitând gestionarea complexității în proiectele software. Limbajul Python integrează complet aceste principii, oferind suport nativ pentru definirea și manipularea claselor și obiectelor, ceea ce îl face ideal pentru proiectele complexe și scalabile.

Sarcini

1. **Crearea clasei „Carte”.** Creați o clasă **`Carte`** cu atributele: titlu **`(string)`**; autor **`(string)`**; stoc **`(int)`**.
   Adăugați o metodă **`este_disponibila()`** care returnează **`True`** dacă stocul > 0 și afișează un mesaj „Stoc epuizat” în caz contrar.
2. **Moștenire – clasa „Student” derivată din clasa „Persoană”.** Creați o clasă **`Persoana`** cu următoarele atribute: nume **`(string)`**; varsta **`(int)`**.
   Adăugați un constructor pentru inițializarea obiectului și o metodă **`prezentare()`** care afișează: „Numele meu este X și am Y ani”. Creați clasa **`student`** care moștenește clasa **`Persoana`**. Adăugați un nou atribut: **`universitate`** și o metodă **`descriere()`** care afișează numele, vârsta și universitatea.

BIBLIOGRAFIE

1. Amos D., Bader D., Jablonski J., Heisler F. (2020). *Python Basics: A Practical Introduction to Python 3*. 4th Edition, Real Python. ISBN 978-1-77-509333-6.
2. Bader D. (2018). *Python Tricks: The Book*. ISBN 978-1-77-509330-5.
3. Barry P. (2017). *Head First Python. A Brain-Friendly Guide*. O'Reilly. ISBN 978-1-491-91953-8.
4. Beazley D., Jones B.K. (2013). *Python Cookbook*. Third Edition, O'Reilly Media. ISBN 978-1-44-934037-7.
5. Cotelea V., Ungur C. (2020). *Python: prima mea carte*, Universitatea Tehnica a Moldovei. Chisinau: S. n., Tipogr. "Foxtrot". ISBN 978-9-97-589161-5.
6. Das U., Lawson A., Mayfield C., Norouzi N. (2024). *Introduction to Python Programming*. Open Stax, Rice University. ISBN 978-1-96-158445-7.
7. Gaddis T. (2019). *Starting out with Python*. Fourht Editon. Pearson Education. ISBN 978-0-13-444432-1.
8. Halvorsen H.P. (2020). *Python Programming*. ISBN 978-8-26-911064-7.
9. Hunt J. (2019). *A Beginners Guide to Python 3 Programming*. Springer. ISBN 978-3-03-020290-3.
10. *Learn Python – Free Interacitve Python Tutorial* - https://www.learnpython.org/.
11. Lutz M. (2009). *Learning Python, Fourth Edition*. O'Reilly Media. ISBN 978-0-59-615806-4
12. Matthes E. (2019). *Python Crash Course: A Hands-On, Project-Based Introduction to Programming*, Second Edition, No Starch Press. ISBN 978-1-59-327928-8.
13. McKinney W. (2018). *Python for Data Analysis*. O'Reilly Media. Second Edition. ISBN 978-1-49-195766-0.
    Media. Second Edition. ISBN 978-1-49-195865-0.
14. Percival H. (2017). *Test-Driven Development with Python*, O'Reilly.

15. *Python 3 pentru începători – Curs online interactiv* - https://www.pythonisti.ro/python-3-lectii-online-interactive.php.
16. *Python for beginners* - https://www.python.org/about/gettingstarted/
17. *Python Tutorial* - https://www.w3schools.com/python/default.asp.
18. Sedgewik R., Wayne K., Dondero R. (2015). *Introduction to programming in Python: and interdisciplinary approach*. Pearson Education, US. ISBN 978-0-13-407643-0.
19. *The Python Language Reference* - https://docs.python.org/3/reference/index.html#reference-index.
20. Tudor V. (2020). *Curs de programare în Python 3 – Fundamentele pentru începători*. L&S SOFT. ISBN 978-6-06-948981-9.
21. Tudor V. (2023). *Curs de programare în Python 3 – Structuri de date şi algoritmi*. L&S SOFT. ISBN 978-6-30-655901-5.
22. Дауни А.Б. (2021). *Основы Python. Научитесь думать как программист*. Москва: Манн, Иванов и Фарбер. ISBN 978-5-00-145798-4.
23. Любанович Б. (2016). *Простой Python. Современный стиль программирования.* СПб.: Питер. ISBN 978-5-49-602088-6.

Bazele programării calculatoarelor
în limbajul Python

Suport de curs

Autori: Rodica Cujba
Olesea Borozan

Redactor E.Balan

Bun de tipar 29.08.25	Formatul hârtiei 60x84 1/16
Hârtie ofset. Tipar RISO	Tirajul 50 ex.
Coli de tipar 12,25	Comanda nr.72

MD-2004, Chișinău, bd. Ștefan cel Mare și Sfânt, 168, UTM
MD-2045, Chișinău, str. Studenților, 9/9, Editura „Tehnica-UTM”