MINISTERUL EDUCAȚIEI ȘI CERCETĂRII

Carmen-Diana Cosman • Melinda-Emilia Coriteac
Amelia Stoian • Isabela-Patricia Coman
Miruna-Andreea Zăvelcă • Cristina Elena Anton
Violeta Marilena Grecea • Andreea Floriana Bontea

Informatică

curriculum de specialitate (CS), filiera teoretică, profilul real, specializarea matematică-informatică

9

Manual pentru clasa a IX-a



SIGMA

Acest manual este proprietatea Ministerului Educației și Cercetării.

Acest proiect de manual școlar este realizat în conformitate cu Programa școlară aprobată prin Ordinul ministrului educației și cercetării nr. 6930/19.12.2025

Numărul de telefon **119** – număr unic de telefon la nivel național pentru cazurile de abuz împotriva copiilor.

Numărul de telefon **116.111** – număr de asistență pentru copii, în conformitate cu Legea nr. 272/2004 privind protecția și promovarea drepturilor copilului, republicată, cu modificările și completările ulterioare.

Ministerul Educației și Cercetării

Carmen-Diana Cosman • Melinda-Emilia Coriteac • Amelia Stoian
Isabela-Patricia Coman • Miruna-Andreea Zăvelcă • Cristina Elena Anton
Violeta Marilena Grecea • Andreea Floriana Bontea

Informatică

curriculum de specialitate (CS), filiera teoretică, profilul real, specializarea matematică-informatică

Manual pentru clasa a IX-a

SIGMA
București, 2026

Manualul școlar a fost aprobat prin Ordinul de Ministru al Educației și Cercetării nr. 5779 din data de 09.10.2026.
Manualul este distribuit elevilor în mod gratuit, atât în format tipărit, cât și în format digital, și este transmisibil timp de patru ani școlari, începând din anul școlar 2026-2027.
Inspectoratul Școlar .....
Școala / Colegiul / Liceul .....

ACEST MANUAL A FOST FOLOSIT DE:

Anul	Numele elevului	Clasa	Anul școlar	Aspectul manualului*			
				Format tipărit		Format digital	
				la primire	la predare	la primire	la predare
1							
2							
3							
4							

* Pentru precizarea aspectului manualului se va folosi unul dintre următorii termeni: ***nou, bun, îngrijit, neîngrijit, deteriorat***.

Cadrele didactice vor verifica dacă informațiile înscrise în tabelul de mai sus sunt corecte.
Elevii nu vor face niciun fel de însemnări pe manual.

Referenți științifici:
- ***Angela Blaga***: profesor, grad didactic I în Informatică, doctor în Inginerie și management, la Colegiul Național „Alexandru Papiu Ilarian” Târgu Mureș, județul Mureș.
- ***Horea-Gavril Oros***: lector universitar doctor, titular la Universitatea din Oradea, Facultatea de Informatică și Științe, Departamentul de Matematică și Informatică.
- ***Delia Tușe***: lector universitar, doctor în Informatică, la Universitatea din Oradea, Facultatea de Informatică și Științe, Departamentul de Matematică și Informatică.

Manager proiect: Radu Răcăreanu
Redactor: Dominic Burghelea
Grafică și tehnoredactare: Adrian-Viorel Rașca
Coperta: Adrian-Viorel Rașca
Credite foto și video: Canva, Magnific
Platforma e-learning, activități digitale interactive, voce, procesare sunet și animații:
IMAGINE INFINITY SRL

ISBN: 978-606-727-693-0

© 2026 – EDITURA SIGMA
Toate drepturile asupra prezentei ediții aparțin Editurii SIGMA. Nicio parte a acestei lucrări nu poate fi reprodusă fără acordul scris al Editurii SIGMA.

Editura SIGMA
Sediul central:
Str. Nicolae Cartojan, nr. 11-13-15, sector 2, București, cod poștal 023815
Tel.: 0758 044 829; 0723 541 020
E-mail: office@editurasigma.ro
Site: www.editurasigma.ro
Distribuție:
Str. Nicolae Cartojan, nr. 11-13-15, sector 2, București, cod poștal 023815
Tel.: 0758 044 825; 0758 044 828
E-mail: comenzi@editurasigma.ro

Cuvânt-înainte

Manualul de față a fost conceput pentru a vă sprijini în formarea gândirii computaționale, algoritmice și creative, esențiale în domeniul informaticii. Acesta își propune să vă ajute să proiectați soluții corecte, eficiente și bine organizate, valorificând limbajul Python și instrumentele moderne ale programării.

Lucrarea este structurată în șase capitole: Fundamentele programării și reprezentarea algoritmilor, Prelucrarea datelor numerice, Modularitate și programare orientată pe obiecte, Interfețe grafice și fișiere text, Organizarea datelor în liste, respectiv Strategii avansate de sortare și eficiență algoritmică.

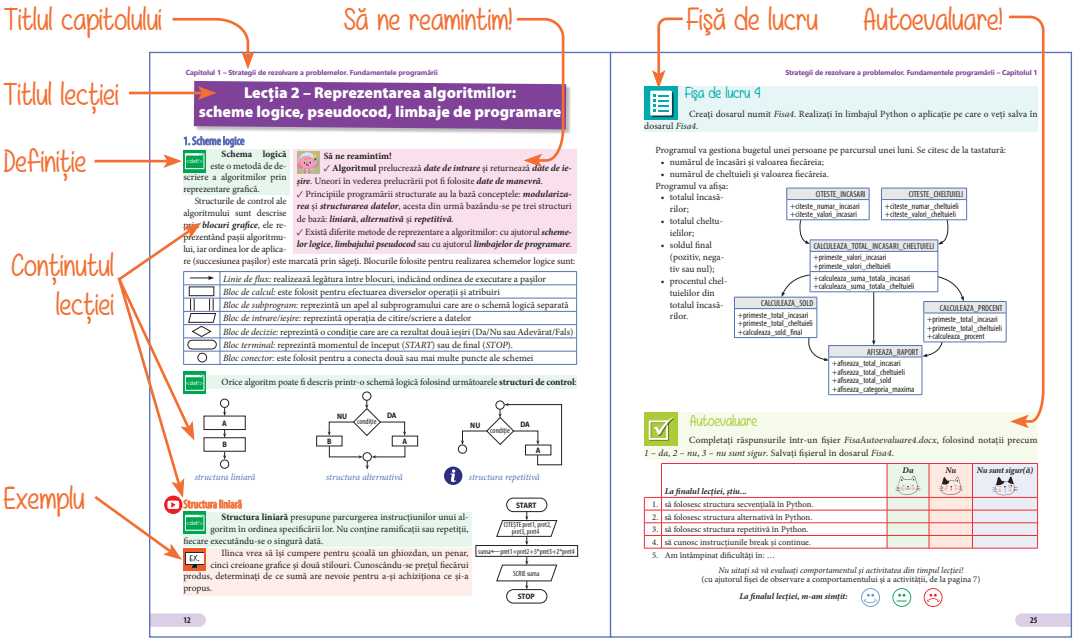
Lecțiile sunt construite pentru a favoriza învățarea prin descoperire, prin exemple și activități practice. Fișele de lucru de la finalul fiecărei lecții vă oferă ocazia să aplicați noțiunile în situații concrete, iar activitățile de autoevaluare vă ajută să vă urmăriți progresul și să vă formați o atitudine responsabilă față de propria învățare.

Vă încurajăm să folosiți acest manual ca pe un punct de plecare pentru curiozitate, explorare și dezvoltare personală. Scrieți cod cu atenție și creativitate — fiecare problemă rezolvată este o ocazie de a gândi mai clar.

Autorii

Manualul digital

Varianta digitală a manualului conține pe lângă varianta tipărită și activități multimedia interactive de învățare (AMII). Acestea vin în completarea noțiunilor și exemplelor prezentate în manualul tipărit și sunt de trei tipuri: AMII statice (desene, fotografii sau planșe didactice); AMII dinamice (filme sau animații) și AMII interactive (diverse exerciții, jocuri educative).



CUPRINS

Cuvânt-înainte. Prezentarea manualului 3
Competențe generale și competențe specifice 5
Instrucțiuni privind organizarea modului de lucru al fișelor/aplicațiilor din manual 6
Fișă de observare sistematică a activității și a comportamentului elevului 7
Evaluare inițială 8

Capitolul 1 – Strategii de rezolvare a problemelor. Fundamentele programării (1.3.; 1.4.; 2.1.; 2.3.; 2.4.; 3.3.; 3.4.; 4.1.; 4.3.; 4.4.; 5.3.; 6.3.)
Lecția 1 *– Etapele de elaborare a unui program: de la analiză și proiectare la testare și depanare* 9
Fișa de lucru 1 11
Lecția 2 *– Reprezentarea algoritmilor: scheme logice, pseudocod, limbaje de programare* 12
Fișa de lucru 2 17
Lecția 3 *– Arhitectura unui program Python. Variabile, tipuri de date și expresii* 19
Fișa de lucru 3 22
Lecția 4 *– Structuri de control: liniară, alternativă și repetitivă* 23
Fișa de lucru 4 25
Evaluare sumativă – Capitolul 1 26

Capitolul 2 – Strategii de rezolvare a problemelor. Prelucrări ale datelor numerice (1.2.; 2.2.; 3.2.; 4.2.; 5.2.; 6.2.)
Lecția 5 *– Operații cu cifrele unui număr* 28
Fișa de lucru 5 29
Lecția 6 *– Algoritmi de divizibilitate: Divizorii unui număr; Primalitate* 32
Fișa de lucru 6 34
Lecția 7 *– Descompunerea în factori primi; CMMDC, CMMMC (Euclid)* 36
Fișa de lucru 7 39
Lecția 8 *– Baze de numerație* 42
Fișa de lucru 8 44
Evaluare sumativă – Capitolul 2 46

Capitolul 3 – Modularitate și programare orientată pe obiecte (1.5.; 2.5.; 3.5.; 4.5.; 5.5.; 6.5.)
Lecția 9 *– Subprograme: antet, corp, parametri* 48
Fișa de lucru 9 51
Lecția 10 *– Mecanismul de execuție: variabile locale vs variabile globale* 52
Fișa de lucru 10 55
Lecția 11 *– Subprograme predefinite pentru colecții (Python)* 56
Fișa de lucru 11 59
Lecția 12 *– Introducere în POO: clase, obiecte și instanțiere* 61
Fișa de lucru 12 65
Evaluare sumativă – Capitolul 3 67

Capitolul 4 – Interfețe grafice și fișiere text (1.4.; 2.3.; 2.4.; 3.3.; 3.4.; 4.4.; 5.4.; 6.4.)
Lecția 13 *– Biblioteca Tkinter: crearea ferestrelor și managementul așezării* 69
Fișa de lucru 13 72
Lecția 14 *– Widget-uri interactive: butoane, etichete și casete text* 73
Fișa de lucru 14 79
Lecția 15 *– Gestionarea evenimentelor în interfețe grafice (afișarea mesajelor la clic)* 80
Fișa de lucru 15 83
Lecția 16 *– Fișiere text: deschidere, închidere și transfer de date* 85
Fișa de lucru 16 88
Lecția 17 *– Clasa TextIOWrapper: metode de citire și scriere eficientă* 90
Fișa de lucru 17 93
Evaluare sumativă – Capitolul 4 96

Capitolul 5 – Organizarea conceptuală a datelor (liste) (1.1.; 1.4., 2.1.; 2.4., 3.1.; 3.4., 4.1., 4.4.; 5.1.; 6.1.)
Lecția 18 *– Modelul conceptual liniar: caracteristici, acces direct vs secvențial* 98
Fișa de lucru 18 101
Lecția 19 *– Liste de frecvență, stiva (LIFO) și coada (FIFO)* 103
Fișa de lucru 19 105
Lecția 20 *– Clasa list din Python: operatori specifici și metode de bază* 107
Fișa de lucru 20 111
Lecția 21 *– Metode avansate ale clasei list și parcurgerea elementelor* 112
Fișa de lucru 21 116
Evaluare sumativă – Capitolul 5 118

Capitolul 6 – Strategii avansate de rezolvare. Generare sistematică, sortare și eficiență algoritmică (1.2.; 2.1.; 2.2.; 3.2.; 3.4.; 4.2.; 5.2.; 6.2.)
Lecția 22 *– Eficiența algoritmilor: timp de execuție, memorie și notația O(n)* 120
Fișa de lucru 22 123
Lecția 23 *– Generare sistematică: șiruri recurente și secvențe cu proprietăți date* 125
Fișa de lucru 23 128
Lecția 24 *– Metode de sortare: selecția minimului* 130
Fișa de lucru 24 132
Lecția 25 *– Metode de sortare: metoda bulelor* 134
Fișa de lucru 25 137
Lecția 26 *– Metode de sortare: sortarea cu listă de frecvențe* 139
Fișa de lucru 26 142
Evaluare sumativă – Capitolul 6 144
Evaluare finală 146
Indicații și răspunsuri 148

Competențe generale și competențe specifice

CG1 – Identifică principalele caracteristici ale modelelor conceptuale și operaționale ale dezvoltării produselor software, pentru înțelegerea fundamentelor programării

- **CS 1.1.** Identifică principalele caracteristici ale organizării datelor în cadrul unor modele conceptuale fundamentale – date simple sau liste, pentru a structura și accesa date în vederea prelucrării acestora
- **CS 1.2.** Identifică specificul, caracteristicile și etapele unor algoritmi specializați pe clase de probleme, pentru a îi putea utiliza în prelucrarea algoritmică a numerelor, sortarea sau generarea sistematică a unor secvențe de valori
- **CS 1.3.** Identifică specificul, caracteristicile și etapele unor strategii de rezolvare a problemelor prin proiectarea modulară a algoritmilor
- **CS 1.4.** Identifică principalele elemente ale limbajului de programare utilizate pentru prelucrarea datelor organizate în modele fundamentale – date simple sau liste, respectiv pentru transmiterea datelor de la și către un program, în vederea rezolvării eficiente a problemelor informatice
- **CS 1.5.** Identifică elementele de sintaxă și componentele din definiția și apelul subprogramelor și subprogramelor predefinite pentru operații uzuale, pentru a le utiliza în implementarea algoritmilor în limbaj de programare

CG2 – Explică principii care stau la baza modelelor conceptuale și operaționale ale dezvoltării produselor software, pentru a fundamenta în mod logic proiectarea și implementarea soluțiilor informatice

- **CS 2.1.** Explică principiile de organizare a datelor în cadrul unor modele conceptuale fundamentale – date simple sau liste, pentru a le gestiona eficient
- **CS 2.2.** Explică etapele algoritmilor specializați pe clase de probleme, pentru a îi putea utiliza în prelucrarea algoritmică a numerelor, sortarea sau generarea sistematică a unor secvențe de valori
- **CS 2.3.** Explică etapele unor strategii de rezolvare a problemelor prin proiectarea modulară a algoritmilor
- **CS 2.4.** Explică rolul principalelor elemente ale limbajului de programare utilizate pentru prelucrarea datelor organizate în modele fundamentale – date simple sau liste, respectiv pentru transmiterea datelor de la și către un program în vederea rezolvării eficiente a problemelor informatice
- **CS 2.5.** Explică mecanismul de executare a subprogramelor și subprogramelor predefinite pentru operații uzuale, pentru a le utiliza în implementarea algoritmilor în limbaj de programare

CG3 – Utilizează modele conceptuale și operaționale ale dezvoltării produselor software, în scopul obținerii de soluții informatice funcționale și eficiente

- **CS 3.1.** Utilizează modul de organizare și prelucrare a datelor în cadrul unor modele conceptuale fundamentale – date simple sau liste, pentru a rezolva probleme de gestionare a datelor
- **CS 3.2.** Aplică algoritmi specializați pe clase de probleme, pentru a îi putea utiliza în prelucrarea algoritmică a numerelor, sortarea sau generarea sistematică a unor secvențe de valori
- **CS 3.3.** Aplică strategii de rezolvare a problemelor prin proiectarea modulară a algoritmilor
- **CS 3.4.** Utilizează principalele elemente ale limbajului de programare utilizate pentru prelucrarea datelor organizate în modele fundamentale – date simple sau liste, respectiv pentru transmiterea datelor de la și către un program, în vederea rezolvării eficiente a problemelor informatice
- **CS 3.5.** Utilizează elementele de sintaxă și componentele din definiția și apelul subprogramelor și subprogramelor predefinite pentru operații uzuale, în implementarea algoritmilor în limbaj de programare

CG4 – Analizează caracteristicile și aplicabilitatea modelelor conceptuale și operaționale ale dezvoltării produselor software, pentru a selecta soluțiile cele mai potrivite în funcție de contextul informatic dat

- **CS 4.1.** Analizează caracteristicile, modul de prelucrare, avantajele și dezavantajele utilizării unor modele conceptuale fundamentale – date simple sau liste, pentru a alege structura adecvată în rezolvarea unor probleme concrete
- **CS 4.2.** Analizează comportamentul, aplicabilitatea, avantajele și dezavantajele utilizării unor algoritmi specializați pe clase de probleme pentru prelucrarea numerelor, sortarea sau generarea sistematică a unor secvențe de valori
- **CS 4.3.** Analizează aplicabilitatea, avantajele și dezavantajele utilizării unor strategii de rezolvare a problemelor prin proiectarea modulară a algoritmilor
- **CS 4.4.** Analizează aplicabilitatea, avantajele și dezavantajele utilizării unor elemente ale limbajului de programare, pentru a identifica soluția adecvată prelucrării datelor organizate în modele fundamentale – date simple sau liste, respectiv pentru transmiterea datelor de la și către un program, în vederea rezolvării eficiente a problemelor informatice
- **CS 4.5.** Analizează aplicabilitatea, avantajele și dezavantajele utilizării subprogramelor, în implementarea algoritmilor în limbaj de programare

CG5 – Evaluează corectitudinea și eficiența soluțiilor informatice, în vederea optimizării și asigurării funcționalității în diverse scenarii de utilizare

- **CS 5.1.** Argumentează alegerea unor modele conceptuale fundamentale – date simple sau liste – și a modurilor de prelucrare a acestora pentru rezolvarea unor probleme informatice concrete
- **CS 5.2.** Evaluează corectitudinea și eficiența soluțiilor cu algoritmi specializați pe clase de probleme pentru prelucrarea numerelor, sortarea sau generarea sistematică a unor secvențe de valori
- **CS 5.3.** Evaluează corectitudinea și eficiența algoritmilor care utilizează strategii de rezolvare a problemelor prin proiectarea modulară a algoritmilor
- **CS 5.4.** Evaluează corectitudinea și eficiența aplicațiilor care utilizează elemente specifice de limbaj de programare pentru prelucrarea datelor organizate în modele fundamentale – date simple sau liste, respectiv pentru transmiterea datelor de la și către un program, în vederea rezolvării eficiente a problemelor informatice
- **CS 5.5.** Evaluează corectitudinea și eficiența programelor care utilizează subprograme, pentru a rezolva probleme informatice

CG6 – Elaborează algoritmi și programe personalizate, pentru a crea soluții informatice coerente și adaptate cerințelor

- **CS 6.1.** Proiectează algoritmi personalizați care utilizează modele conceptuale fundamentale – date simple sau liste pentru organizarea și prelucrarea eficientă a datelor, utilizând algoritmi de bază, în vederea rezolvării unor probleme
- **CS 6.2.** Proiectează soluții cu aplicarea personalizată a algoritmilor specializați pe clase de probleme pentru prelucrarea numerelor, sortarea sau generarea sistematică a unor secvențe de valori
- **CS 6.3.** Proiectează modular algoritmi personalizați pentru rezolvarea problemelor
- **CS 6.4.** Implementează aplicații care integrează în mod personalizat elemente specifice de limbaj de programare pentru prelucrarea datelor organizate în modele fundamentale – date simple sau liste, respectiv pentru transmiterea datelor de la și către un program, în vederea realizării unor soluții funcționale și performante
- **CS 6.5.** Implementează aplicații în limbaj de programare care integrează subprograme personalizate, pentru a modulariza și organiza eficient codul în cadrul soluțiilor informatice

Instrucțiuni privind organizarea modului de lucru al fișelor/aplicațiilor din manual

Prezentul manual este organizat în șase capitole și 26 de lecții, fiecare având la final o fișă de lucru.

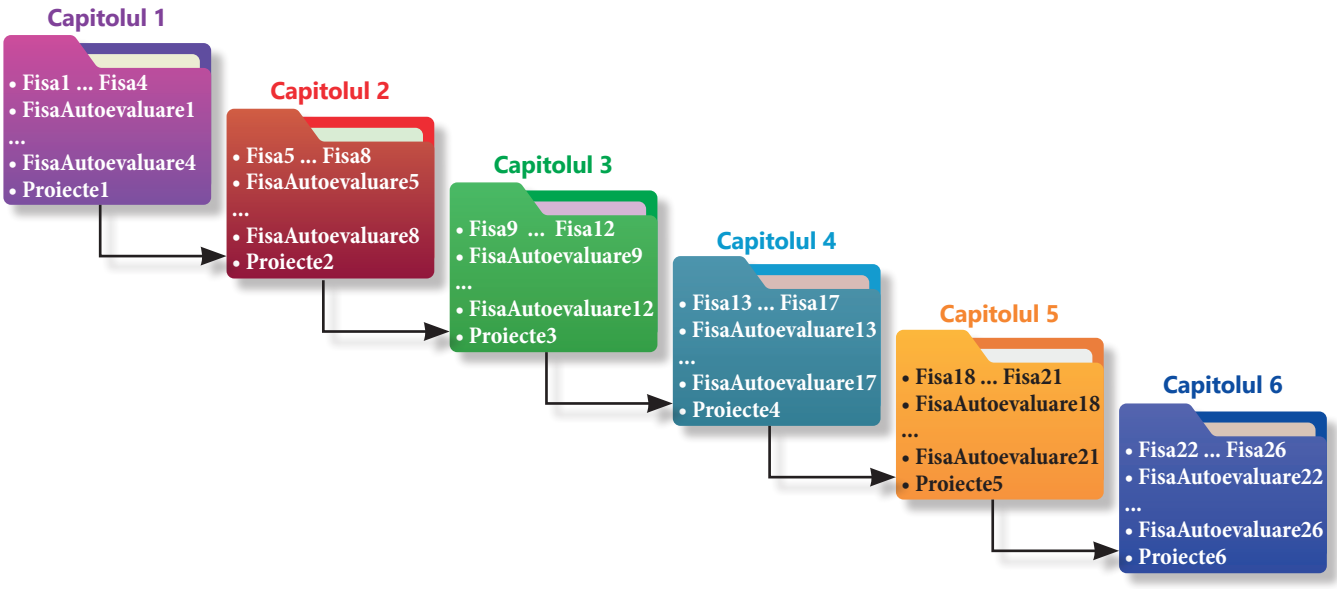
1. Creați pentru fiecare unitate un dosar numit ***CapitolNr.***, unde nr. reprezintă numărul de ordine al capitolului (1, 2, 3, 4, 5 sau 6), într-un loc din memoria externă, unde aveți drept de scriere.

2. Creați pentru fiecare fișă de lucru un dosar numit ***FisaNr.***, unde nr. reprezintă numărul de ordine al fișei de lucru. Fiecare dosar *FisaNr.* va fi salvat în dosarul *CapitolNr.* corespunzător. (Dosarele *Fisa1*, …, *Fisa4* se vor salva în dosarul *Capitol1*, dosarele *Fisa5*, …, *Fisa8* în dosarul *Capitol2*, dosarele *Fisa9*, …, *Fisa12* în dosarul *Capitol3*, dosarele *Fisa13*, …, *Fisa17* în dosarul *Capitol4*, dosarele *Fisa18*, …, *Fisa21* în dosarul *Capitol5*, iar dosarele *Fisa22*, …, *Fisa26* în dosarul *Capitol6*). În aceste dosare salvați fiecare fișier solicitat în fișele de lucru.

3. La finalul fiecărei lecții veți avea la dispoziție o fișă de autoevaluare. Fișele de autoevaluare *1*, …, *4* vor fi salvate în dosarul *FisaNr.* cu același număr aflat în dosarul *Capitol1*; Fișele de autoevaluare *5*, …, *8*, în dosarul *Capitol2*; Fișele de autoevaluare *9*, …, *12*, în dosarul *Capitol3*; Fișele de autoevaluare *13*, …, *17*, în dosarul *Capitol4*; Fișele de autoevaluare *18*, …, *21*, în dosarul *Capitol5*; Fișele de autoevaluare *22*, …, *26*, în dosarul *Capitol6*.

4. Pentru proiectele (Explorează/Investighează) pe care trebuie să le faceți pe parcurs, creați dosarele ***Proiecte1***, ***Proiecte2***, ***Proiecte3***, ***Proiecte4***, ***Proiecte5***, respectiv ***Proiecte6***, în care salvați fișierele solicitate.

La final, **portofoliul** vostru trebuie să aibă următoarea structură:



La finalul fiecărei lecții vă propunem să vă autoevaluați comportamentul, cu ajutorul fișei specifice, existentă la pagina următoare.

☑ Fișă de observare sistematică a activității și a comportamentului elevului*

Nume și prenume: _____ Data: _____
Tema lecției: _____
Clasa: _____

Scală de evaluare:

1 = deloc	2 = puțin	3 = moderat	4 = bine	5 = foarte bine

Autoevaluare

Activitate / Criteriu	Elev	Profesor
Am fost atent la lecție	☐1 ☐2 ☐3 ☐4 ☐5	☐1 ☐2 ☐3 ☐4 ☐5
Am înțeles conceptele predate	☐1 ☐2 ☐3 ☐4 ☐5	☐1 ☐2 ☐3 ☐4 ☐5
Am răspuns la întrebări	☐1 ☐2 ☐3 ☐4 ☐5	☐1 ☐2 ☐3 ☐4 ☐5
M-am implicat activ	☐1 ☐2 ☐3 ☐4 ☐5	☐1 ☐2 ☐3 ☐4 ☐5
Am cerut ajutor când nu am înțeles	☐1 ☐2 ☐3 ☐4 ☐5	☐1 ☐2 ☐3 ☐4 ☐5
Am înțeles sarcinile de lucru	☐1 ☐2 ☐3 ☐4 ☐5	☐1 ☐2 ☐3 ☐4 ☐5
Am rezolvat exercițiile / problemele	☐1 ☐2 ☐3 ☐4 ☐5	☐1 ☐2 ☐3 ☐4 ☐5
Am lucrat eficient în echipă	☐1 ☐2 ☐3 ☐4 ☐5	☐1 ☐2 ☐3 ☐4 ☐5
Am ajutat colegii	☐1 ☐2 ☐3 ☐4 ☐5	☐1 ☐2 ☐3 ☐4 ☐5
Am colaborat constructiv	☐1 ☐2 ☐3 ☐4 ☐5	☐1 ☐2 ☐3 ☐4 ☐5
Am scris / testat codul corect	☐1 ☐2 ☐3 ☐4 ☐5	☐1 ☐2 ☐3 ☐4 ☐5
Am depanat erorile independent	☐1 ☐2 ☐3 ☐4 ☐5	☐1 ☐2 ☐3 ☐4 ☐5



Reflecție

✅ Ce am făcut bine: _____

_____

🔧 Ce pot îmbunătăți: _____

_____

💡 O idee sau curiozitate din lecție: _____

_____

*Această fișă te ajută să îți dai seama cât de bine ai înțeles lecția și cât de implicat ai fost. Nu este un test, ci o evaluare personală care te ajută să progresezi. Fișa se completează de elev și profesor, iar diferențele de opinie vor fi discutate pentru a identifica direcțiile necesare pentru progres.

Evaluare inițială

Creați dosarul numit *evaluare_initiala*. Rezolvați exercițiile 1, 2, 3, 4, 5 și 6 de mai jos, notând răspunsurile într-un fișier: *raspunsuri.docx*.

Pentru cerințele 1, 2, 3, 4 și 5 alegeți răspunsul corect (un singur răspuns) **(20 puncte)**

1. Algoritmul este:
a) un joc; **b)** o imagine; **c)** o succesiune finită de pași; **d)** un calculator.

2. Care este ordinea de efectuare a operațiilor în expresia: ***x+2*(z-y)***:
a) +,*, -; **b)** -,*,+; **c)** -,+,*; **d)** *,-,+.

3. Care dintre următoarele variante reprezintă o secvență corectă prin care se interschimbă valorile a două variabile a și b, folosind o variabilă auxiliară ***aux***?
a) *aux=a; a=b; b=aux*; **b)** *a=b; b=aux; aux=a*;
c) *a=aux; aux=b; b=a*; **d)** *b=a; aux=b; a=b*.

4. Care va fi valoarea variabilei **x** după secvența de atribuiri alăturată?
a) 3; **b)** 5; **c)** 2; **d)** 1.

```
y←2
x←y+1
x←x+y
```

5. În secvența alăturată există:
a) 2 instrucțiuni de atribuire; **b)** 3 variabile; **c)** 2 variabile; **d)** 1 instrucțiune de atribuire.

6. Analizați algoritmul următor și răspundeți la cerințe: **(20 puncte)**

```
citește a, b
s ← a + b
scrie s
```

a) Datele de intrare sunt: ....., iar data de ieșire este ..... .
b) Valoarea afișată pentru a = 7 și b = 5 este ..... .
c) Scrieți pe scurt care ar putea fi enunțul problemei rezolvate de acest algoritm.

7. Scrieți un program care citește de la tastatură două numere întregi ***a*** și ***b*** și calculează:
a) minimul dintre ele; **b)** cifra unităților produsului celor 2 numere. **(20 puncte)**

8. Scrieți un program care citește de la tastatură un număr natural nenul ***n*** și afișează, în ordine descrescătoare, primele ***n*** numere naturale nenule. **(15 puncte)**

9. Scrieți un program care citește de la tastatură un număr natural ***n*** și apoi determină și afișează numărul său de cifre impare. **(15 puncte)**

Barem de corectare: Se acordă ***4 puncte*** pentru fiecare răspuns corect la exercițiile **1**-**5** și ***10 puncte*** din oficiu!
6. a) (3 puncte), (2 puncte), **b)** (5 puncte), **c)** pentru enunț corect – **10 puncte**
7. declarații variabile (**2p**), citire date (**3p**), **a)** determinare: minim (**5p**), **b)** produs (**2p**), ultima cifră (**3p**), afișare rezultat (**3p**), corectitudine sintactică (**2p**).
8. declarații variabile (**2p**), citire date (**3p**), parcurgere primele ***n*** numere naturale nenule (**5p**), ordine descrescătoare (**3p**), afișare rezultat (**2p**).
9. declarații variabile (**2p**), citire date (**3p**), determinarea unei cifre (**3p**), verificarea imparității unei cifre (**2p**), numărul de cifre impare (**3p**), afișare rezultat (**2p**).

Capitolul 1 – Strategii de rezolvare a problemelor. Fundamentele programării

Lecția 1 – Etapele de elaborare a unui program: de la analiză și proiectare la testare și depanare

*„Nu mă tem de calculatoare. Mă tem de lipsa lor.” – **Isaac Asimov***

Informatica. Datele și algoritmul

<def/> **Informatica** este știința care studiază prelucrarea automată a datelor cu ajutorul calculatoarelor. Ea se ocupă de dezvoltarea algoritmilor, programelor și sistemelor care stochează, procesează și transmit date.

OBS *Gândirea computațională* este abilitatea omului prin care acesta dezvoltă metode de rezolvare a problemelor prin secvențe de instrucțiuni și ***algoritmi***.

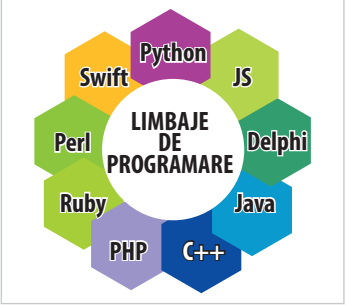
Principalele **caracteristici ale gândirii computaționale** sunt: ***descompunerea*** (problemele complexe sunt divizate în subprobleme, care sunt mai simple și mai ușor de rezolvat), ***generalizarea*** (sunt identificate modele sau soluții similare, pentru a putea rezolva rapid problemele noi folosind soluțiile deja cunoscute), ***abstractizarea*** (sunt reduse detaliile inutile), ***algoritmizarea*** (sunt elaborați pașii pentru rezolvarea problemei) și ***evaluarea*** (se verifică dacă soluția propusă este corectă și eficientă).

<def/> **Algoritmul** este o succesiune finită de pași executați într-o ordine bine definită, care prelucrează *date*, acestea identificându-se prin ***nume***, ***tip*** și ***valoare***. Algoritmii pot rezolva probleme diverse din domeniul matematicii, al fizicii și al chimiei, dar și din viața reală.

OBS **Datele** se clasifică în **funcție de tip** (*date elementare* sau *date structurate*), **în funcție de momentul utilizării lor** (*date de intrare*, *intermediare/de manevră* și de *ieșire*) și în **funcție de valoare** (*date variabile* și *date constante*).

<def/> **Datele** folosite de calculator sunt reprezentate în format digital (cod binar – 0 și 1), deoarece mașinile nu pot interpreta direct limbajul uman. Calculatorul operează astfel cu un limbaj cod-mașină.

OBS Limbajele de programare au evoluat de la limbajul de asamblare la cele de nivel înalt și orientate pe obiecte, ajungând astăzi la limbaje utilizate în inteligența artificială. Cu ajutorul lor, algoritmii pot fi implementați în programe care rezolvă eficient o clasă de probleme.



Etapele de elaborare a unui program

Pentru a realiza un program, care implementează un algoritm este necesară parcurgerea etapelor:

- **Analiza problemei:** presupune identificarea datelor de intrare, de ieșire și a relațiilor dintre ele.
- **Elaborarea modului de rezolvare a problemei sau proiectarea modulară:** se descompune problema în subprobleme și se alege metoda de rezolvare, descriindu-se succesiunea pașilor.
- **Implementarea algoritmului de rezolvare al problemei într-un limbaj de programare:** se realizează respectând sintaxa acestuia (*Exemplu:* Python).
- **Testarea și depanarea programului:** presupune rularea programului pentru a identifica și corecta erorile de sintaxă sau de logică.
- **Analiza complexității și optimizarea programului:** presupun evaluarea eficienței algoritmului din punctul de vedere al memoriei folosite și al timpului de execuție.

Repere pentru aplicarea acestor etape în rezolvarea problemelor

1. Analiza
- identificarea datelor de intrare și de ieșire;
- înțelegerea cerințelor problemei;
- stabilirea restricțiilor și a condițiilor impuse.

2. Proiectarea
- alegerea structurilor de date potrivite;
- elaborarea algoritmului (schiță în pseudocod sau diagramă);
- descompunerea problemei în subprobleme (proiectare modulară).

3. Implementarea
- transpunerea algoritmului într-un limbaj de programare;
- respectarea sintaxei limbajului;
- organizarea clară și lizibilă a codului.

4. Testarea
- verificarea programului cu date de intrare variate (cazuri normale, limită, eronate);
- compararea rezultatelor obținute cu cele așteptate;
- identificarea erorilor de sintaxă, logică sau execuție.

5. Depanarea
- localizarea și corectarea erorilor identificate în etapa de testare;
- urmărirea evoluției valorilor variabilelor;
- reverificarea programului după fiecare corecție.

Criterii de elaborare a testelor

– Pentru **validarea datelor de intrare**:
- verificarea dacă datele introduse respectă tipul așteptat (număr întreg, șir de caractere etc.);
- verificarea dacă valorile se încadrează în intervalul permis (*Exemplu:* nota între 1 și 10);
- testarea cu date limită (valoarea minimă, maximă, zero);
- testarea cu date invalide (valori negative, șiruri goale, caractere neașteptate).

– Pentru **verificarea comportamentului algoritmului**:
- testarea cu date obișnuite (cazul general);
- testarea cazurilor speciale (listă vidă, un singur element, valori egale);
- compararea rezultatelor obținute cu cele așteptate;
- verificarea că programul produce același rezultat la rulări repetate cu aceleași date.

Repere pentru urmărirea evoluției valorilor variabilelor

- utilizarea instrucțiunilor de afișare pentru vizualizarea valorilor variabilelor în puncte-cheie ale programului;
- urmărirea pas cu pas a execuției algoritmului (trasarea manuală pe hârtie sau cu debugger);
- identificarea tipului de eroare:
  - eroare de **sintaxă** – cod incorect din punct de vedere al regulilor limbajului;
  - eroare de **logică** – programul rulează, dar produce rezultate greșite;
  - eroare de **execuție** – programul se oprește în timpul rulării (ex: împărțire la zero).
- corectarea erorii și retestarea programului.

EX. Pentru a ilustra **etapele de elaborare a unui program**, vom analiza următoarea problemă: *Mihai are un număr de **n** note la disciplina Informatică. Trebuie să determinați media notelor elevului și să verificați dacă a promovat sau nu la disciplina Informatică.*

ANALIZA →
- IDENTIFIC DATELE DE INTRARE: NUMĂRUL NATURAL N CARE REPREZINTĂ NUMĂRUL DE NOTE ȘI FIECARE NOTĂ ÎN PARTE
- IDENTIFIC DATELE DE IEȘIRE: MEDIA NOTELOR CARE VA FI DE TIP REAL
- IDENTIFIC RELAȚIA DINTRE DATE: FOLOSIM O DATĂ DE MANEVRĂ, ÎN CARE DETERMINĂM SUMA NOTELOR

PROIECTAREA
- SE VA REALIZA MODULAR
- CITIM NUMĂRUL DE NOTE CARE VA FI TESTAT DACĂ ESTE POZITIV ȘI NU DEPĂȘEȘTE O ANUMITĂ LIMITĂ
- CITIM NOTELE CARE TREBUIE SĂ FIE CUPRINSE ÎNTRE 1 ȘI 10
- CALCULĂM SUMA NOTELOR ȘI APOI MEDIA NOTELOR PRIN ÎMPĂRȚIREA SUMEI LA NUMĂRUL DE NOTE, AVÂND GRIJĂ SĂ FACEM CONVERSIA SUMEI ÎN DATĂ DE TIP REAL
-VERIFICĂM DACĂ MEDIA RESPECTĂ CRITERIUL DE PROMOVABILITATE (≥5) ȘI AFIȘĂM PROMOVAT/ NEPROMOVAT

IMPLEMENTAREA
- TRANSCRIEM ALGORITMUL ÎN LIMBAJUL DE PROGRAMARE STUDIAT (PYTHON, C/C++)

TESTAREA ȘI DEPANAREA
- VERIFICĂM PENTRU DIVERSE DATE DE INTRARE OBȚINEREA REZULTATULUI CORECT ȘI VOM CORECTA EVENTUALELE ERORI DE SINTAXĂ SAU DE LOGICĂ
(EX: DACĂ N=3 ȘI CITIM NOTA1=8, NOTA2=9, NOTA3=10 SE OBȚINE MEDIA=9 ȘI SE VA AFIȘA CALIFICATIVUL PROMOVAT; DACĂ N=2 ȘI CITIM NOTA1=4, NOTA2=3 SE OBȚINE MEDIA 3,5 ȘI SE VA AFIȘA CALIFICATIVUL NEPROMOVAT)

OPTIMIZAREA
- ALGORITMUL ESTE EFICIENT DIN PUNCTUL DE VEDERE AL MEMORIEI O(1) ȘI AL TIMPULUI DE EXECUȚIE (O(N))

Fișa de lucru 1

Creați dosarul numit *Fisa1*. Rezolvați exercițiile următoare, notând răspunsurile corecte într-un fișier *raspunsuri.docx*.

1. *Stabiliți valoarea de adevăr a propozițiilor:*

a. Principalele caracteristici ale gândirii computaționale pot fi: descompunerea, generalizarea, abstractizarea, algoritmizarea, evaluarea;

b. Algoritmul este o succesiune infinită de pași prin care se pot rezolva probleme specifice;

c. Datele de ieșire sunt cele care se folosesc pe parcursul programului;

d. În funcție de valoarea lor, datele pot fi variabile sau constante.

2. *Ordonați pașii de mai jos, care constituie algoritmul prin care se pot interschimba conținuturile a două pahare, unul cu suc și altul cu apă.*

Umple paharul albastru cu suc. ☐
Ia un pahar gol de culoare mov. ☐
Toarnă sucul din paharul albastru în paharul mov. ☐
Umple paharul galben cu apă. ☐
Toarnă sucul din paharul mov în paharul galben. ☐
Toarnă apa din paharul galben în paharul albastru. ☐



Autoevaluare

Completați răspunsurile într-un fișier *FisaAutoevaluare1.docx*, folosind notații precum *1 – da*, *2 – nu*, *3 – nu sunt sigur*. Salvați fișierul în dosarul *Fisa1*.

	La finalul lecției, știu...	*Da*	*Nu*	*Nu sunt sigur(ă)*
1.	ce este un algoritm.			
2.	ce sunt datele.			
3.	care sunt etapele de elaborare ale unui program.			
4.	care este ordinea corectă de executare a pașilor unui algoritm.			

5. Am întâmpinat dificultăți în: …

Nu uitați să vă evaluați comportamentul și activitatea din timpul lecției!
(cu ajutorul fișei de observare a comportamentului și a activității, de la pagina 7)

La finalul lecției, m-am simțit:

Lecția 2 – Reprezentarea algoritmilor: scheme logice, pseudocod, limbaje de programare

1. Scheme logice

<def/> **Schema logică** este o metodă de descriere a algoritmilor prin reprezentare grafică.

Structurile de control ale algoritmului sunt descrise prin ***blocuri grafice***, ele reprezentând pașii algoritmului, iar ordinea lor de aplicare (succesiunea pașilor) este marcată prin săgeți. Blocurile folosite pentru realizarea schemelor logice sunt:



Să ne reamintim!

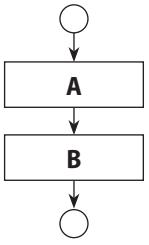
✓ **Algoritmul** prelucrează ***date de intrare*** și returnează ***date de ieșire***. Uneori în vederea prelucrării pot fi folosite ***date de manevră***.

✓ Principiile programării structurate au la bază conceptele: ***modularizarea*** și ***structurarea datelor***, acesta din urmă bazându-se pe trei structuri de bază: ***liniară***, ***alternativă*** și ***repetitivă***.

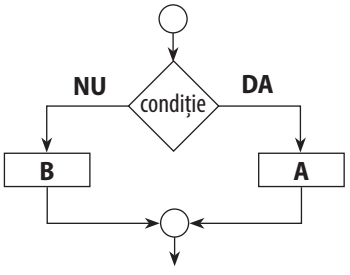
✓ Există diferite metode de reprezentare a algoritmilor: cu ajutorul ***schemelor logice***, ***limbajului pseudocod*** sau cu ajutorul ***limbajelor de programare***.

→	*Linie de flux:* realizează legătura între blocuri, indicând ordinea de executare a pașilor
▭	*Bloc de calcul:* este folosit pentru efectuarea diverselor operații și atribuiri
▯▯	*Bloc de subprogram:* reprezintă un apel al subprogramului care are o schemă logică separată
▱	*Bloc de intrare/ieșire:* reprezintă operația de citire/scriere a datelor
◇	*Bloc de decizie:* reprezintă o condiție care are ca rezultat două ieșiri (Da/Nu sau Adevărat/Fals)
⬭	*Bloc terminal:* reprezintă momentul de început (*START*) sau de final (*STOP*).
○	*Bloc conector:* este folosit pentru a conecta două sau mai multe puncte ale schemei

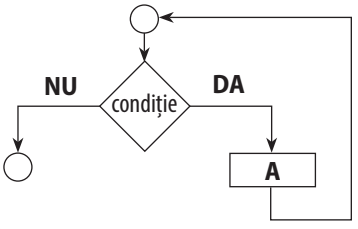
<def/> Orice algoritm poate fi descris printr-o schemă logică folosind următoarele **structuri de control**:



structura liniară



structura alternativă

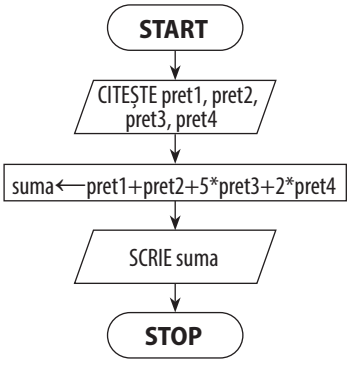


structura repetitivă

Structura liniară

<def/> **Structura liniară** presupune parcurgerea instrucțiunilor unui algoritm în ordinea specificării lor. Nu conține ramificații sau repetiții, fiecare executându-se o singură dată.

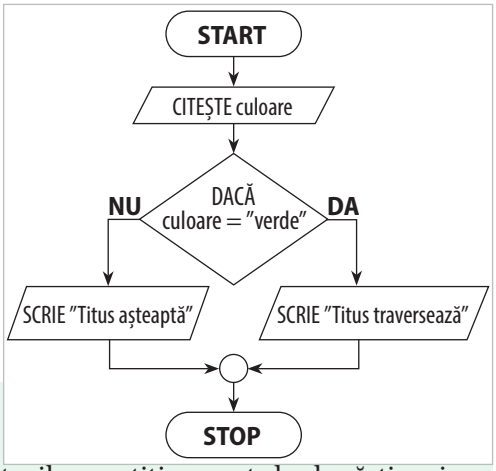
EX. Ilinca vrea să își cumpere pentru școală un ghiozdan, un penar, cinci creioane grafice și două stilouri. Cunoscându-se prețul fiecărui produs, determinați de ce sumă are nevoie pentru a-și achiziționa ce și-a propus.



Structura alternativă

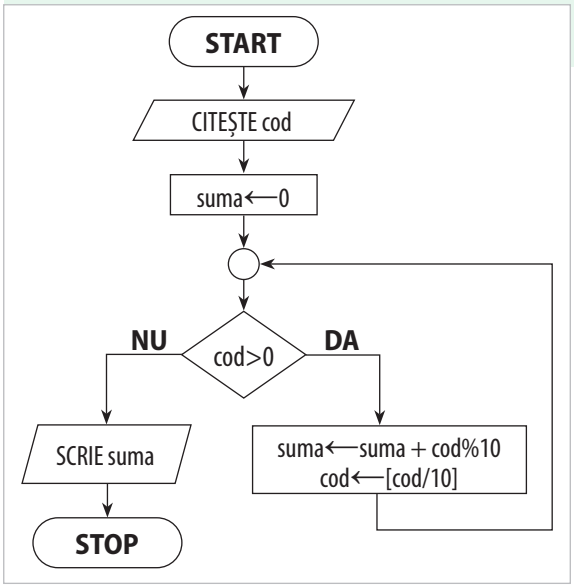
<def/> **Structura alternativă** permite selectarea instrucțiunii care va fi executată în funcție de o condiție dată.

EX. Titus este la semafor și dorește să traverseze trecerea de pietoni. În funcție de culoarea semaforului, el va decide ce va face. Dacă semaforul arată culoarea roșie, Titus așteaptă, iar dacă semaforul arată verde, băiatul poate traversa. Ajutați-l să urmeze pașii corecți!



Structura repetitivă

<def/> **Structura repetitivă** permite executarea repetată a unor instrucțiuni, execuție controlată de valoarea de adevăr a unei expresii. Structurile repetitive sunt de două tipuri: *cu număr necunoscut de pași* și *cu număr cunoscut de pași*.



OBS - Din categoria structurilor repetitive *cu număr necunoscut de pași* fac parte structura repetitivă cu **testare inițială** (*condiționată anterior*) și cea cu **testare finală** (*condiționată posterior*).

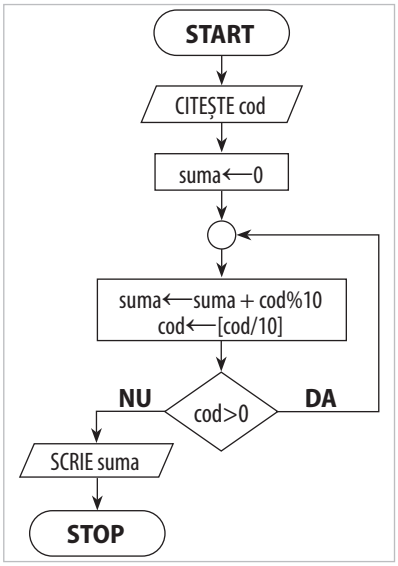
- În structura repetitivă cu **testare inițială** condiția de executare a ciclului repetitiv se pune la *început*, iar în cazul structurii repetitive cu **testare finală** condiția se pune la *finalul* ciclului repetitiv.

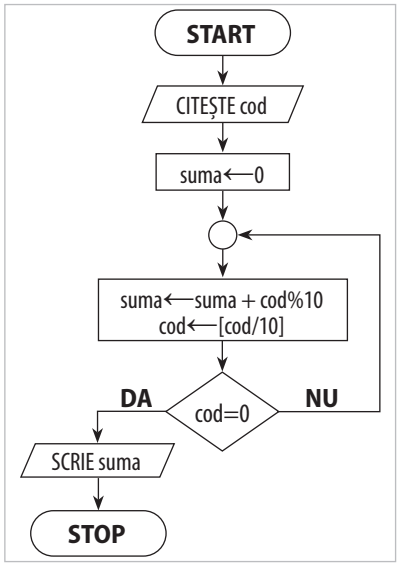
- Diferența între structura repetitivă cu testare inițială și cea cu testare finală este aceea că, în cazul celei de-a doua instrucțiunile din interiorul structurii se execută *cel puțin o dată*, iar în cazul *primeia*, instrucțiunile se execută doar dacă este *adevărată* condiția.

EX. Miruna are o casetă care este închisă cu un lacăt ce are un cod format din 9 cifre. Pentru a deschide caseta, Miruna trebuie să calculeze și să introducă suma cifrelor codului. Ajutați-o pe Miruna să urmeze pașii necesari pentru a deschide caseta.

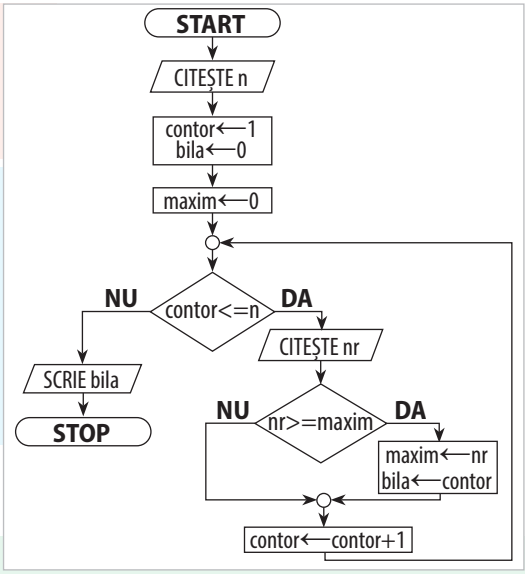
OBS Structura repetitivă cu **testare finală** are două forme, în funcție de îndeplinirea sau neîndeplinirea condiției, acestea sunt ilustrate în schemele logice alăturate.

OBS Structura repetitivă **cu număr cunoscut de pași** are condiția la începutul structurii și se folosește atunci când se cunoaște de câte ori trebuie repetate instrucțiunile.





EX. Oana are ***n*** cutii. În fiecare cutie este ascunsă o bilă, pe care e scris un număr natural. Scrieți un algoritm pentru a o ajuta pe Oana să afle în care cutie se află bila cu numărul cel mai mare.



OBS - Orice structură repetitivă cu număr cunoscut de pași poate fi transformată într-o structură repetitivă cu număr necunoscut de pași. ***Reciproca nu este mereu adevărată!***

- Reprezentarea algoritmilor cu ajutorul schemelor logice constituie ***un avantaj*** deoarece oferă o vizualizare clară a pașilor și ajută la identificarea erorilor. Această reprezentare poate fi ***un dezavantaj*** pentru algoritmii de dimensiuni mari.

2. Limbajul pseudocod

<def/> Limbajul ***pseudocod*** (din gr. *pseudos* = *fals*) este folosit pentru proiectarea algoritmilor, fiind format dintr-un set de reguli de scriere și o sintaxă specifică instrucțiunilor care îl compun. Instrucțiunile se execută în ordinea în care sunt scrise.

<def/> Orice algoritm poate fi descris în limbajul pseudocod, folosind următoarele **structuri de control**:

```
tip_ date identificatori
citire date
instrucțiune 1
instrucțiune 2
.....
afișare date
```

structura secvențială/liniară

```
dacă <condiție> atunci
    <instrucțiuni 1>
[altfel <instrucțiuni 2>]
sfârșit dacă
```

structura alternativă

```
cât timp <condiție> execută
    <instrucțiuni>
sfârșit cât timp
```

structura repetitivă

Structura liniară

EX. Mihai se plimbă pe o stradă pe care se află 8 arbori. Pe fiecare arbore se află o plăcuță pe care sunt notate specia acestuia și înălțimea.

Pentru că este un băiat foarte curios, Mihai dorește să afle înălțimea medie a arborilor. Ajutați-l să rezolve această problemă, realizând în limbajul pseudocod algoritmul de rezolvare.

```
algoritm înălțime_ medie
real h1, h2, h3, h4, h5, h6, h7, h8, medie
citește h1, h2, h3, h4, h5, h6, h7, h8
medie ← (h1+h2+h3+h4+h5+h6+h7+h8) / 8
scrie medie
sfârșit algoritm
```

Structura alternativă

EX. Într-o zi, Ania se plimba pe stradă. Ea s-a întâlnit cu două fetițe, care aveau vârste diferite. Curioase care dintre ele este mai mare, Ania le-a întrebat pe fetițe ce vârstă au și le-a spus apoi care dintre ele este mai mare. Ajutați-o pe Ania să ilustreze în pseudocod pașii care au dus la rezolvarea misterului.

```
algoritm ani
natural x, y
citește x, y
dacă x > y atunci scrie "Prima fetiță este mai mare decât a doua."
altfel scrie "A doua fetiță este mai mare decât prima."
sfârșit dacă
sfârșit algoritm
```

Structura repetitivă

Exemplu de algoritm care utilizează structura repetitivă cu număr necunoscut de pași, cu testare inițială: Ana extrage bilețele cu numere naturale dintr-un săculeț, până când întâlnește numărul 0. Ajutați-o pe Ana să scrie pașii algoritmului prin care poate determina suma numerelor pare din șirul numerelor extrase.

```
algoritm numere_pare
natural x, s ← 0
citește x
cât timp x > 0 execută
    dacă x % 2 = 0 atunci s ← s + x
    sfârșit dacă
    citește x
sfârșit cât timp
scrie s
sfârșit algoritm
```

Structura repetitivă cu **testare finală**, în care condiția se pune la finalul ciclului repetitiv, are două forme de reprezentare. Cele două forme se diferențiază astfel: în ***prima***, instrucțiunile se repetă ***cât timp condiția este adevărată***, iar în ***a doua***, instrucțiunile se repetă ***până când condiția devine adevărată***.

```
algoritm numere_pare
natural x, s ← 0
repetă
    citește x
    dacă x % 2 = 0 atunci s ← s + x
    sfârșit dacă
până când x = 0
scrie s
sfârșit algoritm
```

```
algoritm numere_pare
natural x, s ← 0
execută
    citește x
    dacă x % 2 = 0 atunci s ← s + x
    sfârșit dacă
cât timp x > 0
scrie s
sfârșit algoritm
```

Exemplu de algoritm care utilizează structura repetitivă cu număr cunoscut de pași: Freya este o fetiță jucăușă, dar și foarte inteligentă, căreia îi plac numerele. Ea se joacă cu fratele ei mai mic Elias și dorește să îl învețe să adune numere cu două cifre. Astfel, îi arată ***n*** carduri, pe care sunt scrise numere naturale și îl provoacă să le aleagă pe cele care au exact două cifre și să determine suma lor. Algoritmul de rezolvare este ilustrat în imaginea alăturată.

```
algoritm suma_numere
natural n, i, s ← 0, nr
citește n
pentru i ← 1, n execută
    citește nr
    dacă nr >= 10 și nr <= 99 atunci
        s ← s + nr
    sfârșit dacă
sfârșit pentru
scrie s
sfârșit algoritm
```

Reprezentarea algoritmilor cu ajutorul limbajului pseudocod poate fi ***un avantaj*** deoarece este ușor de înțeles, intuitiv și independent de limbajele de programare. Un ***dezavantaj*** îl reprezintă însă imposibilitatea testării directe a algoritmului, acesta necesitând ulterior transpunerea într-un limbaj de programare sau testare manuală.

3. Limbaje de programare

Limbajul de programare este un mijloc de comunicare între programator și calculator. El este alcătuit din **vocabular** și un **set de reguli de scriere** a instrucțiunilor. Cu ajutorul limbajului de programare pot fi implementați algoritmi sub forma unui **program**.

Pentru a scrie un algoritm în limbajul C/C++ se poate folosi un mediu de dezvoltare software (CodeBlocks/MinGW), care permite ***editarea*** codului sursă, ***compilarea***, ***depanarea*** și ***testarea***.

Vocabularul limbajului C/C++ este alcătuit din: *alfabetul limbajului* (litere mari și mici ale alfabetului, cifre și caractere speciale), *identificatori*, *cuvinte-cheie*, *separatori* și *comentarii*.

Datele cu care lucrează un algoritm se identifică prin: *nume*, *tip* și *valoare*. Un **tip de date** definește mulțimea valorilor pe care le poate lua data respectivă și modul de reprezentare în memorie. Există tipuri de date: *elementare* și *structurate*. Tipuri de date elementare sunt: *numerice întregi* (int, short, long, unsigned int, unsigned long, long long), *numerice reale* (float, double), *caracter* (char, unsigned char), *logice* (bool).

OBS În funcție de valoarea pe care o au datele există ***date variabile*** (care își modifică valoarea pe parcursul programului) și ***date constante*** (care nu își modifică valoarea).

<def/> **Expresia** este alcătuită din unul sau mai mulți **operanzi** separați între ei prin **operatori**.

Expresiile pot fi **aritmetice** sau **logice** în funcție de tipul operatorilor folosiți.



Să ne reamintim! Tipuri de operatori!

✓ Aritmetici: unari (+, -), binari (+,-, *, /, %)
✓ Relaționali: de inegalitate (<, <=, >, >=), de egalitate (==, !=)
✓ Incrementare/Decrementare: ++, --
✓ Logici: !, &&, ||
✓ De atribuire: simplă (=), compusă (+=, -=, *=, /=, %=)
✓ De conversie explicită: (tip) operand
✓ Condițional: e ? expresie1: expresie2

Structurile de control implementate în limbajul de programare C/C++ sunt:

Structura secvențială	Structura alternativă	Structura repetitivă cu testare inițială
int nr1, nr2, s; cin>>nr1>>nr2; s=nr1+nr2; cout<<s;	**if** (condiție) instrucțiune1; **else** instrucțiune2;	**while**(condiție) instrucțiune;
Exemplu: Anais a primit trei note la disciplina Biologie. *Ajutați-o pe Anais să își calculeze media.*	*Exemplu: Ianis scrie pe o foaie un număr. În funcție de valoarea acestuia, prietenul său Andrei va spune dacă este par sau impar. Ajutați-l pe Ianis să realizeze un program prin care să se afișeze rezultatul dorit.*	*Exemplu: Tania primește un bilețel pe care este scris un număr natural, cu mai multe cifre și sarcina de lucru: "Determină câte cifre impare are numărul!". Scrieți un program care să determine acest lucru, iar dacă numărul nu conține nicio cifră impară să se afișeze mesajul "Nu există".*
#include <iostream> **using namespace** std; **int** main() { **unsigned int** nt1, nt2, nt3; **double** media; cin>>nt1>>nt2>>nt3; media=(nt1+nt2+nt3)/3.0; cout<<media; **return** 0; }	*#include <iostream>* **using namespace** std; **int** main() { **unsigned int** nr; cin>>nr; **if**(nr%2==0) cout<<"Nr. este par"; **else** cout<<"Nr. este impar"; **return** 0; }	*#include <iostream>* **using namespace** std; **int** main() { **unsigned int** nr, k=0; cin>>nr; **while**(nr!=0) { **if**(nr%2==1) k++; nr/=10; } **if**(k!=0) cout<<k; **else** cout<<"Nu există"; **return** 0; }

Structura repetitivă cu testare finală	Structura repetitivă cu număr cunoscut de pași
do instrucțiune; **while**(condiție);	**for(inițializare; condiție; continuare)** instrucțiune;
Structura repetitivă cu **testare finală**, în care condiția se pune la finalul ciclului repetitiv are o singură formă în limbajul C/C++.	*Exemplu: Profesorul de sport vrea să vadă care este cel mai înalt copil din clasă și câți copii sunt la fel de înalți ca și el. Scrieți un program care realizează aceste operații.*

```
#include <iostream>
using namespace std;
int main()
{
    unsigned int nr, k=0; cin>>nr;
    do
    {
        if(nr%2==1)
            k++;
        nr/=10;
    }while(nr!=0);
    if(k!=0)
        cout<<k;
    else
        cout<<"Nu exista";
    return 0;
}
```

```
#include <iostream>
using namespace std;
int main()
{
    unsigned int n, h, hmax=0, contor=0; cin>>n;
    for(int i=1;i<=n;i++)
    {
        cin>>h;
        if(h>hmax)
        {
            hmax=h;
            contor=1;
        }
        else
            if(h==hmax)
                contor++;
    }
        cout<<hmax<<'    '<<contor;
        return 0;
}
```

Exemplele în C/C++ din această lecție au rol informativ, pentru elevii care au studiat acest limbaj în gimnaziu; în continuare, manualul folosește limbajul Python.

OBS Reprezentarea algoritmilor cu ajutorul limbajului de programare prezintă ***avantaje*** precum: precizie, posibilitatea executării algoritmului, performanță și eficiență, reutilizarea codului, dar poate constitui și ***dezavantaje*** deoarece limbajele de programare folosesc o sintaxă strictă, care trebuie respectată cu exactitate.

Fișa de lucru 2

Creați dosarul numit *Fisa2*. Rezolvați exercițiile următoare, notând răspunsurile corecte într-un fișier *raspunsuri.docx*.

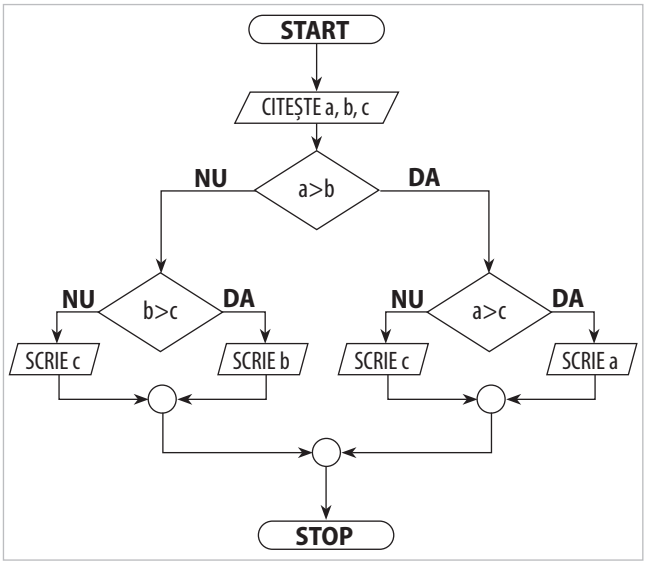
1. *Parcurgeți algoritmul reprezentat prin schema logică alăturată și rezolvați următoarele cerințe:*

a. Stabiliți ce se va afișa dacă se introduc valorile 175, 100, 23. Dar dacă se introduc valorile 2, 2, 0?

b. Descrieți ce face algoritmul: ..... .

c. Transformați schema logică în limbajul pseudocod.

d. Rescrieți algoritmul în limbajul de programare studiat în gimnaziu.



2. ***Stabiliți valoarea de adevăr a următoarelor propoziții:*** **a.** Structura repetitivă cu număr cunoscut de pași poate fi transformată într-o structură repetitivă cu număr necunoscut de pași. Reciproca este mereu adevărată! **b.** Structura liniară permite executarea instrucțiunilor în ordinea în care au fost scrise. **c.** Structura repetitivă condiționată posterior are 2 modalități de scriere atât în pseudocod, cât și în limbajul de programare C/C++. **d.** Structurile de control sunt: secvențială, alternativă și repetitivă.

3. ***Stabiliți ordinea corectă a următoarelor instrucțiuni din cadrul algoritmului prin care se va determina cel mai mare divizor comun a două numere întregi.***

sfârșit algoritm
r ← a % b
citește a, b
întregi a, b, r
algoritm cmmdc
a ← b
r ← a % b
cât timp r!=0 execută
b ← r
scrie b
sfârșit cât timp

4. ***Parcurgeți algoritmul reprezentat în limbajul pseudocod din imaginea alăturată și rezolvați următoarele cerințe:*** **a.** Stabiliți ce se va afișa dacă se introduc în ordine valorile 5, 13, 2, 0, 28, 29. **b.** Știind că valoarea lui n este 6 scrieți un șir de numere distincte care pot fi citite în continuare astfel încât valoarea afișată să fie 0. **c.** Scrieți în pseudocod un algoritm echivalent cu cel dat în care să rescrieți structura **pentru... execută** cu o structura repetitivă de tipul **cât timp... execută**. **d.** Scrieți, în limbajul de programare studiat în gimnaziu, programul corespunzător algoritmului dat.

```
algoritm problemă
întreg n, nr, k, ok, i, d
citește n
k ← 0
pentru i ← 1, n execută
    citește nr
    ok ← 1
    dacă nr < 2 atunci ok ← 0
    sfârșit dacă
    dacă nr % 2 = 0 && nr<>2 atunci ok ← 0
    sfârșit dacă
    pentru d ← 3, [nr/2], 2 execută
        dacă nr % d = 0 atunci ok ← 0
        sfârșit dacă
    sfârșit pentru
    dacă ok = 1 atunci k ← k + 1
    sfârșit dacă
sfârșit pentru
scrie k
sfârșit algoritm
```

5. ***Scrieți în pseudocod și în limbajul de programare studiat în gimnaziu algoritmul de rezolvare pentru următoarea problemă:*** Se citește un șir de ***n*** numere naturale. Să se afișeze pentru fiecare număr citit câți divizori naturali are.

Autoevaluare

Completați răspunsurile într-un fișier *FisaAutoevaluare2.docx*, folosind notații precum *1 – da*, *2 – nu*, *3 – nu sunt sigur*. Salvați fișierul în dosarul *Fisa2*.

	La finalul lecției, știu...	***Da***	***Nu***	***Nu sunt sigur(ă)***
1.	să identific structurile de control necesare pentru rezolvarea unui algoritm.			
2.	să reprezint un algoritm cu ajutorul schemei logice.			
3.	să reprezint un algoritm cu ajutorul limbajului pseudocod.			
4.	să reprezint un algoritm cu ajutorul limbajului de programare studiat în gimnaziu.			
5.	să identific avantajele/ dezavantajele diverselor metode de reprezentare ale algoritmilor.			

6. Am întâmpinat dificultăți în: …

Nu uitați să vă evaluați comportamentul și activitatea din timpul lecției!
(cu ajutorul fișei de observare a comportamentului și a activității, de la pagina 7)

La finalul lecției, m-am simțit:

Lecția 3 – Arhitectura unui program Python. Variabile, tipuri de date și expresii

Arhitectura unui program Python

Limbajul Python este un limbaj interpretat, în care instrucțiunile sunt executate pe rând, de către interpretor.

Spre deosebire de limbajele compilate, programele Python nu necesită o etapă de compilare separată înainte de execuție.

- Pentru a putea lucra în **Python** trebuie instalat mediul de programare de la adresa: https://www.python.org/downloads/.

- Pe lângă **IDLE** (**I**ntegrated **D**evelopment and **L**earning **E**nvironment), pot fi utilizate și alte medii de dezvoltare:
  - *PyCharm* – editor modern pentru dezvoltarea programelor Python;
  - *Google Colab* – mediu online care permite scrierea și executarea programelor direct în browser, fără instalare.

Știați că?
✓ Programele scrise în Python sunt portabile
✓ Python este un limbaj ușor de învățat și de folosit, fiind gratuit și orientat pe obiecte
✓ Limbajul Python a fost creat în anul 1989 de Guido van Rossum

- Odată instalat, pentru a porni **IDLE** apăsați butonul Start din Windows și căutați **IDLE**. Se observă că se lansează consola (Interactive Python Shell), în care pot fi scrise liniile de comandă imediat după caracterele „>>>”. Pentru a se afișa un mesaj se folosește funcția **print( )**.

```
Python 3.14.4 (tags/v3.14.4:23116:
A11D64) J on win32
Enter "help" below or click "Help
>>> print ("Bine ati venit!")
Bine ati venit!
```

Pentru a crea un program nou se selectează din meniul **File – New File** (Ctrl+N). În fereastra deschisă se poate scrie codul Python.

```
print("Bine ati venit!")
print("Acesta este primul meu program!")
print("******************")
```

Pentru salvarea programului se utilizează comanda **File – Save As** (Ctrl+Shift+S). Extensia fișierelor create în Python este ***.py***. Executarea programului se realizează din meniul **Run – Run Module** (F5). Rezultatul este afișat în consola Python.

- Cel mai simplu program afișează un șir de caractere cu ajutorul funcției **print**. În Python, instrucțiunile pot fi separate prin caracterul **;**, însă acest lucru nu este obligatoriu. Pentru o mai bună lizibilitate, este recomandat ca fiecare instrucțiune să fie scrisă pe un rând separat. Limbajul Python face diferența între literele mari și mici (case-sensitive).

Vocabularul limbajului Python

Vocabularul limbajului Python este alcătuit la fel ca și în cazul altor limbaje de programare din:

– *Setul de caractere* utilizat pentru scrierea programelor (litere mari și mici ale alfabetului englez, cifrele de la 0..9, caractere speciale).

– *Identificatori* folosiți pentru denumirea variabilelor, constantelor, funcțiilor sau modulelor și pot conține litere, cifre sau „_”, dar nu pot începe cu o cifră.

– *Cuvinte cheie* (*reserved words*) care au un înțeles special în limbaj (if, else, elif, for, True, while, False, continue, break, from, return, not, and, or)

– *Separatori* utilizați pentru delimitarea instrucțiunilor sau elementelor unui program, cum ar fi: spațiu, indentarea, parantezele, „:”, „ ,” sau „ ;”.

– *Comentarii* care sunt introduse cu ajutorul caracterului „#” și au rolul de a explica secvențe de cod. Dacă se dorește comentarea mai multor linii de cod, se vor folosi ghilimelele triple (""" … """).

În Python, indentarea este foarte importantă deoarece delimitează blocurile de instrucțiuni și face parte din sintaxă.

Pentru ***declararea*** unei variabile în Python, se alege un identificator (nume), apoi i se atribuie o valoare. În Python, tipul variabilei este stabilit automat în funcție de valoarea atribuită, astfel că nu este necesară declararea explicită a tipului de date.

```
variabila = 23
print(variabila)    # se afișează 23
variabila = -55
print(variabila)    # se afișează -55
variabila = "un șir"
print(variabila)    # se afișează un șir
```

- Funcția **input()** afișează un mesaj în consolă și citește date de la tastatură, returnând întotdeauna un șir de caractere (**string**). Rezultatul se atribuie unei variabile. Funcția **print()** afișează mesaje, valori ale variabilelor sau combinații ale acestora. Șirurile se concatenează cu operatorul **+**.

```
nume = input('Cum te numesti? ')
varsta = input('Ce varsta ai? ')
print(nume + ' ai' + varsta + ' ani.')
```

- În Python, valorile variabilelor pot fi afișate împreună cu texte folosind șiruri formatate. Acest lucru se poate realiza: cu ajutorul funcției **format()** sau utilizând **f-string-uri**, prin scrierea literei ***f*** sau ***F*** înaintea ghilimelelor/apostrofurilor. În cazul *f-string-urilor*, variabilele sau expresiile se scriu între acolade *{ }* și sunt înlocuite automat cu valorile lor.

```
nume = input('Cum te numesti? ')
varsta = input('Ce varsta ai? ')
print(f'Buna {nume}. Ai {varsta} ani.')
```

```
nume = input('Cum te numesti? ')
varsta = input('Ce varsta ai? ')
print('Buna {0}. Ai {1} ani.'.format(nume,varsta))
```

Tipuri de date

În limbajul Python există tipurile de date: ***numerice, logice, șir de caractere***, ***liste***, ***dicționare, mulțimi*** și ***tupluri***. În Python, tipul unei variabile este stabilit automat în funcție de valoarea atribuită, fără a fi necesară declararea explicită a tipului de date.

```
nr1 = int(input("Primul numar:"))
nr2 = int(input("Al doilea numar:"))
print(nr1 + nr2)
print(nr1 - nr2)
print(nr1 * nr2)
```

Tipurile numerice sunt: **int** (numere întregi), **float** (numere reale) și **complex** (numere complexe). Funcțiile **int()**, **float()** și **complex()** pot fi utilizate pentru conversia valorilor la tipul de date dorit. Valorile citite cu **input()** sunt memorate ca șiruri de caractere și pot fi transformate în numere folosind aceste funcții.

```
a = int(input("a="))        # a=2
b = float(input("b="))      # b=3.5
print(a + b)                # 5.5
```

Dacă o operație implică atât un număr întreg, cât și un număr real, rezultatul va fi de tip real. Pentru numerele reale, Python afișează rezultatul cu precizia necesară, în funcție de valorile utilizate în calcul.

Variabilelor li se poate atribui direct un șir de caractere, scris între apostrofuri sau ghilimele. Ambele au același rol. Funcția ***str()*** este opțională – convertește alte tipuri de date în șir de caractere.

```
sir1 = str("Acesta e un sir!")
print(sir1)
sir2 ="********"
print(sir2)
sir3 = str('''Aici avem un exemplu de
sir de caractere scris pe mai multe
randuri.''')
print(sir3)
```

În cazul în care se dorește atribuirea unui șir de caractere care se află pe mai multe rânduri, acesta trebuie încadrat între trei apostrofuri sau trei ghilimele.

Lista este un tip de date Python care stochează o colecție de elemente separate prin **,** încadrate în **[ ]**. Elementele sunt indexate de la ***0*** la ***len(listă) - 1***, unde ***len()*** returnează lungimea listei. O listă se poate crea direct cu **[ ]** sau cu funcția **list()**.

```
Lista1 = list()
Lista2 = list(('Mate', 'Info', 'Fizica'))
print(Lista1) # se va afișa []
print(Lista2) # se va afișa ['Mate', 'Info', 'Fizica']
```

EX.

```
culori = ["rosu", "verde", "albastru"]
print(culori[0]) # rosu
```

Listele sunt ordonate, au număr finit de elemente și pot fi modificate. Elementele nu trebuie să fie de același tip. Accesul se face prin index: de la ***0*** la ***len-1*** înainte, sau de la ***-1*** (ultimul) în sens invers.

Expresii

<def/> **Expresia** este o **succesiune** alcătuită din **operanzi**, legați prin **operatori** ex: 12-3*2. O expresie se evaluează ținând cont de *precedența operatorilor* (ordinea efectuării operațiilor) și de *asociativitatea operatorilor* (de la stânga la dreapta și de la dreapta la stânga). Operatorii cu aceeași prioritate au aceeași asociativitate.

• Operatori aritmetici

<def/> Se clasifică în **operatori unari**, care acționează asupra unui singur operand (+, -) și **operatori binari**, care acționează asupra a doi operanzi (+, -, *, **, /, //, %). Operanzii pot fi întregi, reali sau complecși.

- Operatorul ** permite ridicarea unui număr la putere. Asociativitatea acestui este operator este de la dreapta la stânga. (Exemplu: 2**3**2=512 spre deosebire de (2**3)**2=64).
- Operatorul / determină rezultatul împărțirii a două numere întregi sau reale, rezultatul fiind de tip real.
- Operatorul // este operatorul de împărțire întreagă (rezultatul împărțirii se rotunjește în jos la cel mai apropiat număr întreg).
- Operatorul % este operatorul de împărțire cu rest.
- Operatorii % și // sunt utili în prelucrarea cifrelor unui număr, cu număr necunoscut de cifre.

```
a = 5
print(+ a)          # 5
print(- a)          # -5
b = 3
print(a + b)        # 8
print(a - b)        # 2
print(a * b)        # 15
print(a / b)        # 1.6666666666666667
print(a % b)        # 2
print(a // b)       # 1
c=2.25
print(a + c)        # 7.25
print(a - c)        # 2.75
print(a * c)        # 11.25
print(a / c)        # 2.2222222222222223
print(a % c)        # 0.5
print(a // c)       # 2.0
```

1.1. Ultima cifră a unui număr – operatorul %

Operatorul % (modulo) returnează restul împărțirii. Deoarece orice număr natural se scrie ca **n = q × 10 + r**, restul împărțirii la **10** este exact ultima cifră ($0 \le r \le 9$).

```
n = 712345; ultima = n % 10
# Ultima cifra
print('Ultima cifra:', ultima)    # 5
# 4732=473×10+2; 4732 % 10 = 2
# 8900=890×10 + 0; 8900 % 10 = 0
```

1.2. Ștergerea ultimei cifre a unui număr – operatorul //

Operatorul // (împărțire întreagă) returnează câtul întreg al împărțirii. Împărțind la **10**, eliminăm ultima cifră: 4732 // 10 = 473, deoarece 4732 = 473 × 10 + 2.

```
n = 712345
ultima = n % 10    # Ultima cifră
print('Ultima cifra:', ultima)    # 5
n = n // 10    # Eliminarea ultimei cifre
print('n fara ultima cifra:', n)    # 71234
```

1.3. Cifra de pe poziția k a unui număr

Fiind dat **n** și o poziție **k** (numărată de la dreapta, începând cu 0), cifra de pe poziția **k** se obține cu formula: **cifra_k = (n // 10^k) % 10.**

```
n = int(input())
k = int(input())
cifra_k = (n // (10 ** k)) % 10
print(f'Cifra k de la dreapta: {cifra_k}')
# n = 4732, k = 2
# 4732 // 100 = 47; 47 % 10 = 7
```

• Operatori relaționali

<def/> Operatorii relaționali sunt binari. Aceștia sunt: >, >=, <, <=, == și !=. Aceștia pot fi aplicați pe orice tip de date. Rezultatul aplicării lor este o valoare logică (True sau False), numită și valoare booleană. În exemplul dat este folosit operatorul de atribuire multiplă, iar a primește valoarea 5, b primește valoarea 3, iar variabila c primește valoarea 5.0.

```
a, b, c = 5, 3, 5.0
print(a < b)     # False
print(a > c)     # False
print(a == c)    # True
print(b != c)    # True
print(a >= b)    # True
```

• Operatori logici

<def/> Operatorii logici sunt: **not**, **and** și **or**. Operatorii **and** și **or** sunt binari, iar operatorul **not** este unar. Operanzii pot fi de tip întreg, real, logic etc., rezultatul fiind o valoare logică.

and	True	False
True	True	False
False	False	False

or	True	False
True	True	True
False	True	False

not	True	False
	False	True

• Operatori de atribuire

Operatorii de atribuire pot fi: operator de atribuire simplă (=) sau operatori de atribuire compusă (+=, -=, *=, /=, %=, //=, **=). Operatorii de atribuire au asociativitatea de la dreapta la stânga. Întâi se evaluează expresia din dreapta operatorului și se atribuie rezultatul variabilei aflate în stânga operatorului.

Tabelul de prioritate al operatorilor

Prioritate	Operatori	Asociativitate
1	( )	Stânga - Dreapta
2	**	Dreapta - Stânga
3	+a, -a	Stânga - Dreapta
4	*, /, //, %	Stânga - Dreapta
5	+, -	Stânga - Dreapta
6	==, !=, >, >=, <, <=	Stânga - Dreapta
7	not	Stânga - Dreapta
8	and	Stânga - Dreapta
9	or	Stânga - Dreapta

Fișa de lucru 3

Creați dosarul numit *Fisa3*. Rezolvați exercițiile următoare, notând răspunsurile corecte într-un fișier *raspunsuri.docx*.

1. *Parcurgeți următorul program scris în limbajul Python și stabiliți ce se va afișa dacă se citește numărul 209.* Explicați ce face acest program.

```
nr = int(input("Scrie un nr. de 3 cifre: "))
a = nr % 10; nr = nr // 10
b = nr % 10; nr = nr // 10
c = nr
nrnou = a * 100 + b * 10 + c
print("Nr. nou format este: ", nrnou)
```

2. *Transcrieți în limbajul Python, algoritmul prezentat în limbajul pseudocod. Testați programul.*

```
algoritm înălțime_medie
real h1, h2, h3, h4, h5, h6, h7, h8, medie
citește h1, h2, h3, h4, h5, h6, h7, h8
medie ← (h1+h2+h3+h4+h5+h6+h7+h8) / 8
scrie medie
sfârșit algoritm
```

3. *Rezolvați în limbajul Python următoarele probleme:* **a.** Se citesc două numere întregi. Să se realizeze cu aceste numere operațiile de adunare, scădere, înmulțire, împărțire cu cât și cu rest. **b.** Se citește un număr întreg cu exact 4 cifre. Să se determine suma cifrelor sale. **c.** Se citește un număr întreg cu exact 2 cifre. Să se determine suma tuturor numerelor de la 1 până la valoarea numărului întreg și să se afișeze produsul pătratelor cifrelor numărului citit.

Autoevaluare

Completați răspunsurile într-un fișier *FisaAutoevaluare3.docx*, folosind notații precum *1 – da, 2 – nu, 3 – nu sunt sigur*. Salvați fișierul în dosarul *Fisa3*.

	La finalul lecției, știu...	*Da*	*Nu*	*Nu sunt sigur(ă)*
1.	să scriu un program în limbajul Python.			
2.	care sunt tipurile de date în Python.			
3.	cum se declară variabilele.			
4.	cum se clasifică operatorii și ce sunt expresiile.			
5.	să parcurg un program în Python.			

6. Am întâmpinat dificultăți în: …

Nu uitați să vă evaluați comportamentul și activitatea din timpul lecției!
(cu ajutorul fișei de observare a comportamentului și a activității, de la pagina 7)

La finalul lecției, m-am simțit:

Lecția 4 – Structuri de control: liniară, alternativă și repetitivă

Structura liniară/secvențială

<def/> În limbajul Python ***structura liniară*** presupune, la fel ca și în alte limbaje de programare, executarea instrucțiunilor, una câte una, în ordinea în care sunt scrise.

EX. Să se scrie un program care determină aria totală și volumul unui cub, pentru care se citește din consolă lungimea laturii.

```
#structura secventiala
l = int(input("Lungimea laturii: "))
aria = 6 * l * l
vol = l * l * l
print(f"Aria cubului este {aria} si volumul este {vol}")
```

Structura alternativă/decizională

<def/> Structura ***alternativă*** evaluează o expresie logică și execută un anumit set de instrucțiuni în funcție de rezultatul obținut. Dacă expresia este adevărată, se execută setul de ***instrucțiuni_1***, iar dacă este falsă, se execută setul de ***instrucțiuni_2***. Sintaxa structurii în limbajul Python o aveți în imaginea alăturată. Blocul ***else*** poate să lipsească.

Sintaxa:
if *expresie_logică:*
    instrucțiuni_1
else:
    instrucțiuni_2

EX. Scrieți un program care citește vârstele a doi copii. Cei doi copii au vârste diferite. Afișați care dintre copii este mai mare.

```
varsta1 = int(input("Varsta primului copil: "))
varsta2 = int(input("Varsta celui de-al doilea copil: "))
if varsta1 > varsta2:
    print("Primul copil e mai mare decat al doilea")
else:
    print("Al doilea copil e mai mare decat primul")
```

OBS Structura alternativă are și o formă **if.. elif .. else** care constă în două structuri alternative imbricate. Sintaxa acestei forme o puteți vedea în imaginea alăturată. ***Modul de execuție*** este: se evaluează ***expresie_logică1***, dacă este adevărată se execută setul de ***instrucțiuni_1***, iar dacă este falsă se evaluează expresia de pe ramura ***elif***. Dacă ***expresie_logică2*** este adevărată se execută ***instrucțiuni_2***, altfel se execută ***instrucțiuni_3***.

Sintaxa:
if *expresie_logică1:*
    instrucțiuni_1
elif *expresie_logică2:*
    instrucțiuni_2
else:
    instrucțiuni_3

EX. Scrieți un program în care se determină ultima cifră a numărului 2^n, unde **n** este o valoare numerică întreagă.

```
n = int(input("n="))
if n == 0:
    print(f"Ultima cifra a lui 2^{n} este 1")
elif n % 4 == 1:
    print(f"Ultima cifra a lui 2^{n} este 2")
elif n % 4 == 2:
    print(f"Ultima cifra a lui 2^{n} este 4")
elif n % 4 == 3:
    print(f"Ultima cifra a lui 2^{n} este 8")
else:
    print(f"Ultima cifra a lui 2^{n} este 6")
```

OBS Instrucțiunile din blocul if se scriu indentate (de regulă cu 4 spații sau Tab), chiar și atunci când este una singură, dacă este pe linie separată, pentru a forma o instrucțiune compusă – altfel apare eroare de sintaxă. Instrucțiunile din bloc se pot indenta ***consecvent*** și cu spații (de preferat 4). Nu indentați și cu Tab și cu spații!

Folosiți mereu același număr de spații!

Structura repetitivă

<def/> Structura repetitivă cu număr necunoscut de pași este implementată cu ajutorul instrucțiunii **while**. Aceasta evaluează repetat o condiție: cât timp condiția este adevărată, se execută blocul de instrucțiuni.

Sintaxa:
while *expresie_logică:*
    instrucțiuni

OBS - Dintre structurile repetitive condiționate, Python oferă doar structura cu test inițial (while), nu și una cu test final (do…while). Nu există o implementare pentru structura repetitivă cu test final, ca în limbajul de programare C/C++.

- La fel ca în cazul structurii alternative este foarte importantă indentarea instrucțiunilor din cadrul structurii repetitive!

<def/> ***Structura repetitivă cu număr cunoscut de pași*** este folosită pentru a accesa pe rând elementele din cadrul unei liste, a unui șir de caractere etc. și pentru a le prelucra, în ordinea în care apar. Sintaxa structurii în limbajul Python o aveți în imaginea alăturată.

Sintaxa:
for *variabila* **in** *colecție_de_obiecte:*
    instrucțiuni

EX. Scrieți un program în care se parcurg elementele unei liste și se determină câte dintre ele sunt pozitive, câte negative și câte nule.

```
lista = list((0, -3, 12, 45, 3, 0, -5, 9))
poz = 0
neg = 0
nule = 0
for nr in lista:
    if nr > 0:
        poz = poz + 1
    elif nr < 0:
        neg = neg + 1
    else:
        nule = nule + 1
print(f"Exista {poz} nr. pozitive, {neg} nr. negative si {nule} nr. nule")
```

OBS În exemplul alăturat s-a folosit instrucțiunea **for** pentru a parcurge elementele listei. Dacă se dorește parcurgerea unei secvențe de numere de la **0** până la o **valoare finală – 1** și se cunoaște pasul de creștere al valorilor din secvență, se poate realiza această parcurgere cu ajutorul funcției **range(val_inițială, val_finală, pas_creștere)**. Valoarea inițială de la care începe secvența și pasul de creștere sunt opționale, valorile lor implicite fiind **0** și **1**.

EX. Scrieți un program în care se determină suma tuturor numerelor de la **1** la **n**, n fiind o valoare care se citește.

```
n = int(input('n='))
s = 0
for nr in range(1, n+1):
    s = s + nr
print(f'Suma numerelor de la 1 la {n} este: {s}')
```

Instrucțiunile break și continue

<def/> Instrucțiunea **break** este folosită în cadrul unei instrucțiuni repetitive dacă se dorește oprirea forțată a acesteia.

EX. Scrieți un program în care se vor citi numere până la întâlnirea valorii **0** și se va determina suma numerelor citite.

```
suma = 0
while True:
    nr = int(input("nr="))
    suma = suma + nr
    if nr == 0:
        break
print(f"Suma nr. citite este {suma}")
```

<def/> Instrucțiunea **continue** este folosită în cadrul unei instrucțiuni repetitive dacă se dorește oprirea forțată a iterației curente, continuându-se cu iterația următoare. Nu se va încheia însă și structura repetitivă când este folosită instrucțiunea **continue**.

EX. Scrieți un program în care se va citi o valoare **n** și se vor afișa doar numerele impare de la **1** la **n**.

```
n = int(input("n="))
for i in range(1, n + 1):
    if i % 2 == 1:
        print(i, end=" ")[1]
    else:
        continue
```

[1] Parametrul end din print() stabilește ce apare după numărul afișat – în loc să treacă pe linie nouă, pune caracterul specificat (în acest caz, un spațiu).

Fișa de lucru 4

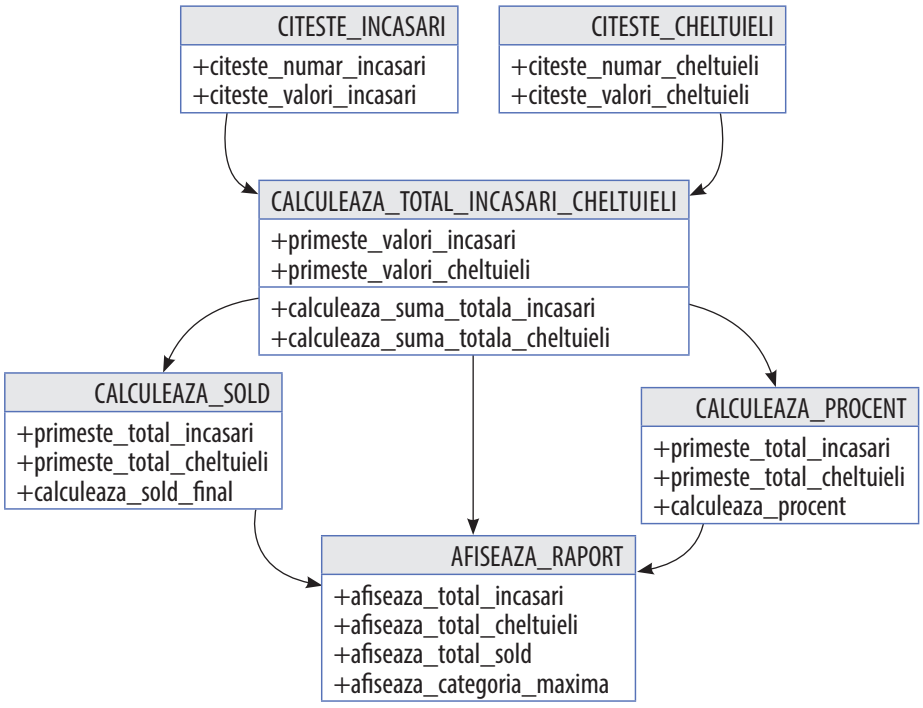
Creați dosarul numit *Fisa4*. Realizați în limbajul Python o aplicație pe care o veți salva în dosarul *Fisa4*.

Programul va gestiona bugetul unei persoane pe parcursul unei luni. Se citesc de la tastatură:

- numărul de încasări și valoarea fiecăreia;
- numărul de cheltuieli și valoarea fiecăreia.

Programul va afișa:

- totalul încasărilor;
- totalul cheltuielilor;
- soldul final (pozitiv, negativ sau nul);
- procentul cheltuielilor din totalul încasărilor.



Autoevaluare

Completați răspunsurile într-un fișier *FisaAutoevaluare4.docx*, folosind notații precum *1 – da*, *2 – nu*, *3 – nu sunt sigur*. Salvați fișierul în dosarul *Fisa4*.

	La finalul lecției, știu...	***Da***	***Nu***	***Nu sunt sigur(ă)***
1.	să folosesc structura secvențială în Python.			
2.	să folosesc structura alternativă în Python.			
3.	să folosesc structura repetitivă în Python.			
4.	să cunosc instrucțiunile break și continue.			

5. Am întâmpinat dificultăți în: …

Nu uitați să vă evaluați comportamentul și activitatea din timpul lecției!
(cu ajutorul fișei de observare a comportamentului și a activității, de la pagina 7)

La finalul lecției, m-am simțit:

Evaluare sumativă - Capitolul 1

Investighează!

(Punctaj maxim: 100 puncte)

Studiați programul alăturat și răspundeți la următoarele întrebări. Răspunsurile le veți nota într-un fișier ***investigheaza1.docx***:

```
intrebari = {
    "Care e capitala Bulgariei?": "Sofia",
    "Care e capitala Ungariei?": "Budapesta",
    "Care e capitala Greciei?": "Atena",
    "Care e capitala Croatiei?": "Zagreb",
    "Care e capitala Sloveniei?": "Ljubljana"
}
k = 0
for intrebare, raspuns in intrebari.items():
    aux = input(intrebare + " ").strip().capitalize()
    if aux == raspuns:
        print("Raspuns corect!")
        k = k + 1
    else:
        print(f"Raspuns gresit!!! {raspuns} este raspunsul corect.")
print(f"Rezultat final: {k}/{len(intrebari)}")
```

1. Care sunt datele de intrare, datele de ieșire și datele de manevră ale programului la final?

2. a. Ce ramură a structurii decizionale se execută atunci când utilizatorul introduce un răspuns incorect? Descrieți ce se întâmplă în acea ramură.

b. Cum ați modifica programul astfel încât răspunsul utilizatorului să fie acceptat indiferent de majuscule sau diacritice? (exemplu: "Da", "da", "DA" să fie tratate la fel)

3. Dacă ar fi să propuneți două teste: unul care verifică validarea datelor și unul care verifică comportamentul corect al algoritmului, care ar fi acelea?

4. Cum ați reorganiza programul folosind proiectare modulară? Identificați cel puțin 2 module distincte și descrieți rolul fiecăruia.

Sugestii: Lucrați în echipe de 2-3 elevi. Rescrieți programul, modificându-l, astfel încât să aibă cel puțin 10 întrebări. De asemenea, modificați formula de calcul a rezultatului final astfel încât să fie afișat punctajul obținut din 100 de puncte, fiecare întrebare valorând câte 10 puncte. Copiați programul în fișierul ***investigheaza1.docx.***

Modalitate de evaluare:
- Dosarul ***Capitol1*** conține fișierul ***investigheaza1.docx*** – **5p**
- Cerințele **1**, **2**, **3**, **4** sunt îndeplinite – **60p** (fiecare cerință valorează **15p**)
- Rescrierea și testarea programului – **25p**
- Modificarea formulei de calcul a punctajului din program – **10p**

Verifică-ți cunoștințele!

(Punctaj: 90 de puncte + 10 puncte din oficiu)

Cerință: Rezolvați exercițiile 1 și 2 într-un fișier ***evaluare_sumativa1.docx***. Salvați toate fișierele pe care le creați, în dosarul ***Capitol1***.

1. Stabiliți valoarea de adevăr a următoarelor afirmații: (5p × 6 = 30p)

a) Operatorul **%** returnează câtul împărțirii a două numere întregi.

b) Instrucțiunea **break** oprește execuția unei bucle.

c) Limbajul Python permite concatenarea șirurilor de caractere cu operatorul +.

d) Funcția **input( )** returnează implicit valori de tip **int**.

e) Structura **if** nu poate exista fără **else**.

f) Expresia 3 + 5.0 are tipul de date **float**.

2. Stabiliți ce se va afișa în urma parcurgerii secvențelor de cod următoare: (5p × 4 = 20p)

a)
```
a = 12
a += 2 * 5
print(a)
```

b)
```
for i in range(1, 5):
    print(i, end=" ")
```

c)
```
x = 5
if x % 2 == 0:
    print("*")
elif x > 10:
    print("**")
else:
    print("***")
```

d)
```
print(len("Limbajul Python"))
```

3. Scrieți un program în Python, numit ***numere_jucause.py*** și rezolvați cerințele: (4 × 10p = 40p)

Se citește un șir de numere întregi până la întâlnirea valorii 0. Determinați: **a)** Cel mai mare și cel mai mic număr din șir, precum și media lor aritmetică. **b)** Câte numere pozitive și negative există până la întâlnirea valorii 0. **c)** Suma numerelor pare din șir și produsul numerelor impare. **d)** Câte dintre perechile de numere citite consecutiv au ultima cifră egală.



Proiect

(Punctaj: 100 puncte)

Realizați un proiect în care să tratați o problemă din viața de zi cu zi. Implementați proiectul în limbajul Python. Ca și exemple de proiecte puteți lua în considerare: planificarea bugetului unei familii, realizarea unui chestionar sau a unui joc cu întrebări, implementarea unui convertor de unități de măsură etc. Proiectul va fi salvat cu numele ***proiect.docx***, în dosarul cu același nume.

1. Analiza proiectului – va consta în descrierea problemei identificate și clasificarea datelor de intrare, a celor de ieșire și a celor de manevră.

2. Proiectarea – va fi o diagramă cu minim două module în care vor fi ilustrate operațiile principale și rolul fiecăruia.

3. Implementarea – va fi reprezentată de programul în limbajul Python, care va conține cel puțin o structură alternativă și una repetitivă.

4. Testarea – vor fi prezentate cel puțin trei teste de intrare care vor fi documentate. Unul din teste va consta în date invalide, altul va reprezenta un caz limită pentru problema voastră și unul va fi cu date valide.

5. Concluzii – veți identifica dificultățile întâlnite în proiectarea și implementarea programului, care sunt avantajele proiectării propuse și ce ați îmbunătăți.

(Evaluare: **10p**/fiecare cerință, **15p**/existența programului, **15p**/funcționalitate, **10p**/creativitate, **5p**/utilitate, **5p**/design)



Portofoliu

(Punctaj: 100 puncte)

La finalul acestui capitol, portofoliul personal al elevului trebuie să cuprindă, în dosarul ***Capitol1***, următoarele materiale:

- Dosarul ***Fisa1***, cu fișierele: ***raspunsuri.docx***, ***FisaAutoevaluare1.docx***
- Dosarul ***Fisa2***, cu fișierele: ***raspunsuri.docx***, ***FisaAutoevaluare2.docx***
- Dosarul ***Fisa3***, cu fișierele: ***raspunsuri.docx***, ***FisaAutoevaluare3.docx***
- Dosarul ***Fisa4***, cu fișierele: ***aplicație.py***, ***FisaAutoevaluare4.docx***
- Dosarul ***Proiecte1***, cu fișierele: ***investigheaza1.docx***, ***proiect.docx***
- Fișierele ***evaluare_sumativa1.docx***, ***numere_jucause.py***

Modalitate de evaluare:
- Există dosarul ***Capitol1*** cu structura de directoare specificată – **20p**
- Există în dosare:
  - Peste 95% din fișiere – **80p**
  - Între 75% și 94% din fișiere – **60p**
  - Între 50% și 74% din fișiere – **40p**
  - Între 25% și 49% din fișiere – **20p**
  - Sub 25% – **0p**

Capitolul 2 – Strategii de rezolvare a problemelor. Prelucrări ale datelor numerice

Lecția 5 – Operații cu cifrele unui număr

Prelucrarea cifrelor unui număr natural reprezintă o clasă importantă de probleme în algoritmică. Algoritmii din această categorie exploatează proprietățile sistemului pozițional de numerație: fiecare cifră poate fi extrasă prin operațiile de împărțire întreagă și modulo (restul împărțirii %).

5.1. Scrierea pozițională a numerelor naturale

Orice număr natural poate fi privit ca o succesiune de cifre scrise în baza 10. De exemplu, 4732 este numărul patru mii șapte sute treizeci și doi. Valoarea sa se descompune astfel:

Cifra	4	7	3	2
Poziție (de la stânga)	mii (10^3)	sute (10^2)	zeci (10^1)	unități (10^0)
Valoare pozițională[1]	$4 \times 1000 = 4000$	$7 \times 100 = 700$	$3 \times 10 = 30$	$2 \times 1 = 2$

Suma valorilor poziționale: 4000 + 700 + 30 + 2 = 4732. Aceasta este descompunerea poziționala în baza 10 pe care o exploatăm în toți algoritmii din această lecție.

5.2. Accesul la cifrele unui număr

În capitolul anterior ați învățat cum să extrageți ultima cifră a unui număr utilizând operatorul %, cum să ștergeți ultima cifră a unui număr folosind operatorul //.

Repere pentru parcurgerea cifrelor unui număr

Pentru a prelucra cifrele unui număr natural, se aplică în mod repetat două operații aritmetice de bază:

- restul împărțirii la 10 (n % 10) → ***extrage ultima cifră*** a numărului
- împărțirea întreagă la 10 (n // 10) → ***elimină ultima cifră***, scurtând numărul

Parcurgerea cifrelor de la dreapta la stânga

Dacă aplicăm în mod repetat cele două operații, de extragere a ultimei cifre și de ștergere a acesteia, vom putea accesa toate cifrele pe rând. Bucla **while n > 0** se termină când nu mai există cifre de procesat. Urmăriți pas cu pas ce se întâmplă la fiecare iterație a buclei în tabelul de mai jos:

```
n = int(input("n: ")) # citim n
if n == 0:
    print (0)
while n > 0:
    c = n % 10 # extrage ultima cifra
    print (c)  # afișează cifra
    n //= 10   # șterge ultima cifra
```

⚠ **Atenție!** Dacă `n = 0`, bucla `while n > 0` nu se execută deloc. Tratați întotdeauna acest caz separat, înainte de buclă, cu `if n == 0`.

Simularea procesului de obținere a cifrelor

Iterație	n	c = n % 10	n //= 10 (după)	Cifra afișată
1	473	473 % 10 = 3	473 // 10 = 47	3
2	47	47 % 10 = 7	47 // 10 = 4	7
3	4	4 % 10 = 4	4 // 10 = 0	4
STOP	**0**	**condiția while n > 0 este falsă**	**---**	**Bucla se încheie**

Parcurgerea cifrelor de la stânga la dreapta

Dacă ordinea cifrelor este importantă, acestea pot fi parcurse de la stânga la dreapta. Folosind o putere a lui 10 (p) care indică poziția primei cifre rămase, extragem repetat această cifră și o eliminăm din număr, cât timp p > 0, păstrând astfel și cifrele 0 de la final.

```
n = int(input())
if n == 0:
print (0)
else:
    p = 10**9 # presupunând că n are 9 cifre
    while p > n:
        p = p // 10
    while p > 0:
        c = n // p # extrage prima cifra
        print (c)  # prelucreaza cifra
        n %= p   # sterge prima cifră
        p //= 10 # ajustăm p pentru următorul pas
```

[1] Valoare pozițională este valoarea pe care o are o cifră în funcție de locul (poziția) pe care îl ocupă într-un număr.

Aplicabilitate – clase de probleme

Algoritmul de parcurgere a cifrelor poate fi folosit pentru a calcula suma, produsul sau numărul cifrelor unui număr, pentru a găsi cifra maximă sau pentru a selecta doar cifrele care respectă anumite condiții. Acest tip de algoritm este utilizat în aplicații din viața reală precum verificarea CNP-ului sau a unui cod de card bancar.

EX. Algoritmul alăturat afișează suma cifrelor lui **n**. *Exemplu:* pentru **n = 245** se va afișa **11**.

```
n = int(input("n: "))
suma = 0
while n > 0:
    suma += n % 10   # adaugă ultima cifră la sumă
    n //= 10         # elimină ultima cifră
print("Suma cifrelor:", suma)
```

5.3. Construirea unui număr prin adăugarea de cifre

Pornind de la un număr existent, putem construi numere noi adăugând o cifră fie la sfârșitul său (la dreapta), fie la începutul său (la stânga).

Adăugarea unei cifre la dreapta

Adăugând cifra **c** la dreapta lui **n**, obținem un număr care are cu o cifră mai mult la final. Fiecare cifră veche urcă o poziție (valoarea sa pozițională se înmulțește cu 10), iar noua cifră ocupă poziția unităților.

```
n = 456
c = 9
n_nou = n * 10 + c
print(n_nou)   # 4569
```

Oglinditul unui număr

Oglinditul unui număr natural este numărul obținut prin inversarea ordinii cifrelor sale. De exemplu, oglinditul lui 1234 este 4321, iar al lui 1200 este 21 (zerourile de la început se pierd în reprezentarea numerică).

Pașii algoritmului: Extrage cifra unităților din numărul curent cu n % 10, adaugă cifra la sfârșitul oglinditului construit până în acel moment, apoi elimină cifra procesată prin n // 10. Procesul se repetă cât timp mai există cifre.

```
n = int(input("n = "))
ogl = 0
while n > 0:
    ogl = ogl * 10 + n % 10
    n //= 10
print(ogl)
```

Adăugarea unei cifre la stânga

Adăugând cifra **c** la stânga lui **n**, cifra nouă devine cea mai semnificativă. Determinăm mai întâi puterea lui 10 corespunzătoare în sistemul pozițional, apoi calculăm noul număr.

EX. Dacă adăugăm cifra 7 la stânga lui 456 determinăm mai întâi puterea lui 10 corespunzătoare, p = 1000.
Noul număr îl calculăm ca fiind 7 * 1000 + 456 = 7456.

Formula: n_nou = c × 10^x + n

```
n = int(input()) # 456
c = int(input()) # 7
p = 1
while p <= n:
    p *= 10
n_nou = c * p + n  # 7*1000+456
print(n_nou)  # 7456
```

5.4. Evaluarea corectitudinii – teste adecvate, surse frecvente de erori

Orice algoritm trebuie verificat și pentru cazuri limită. Pentru **n = 0**, bucla while **n > 0** nu se execută: suma cifrelor rămâne corect 0, dar numărul de cifre devine greșit 0 în loc de 1. La numerele cu zerouri la final, prin oglindire se pierd zerourile (4300 → 34 → 43).

Fișa de lucru 5

Creați dosarul numit *Fisa5*. Rezolvați exercițiile următoare, notând răspunsurile corecte într-un fișier *raspunsuri.docx*.

1. *Cifra maximă:* Algoritmul alăturat citește un număr natural ***n*** și afișează cifra maximă a acestuia. *Exemplu:* pentru ***n*** = 230 se va afișa 3. Modificați acest algoritm astfel încât să afișați cifra minimă impară. Dacă numărul nu are cifre impare, se va afișa un mesaj "Nu există". *Exemplu:* pentru ***n*** = 2301 se va afișa 1, iar pentru ***n*** = 240 se va afișa mesajul "Nu există".

```
n = int (input ())
cmax = 0
while n > 0:
    c = n % 10  # extrage ultima cifra
    if c > cmax:
        cmax = c
    n //= 10    # șterge ultima cifra
print (cmax)
```

```
n = int (input ())
nr_cifre = 1
while n > 9:
    nr_cifre += 1
    n //= 10
print( nr_cifre )
```

2. ***Numărul de cifre:*** Algoritmul alăturat citește un număr natural ***n*** și afișează numărul de cifre ale lui ***n***. Modificați algoritmul astfel încât să afișați numărul de cifre divizibile cu 3.

3. Se dă algoritmul alăturat care citește un număr natural nenul ***n*** și afișează produsul ***p*** al cifrelor nenule ale lui, respectiv suma ***s*** a cifrelor ce au în față cifra ***3***.

Modificați acest algoritm astfel încât acesta să numere câte cifre sunt urmate de o cifră nulă.

Exemplu: pentru n = **1035402** cifra 1 și cifra 4 sunt urmate de o cifră 0, deci vom afișa 2.

```
n = int (input ())
p = 1
s = 0
while n > 0:
    c = n % 10
    if c != 0:
        p = p * c
    if n > 9 and n // 10 % 10 == 3:
        s += c
    n = n // 10
print( p )
print ( s )
```

Probleme propuse - Parcurgerea cifrelor

Implementați algoritmii de mai jos și testați algoritmii cu 3 date de test.

1. Citiți un număr natural **n**. Afișați suma cifrelor și numărul de cifre. *Exemplu:* n = 12345 afișează s = 15, cifre = 5; n = 0 afișează s = 0, cifre = 1 (caz particular!).

2. Citiți un număr natural **n** și o cifră **k**. Determinați câte cifre sunt egale cu cifra **k**.

3. Verificați dacă un număr citit de la tastatură este palindrom (se citește la fel de la stânga și de la dreapta).

4. Citiți un număr natural **n**. Afișați numărul obținut din toate cifrele impare ale lui **n**, în ordine crescătoare. *Exemplu:* **n** = 429353 se va afișa 3359.

5. Mara și Mihai își construiesc coduri pornind de la un număr natural **n**. Fiecare cifră a lui **n** se înlocuiește cu ultima cifră a sumei dintre ea și cifra următoare; ultima cifră Citiți un număr **n** și afișați codul obținut.

Exemplu: n = 1934 → (1+9)%10=0, (9+3)%10=2, (3+4)%10=7, 4→4 → codul este 0274.

6. ***Mini-proiect:*** Realizați un proiect despre doi algoritmi utilizați în viața reală, care implică prelucrarea cifrelor (*Exemplu:* algoritmul lui Luhn, Cifrul Caesar etc.). Proiectul trebuie să conțină o prezentare a contextului în care este folosit algoritmul, descrierea pas cu pas a metodei de funcționare și o implementare în limbajul Python.

Autoevaluare! Ce am învățat?

[CUNOAȘTERE] [ÎNȚELEGERE] [APLICARE] [ANALIZĂ] [EVALUARE] [CREARE]

CUNOAȘTERE

1. Asociați:

`n // 10`	adaugă cifra 5 la sfârșitul numărului n
`n % 10`	șterge ultima cifră din n
`n * 10`	adaugă o cifră 0 la finalul numărului n
`n = n * 10 + 5`	extrage ultima cifră din n

ÎNȚELEGERE

2. Fie algoritmul alăturat. Ce valoare va afișa pentru ***n = 256***?

a. 256; b. 13;
c. 62; d. 652.

```
n = 256
s = 0
while n != 0:
    s = s + n % 10
    n = n // 10
print(s)
```

APLICARE

3. Completați fragmentul care numără cifrele pare ale lui ***n***:

a. c % 2 == 0; b. c % 2 == 1;

c. n % 2 == 0; d. c == 0.

```
n = 12045
nr = 0
while n != 0:
    c = n % 10
    if ____________:
        nr = nr + 1
    n = n // 10
print(nr)
```

ANALIZĂ

4. Analizați algoritmul următor și identificați ce calculează, considerând pozițiile cifrelor numerotate de la 1:

a. produsul cifrelor pare;

b. produsul cifrelor de pe pozițiile pare, numerotate de la dreapta la stânga;

c. produsul cifrelor de pe poziții pare, numerotate de la stânga la dreapta;

d. produsul cifrelor de pe poziții impare, numerotate de la dreapta la stânga.

```
p = 1
d = 1
while n > 0:
    c = n % 10
    if d % 2 == 0:
        p = p * c
    d = d + 1
    n = n // 10
print (p)
```

EVALUARE

5. Un elev susține că algoritmul de alături verifică corect dacă ***n*** este palindrom. Are sau nu dreptate? Argumentați și identificați cazul în care eșuează. Algoritmul funcționează totuși pe un caz particular. Care este acela?

```
n = int (input())
ogl = 0
while n > 0:
    c = n % 10
    ogl = ogl * 10 + c
    n = n // 10
if ogl == n:
    print("DA")
else:
    print("NU")
```

CREARE

6. Se dă un număr natural ***n***. Construiți un algoritm care generează o „parolă" extrăgând alternativ **prima** și **ultima** cifră a lui ***n***, în această ordine, până când nu mai rămân cifre. Cifrele extrase se concatenează formând parola. Pentru ***n*** = 12345, parola este 15243

Barem: 2p (din oficiu) + 1p (1) + 1p (2) + 1p (3) + 0.5p (4) + 2p (5) + 2.5p (6)

	La finalul lecției, știu ...	***Da***	***Nu***	***Nu sunt sigur(ă)***
1.	să extrag ultima cifră a unui număr și să o șterg folosind operatorii % și //.			
2.	să parcurg toate cifrele unui număr și să calculez suma, produsul sau numărul lor.			
3.	să adaug o cifră la dreapta unui număr și la stânga.			
4.	să construiesc oglinditul unui număr.			
5.	să filtrez cifrele după o condiție și să le prelucrez selectiv.			
6.	să identific și să tratez cazurile limită.			

7. Am întâmpinat dificultăți în: …

Nu uitați să vă evaluați comportamentul și activitatea din timpul lecției!
(cu ajutorul fișei de observare a comportamentului și a activității, de la pagina 7)

La finalul lecției, m-am simțit:

Lecția 6 – Algoritmi de divizibilitate: Divizorii unui număr; Primalitate

6.1. Divizorii unui număr

Ce este un divizor?

Un număr natural ***d*** este ***divizor*** al numărului ***n*** natural nenul dacă împărțirea lui ***n*** la ***d*** dă restul ***0***, adică ***n%d==0***. Cu alte cuvinte, ***d*** îl împarte exact pe ***n***, fără rest.

EX. D(12) = {1, 2, 3, 4, 6, 12} | D(7) = {1, 7} | D(36) = {1, 2, 3, 4, 6, 9, 12, 18, 36}

Algoritmul de bază – parcurgere 1 … n

Fie ***n*** un număr natural. Dorim să afișăm în ordine toți divizorii acestuia.
Idee de rezolvare: testăm fiecare număr ***d*** de la ***1*** la ***n*** și îl afișăm dacă divide ***n***.

```
n = int(input())
d = 1
while d <= n:
    if n % d == 0:
        print(d)
    d = d + 1
```

Algoritmul optimizat – parcurgere 1 … √n

Fie ***n*** un număr natural nenul. Să se determine suma divizorilor numărului ***n***.

OBS Dacă ***d*** | ***n***, atunci și ***(n // d)*** | ***n***. Divizorii vin în perechi ***(d, n//d)***. Ajunge să parcurgem ***d*** de la ***1*** la $\sqrt{n}$, reducând numărul de operații de la ***n*** la $\sqrt{n}$. Avem un caz particular însă, numerele pătrate perfecte pentru care radicalul lor trebuie adunat doar o singură dată.

```
# varianta lentă
n = int(input())
s = 0
d = 1
while d <= n:
    if n % d == 0:
        s = s + d
    d = d + 1
print (s)
```

```
# varianta optimizată
n = int(input())
s = 0
d = 1
while d * d < n:
    if n % d == 0:
        s = s + d + n // d
    d = d + 1
if d * d == n:
    s = s + d
print (s)
```

Comparație algoritmică

Să analizăm cele două metode propuse pentru determinarea sumei divizorilor:

Criteriu	Varianta 1: 1…n	Varianta 2: 1…√n	Varianta preferabilă
Nr. operații	n operații	≈ √n operații	V2
Număr pași pentru n = 1 000 000	1 000 000 pași	1 000 pași	V2
Implementare	Simplă	Puțin mai complexă	V1
Cazul $d^2 = n$	Nu e problemă	Necesită verificare	V1

Aplicații practice – Contexte reale de utilizare a divizorilor:

- Tăierea unei benzi de ***L*** cm în bucăți întregi, fără pierderi: lungimile posibile ale bucăților = divizorii lui ***L***;
- Distribuirea a ***N*** caiete: numărul de elevi posibili = orice divizor ***d*** al lui ***N*** (fiecare primește ***N/d*** caiete).

6.2. Primalitate

Ce este un număr prim

<def/> **Număr prim**: ***n > 1*** care nu are alți divizori decât ***1*** și ***n***. *Exemple:* 2, 3, 5, 7, 11, 13, 97, 101.
Număr compus: ***n > 1*** care are cel puțin un divizor ***d*** cu ***1 < d < n***. *Exemple:* $4=2^2$, $6=2\times3$, $15=3\times5$.
Cazuri speciale: ***0*** și ***1*** nu sunt nici numere prime, nici numere compuse – acestea nu intră în clasificare.

Determinarea primalității

Am arătat mai sus că putem obține toți divizorii unui număr, **parcurgând doar numerele de la 1 la $\sqrt{n}$**, (în perechi ***d*** și ***n/d***). Acum aplicăm direct această observație: un număr este prim dacă și numai dacă ***nu are niciun divizor d cu $2 \le d \le \sqrt{n}$***. Nu are sens să căutăm divizori mai mari decât $\sqrt{n}$, dacă un astfel de ***d*** ar exista, atunci ***n/d*** ar fi un al doilea divizor $\le \sqrt{n}$. Pentru analiză, prezentăm și varianta lentă. Ambele variante pornesc cu ***d=2*** și avansează câte un pas. **Condiția de oprire** este cea care diferențiază eficiența:

Python – V1 parcurgere 1…n (lentă)	Python – V2 parcurgere 1…$\sqrt{n}$ (optimizată)
`n = int(input('n = '))` `d = 2` `while d < n and n % d != 0:` `  d += 1` `if d == n: # nu am găsit nici un divizor` `  print('PRIM')` `else:` `  print('nu este prim')` `# Dacă n este prim => d merge pana la n - 1`	`n = int(input('n = '))` `d = 2` `while d * d <= n and n % d != 0:` `  d += 1` `if d * d > n and n > 1: # niciun divizor gasit` `  print('PRIM')` `else:` `  print('nu este prim')` `# Dacă n este prim => d merge pana la √n`

De ce condiția d*d > n echivalează cu «niciun divizor până la $\sqrt{n}$»?

Bucla se oprește când:

(a) ***d*d > n*** – am epuizat toți candidații $\le \sqrt{n}$ fără să găsim un divizor, deci ***n*** e prim;

(b) ***n%d==0*** – am găsit un divizor, deci ***n*** e compus.

Testul ***d*d <= n*** evită calculul ***sqrt()*** și funcționează corect și la valori mari.

⚠ **Atenție!** ***n=1*** trebuie tratat separat (***1*** nu e prim). Condiția ***«n > 1»*** din ***if*** asigură asta.

Comparație algoritmică

n	Este prim?	Pași V1 (1…n)	Pași V2 (1…$\sqrt{n}$)	Raport V1/V2
7	DA	5	2	2.5×
25	NU	3	3	1× (egal)
97	DA	95	8	~12×
1 009	DA	1 007	31	~32×
1 000 003	DA	~10^6	~1000	~1000×

Aplicații practice – Contexte reale de utilizare a numerelor prime

- Securitatea pe Internet: Numerele prime sunt folosite în metode de protejare a datelor (*Exemplu:* **RSA**[2]). E ușor să înmulțești două numere prime, dar foarte greu e procesul invers. (Vezi descompunerea în factori primi din lecția ce urmează)
- Hashing: se folosește ca să transformi rapid date (cuvinte, parole, fișiere) într-un „cod” scurt.
- Numerele prime sunt ca niște „cărămizi de bază” ale tuturor numerelor. Vom vedea în lecția ce urmează descompunerea numerelor în produs de factori primi.

Reține!

✔ Divizorii lui ***n*** apar în perechi ***(d, n//d)***, deci ajunge să căutăm până la $\sqrt{n}$.

✔ Un număr este prim dacă nu are niciun divizor ***d*** cu ***$2 \le d \le \sqrt{n}$***.

✔ Cazurile speciale: ***0*** și ***1*** nu sunt nici prime, nici compuse.

✔ Condiția ***d*d <= n*** este preferată în cod față de ***d <= $\sqrt{n}$*** pentru eficiență și acuratețe.

[2] RSA este un algoritm de criptare folosit pentru securizarea datelor pe internet.



Fișa de lucru 6

Creați dosarul numit *Fisa6*. Rezolvați exercițiile următoare, notând răspunsurile corecte într-un fișier *raspunsuri.docx*.

1. Fie ***n*** un număr natural. Să se afișeze divizorii acestuia, în ordine crescătoare. Folosiți un algoritm optim de parcurgere a divizorilor.

Idee de rezolvare: Putem parcurge intervalul de la ***1 … $\sqrt{n}$*** de două ori, o dată pentru a afișa divizorii de până în radical și a doua oară pentru a afișa perechile acestora.

În ce ordine sunt parcurși divizorii a doua oară și de ce?

```
n = int(input())
# afișăm mai intâi divizorii de la 1 până la radical
# vom parcurge aproximativ sqrt(n) numere
d = 1
while d * d < n:
    if n % d == 0:
        print(d)
    d = d + 1
# afișăm divizorii de după radical
# și afișăm perechea fiecăruia
if d * d > n:    # dacă am depășit radicalul
    d -= 1       # devansăm d
while d >= 1:
    if n % d == 0:
        print(n // d)
    d = d - 1
```

2. Un ***număr*** (n>1) este **perfect** dacă suma divizorilor săi (fără el însuși) este egală cu numărul. *Exemplu:* 6 = 1 + 2 + 3. Scrieți un algoritm care verifică dacă un număr este perfect.

Idee de rezolvare: Vom calcula suma divizorilor folosind algoritmul prezentat în lecție și vom testa dacă suma obținută este egală cu ***n***.

```
n = int(input())
s = 1
d = 2
# ne oprim cu d până la radicalul numărului n
while d * d < n:
    if n % d == 0:
        s = s + d + n // d
    d = d + 1
# aici d va fi fie egal cu radical din n, dacă n e pătrat perfect
# fie mai mare decât radical din n, în caz contrar
if d * d == n:    # dacă n e pătrat perfect
        s = s + d    # adăugăm radicalul doar o dată
if n == s:
        print('Numarul este perfect')
else:
        print('Numarul nu este perfect')
```

De ce suma a fost inițializată cu 1?

3. Determinați cel mai mic divizor al lui ***n***, care este și număr prim.

4. Se citește un număr ***n***. Să se determine dacă ***n*** este pătrat prim, adică un număr care poate fi scris ca pătratul unui număr prim. *Exemplu:* 9 este pătrat prim, căci poate fi scris ca fiind 3^2.

Probleme propuse

1. Scrieți un program care citește ***n*** și afișează: numărul de divizori ai lui ***n***, cel mai mare divizor propriu al lui ***n*** și suma divizorilor impari ai lui ***n***.

2. O bandă de ***L*** cm (***L***>3) trebuie tăiată în bucăți egale (lungimi naturale, fără pierderi). Afișați toate lungimile posibile, dacă banda se taie în cel puțin ***3*** bucăți.

3. Citiți ***N***. Determinați în câte moduri ***N*** caiete pot fi distribuite în mod egal elevilor dintr-un grup care are cel puțin ***2*** și cel mult ***10*** elevi. Afișați fiecare variantă (nr. elevi, nr. caiete/elev).

4. Un număr ***n*** se numește ***abundent*** dacă suma divizorilor săi proprii este mai mare decât ***n***. Afișați toate numerele abundente ≤ 500.

5. Se citește un număr natural ***n***. Să se determine cel mai mic număr prim, strict mai mare decât ***n***.

Autoevaluare! Ce am învățat?

[CUNOAȘTERE] [ÎNȚELEGERE] [APLICARE] [ANALIZĂ] [EVALUARE] [CREARE]

CUNOAȘTERE

1. Care din variantele de mai jos reprezintă definiția corectă a unui **divizor**?
a. d este divizor al lui n dacă n % d > 0 b. d este divizor al lui n dacă n % d == 0
c. d este divizor al lui n dacă d % n == 0 d. d este divizor al lui n dacă n // d == 0

ÎNȚELEGERE

2. Ce afișează algoritmul următor?
a. 26 b. 28
c. 16 d. 12

```
s = 0
d = 1
while d * d < 12:
    if 12 % d == 0:
        s = s + d + 12 // d
    d += 1
if d * d == 12:
    s = s + d
print(s)
```

APLICARE

3. Scrieți un algoritm care citește un număr natural ***n*** și afișează suma divizorilor impari.

ANALIZĂ

4. Sunt dați doi algoritmi de verificare a primalității unui număr n. Analizați și identificați o diferență și o asemănare între ei în ceea ce privește divizorii testați.

```
# Algoritmul A
d = 2
e_prim = True
while d <= n - 1:
    if n % d == 0:
        e_prim = False
    d += 1
```

```
# Algoritmul B
d = 2
e_prim = True
while d <= n // 2:
    if n % d == 0:
        e_prim = False
    d += 1
```

EVALUARE

5. Un elev verifică primalitatea cu while ***d <= n // 2***. Este mai bună sau mai slabă decât while ***d * d <= n***? Argumentați cu numărul de pași pentru ***n = 97***.

CREARE

6. Scrieți un algoritm care citește un număr natural ***n*** și afișează dacă ***n*** este „aproape prim” (poate fi scris ca produs a exact două numere prime, nu neapărat distincte).

Barem: 2p (din oficiu) + 1p (1) + 1p (2) + 1p (3) + 0.5p (4) + 2p (5) + 2.5p (6)

	La finalul lecției, știu...	***Da***	***Nu***	***Nu sunt sigur(ă)***
1.	să explic ce este un divizor și să dau exemple.			
2.	să implementez algoritmul de parcurgere ***1…n*** în Python.			
3.	să implementez algoritmul optimizat ***1…$\sqrt{n}$*** cu afișare ordonată.			
4.	de ce Varianta 2 este mai eficientă și să compar numărul de pași.			
5.	să aplic algoritmul de divizori pentru rezolvarea problemelor practice.			
6.	să implementez și să explic algoritmul numărului perfect.			

7. Am întâmpinat dificultăți în: …

Nu uitați să vă evaluați comportamentul și activitatea din timpul lecției!
(cu ajutorul fișei de observare a comportamentului și a activității, de la pagina 7)

La finalul lecției, m-am simțit:

Lecția 7 – Descompunerea în factori primi; CMMDC, CMMMC (Euclid)

7.1. Descompunerea în factori primi

1. Studiu de caz: afișarea divizorilor primi ai lui n

În lecția anterioară am parcurs ***d*** de la ***1*** la $\sqrt{n}$ pentru a găsi toți divizorii lui ***n***. Acum întrebăm: cum găsim doar divizorii ***primi***, fără a verifica explicit primalitatea? ***Ideea cheie:*** pe măsură ce găsim un divizor, îl și ***eliminăm*** din număr – împărțim ***n*** la divizorul găsit, repetat, până nu îl mai divide. Astfel, când ajungem la un ***d*** compus, toți factorii lui primi au fost deja extrași, deci ***d*** nu va mai divide ***n***-ul rămas – **posibilii divizori compuși se elimină singuri**.

EX.

Exemplu pas cu pas: n = 180

d=2: 180÷2=90, 90÷2=45 → factor 2^2, n devine 45

d=3: 45÷3=15, 15÷3=5 → factor 3^2, n devine 5

d=4: 5 nu se divide cu 4

d=5: 5÷5 = 1, n=1 → factor 5 (descompunerea se oprește)

Rezultat: $180 = 2^2 \times 3^2 \times 5^1$ | 3 factori primi distincți | pași efectuați: ~4

Această idee stă la baza algoritmului următor: descompunerea în factori primi.

2. Descompunerea în factori primi

Teorema fundamentală a aritmeticii garantează că orice număr natural $n > 1$ se scrie **unic** ca produs de numere prime: $n = p_1^{a_1} \cdot p_2^{a_2} \cdot \ldots \cdot p_n^{a_n}$. De exemplu: $360 = 2^3 \cdot 3^2 \cdot 5^1$.

Spre deosebire de algoritmul de găsire a divizorilor, unde ***n* rămânea fix**, aici ***n* se transformă** la fiecare pas: când găsim un factor prim, îl scoatem complet prin împărțiri repetate și continuăm căutarea în numărul rămas. Pentru ***360***: extragem ***2*** de ***trei*** ori (360→180→90→45), apoi căutăm factori în ***45***, nu în ***360***.

Varianta naivă – lentă: Pornim cu $d = 2$ și creștem ***d*** până când ***n*** ajunge la ***1***. La fiecare ***d***, dacă divide ***n***, îl extragem complet și afișăm factorul cu puterea lui. Funcționează, dar pentru un număr prim foarte mare ***d*** va trebui să ajungă până la ***n*** însuși.

Varianta optimizată: După ce am scos toți factorii, poate rămâne în ***n*** un factor prim negăsit – dar **cel mult unul singur**. Iată de ce: dacă ar rămâne doi factori ***p*** și ***q***, cel mai mic dintre ei, ***d***, ar satisface $d \times d \leq p \times q \leq n$, deci $d \leq \sqrt{n}$ – și bucla l-ar fi găsit deja, ***contradicție***. Prin urmare ne oprim când $d \times d > n$: toți factorii care apar de două ori sau mai mult au fost deja extrași, reducând verificările de la ***n*** la doar ~$\sqrt{n}$.

Python – V1 parcurgere 1…n (lentă)

```
n = int(input())
d = 2

while n > 1:
    if n % d == 0:
        p = 0
        while n % d == 0:
            n //= d
            p += 1
        print(f'{d}^{p}')
    d += 1
# Cel mai rau caz: n prim => d merge pana
# la n-1
```

Python – V2 parcurgere 1…√n (optimizată)

```
n = int(input('n = '))
d = 2
while d * d <= n:       # parcurgem pana la √n
    p = 0               # puterea factorului d
    while n % d == 0:   # cat timp d divide n
        n //= d         # impartim n la d
        p += 1          # crestem puterea
    if p > 0:           # d a fost factor
        print(f'{d}^{p}')
    d += 1
if n > 1:       # a ramas un factor prim
    print(f'{n}^1', end='')
# Cel mai rau caz: n prim => d merge pana
# la √n
```

n	Factorizare	Pași V1	Pași optim
84	$2^2 \times 3 \times 7$	82	8
360	$2^3 \times 3^2 \times 5$	358	18
97	97^1 (prim)	96	9
1000003	1000003^1 (prim)	1000002	~1000

3. Aplicații practice ale descompunerii

Descompunerea în factori primi stă la baza multor algoritmi fundamentali:

cmmdc și cmmmc cmmmc(a,b) = produs al factorilor comuni și necomuni la puterea maximă. **Exemplu:** cmmmc(12,18) = $2^2 \times 3^2 = 36$	**Criptografie RSA** Securitatea RSA se bazează pe dificultatea descompunerii unui număr mare (produs de 2 numere prime mari) în factori.
Numărul de divizori și suma lor Pornind de la descompunerea în factori primi, putem calcula numărul de divizori, respectiv suma acestora.	**Simplificarea fracțiilor** cmmdc(a,b) prin factorizare → a/cmmdc și b/cmmdc sunt termenii reduși la forma cea mai simplă.

7.2. CMMDC, CMMMC

Algoritmul lui Euclid – cmmdc

Algoritmul lui Euclid este cea mai eficientă metodă clasică de calcul al celui mai mare divizor comun (CMMDC) a două numere nenule, fără a fi nevoie să le determinăm factorii primi. Ideea lui se bazează pe faptul că divizorii comuni se păstrează atunci când reducem problema la una mai simplă.

CMMDC : Pentru două numere întregi pozitive a și b, cel mai mare divizor comun al lor, notat cmmdc(a,b), este cel mai mare $d>0$ astfel încât d divide atât a cât și b.

EX. cmmdc(12,8)=4 (divizorii comuni sunt 1, 2, 4) cmmdc(17,5)=1 (numere prime între ele) cmmdc(48,18)=6	Proprietăți utile: cmmdc(a,0) = a pentru $a \geq 0$. cmmdc(a,b) = cmmdc(b,a) (comutativitate). Dacă d = cmmdc(a,b) atunci există numere întregi x,y cu $ax+by=d$ (identitatea lui Bézout).

Rațiunea inițială – Algoritmul cu scăderi

Euclid (aprox. 300 î.Hr.) a observat următorul principiu: Dacă un număr d divide pe a și pe b, atunci el divide și diferența lor:

$a = d \times x$

$b = d \times y$

$a - b = d(x - y)$

De aici rezultă regula fundamentală: ***cmmdc(a, b) = cmmdc(a - b, b)***. Procedura continuă prin scăderi repetate până când cele două numere devin egale. Atunci acel număr este ***CMMDC***.

EX.

cmmdc(48, 18):

48 – 18 = 30 → cmmdc(30, 18)

30 – 18 = 12 → cmmdc(12, 18)

18 – 12 = 6 → cmmdc(6, 12)

12 – 6 = 6 → cmmdc(6, 6) = 6

Optimizarea – Algoritmul cu împărțiri

Observația cheie este că scăderile repetate reprezintă, de fapt, operația de împărțire cu rest. Înseamnă că: 48 = 2 × 18 + 12 → 48 mod 18 = 12 Prin urmare, în loc să facem scăderi una câte una, putem trece direct la restul împărțirii: ***cmmdc(a, b) = cmmdc(b, a mod b)***

Divizorii comuni rămân aceiași deoarece orice divizor comun al lui ***a*** și ***b*** divide și restul ***a − q·b***. Algoritmul continuă până când restul devine ***0***, moment în care ultimul număr nenul este ***CMMDC***.	cmmdc(48, 18): 48 mod 18 = 12 → cmmdc(18, 12) 18 mod 12 = 6 → cmmdc(12, 6) 12 mod 6 = 0 → cmmdc(6, 0) = 6

Comparând cele două metode:

- Cu scăderi: 4 pași
- Cu împărțiri: 3 pași

Diferența devine uriașă pentru numere mari. *Exemplu:* cmmdc(1.000.000, 1)

- Cu scăderi: 999.999 operații
- Cu împărțiri: 1 operație (1.000.000 mod 1 = 0)

Explicația intuitivă

Imaginează două șnururi: unul lung de ***48*** cm și unul de ***18*** cm. Vrei să le tai în bucăți egale cât mai mari.
– Tai bucăți de ***18*** cm din cel de ***48*** → îți rămâne ***12*** cm. Acum ai șnururile de ***18*** cm și ***12*** cm.
– Tai bucăți de ***12*** cm din cel de ***18*** → îți rămâne ***6*** cm. Acum ai ***12*** cm și ***6*** cm.
– Tai bucăți de ***6*** cm → rest ***0***.
Ultima bucată care „intră perfect" în ambele este ***6*** cm, adică ***CMMDC***.

Activitate practică: Vom tăia 2 panglici cu dimensiuni numere întregi, astfel încât la final să obținem benzi de aceeași dimensiune (număr întreg)

Aplicabilitate – alte probleme din viața reală:

– *Distribuirea materialelor:* Laboratorul are două role de sârmă, una de ***72*** cm și una de ***120*** cm. Se vor tăia bucăți identice, cât mai lungi, fără rest. Care poate fi lungimea maximă?

– *Gruparea elevilor:* Două clase au ***24*** și ***30*** de elevi. Profesorul vrea să formeze grupe mixte, toate cu același număr de elevi și fără ca cineva să rămână negrupat. Care este mărimea maximă a unei grupe?

– *Împărțirea exercițiilor pe capitole:* Un manual are un capitol cu ***42*** de exerciții și altul cu ***63***. Se vor crea seturi de fișe astfel încât fiecare fișă să conțină același număr de exerciții din fiecare capitol, fără rest. Care este numărul maxim de exerciții dintr-o fișă?

– *Acoperirea unei suprafețe dreptunghiulare cu pătrate de latură maximă* (aplicabilă la pavarea suprafețelor).

Python – cu scăderi repetate (lent)

```
a = int(input())
b = int(input())
if a == 0:
    print(b)
elif b == 0:
    print(a)
else:
    while a != b:
        if a > b:
            a = a - b
        else:
            b = b - a
    print(a)
```

Python – cu împărțiri repetate (optim)

```
a = int(input())    # a,b pozitive
b = int(input())
while b > 0:
    r = a % b
    a = b
    b = r
print(a)
```

Cel mai mic multiplu comun

Cel mai mic multiplu comun a două numere este cel mai mic număr strict pozitiv care este multiplu pentru ambele. Cu alte cuvinte, este primul număr la care „se întâlnesc" cele două serii de multipli, multiplii lui ***a*** și multiplii lui ***b***. Se calculează eficient folosind relația dintre ***CMMMC*** și ***CMMDC***:

CMMMC(a, b) = a * b / CMMDC(a, b)

Motivul este că produsul a×b include toți factorii ambelor numere, iar împărțirea prin ***CMMDC*** elimină factorii comuni dublați.

Explicație intuitivă

Îți poți imagina două roți dințate: una face câte un „clic” la fiecare ***4*** secunde, cealaltă la fiecare ***6*** secunde. Momentul în care fac clic simultan pentru prima dată este ***CMMMC(4, 6) = 12 secunde***, deoarece ***12*** este primul multiplu comun al lui ***4*** și ***6*** (multiplii lui ***4: 4, 8, 12…***; multiplii lui ***6: 6, 12, 18…***). Astfel, CMMMC ne spune la ce interval se aliniază două ritmuri, cicluri sau multiplicări.

Se poate observa că implementarea algoritmului lui Euclid modifică numerele introduse inițial, astfel că, pentru determinarea celui mai mic multiplu comun este necesară păstrarea unor copii ale valorilor inițiale, pentru a putea fi refolosite la final.

Python – CMMMC

```
a = int(input())
b = int(input())
ca = a
cb = b
while b > 0:
    r = a % b
    a = b
    b = r
print(ca * cb // a)
```

Fișa de lucru 7

Creați dosarul numit *Fisa7*. Rezolvați exercițiile următoare, notând răspunsurile corecte într-un fișier *raspunsuri.docx*.

1. *Descompunere – cel mai mare divizor prim.*

a. Parcurgeți algoritmul de mai jos pentru ***n=84*** și completați un tabel de forma celui alăturat.

n	d	p	dmax
84	...		

b. De ce linia ***if n > 1: dmax = n*** se află **în afara** buclei while ***d * d <= n***?

```
n = int(input())
d = 2;
dmax = 0
while d * d <= n:
    p = 0
    while n % d == 0:
        n = n // d; p += 1
    if p > 0: dmax = d
    d += 1
if n > 1: dmax = n
print(dmax)
```

2. *Descompunere – factorul prim cu puterea maximă.*

a. Ce va afișa algoritmul de mai jos pentru ***n = 84***? Dar pentru ***n = 18***?

b. Urmăriți manual execuția pentru ***n=2*** și explicați de ce tratarea cazului final if ***n*** > 1 and ***1 >= pmax*** este necesară.

```
n = int(input())
d = 2;
pmax = 0
dmax = 0
while d * d <= n:
    p = 0
    while n % d == 0:
        n //= d; p += 1
    if p >= pmax:
        pmax = p; dmax = d
    d += 1
if n > 1 and 1 >= pmax: dmax = n
print(dmax)
```

3. *CMMDC – cu scăderi*

```
a, b = map(int, input().split())
while a != b:
    if a > b: a = a - b
    else:     b = b - a
print(a)
```

a. Completați tabelul pas cu pas (coloane: a | b | a>b? | operație).

a	b	a>b?	Operație
	...		

b. De ce bucla se oprește când ***a == b*** și nu când ***a == 0***? Ce face algoritmul pentru ***a = 0*** și ***b = 5***

4. *CMMDC – cu împărțiri*

```
a, b = map(int, input().split())
while b != 0:
    a, b = b, a % b
print(a)
```

a. Completați tabelul pas cu pas (coloane: a | b | a%b → devine).

a	b	a%b → devine
	...	

b. Ce s-ar întâmpla dacă am înlocui ***a, b = b, a % b*** cu ***a = a%b; a, b = b, a***?

5. *Cel mai mic factor prim al lui* n.
Scrieți un program Python care citește un număr natural **n** și afișează cel mai mare factor prim din descompunerea acestuia. *Exemplu:* n=20 → 5; n=17 → 17.

6. *Calculul CMMMC.*
Scrieți un program Python care citește 2 numere **a** și **b** și calculează cel mai mic multiplu comun al acestora, știind că ***CMMMC(a, b) = a × b ÷ CMMDC(a, b)***. *Exemplu:* pentru a=12 și b=18 → CMMDC=6 → CMMMC=36.

7. *Fracție ireductibilă.*
Citiți două numere naturale **a** și **b** (numărătorul și numitorul unei fracții) și afișați fracția a/b adusă la forma ireductibilă folosind ***CMMDC***. *Exemplu:* 18/24 → 3/4; 36/48 → 3/4.

8. *Factorizare – variante A (până la sqrt(n)) vs. B (până la n)*
Două programe pretind că rezolvă aceeași problemă: determinarea numărului de factori primi distincți ai lui ***n***.

a. Urmăriți manual ambele variante alăturate (A și B) pentru ***n = 12*** și ***n = 13***. Produc același rezultat?

```
# A:
n = int(input())
contor = 0
d = 2
while d * d <= n:
    if n % d == 0:
        contor += 1
        while n % d == 0: n //= d
    d += 1
if n > 1:
contor += 1
print( contor )
```

```
# B:
n = int(input())
contor = 0
d = 2
while d * d <= n:
    if n % d == 0:
        contor += 1
        while n % d == 0: n //= d
    d += 1
if n > 1:
contor += 1
print(contor)
```

b. Estimați numărul de pași pentru ***n = 1.000.003*** (prim). Care variantă e mai eficientă?

9. *CMMDC – variante A (scăderi) vs. B (împărțiri)*

a. Urmăriți manual ambele variante alăturate (A și B) pentru ***a = 100***, ***b = 75***, respectiv pentru ***a = 0*** și ***b = 24***. Produc același rezultat?

b. Estimați pașii pentru ***a = 1.000.000***, ***b = 1***. Care variantă e mai eficientă și de ce?

```
# A:
a = int(input())
b = int(input())

while a != b:
    if a > b:
        a = a - b
    else:
        b = b - a
print(a)
```

```
# B:
a = int(input())
b = int(input())

while b != 0:
    a, b = b, a % b
print(a)
```

10. *Andrei merge la antrenament la fiecare 4 zile, Bogdan la fiecare 6 zile. Un coleg susține că se vor întâlni după* **CMMDC(4,6)=2 zile**. *Identificați ce calcul trebuia făcut și verificați manual rezultatul corect.*

11. Scrieți un program care citește un număr natural *n* nenul și afișează divizorul cel mai mare al lui *n* care este și pătrat perfect. *Exemplu:* pentru n = 20 → 4.

12. *Semafoare sincronizate.*
Semaforul ***A*** se schimbă la fiecare ***45*** s, ***B*** la fiecare ***60*** s. Ambele sunt verzi la ***t=0***. Scrieți un program care determină după câte secunde vor fi din nou verzi simultan. Generalizați: citiți ***n*** (numărul de semafoare) și intervalul de schimbare al fiecăruia, apoi determinați după câte secunde toate semafoarele vor fi verzi simultan. *Indicație*: *CMMMC(a,b,c) = CMMMC(CMMMC(a,b), c).*

Autoevaluare! Ce am învățat?

[CUNOAȘTERE] [ÎNȚELEGERE] [APLICARE] [ANALIZĂ] [EVALUARE] [CREARE]

CUNOAȘTERE

1) Care dintre următoarele reprezintă descompunerea corectă în factori primi a lui **36**?
a) $36 = 2 \cdot 18$ b) $36 = 2^2 \cdot 3^2$ c) $36 = 4 \cdot 9$ d) $36 = 2^3 \cdot 3$
Care dintre următoarele afirmații despre CMMDC și CMMMC este **corectă**?
a) CMMDC(a, b) este întotdeauna mai mare decât a și b; b) CMMDC(a, b) × CMMMC(a, b) = a × b; c) CMMMC(a, b) este întotdeauna mai mic decât a + b; d) CMMDC(a, b) = a dacă a divide pe b, altfel CMMDC(a,b) = b.

ÎNȚELEGERE

2) Ce afișează algoritmul alăturat pentru ***n = 12***?
a) 2
b) 3
c) 4
d) 6

```
n = 12; contor = 0
d = 2
while d * d <= n:
    p = 0
    while n % d == 0:
        n = n // d
        p += 1
    if p > 0:
        contor += 1
    d += 1
if n > 1:
    contor += 1
print(contor)
```

Ce afișează algoritmul lui Euclid de mai jos pentru ***a = 48***, ***b = 36***?
a) 48 b) 12 c) 6 d) 3

```
a = int(input())
b = int(input())
while b > 0:
    a, b = b, a % b
print(a)
```

APLICARE

3) Scrieți un program care citește ***n*** și afișează puterea la care apare factorul ***2*** în descompunerea lui ***n***. Dacă ***2*** nu divide ***n***, afișați ***0***.
Citiți două numere a și b și afișati fracția a/b adusă la forma ireductibilă.

ANALIZĂ

4) Se dau doi algoritmi pentru numărul de factori primi distincți (A: până la $\sqrt{n}$, B: până la ***n***). Identificați o diferență și o asemănare privind eficiența și corectitudinea.

EVALUARE

5) Un elev susține că algoritmul de descompunere în factori primi funcționează corect și pentru ***n = 1***, deoarece bucla while ***d * d <= n*** nu se execută deloc. Este afirmația corectă? Argumentează ce afișează programul pentru ***n = 1*** și dacă rezultatul este cel așteptat.

CREARE

6) Un număr natural **n>1** se numește **„echilibrat”** dacă în descompunerea sa în factori primi toți exponenții sunt egali. (*Exemplu:* $36 = 2^2 \cdot 3^2$ – *echilibrat*; $12 = 2^2 \cdot 3$ – *nu este echilibrat*). Scrieți un program care citește **n** și afișează cel mai mic număr **m > n** care este echilibrat.

Barem: 2p (din oficiu) + 0.5p (1) + 1p (2) + 1p (3) + 1p (4) + 2p (5) + 2.5p (6)

	La finalul lecției, știu...	*Da*	*Nu*	*Nu sunt sigur(ă)*
1.	să aplic algoritmul de descompunere în factori primi.			
2.	să explic principiul algoritmului lui Euclid (scăderi → împărțiri).			
3.	să implementez algoritmul optimizat ***1…*** $\sqrt{n}$ cu afișare ordonată.			
4.	să identific erori logice în algoritmi alternativi și le argumentez.			
5.	să calculez CMMDC cu Euclid și CMMMC cu formula.			

6. Am întâmpinat dificultăți în: …

Nu uitați să vă evaluați comportamentul și activitatea din timpul lecției!
(cu ajutorul fișei de observare a comportamentului și a activității, de la pagina 7)

La finalul lecției, m-am simțit:

Lecția 8 – Baze de numerație

8.1. Sistemul poziţional – analogia fundamentală

Orice număr în baza **b** se scrie ca sumă de **puteri ale lui b**, ponderate cu cifrele sale (0 … b–1).

Baza 10	**Baza 2**
$347 = 3 \cdot 10^2 + 4 \cdot 10^1 + 7 \cdot 10^0$ cifre: 0 – 9	$1011_2 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 11$ cifre: 0 și 1

⚠ **Atenție!** Cifrele într-o bază b sunt întotdeauna cuprinse între 0 și b-1:
- baza 2 → cifre: 0, 1
- baza 8 → cifre: 0, 1, 2, 3, 4, 5, 6, 7
- baza 16 → cifre: 0, 1, ..., 9, A, B, C, D, E, F

8.2. Extragerea ultimei cifre a unui număr n – analogie cu baza 10

Baza 10 Ultima cifră = n % 10 Numărul fără ultima cifră = n // 10	**Baza 2** Ultima cifră = n % 2 Numărul fără ultima cifră = n // 2
`n = 347` `cifra = n % 10  # 7` `n = n // 10  # 34`	`n = 11` `bit = n % 2  # 1` `n = n // 2  # 5`

8.3. Parcurgerea cifrelor de la dreapta la stânga – analogie cu baza 10

Baza 10 Rulare pas cu pas - n = 13	**Baza 2** Rulare pas cu pas - n = 13 $n_2 = 1101$
`n = int(input())` `while n > 0:` `  c = n % 10` `  print(c, end='')` `  n = n // 10` `# Cifrele zecimale vor fi afișate începând cu ultima cifră până la prima cifră`	`n = int(input())` `while n > 0:` `  c = n % 2` `  print(c, end='')` `  n = n // 2` `# Cifrele binare vor fi afișate începând cu ultima cifră binară până la ultima`

8.4. Parcurgerea cifrelor de la stânga la dreapta – Conversie din baza 10 în baza 2

Baza 10 Parcurgem cifrele pornind de la prima cifră, afișând cifrele în ordine, de la stânga la dreapta.	Rulare pas cu pas - n = 13

```
n = int(input())
p = 10**9
while (p > n):
    p = p // 10
while p > 0:
        c = n // p
        print(c, end='')
        n = n % p
        p = p // 10
```

Baza 2
Parcurgem cifrele binare, analog, pornind de la prima cifră binară, afișând cifrele în ordine, de la stânga la dreapta. Programul va afișa de fapt numărul **n scris în baza 2**, cifră cu cifră.

Rulare pas cu pas - n = 13
Afișăm: n_2 = 1101 cifră cu cifră

n	p	n // p (c)	n % p
13	8	1	6
6	4	1	3
3	2	0	1
1	1	1	0
0	0		

```
n = int(input())
p = 2**31  # 2^31
while (p > n):
    p = p // 2
while p > 0:
    c = n // p
    print(c, end='')
    n = n % p
    p = p // 2
```

OBS

De ce pornim de la 2^{31}?
Avem nevoie de o putere a lui ***2*** mai mare decât orice număr cu care vom lucra.
2^{31} = 2.147.483.648 — suficient pentru numerele cu care lucrăm în exemple.
Bucla ***while p > n*** coboară ***p*** până la cea mai mare putere a lui ***2*** care nu depășește ***n***.

EX.

n=13 → pornim de la 2^{31}, coborâm: ..., 16 > 13, 8 ≤ 13 → p=8. Primul bit afișat: 13//8 = 1

8.5. Conversie din baza 10 într-o bază k

Algoritmul este identic cu cel pentru baza ***2***, înlocuind doar ***2*** cu ***k***, fie că este vorba de parcurgerea cifrelor de la dreapta la stânga sau invers. În cazul în care ***k > 10***, resturile de la ***0*** la ***9*** rămân cifre, iar resturile mai mari ca ***10*** vor fi înlocuite de litere. De exemplu, pentru ***k = 16***, în locul resturilor de la ***10*** la ***15*** vom folosi literele de la ***A*** la ***F*** (***10→A***, ***11→B***, ***12→C***, ***13→D***, ***14→E***, ***15→F***). Vom utiliza funcțiile predefinite: ord(c) – codul ASCII al caracterului c. Exemplu: ord('A') → 65. chr(k) – care returnează caracterul corespunzător codului ASCII k. Exemplu: chr(65) → 'A'.

```
# Ex. conversie din baza 10 în baza 16
n = int(input())
p = 1
while p <= n:
    p = p * 16
p = p // 16
while p > 0:
    c = n // p
    if ( c < 10):
        print(c, end = '')
    else:
        print(chr(ord('A') + c - 10), end ='')
    n = n % p
    p = p // 16
```

8.6. Conversie din baza k în baza 10

Pentru a converti un număr din baza ***k*** în baza ***10***, parcurgem cifrele de la stânga la dreapta și construim valoarea într-un număr nou format, inițial 0, după ***formula: nr = nr * k + cifra***.

Rulare pas cu pas n_2 = 1101, k = 2

nr	cifra	nr*k	nr*k + cifra
0	1	0	1
1	1	2	3
3	0	6	6
6	1	12	13
13			

```
n = input().upper()  # Citim n sub formă de șir și
# transformăm literele în majuscule
k = int(input())    # Citim baza
nr = 0          # Rezultatul in baza 10
for c in n:    # Parcurgem fiecare cifră din număr
    if '0' <= c <= '9': # Caracterul c este o posibilă cifră
        cifra = int(c)   # Transformăm caracterul în cifră
    else:           # Caracterul c este literă
        cifra = ord(c) - ord('A') + 10   # Transformă o
# literă în valoarea ei numerică
    nr = nr * k + cifra
print(nr)
```

8.7. Aplicabilitate – clase de probleme

Bazele de numerație nu sunt un subiect pur teoretic – ele stau la temelia informaticii practice. La nivel hardware, procesoarele operează exclusiv în baza 2, iar înțelegerea reprezentării biților este esențială pentru

a lucra cu registre, porți logice și circuite. În programare, operațiile pe ***biți*** (*bitwise*) folosite în criptografie, compresia datelor și grafică se bazează direct pe manipularea reprezentării binare sau hexazecimale. Baza ***16*** este standard în debuggere, editoare hex și protocoale de rețea (adrese MAC, IPv6, culori RGB în HTML). Baza ***8*** apare în sistemele Unix/Linux la permisiunile fișierelor (chmod 755). La nivel algoritmic, problemele de tip „câți biți de 1 are *n*?", „este *n* putere a lui ***2***?", „care este cel mai semnificativ bit?" sunt clasice în interviuri tehnice și competitive programming. Înțelegerea profundă a bazelor permite, de asemenea, detectarea și corectarea erorilor prin coduri redundante (Hamming, paritate) – aplicații critice în comunicații și stocare.

Fișa de lucru 8

Creați dosarul numit *Fisa8*. Rezolvați exercițiile următoare, notând răspunsurile corecte într-un fișier *raspunsuri.docx*.

1. *Conversie zecimal→binar.* Convertiți ***173*** în binar prin împărțiri succesive la ***2***. Scrieți toți pașii parcurși, folosind un tabel. *Exemplu ajutător:* $25_{10} \rightarrow 11001_2$

2. *Conversie binar→zecimal.* Calculați valoarea zecimală a numărului binar $\mathbf{11010110_2}$ folosind formula pozițională (suma puterilor lui ***2***). *Exemplu ajutător:* $1011_2 = 8+0+2+1 = 11$

3. *Gruparea biților.* Convertiți $\mathbf{10110111_2}$ în octal și în hexazecimal prin gruparea biților (câte ***3***, respectiv câte ***4***). *Exemplu ajutător:* $110\ 101_2 \rightarrow 65_8$

4. *Conversie din baza 10 în baza 3.* Se citește un număr natural **n**. Să se afișeze transformarea acestuia în baza 3. *Exemplu*: pentru n = 230 se va afișa 22112_3

a. Modifică "***Conversie din baza 10 în baza 3***" astfel încât să afișezi transformarea numărului ***n*** într-o bază ***k***, k <10. *Exemplu:* pentru n = 12 și k = 2 se va afișa 1100.

b. Modifică "***Conversie din baza 10 în baza 3***" astfel încât să afișezi transformarea numărului ***n*** într-o bază ***16***. *Exemplu:*pentru n = 200 se va afișa $C8_{16}$

```
n = int (input())
p = 3**20
while (p > n):
    p = p // 3
while p > 0:
    c = n // p
    print(c, end='')
    n = n % p
    p = p // 3
```

5. *Cercetează:* Realizați o cercetare folosind un AI (ChatGPT, Claude, Gemini etc.) despre Hexadecimal și rolul său în Informatică. Identificați exemple concrete din viața reală (culori RGB/HEX, memorii, adrese MAC, coduri Unicode etc.) faceți o scurtă prezentare despre cum „vede" calculatorul imaginile și culorile. Puteți include și o aplicație realizată în Python care să efectueze conversii între baze de numerație sau să transforme culori din format RGB în HEX și invers. Verificați rezultatele din cel puțin două surse, citați sursele și nu copiați răspunsul AI fără verificare.

Probleme propuse

1. Scrieți un program Python care citește un număr zecimal ***n*** și afișează suma cifrelor sale în baza ***2*** (adică numărul de biți de valoare ***1***). *Exemplu:* n=23 → $23=10111_2$ → suma biților = 4

2. Scrieți un program care citește un număr zecimal și verifică dacă este putere a lui 2 (un număr este putere a lui ***2*** dacă în binar are exact un singur bit de ***1***). *Exemplu:* 32 → DA (100000_2), 12 → NU (1100_2)

3. *Opțional* **(Conversie octal→binar→hex)** Convertiți $\mathbf{374_8}$ mai întâi în binar (fiecare cifră octală → 3 biți), apoi în hexazecimal (grupe de 4 biți). *Exemplu ajutător:* $52_8 \rightarrow 101\ 010_2 \rightarrow 0010\ 1010_2 \rightarrow 2A_{16}$

4. *Opțional* Calculează suma a două numere binare bit cu bit (adunare în baza 2) folosind doar variabile simple și while. Testează $1011_2 + 0110_2$. *Ex.*: $10001_2 = 17$

5. *Mini-proiect:* **Realizați o cercetare despre sistemul sexagesimal și aplicațiile sale în viața reală.** Proiectul trebuie să conțină: originea sistemului și civilizația care l-a folosit, explicația modului de funcționare al bazei 60 și exemple concrete de utilizare în prezent (măsurarea timpului, unghiuri, coordonate geografice, astronomie etc.). Includeți și o aplicație în Python care să realizeze conversii între ore/minute/secunde sau grade/minute/secunde și o reprezentare numerică obișnuită.

Autoevaluare! Ce am învățat?

[CUNOAȘTERE] [ÎNȚELEGERE] [APLICARE] [ANALIZĂ] [EVALUARE] [CREARE]

CUNOAȘTERE	1) Care este formula pentru a extrage ultima cifră a unui număr ***N*** în baza ***10***? Dar în baza ***2***?
ÎNȚELEGERE	2) Care este restul împărțirii lui ***25*** la ***2*** și ce semnificație are în conversia în binar?
APLICARE	3) Se dă un număr zecimal ***n***. Să se afișeze numărul de cifre binare ***1*** din reprezentarea lui în baza ***2***. Exemplu: pentru n = 13 (= 1101_2), se va afișa 3 (numărul conține 3 biți de 1).
ANALIZĂ	4) Programul următor se execută pentru ***n = 100***. Urmăriți evoluția variabilelor ***n*** și ***p*** la fiecare pas și completați tabelul de mai jos. Ce proprietate matematică a lui 100 calculează programul? Același program aplicat pe ***n = 36*** va afișa același rezultat? Dar pentru ***n = 7***? Justificați. **Pas / n / p**: 1 / 100 / 1; ...
EVALUARE	5) Un coleg susține că pentru a converti un număr din baza ***8*** în baza ***16*** este mai eficient să-l converti mai întâi în baza ***10*** și apoi în baza ***16***. Sunteți de acord? Argumentați cu un exemplu concret.
CREARE	6) Se definește complementul unui număr ***n*** astfel: se convertește ***n*** la baza ***2***, apoi fiecare cifră binară a reprezentării se neagă: dacă era ***1*** este transformată în ***0*** și invers. Ceea ce rezultă se convertește înapoi în baza ***10*** și numărul rezultat este complementul lui ***n***. Se dă un număr ***n***, să se afișeze complementul lui. Exemplu: $45_{(10)}$ se reprezintă în baza doi ca $101101_{(2)}$, a cărui negare este $010010_{(2)}$, care convertită înapoi la baza 10 este $18_{(10)}$, deci se va afișa 18.

```
n = 100
p = 1
while n % 2 == 0:
    n = n // 2
    p = p * 2
print(p)
```

Barem: 2p (din oficiu) + 1p (1) + 1p (2) + 1p (3) + 1p (4) + 2p (5) + 2p (6)

	La finalul lecției, știu...	***Da***	***Nu***	***Nu sunt sigur(ă)***
1.	să explic de ce orice număr se poate scrie ca sumă de puteri ale bazei.			
2.	să convertesc un număr din baza 10 în baza k.			
3.	să convertesc un număr din baza k în baza 10.			
4.	să parcurg cifrele unui număr în orice bază.			
5.	să creez un program care convertește în orice bază k.			

6. Am întâmpinat dificultăți în: …

Nu uitați să vă evaluați comportamentul și activitatea din timpul lecției!
(cu ajutorul fișei de observare a comportamentului și a activității, de la pagina 7)

La finalul lecției, m-am simțit:

Evaluare sumativă - Capitolul 2

Investighează!

(Punctaj maxim: 90 puncte + 10 din oficiu)

Prelucrări ale datelor numerice

Programul de investigat:

```
n = int(input("Introduceti un numar: "))
t=n; s=0; cmax=0; inv=0; nrcif=0
while t > 0:
    c = t % 10; s = s + c
    if c > cmax: cmax = c
    inv = inv * 10 + c
    nrcif = nrcif + 1
    t = t // 10
print("Suma:",s," Max:",cmax)
print("Rasturnat:",inv," NrCifre:",nrcif)
if inv==n:
    print(n,"este palindrom")
else:
    print(n,"nu este palindrom")
p=2**30; t=n
print("In baza 2:", end=" ")
if (n == 0):
    print(0)
while p > n:
    p //= 2
while p > 0:
    print(t//p, end =""); t=t%p; p=p//2
# print('\n')
d=2; prim=1
if n < 2:
    prim=0
while d<n and prim==1:
    if n%d==0:
        prim=0
    d=d+1
if prim==1:
    print("\n",n,"este prim")
else:
    print("\n",n,"nu este prim")
```

1. Identificarea componentelor **(15p)**

Completați tabelul:

Categorie	Variabile	Rol
Date intrare		
Date ieșire		
Date manevră		

2. Urmărirea execuției **(20p)**

Trasați execuția pentru n = 37 (bucla cifrelor):

t	c=t%10	s	cmax	inv	nrcif	t//10
37						

Ieșiri pentru n = 37: ...

3. Analiza erorilor și testarea **(20p)**

a) Analizați comportamentul programului pentru n = 0. Sunt toate rezultatele corecte? Justificați.

b) Propuneți 3 cazuri de test:

Caz de test	Valoare n	Ieșiri așteptate	Ce verifică

c) Identificați un comportament neașteptat (bug) și propuneți corectura.

4. Extinderea programului **(20p)**

a) Modificați programul astfel încât să calculeze cifra minimă în locul cifrei maxime. Precizați liniile modificate și locul inserării.

b) Adăugați calculul cmmdc(n, inv) prin algoritmul lui Euclid.

5. Analiză și Evaluare **(15p)**

Care parte a programului poate fi modificată pentru ca algoritmul să devină mai eficient? Explicați.

Barem: 1. → 15p; 2. → 20p; 3. → 20p; 4. → 20p; 5. → 15p = 90p

Salvați răspunsurile în *investigheaza2.docx* și codul modificat ca *investigat.py* separat.

Verifică-ți cunoștințele!

(Punctaj: 90 de puncte + 10 puncte din oficiu)

Cerință: Rezolvați cerințele de mai jos într-un fișier ***evaluare_sumativa2.docx***. Salvați toate fișierele pe care le creați, în dosarul ***Capitol2***.

CUNOAȘTERE	1) Un număr natural ***n*** este palindrom dacă este egal cu oglinditul său. Care variantă verifică corect această proprietate? A) `n % 10 == n // 10` B) construim `inv` prin `inv*10 + n%10` verificam `inv == n_initial` C) `n // 10 == 0` D) `n % 11 == 0`	(6p)
	2) Definiți noțiunile: *număr prim*, *divizor*, *cmmdc* și *cmmmc*. Dați câte un exemplu numeric pentru fiecare. *Exemplu:* 7 este prim; div(12)={1,2,3,4,6,12}; cmmdc(12,18)=6; cmmmc(4,6)=12.	(8p)
ÎNȚELEGERE	3) Completați spațiile pentru a descrie corect algoritmul de conversie baza 10 → baza 2: a) La fiecare pas se calculează restul n % .....; acesta reprezintă ..... . b) Numărul se înjumătățește prin n = ... c) Resturile se citesc în ordine	(10p)
	4) Explicați de ce, pentru a verifica primalitatea lui ***n***, este suficient să testați divizorii până la $\sqrt{n}$ (nu până la ***n-1***). Ilustrați explicația pentru ***n=36*** și ***n=37***.	(12p)
APLICARE	5) Scrieți cod Python care citește ***n*** și ***k*** și afișează suma cifrelor lui ***n*** în baza ***k***, ***k*** < 10. *Exemplu:* pentru ***n=45*** și ***k=5*** se va afișa suma = 5 ($45_{10} \rightarrow 140_5$, suma = 1 + 4 = 5).	(16p)
EVALUARE	6) Un elev propune verificarea primalității lui ***n*** împărțind la toți divizorii până la ***n-1***. Un alt elev propune împărțirea doar la numerele prime până la $\sqrt{n}$. Comparați cele două variante ca număr de pași pentru ***n=97***. Comparați cele două metode din punctul de vedere al numărului de operații.	(18p)
CREARE	7) Scrieți cod Python care citește ***n*** și afișează descompunerea sa în factori primi, apoi verifică dacă numărul de factori primi distincți este par sau impar. Testați cu ***n = 360*** și ***n = 12***. *Exemplu*: $360 = 2^3 \cdot 3^2 \cdot 5^1$ → 3 factori primi distincți → impar. $12 = 2^2 \cdot 3^1$ → 2 factori → par.	(20p)

Barem: 10p (din oficiu) + 6p (1) + 8p (2) + 10p (3) + 12p (4) + 16p (5) + 18p (6) + 20p (7)

Portofoliu

(Punctaj: 100 puncte)

La finalul acestui capitol, portofoliul personal al elevului trebuie să cuprindă, în dosarul ***Capitol2***, următoarele materiale:

- Dosarul ***Fisa5***, cu fișierele: ***raspunsuri.docx***, ***FisaAutoevaluare5.docx***
- Dosarul ***Fisa6***, cu fișierele: ***raspunsuri.docx***, ***FisaAutoevaluare6.docx***
- Dosarul ***Fisa7***, cu fișierele: ***raspunsuri.docx***, ***FisaAutoevaluare7.docx***
- Dosarul ***Fisa8***, cu fișierele: ***raspunsuri.docx***, ***FisaAutoevaluare8.docx***
- Dosarul ***Proiecte2***, cu fișierele: ***investigheaza2.docx***, ***investigat.py***
- Fișierul ***evaluare_sumativa2.docx***

Modalitate de evaluare:
- Există dosarul ***Capitol2*** cu structura de directoare specificată – **20p**
- Există în dosare:
  - Peste 95% din fișiere – **80p**
  - Între 75% și 94% din fișiere – **60p**
  - Între 50% și 74% din fișiere – **40p**
  - Între 25% și 49% din fișiere – **20p**
  - Sub 25% – **0p**

Capitolul 3 – Modularitate și programare orientată pe obiecte

Lecția 9 – Subprograme: antet, corp, parametri

9.1. Ce este un subprogram? Caracteristici și rol



Un **subprogram** (sau **funcție** în Python) este un bloc de instrucțiuni identificat printr-un ***nume***, care rezolvă o sarcină specifică și poate fi reutilizat în mai multe locuri din program. Un subprogram poate fi apelat ori de câte ori este nevoie, din orice punct al programului.

Rolul subprogramelor este de a permite **organizarea modulară** a codului: un program complex este descompus în subprograme mai mici, fiecare rezolvând o parte a problemei.

Avantajele utilizării subprogramelor:

- ***Reutilizarea codului*** – un subprogram se scrie o singură dată și se apelează de oricâte ori este nevoie
- ***Lizibilitate*** – programul devine mai ușor de citit și înțeles
- ***Testare și depanare mai ușoară*** – fiecare subprogram poate fi testat independent
- ***Colaborare*** – membrii echipei pot lucra independent la subprograme diferite

9.2. Structura unui subprogram în Python

Un subprogram în Python are două componente principale:

Componentă	Descriere
Antetul (header)	Prima linie, conține cuvântul cheie **def**, numele funcției, urmat de o pereche de paranteze rotunde, între care se trec datele cu care lucrează funcția (parametri). **Exemplu:** def min(a, b):
Corpul (body)	Liniile indentate sub antet; conține instrucțiunile care se execută la apelul funcției.
return	Instrucțiunea opțională care permite returnarea unui rezultat. Fără *return*, funcția returnează *None*.

Antetul (sau header) este prima linie a unui subprogram. El conține:

- cuvântul cheie ***def*** – marchează începutul definiției subprogramului;
- **numele** subprogramului – un identificator unic, ales de programator;
- **lista parametrilor** – între paranteze, separați prin virgulă. Lista poate fi și vidă;
- caracterul **:** (două puncte) – marchează sfârșitul antetului.

Sintaxa generală a unui subprogram:

Definim un subprogram prin folosirea cuvântului cheie ***def***:

```
def nume_functie(parametru1, parametru2, ...):
    instructiune1
    instructiune2
    return rezultat  # opțional
```

9.3. Exemplu: suma a două valori

```
def suma(a, b):
    rezultat = a + b
    return rezultat
```

OBS Apelăm subprogramul folosind doar numele acestuia cu lista noastră de valori. Pentru exemplul de mai sus, avem:

```
# Apelul funcției:
s = suma(3, 5)    # apel
print(s)          # → 8
```

Antetul unui subprogram este prima linie a funcției, cea în care sunt precizate numele funcției și parametrii acesteia. El ne transmite cum se numește funcția și ce date primește. Tot ce urmează sub această linie, indentat, face parte din corpul funcției.

9.4. Transmiterea prin intermediul parametrilor

<def/> ***Parametrii*** sunt variabile prin care funcția primește date din exterior. La apel, valorile transmise se numesc ***argumente***. Aceștia permit utilizarea aceleiași funcții pentru valori diferite.

Există situații în care ne așteptăm ca un parametru să aibă o anumită valoare în mod normal, dar vrem să îl putem modifica în situații speciale. De exemplu, nota de la un test va avea în mod normal un punct din oficiu și va fi maximum 10, dar putem avea situații în care vrem să adunăm nota de la test cu nota de la un proiect, caz în care ar putea valora mai puțin, de exemplu ar putea valora maximum 5 puncte, cu 0.5 puncte din oficiu:

```
def calculeaza_nota(nota, oficiu = 1, maxim = 10):
    return min(nota + oficiu, maxim)
```
[1]

Definim valorile implicite ca ***oficiu = 1*** sau ***maxim = 10*** direct din antetul funcției. Dacă vrem să apelăm subprogramul cu valorile implicite putem ignora acești parametri la apelul funcției, nescriindu-i. Altfel le putem da valori ca și până acum:

```
calculeaza_nota(3) → va returna min(3 + 1, 10) = min(4, 10) = 4
calculeaza_nota(10) → va returna min(10 + 1, 10) = min(11, 10) = 10
calculeaza_nota(3, 5) → va returna min(3 + 5, 10) = min(8, 10) = 8
calculeaza_nota(3, 2, 5) → va returna min(3 + 2, 5) = min(5, 5) = 5
```

Subprogramul ***calculeaza_nota*** poate primi între unu și trei parametri. Parametrii care nu sunt specificați la apel vor primi automat valorile implicite definite în antetul funcției, iar ceilalți parametri vor primi valorile transmise, în ordinea în care sunt scrișI, de la stânga la dreapta.

```
calculeaza_nota(nota=3, oficiu=5) → va returna min(3 + 5, 10) = min(8, 10) = 8
calculeaza_nota(3, maxim=5) → va returna min(3 + 1, 5) = min(4, 5) = 4
```

Dacă dorim să modificăm doar anumite valori, fără a le specifica pe toate, putem folosi apelul prin nume, indicând explicit parametrul căruia vrem să îi atribuim o valoare.

9.5. Returnarea rezultatelor

Instrucțiunea ***return*** transmite rezultatul calculat înapoi la locul din program de unde a fost apelat subprogramul. După executarea lui return, subprogramul se încheie imediat.

```
import math
def distanta(x1, y1, x2, y2):
    d = math.sqrt((x2-x1)**2 + (y2-y1)**2)
    return d # ← trimite rezultatul înapoi
r = distanta(0, 0, 3, 4)    # r primește valoarea returnată: 5.0
print(r)                    # → 5.0
```

[1] Funcția min() este o funcție predefinită în Python care returnează valoarea minimă – aceasta face parte din categoria funcțiilor predefinite pentru colecții și va fi studiată în detaliu în lecția următoare.

O funcție poate avea sau nu parametri și poate returna explicit sau nu o valoare.

```
def salut():
    return "Buna"
```

Rezultatul unei funcții este valoarea pe care funcția o produce și o trimite înapoi către locul din program de unde a fost apelată. Această valoare este trimisă folosind instrucțiunea ***return***. Rezultatul poate fi orice: un număr, un text, un tuplu, o altă funcție etc. Dacă nu returnăm nimic Python va presupune automat că am returnat ***None***.

```
def salut():
    print("Buna")
```

Funcția de mai sus va fi apelată simplu cu instrucțiunea ***salut()***. Dacă încerc să salvez rezultatul funcției într-o variabilă x = ***salut()***, codul va afișa pe ecran textul "Buna", iar variabila x va avea valoarea ***None***.

9.6. Apelul unui subprogram

Apelul reprezintă mecanismul prin care un subprogram este invocat (pornit) din programul principal sau din alt subprogram. Sintaxa apelului este:

```
# Apel simplu (fără captarea rezultatului):
salut()
# Apel cu captarea rezultatului într-o variabilă:
aria = calculeaza_aria(5, 3)
# Apel direct în expresie:
print (calculeaza_aria(5, 3))
# Apel cu argumente prin nume:
calculeaza_nota(nota=8, oficiu=2)
```

9.7. Sintaxa pentru definiția și apelul unui subprogram în Python

Sintaxa completă – definiție:

```
def nume_subprogram(param1, param2, param3=valoare_implicita):
    # corpul subprogramului
    instructiune1
    instructiune2
    return rezultat   # opțional
```

Sintaxa completă – apel:

```
# Varianta 1 - argumente poziționale
rezultat = nume_subprogram(arg1, arg2)
# Varianta 2 - argumente prin nume
rezultat = nume_subprogram(param1=arg1, param3=arg3)
# Varianta 3 - apel fără captare (pentru subprograme fără return util)
nume_subprogram(arg1, arg2)
```

9.8. Activitate practică

Rezolvați următoarele exerciții:

1. Scrieți o funcție ***calculeaza_aria***(lungime, latime) care returnează aria unui teren dreptunghiular.
2. Scrieți o funcție ***cel_mai_mare***(a, b, c) care returnează maximul dintre trei numere.
3. Modificați funcția ***calculeaza_nota*** astfel încât să accepte un parametru opțional bonus (implicit 0) care se adaugă la notă.

Reține!

✔ *def* → cuvântul-cheie pentru definirea unui subprogram
✔ ***Antetul*** conține numele și parametrii; ***corpul*** conține instrucțiunile
✔ *return* trimite rezultatul înapoi la locul apelului
✔ ***Parametrii*** cu valori implicite permit apeluri cu argumente opționale
✔ ***Apelul*** prin nume (parametru = valoare) permite specificarea selectivă a argumentelor

Fișa de lucru 9

Creați dosarul numit *Fisa9*. Rezolvați exercițiile următoare, notând răspunsurile corecte într-un fișier *raspunsuri.docx*.

1. Scrie o funcție care primește ca parametri patru valori, a, b, c și d, și returnează elementul cu valoarea maximă dintre ele.

2. Definește un subprogram care afișează mesajul „Bun venit, ”, urmat de numele tău transmis prin parametru.

3. Scrie o funcție care primește două numere și returnează diferența lor în modul.

4. Fie subprogramul de mai jos. Ce va returna dacă este apelat cu *f(5)*?

```
def f(a):
    return a ** 2, a - 3    # o functie poate returna si 2 valori – Python
                            # le împachetează automat într-un tuplu.
```

5. ***Mini-proiect:*** Scrieți o aplicație care calculează costul total al unor cumpărături (preț × cantitate) folosind o funcție cu valoare implicită pentru TVA.

Autoevaluare

Completați răspunsurile într-un fișier *FisaAutoevaluare9.docx*, folosind notații precum *1 – da*, *2 – nu*, *3 – nu sunt sigur*. Salvați fișierul în dosarul *Fisa9*.

	La finalul lecției, știu...	*Da*	*Nu*	*Nu sunt sigur(ă)*
1.	ce este un subprogram (funcție) și care este rolul său în organizarea unui program.			
2.	care sunt componentele unui subprogram: antetul (def, nume, parametri) și corpul (instrucțiunile indentate).			
3.	cum se definește un subprogram în Python.			
4.	ce sunt parametrii.			
5.	cum se apelează un subprogram.			
6.	care sunt avantajele utilizării subprogramelor.			

7. Am întâmpinat dificultăți în: …

Nu uitați să vă evaluați comportamentul și activitatea din timpul lecției!
(cu ajutorul fișei de observare a comportamentului și a activității, de la pagina 7)

La finalul lecției, m-am simțit:

Lecția 10 – Mecanismul de execuție: variabile locale vs variabile globale

10.1. Mecanismul de execuție

La apelul unei funcții, execuția se întrerupe, valorile sunt transmise parametrilor, corpul funcției rulează și returnează rezultatul, apoi programul continuă de unde a rămas. Acest mecanism asigură izolarea funcțiilor între ele și față de programul principal, permițând reutilizarea eficientă a codului.

Mecanismul de execuție a unui subprogram urmează acești pași în ordine:

1. Se oprește execuția programului principal la întâlnirea apelului.
2. Se creează un spațiu de memorie nou (context de execuție) pentru subprogram.
3. Valorile argumentelor se transmit parametrilor funcției.
4. Se execută corpul funcției.
5. Valoarea returnată (return) este trimisă înapoi la locul apelului.
6. Contextul de execuție al subprogramului se eliberează din memorie.
7. Execuția programului principal continuă de unde s-a oprit.

OBS Ne amintim funcția de calcul a unei note, definită în lecția anterioară:

```
def calculeaza_nota(nota, oficiu = 1, maxim = 10):
        return min(nota + oficiu, maxim)

print('Inainte de apel')     # → afișează 'Inainte de apel'
x = calculeaza_nota(8)       # → intră în funcție: nota=8, oficiu=1, maxim=10
                             #   return min(8+1, 10) = 9
print('x =', x)              # → afișează 'x = 9'
```

O parte esențială în înțelegerea funcționării unui program este gestionarea locului în care variabilele sunt create și durata lor de viață. Acest concept se numește vizibilitatea variabilelor. Variabilele pot fi clasificate în două categorii principale: locale și globale.

10.2. Variabile locale

<def/> Variabilele **locale** sunt definite în interiorul unei funcții. Ele ***există doar pe durata execuției funcției*** și sunt inaccesibile din exterior.

```
def test_local():
    mesaj = 'Salut'          # variabilă locală
    print(mesaj)             # OK - în interiorul funcției

test_local()                 # afișează 'Salut'
print(mesaj)                 # EROARE! 'mesaj' nu există în afara funcției
```

OBS În exemplul de mai sus, apelul ***test_local()*** va afișa "Salut", dar instrucțiunea ***print(mesaj)*** de după apel va genera o eroare, deoarece variabila mesaj nu este cunoscută în afara funcției. Parametrii unei funcții sunt, de asemenea, variabile locale.

Caracteristici ale variabilelor locale:

- Sunt vizibile și accesibile doar în interiorul subprogramului în care au fost definite
- Nu pot fi accesate din programul principal sau din alte subprograme
- Parametrii unui subprogram sunt și ei variabile locale
- Două subprograme diferite pot avea variabile locale cu același nume, fără conflict

⚠ **Atenție**

Parametrii unei funcții sunt și ei variabile locale. Ei dispar din memorie când funcția termină execuția.

10.3. Variabile globale

<def/>

Variabilele **globale** sunt definite în afara oricărei funcții. Pot fi **citite** din funcții, dar nu pot fi **modificate** fără a folosi cuvântul-cheie **global**.

```
tva = 19        # variabilă globală
def calculeaza_pret(pret):
    return pret + (pret * tva / 100)    # citire globală – OK
puncte = 0
def adauga_punct():
    global puncte           # declarăm deoarece dorim să MODIFICĂM globala
    puncte = puncte + 1
adauga_punct()
print(puncte)   # → 1
```

OBS

În acest exemplu, funcția ***calculeaza_pret*** utilizează variabila globală ***tva*** pentru a efectua calculul, chiar dacă aceasta nu a fost primită ca parametru. Dacă dorim să modificăm valoarea unei variabile globale în interiorul unei funcții, trebuie să folosim cuvântul-cheie ***global***. Dacă nu am folosi instrucțiunea ***global puncte***, Python ar încerca să creeze o nouă variabilă locală cu același nume, iar variabila globală ar rămâne neschimbată.

Caracteristici ale variabilelor globale:

- Sunt vizibile și accesibile din orice parte a programului, inclusiv din subprograme;
- Pot fi citite direct din interiorul unui subprogram, fără declarație specială;
- Pentru a fi modificate dintr-un subprogram, trebuie declarate explicit cu global;
- Utilizarea excesivă a variabilelor globale poate îngreuna depanarea programului.

10.4. Tabel comparativ: local vs global

Caracteristică	Local vs Global
Unde e definită	**Local**: în interiorul funcției | **Global**: în afara oricărei funcții
Durata de viață	**Local**: dispare după funcție | **Global**: persistă pe toată rularea
Unde e accesibilă	**Local**: doar în funcție | **Global**: din orice parte a codului
Modificare din funcție	**Local**: direct | **Global**: necesită **'global variabila'**
Risc	**Local**: izolată, fără efecte secundare | **Global**: modificabilă accidental

Mecanismul de execuție urmează o logică strictă: atunci când apelăm o funcție, se creează un context nou de execuție (un spațiu în memorie). În acest spațiu sunt încărcați parametrii și variabilele locale. Când se ajunge la ***return***, acest spațiu este șters, iar controlul revine programului principal. Variabilele globale rămân în memorie pe tot parcursul rulării programului.

10.5. Ecranarea variabilelor

<def/>

Atunci când avem o variabilă locală cu același nume ca o variabilă globală, subprogramul o va folosi pe cea locală. Acest fenomen se numește ***ecranare***. Dacă o variabilă locală are același nume cu una globală, funcția o va folosi pe cea locală.

```
x = 100          # variabilă globală
def test():
    x = 5        # variabilă locală – nu afectează variabila globală!
    print(x)     # afișează 5
test()           # afișează 5
print(x)         # afișează 100 (variabila globală e neschimbată)
```

10.6. Cuvântul-cheie nonlocal (funcții imbricate)

Există însă situații, în special în structuri mai complexe, unde definim o funcție în interiorul altei funcții. În acest caz, apare nevoia de a lucra cu un tip intermediar de variabilă, care nu este nici globală, nici locală funcției curente.

Cuvântul-cheie ***nonlocal*** este utilizat în funcțiile imbricate (funcție în funcție) pentru a indica faptul că o variabilă aparține funcției exterioare, dar nu este o variabilă globală.

Când definim o funcție în interiorul altei funcții, ***nonlocal*** permite accesul la variabilele funcției exterioare:

```
def contor():
    n = 0
    def incrementeaza():
        nonlocal n    # accesează 'n' din funcția exterioară
        n = n + 1
    incrementeaza()
    incrementeaza()
    print(n)          # → 2
contor()
```

```
def functie_exterior():
    x = 10
    def functie_interior():
        nonlocal x
        x = 20
        print("Interior x =", x)
    functie_interior()
    print("Exterior x =", x)
functie_exterior()
```

În exemplul alăturat, fără instrucțiunea nonlocal ***x***, funcția din interior ar fi creat o nouă variabilă locală ***x*** cu valoarea ***20***, lăsând variabila ***x*** din exterior neschimbată (ar fi rămas 10). Folosind ***nonlocal***, îi spunem programului să caute variabila ***x*** în contextul imediat superior și să o modifice direct pe aceea.

Acest mecanism este foarte util atunci când vrem să menținem o stare internă a unei funcții (cum ar fi un contor sau o setare) fără a polua spațiul global de variabile al programului, păstrând datele protejate în interiorul funcției principale.

10.7. Activitate practică

1. Scrieți o funcție care calculează temperatura maximă dintr-o zi. Identificați ce variabile ar trebui să fie locale și care ar putea fi globale (Exemplu: pragul de alertă).

2. Explicați de ce codul alăturat generează eroare și cum îl corectați:

```
total = 0
def adauga(val):
    total = total + val
adauga(5)
```

Reține!

- ✔ La apelul funcției se creează un context nou de execuție în memorie
- ✔ Variabilele **locale** există doar pe durata execuției funcției
- ✔ Variabilele **globale** persistă pe toată durata programului și pot fi citite din funcții
  - **global** leagă o variabilă de contextul general al întregului program
  - **nonlocal** leagă o variabilă de contextul funcției părinte, permițând modificarea variabilelor care sunt "prinse" între funcția de bază și cea interioară
- ✔ **Ecranarea** apare când o variabilă locală are același nume cu una globală



Fișa de lucru 10

Creați dosarul numit *Fisa10*. Rezolvați exercițiile următoare, notând răspunsurile corecte într-un fișier *raspunsuri.docx*.

1. Analizați codul de mai jos. Ce va afișa acesta la final?

```
x = 10
def modifica():
    x = 5
modifica()
print (x)
```

2. Scrieți o funcție care să modifice o variabilă globală numită energie, scăzând din ea valoarea unui parametru consum transmis la apel.

3. Ce se întâmplă dacă încercați să accesați o variabilă locală a unei funcții dintr-o altă funcție? Explicați pe baza mecanismului de execuție.

4. Fie subprogramul de mai jos. Ce va afișa dacă este apelat succesiv de două ori?

```
total = 0
def aduna(n):
    global total
    total = total + n
    return total
print (aduna(5))
print (aduna(3))
```

5. Care este diferența principală între o variabilă globală și una declarată ca nonlocal?

6. Scrieți un subprogram numit ***scoala()*** care conține în interior o variabilă elevi = 20 și un alt subprogram ***inscrie()*** care să mărească numărul elevilor cu 1 folosind keyword-ul învățat.

7. Ce va afișa codul de mai jos?

```
def mama():
    nume = "Elena"
    def copil():
        nonlocal nume
        nume = "Andrei"
        print("Interior nume =", nume)
    copil()
    print(nume)
mama()
```

8. ***Mini-proiect:*** Realizați o aplicație de catalog școlar cu variabile globale (lista elevilor) și funcții locale pentru calculul mediei fiecărui elev.



Autoevaluare

Completați răspunsurile într-un fișier *FisaAutoevaluare10.docx*, folosind notații precum *1 – da*, *2 – nu*, *3 – nu sunt sigur*. Salvați fișierul în dosarul *Fisa10*.

	La finalul lecției, știu...	***Da***	***Nu***	***Nu sunt sigur(ă)***
1.	care sunt pașii mecanismului de execuție al unui subprogram.			
2.	ce sunt variabilele locale.			
3.	ce sunt variabilele globale.			
4.	de ce este nevoie de cuvântul-cheie global pentru a modifica o variabilă globală din interiorul unei funcții.			
5.	ce înseamnă ecranarea variabilelor.			
6.	ce este cuvântul-cheie nonlocal și în ce situații se folosește.			
7.	care sunt diferențele esențiale dintre variabilele locale și cele globale (durată de viață, accesibilitate, risc).			

8. Am întâmpinat dificultăți în: …

Nu uitați să vă evaluați comportamentul și activitatea din timpul lecției!
(cu ajutorul fișei de observare a comportamentului și a activității, de la pagina 7)

La finalul lecției, m-am simțit:

Lecția 11 – Subprograme predefinite pentru colecții (Python)

Python include o serie de funcții încorporate care pot fi apelate direct pentru a efectua calcule matematice rapide sau pentru a converti tipurile de date între ele.

Caracteristici și repere de utilizare a unor subprograme predefinite în Python pentru operații matematice uzuale

11.1. Valoarea absolută

Valoarea absolută a unui număr reprezintă distanța față de zero pe axa numerelor, fără semn. Subprogramul predefinit care o calculează este **abs(x)**, disponibil direct în Python fără niciun import.

Caracteristici și repere de utilizare: acceptă argumente de tip ***int*** sau ***float***; returnează întotdeauna o valoare ***pozitivă*** sau ***zero***; utilizări tipice: calculul distanței, validarea erorilor, comparații simetrice

```
abs(-7)          # 7
abs(3.5)         # 3.5
abs(0)           # 0
# Exemplu: distanța dintre două puncte pe o axă
def distanta_1d(a, b):
    return abs(a - b)
distanta_1d(3, 8)     # 5
distanta_1d(8, 3)     # 5  (același rezultat, indiferent de ordine)
```

11.2. Partea întreagă

Partea întreagă a unui număr real reprezintă componenta sa întreagă, fără zecimale. În Python există mai multe subprograme predefinite pentru obținerea ei, cu comportamente diferite.

OBS Pentru operații matematice mai avansate, Python utilizează modulul ***math***. Acesta trebuie importat la începutul programului.

Funcție	Ce face	Exemplu
int(x)	Convertește la întreg; pentru numere reale, trunchiază spre zero.	`int(3.9)` → `3` | `int(-3.9)` → `-3`
math.floor(x)	Rotunjire în jos (returnează partea întreagă)	`math.floor(3.9)` → `3` | `math.floor(-3.1)` → `-4`
math.trunc(x)	Trunchiază spre număr cu zero zecimale (identic cu int())	`math.trunc(3.9)` → `3` | `math.trunc(-3.9)` → `-3`

⚠ Atenție la numere negative

`int(-3.9)` → `-3` (trunchiază spre număr întreg mai MARE)

`math.floor(-3.9)` → `-4` (rotunjire în jos, rezultat mai MIC)

Alegerea corectă depinde de contextul problemei.

```
import math
# Calculul notei școlare (nu se poate da fracțiuni de notă)
nota_calculata = 7.8
print(int(nota_calculata))              # 7 (trunchiază)
print(math.floor(nota_calculata))       # 7 (rotunjeste în jos)
# Câte cutii întregi încap într-un depozit de 47.3 m²?
suprafata = 47.3
suprafata_cutie = 5
print(int(suprafata / suprafata_cutie)) # 9
```

11.3. Rotunjirea

Rotunjirea transformă un număr real într-o valoare aproximată, păstrând un număr dorit de zecimale. Python oferă mai multe subprograme predefinite cu comportamente distincte:

Funcție	Ce face	Exemplu
round(x, n)	Returnează valoarea lui x rotunjită la număr cu ***n*** zecimale. Dacă ***n*** nu este specificat, rezultatul este un număr întreg.	`round(3.567, 2)` → `3.57`
math.floor(x)	Rotunjire întotdeauna în jos	`math.floor(3.9)` → `3` | `math.floor(-3.1)` → `-4`
math.ceil(x)	Rotunjire întotdeauna în sus	`math.ceil(3.1)` → `4` | `math.ceil(-3.9)` → `-3`

Rotunjirea bancară în Python 3

```
round() folosește rotunjirea bancară (la cel mai apropiat număr par):
  round(2.5) → 2   (nu 3!)
  round(3.5) → 4
  round(4.5) → 4   (nu 5!)
```

Acest comportament evită acumularea erorilor de rotunjire în calcule financiare.

```
import math
total_factura = 45.678
print(round(total_factura, 2))                  # 45.68   (standard)
print(math.floor(total_factura*100)/100)        # 45.67   (trunchiere la 2 zec.)
print(math.ceil(total_factura*100)/100)         # 45.68   (rotunjire în sus)
```

11.4. Radicalul

Radicalul (rădăcina pătrată) se calculează cu subprogramul predefinit ***math.sqrt(x)*** din modulul math. Returnează întotdeauna un număr de tip float.

Caracteristici și repere de utilizare:

- Necesită `import math` înainte de utilizare
- Argumentul **x** trebuie să fie **pozitiv** sau **zero** (altfel generează eroare ***ValueError)***
- Returnează ***float*** chiar dacă rezultatul este un număr ***întreg***: `math.sqrt(25)` → `5.0`
- Utilizări tipice: distanța dintre două puncte (teorema lui Pitagora)

```
import math
math.sqrt(16)          # 4.0
math.sqrt(2)           # 1.4142135623730951
math.sqrt(0)           # 0.0
# Exemplu: distanța dintre două puncte în plan (Pitagora)
def distanta_2d(x1, y1, x2, y2):
    return math.sqrt((x2 - x1)**2 + (y2 - y1)**2)
distanta_2d(0, 0, 3, 4)     #  5.0
```

11.5. Conversii

Conversia tipurilor este esențială atunci când primim date de la utilizator (care sunt întotdeauna primite sub formă de șiruri de caractere) și trebuie să le prelucrăm matematic. Python oferă subprograme predefinite pentru cele mai frecvente conversii, disponibile fără import.

Funcție	Ce face	Exemplu
int(x)	Convertește un șir de caractere în număr întreg sau returnează partea întreagă a unui număr real	`int('42') → 42 | int(3.9) → 3`
float(x)	Convertește o valoare într-un număr real	`float('3.14') → 3.14 | float(5) → 5.0`
str(x)	Convertește o valoare într-un șir de caractere	`str(42) → '42' | str(3.14) → '3.14'`
bool(x)	Convertește o valoare într-o valoare logică	`bool(0) → False | bool(5) → True`

Caracteristici și repere de utilizare:

- *int()* aplicat pe un șir funcționează doar dacă șirul reprezintă un număr întreg valid (șirul de caractere se poate scrie și între ghilimele):
  `int('3.14') → eroare deoarece sirul nu contine un număr intreg valid`
- *float()* aplicat pe un șir acceptă atât numere întregi cât și reale: `float('3') → 3.0`
- *str()* funcționează pe orice tip de date, util pentru concatenare și afișare

```
# Citire și conversie date de la utilizator
varsta = int(input('Varsta: '))   # '18' → 18
nota   = float(input('Nota: '))   # '9.5' → 9.5
# Conversie pentru afișare concatenată
mesaj = 'Nota ta este: ' + str(nota)    # float → str
print(mesaj)    # Conversie între tipuri numerice
x = 7 / 2       # 3.5 (float)
y = int(7 / 2)  # 3   (int, prin trunchiere)
```

11.6. Subprograme predefinite pentru colecții de valori

Atunci când lucrăm cu seturi de date, cum ar fi listele, avem nevoie de mecanisme rapide pentru a extrage informații despre întregul grup de elemente. Python oferă subprograme predefinite care scutesc programatorul de scrierea unor bucle pentru calcule simple.

Determinarea numărului de elemente – len()

Subprogramul **len(colectie)** returnează numărul de elemente din colecție.

Caracteristici:

- funcționează pe liste, tupluri, șiruri și orice colecție Python;
- returnează int;
- colecția vidă returnează 0.

```
note = [10, 8, 9, 7, 6]
len(note)              # 5
len([])                # 0 (lista vidă)
len('informatica')     # 11 (returnează lungimea șirului de caractere)
```

Determinarea minimului – min()

Subprogramul **min(colectie)** returnează cea mai mică valoare din colecție.

Caracteristici:

- colecția trebuie să fie nevidă (altfel se generează eroarea ValueError);
- elementele trebuie să fie comparabile între ele;
- poate primi și argumente separate: min(a, b, c).

```
note = [10, 8, 9, 7, 6]
min(note)              # 6
min(3, 1, 7, 2)        # 1 (argumente separate, fără listă)
```

Determinarea maximului – max()

Subprogramul **max(colectie)** returnează cea mai mare valoare din colecție.

Caracteristici:

- identice cu min();
- poate primi și argumente separate: max(a, b, c).

```
note = [10, 8, 9, 7, 6]
max(note)              # 10
max(3, 1, 7, 2)        # 7
```

Determinarea sumei valorilor – sum()

Subprogramul **sum(colectie)** returnează suma tuturor elementelor din colecție.

Caracteristici:

- funcționează pe colecții de numere;
- lista vidă returnează 0;
- acceptă un al doilea argument opțional ca valoare de pornire: sum(lista, start).

```
note = [10, 8, 9, 7, 6]
sum(note)       # 40
sum([])         # 0 (lista vidă)
sum(note, 100)  # 140 (pornește de la 100)
```

11.7. Exemplu: analiza unei liste de note din catalog

```
import math
note = [10, 8, 9, 7, 6, 5, 9, 8]
def analizeaza_catalog(note):
    numar  = len(note)                          # nr. de elemente
    minima = min(note)                          # minimul
    maxima = max(note)                          # maximul
    total  = sum(note)                          # suma
    media  = round(total / numar, 2)            # rotunjire
    extreme = math.sqrt(maxima - minima)        # radical
    return minima, maxima, media, extreme
mn, mx, med, ab = analizeaza_catalog(note)
```

OBS Aceste funcții sunt optimizate să parcurgă colecția și să returneze rezultatul instantaneu, fiind mult mai eficiente decât parcurgerea manuală a listei folosind o structură repetitivă.

Fișa de lucru 11

Creați dosarul numit *Fisa11*. Rezolvați exercițiile următoare, notând răspunsurile corecte într-un fișier *raspunsuri.docx*.

1. Analizați codul de mai jos. Ce va afișa acesta și de ce?

```
x = 15.7
print(int(x))
print(round(x))
```

2. Scrieți un subprogram care primește ca parametru o listă de prețuri și returnează valoarea celui mai scump produs.

3. Ce credeți că s-ar întâmpla dacă apelați funcția ***sum()*** pe o listă care conține și numere și șiruri de caractere? Verificați și apoi explicați raspunsul.

4. Fie subprogramul de mai jos. Ce va afișa dacă lista primită este [10, 20, 30, 40]?

```
def proceseaza(lista):
    return (max(lista) + min(lista)) / 2
print(proceseaza([10, 20, 30, 40]))
```

5. Care este diferența dintre utilizarea funcției ***int()*** și a funcției ***round()*** atunci când argumentul este un număr real pozitiv?

6. Scrieți un subprogram numit ***calculeaza_medii()*** care primește o listă de note, afișează numărul de note introduse (***len***) și media acestora (***sum / len***).

7. Ce va afișa codul de mai jos?

```
import math
x = -25
print(abs(x))
print(math.sqrt(abs(x)))
```

8. Calculați distanța dintre punctele A(1, 2) și B(4, 6) folosind ***math.sqrt()*** și ***abs()***.

9. ***Mini-proiect:*** Creați un calculator de conversii valutare – funcție cu suma și rata implicită EUR/RON = 5.0, care afișează rezultatul rotunjit la 2 zecimale folosind round().



Autoevaluare

Completați răspunsurile într-un fișier *FisaAutoevaluare11.docx*, folosind notații precum *1 – da*, *2 – nu*, *3 – nu sunt sigur*. Salvați fișierul în dosarul *Fisa11*.

	La finalul lecției, știu...	***Da***	***Nu***	***Nu sunt sigur(ă)***
1.	ce este valoarea absolută, cum se calculează cu abs().			
2.	diferența dintre int(), math.floor() și math.trunc() pentru obținerea părții întregi.			
3.	cum funcționează round(), math.floor() și math.ceil() pentru rotunjire.			
4.	cum se calculează radicalul cu math.sqrt().			
5.	ce subprograme predefinite există pentru conversii.			
6.	ce subprograme predefinite există pentru colecții.			

7. Am întâmpinat dificultăți în: …

Nu uitați să vă evaluați comportamentul și activitatea din timpul lecției!
(cu ajutorul fișei de observare a comportamentului și a activității, de la pagina 7)

La finalul lecției, m-am simțit:

Lecția 12 – Introducere în POO: clase, obiecte și instanțiere

Până acum am scris cod procedural, adică o secvență de instrucțiuni executate una după alta. Pe măsură ce programele devin mai complexe, gestionarea datelor și a funcțiilor asociate acestora devine dificilă. Programarea Orientată pe Obiecte (POO) ne permite să modelăm realitatea prin crearea de entități numite ***obiecte***, care combină ***date*** (atribute) și ***subprograme*** (metode).

12.1. Clasa

O clasă este un șablon sau un plan de construcție. Gândește-te la ea ca la o schiță pentru o casă. Un obiect este instanța concretă a acelei clase, adică casa propriu-zisă construită după schiță. Poți construi oricâte obiecte dorești pe baza aceleiași clase, fiecare putând avea propriile date specifice.

Definirea unei clase

<def/> **Clasa** este un șablon care definește structura și comportamentul unui tip de obiecte. Ea specifică ce date poate stoca un obiect și ce operații poate efectua, fără a crea obiecte concrete. Clasele se definesc cu ajutorul cuvântului-cheie ***class***.

Caracteristici ale unei clase:

- Se definește o singură dată, dar poate fi folosită pentru a crea oricâte obiecte
- Conține două categorii de membri: ***date*** (atribute) și ***subprograme*** (metode)
- În Python se definește cu cuvântul-cheie ***class***, urmat de numele clasei (cu majusculă)
- Analogie: clasa este un șablon după care se creează obiecte. Așa cum un tipar/o schiță pentru o casă nu este o casă, o clasă nu este un obiect, ci descrie cum vor arăta obiectele create pe baza ei.

```
class Elev:              # definirea clasei - ȘABLONUL
    pass                 # (corp minim, fără membri deocamdată)
```

12.2. Membrii clasei – date și metode

<def/> **Membrii clasei** sunt componentele definite în interiorul unei clase. Există două categorii:

Date (atribute):

<def/> **Datele** sunt variabilele care aparțin unui obiect și stochează informații despre acesta. Se definesc în constructorul clasei folosind ***self.nume_atribut = valoare***.

- Fiecare obiect are propriile valori ale atributelor, independente de alte obiecte;
- Sunt accesate din exterior prin sintaxa ***obiect.atribut***.

Metode:

<def/> **Metodele** sunt subprogramele (funcțiile) definite în interiorul clasei, care descriu comportamentul obiectelor. Primul lor parametru este întotdeauna ***self*** – referința la obiectul curent.

- Metodele pot accesa și modifica atributele obiectului prin self;
- Sunt apelate din exterior prin sintaxa ***obiect.metoda(argumente)***.

12.3. Constructorul clasei (__init__) și operatorul self

<def/> **Constructorul**[2] este o metodă specială a clasei, care se execută **automat** în momentul instanțierii/ creării unui obiect – adică imediat ce scriem obiect = NumeClasa(argumente). Rolul său este de a inițializa starea internă a obiectului prin atribuirea unor valori specifice atributelor sale încă din momentul creării.

[2] În POO, **constructorul** este mecanismul prin care un obiect nou creat primește valorile inițiale ale datelor sale. Fără constructor, obiectul ar exista în memorie, dar ar fi "gol" — fără nicio dată asociată.

Operatorul self:

self este modul prin care un obiect se referă la el însuși în interiorul metodelor. Când apelăm o metodă a unui obiect, Python transmite automat acel obiect ca prim argument (self), astfel încât metoda știe despre care obiect este vorba concret.

```
class Animal:
    def __init__(self, nume, varsta):
        # inițializarea atributelor specifice fiecărei instanțe
        self.nume = nume      # atribut - persistă în memorie
        self.varsta = varsta # atribut - persistă în memorie
    def descriere(self, x):
        # self.varsta = atribut al obiectului
        # x = parametru extern, există doar pe durata acestei metode
        return self.varsta, x
# Crearea obiectelor (instanțiere) prin apelarea constructorului
caine = Animal("Bobby", 2)
pisica = Animal("Katy", 5)

print(caine.nume)  # Accesăm atributul nume al primului obiect - Bobby
print(pisica.nume) # Accesăm atributul nume al celui de-al doilea obiect - Katy
print(caine.descriere(7))     # (2, 7)
```

- ***self.atribut*** înseamnă 'atributul meu, al obiectului curent';
- Atributele (self.nume, self.varsta) sunt variabile „atașate" fiecărui obiect în parte și persistă în memorie cât timp obiectul există;
- Parametrii obișnuiți ai metodelor (ex: x) sunt independenți de starea obiectului și există doar pe durata execuției metodei.

Metoda de inițializare, atunci când este definită, se numește ***__init__***. Practic, constructorul ne scutește de a scrie caine.nume = 'Bobby' după crearea obiectului, realizând configurarea datelor într-un singur pas. În acest exemplu, ***self*** reprezintă obiectul curent care este utilizat. Observă cum ***x*** și ***self.varsta*** sunt variabile complet separate.

Izolarea obiectelor în memorie

Fiecare obiect ocupă propriul spațiu în memorie. Modificarea unui atribut al obiectului caine nu afectează sub nicio formă atributul corespunzător al obiectului pisica. Această izolare permite gestionarea a mii de entități diferite folosind același șablon: clasa.

```
class Elev:
    def __init__(self, nume, nota_medie):  # constructor
        self.nume = nume                    # dată (atribut)
        self.nota_medie = nota_medie        # dată (atribut)
    def saluta(self):                       # metodă
        print(f'Buna! Sunt {self.nume}.')
    def este_promovat(self):                # metodă
        return self.nota_medie >= 5
```

Metoda ***__init__*** este un constructor special care se apelează automat în momentul creării unui obiect și este folosită pentru a inițializa atributele acestuia.

12.4. Metode speciale (Dunder Methods)

Pe lângă metodele obișnuite, Python oferă **metode speciale** numite și **dunder methods** (dunder fiind abrevierea de la double underscore). Acestea încep și se termină cu caracterul underscore_, scris de două ori și au roluri predefinite în limbaj. Pot fi apelate direct prin operatorul sau funcția corespunzătoare, fără a le numi explicit.

Metodă specială	Rol și declanșator
__init__(self, ...)	Constructorul – se apelează automat la instanțiere: Animal('Bobby', 2)
__str__(self)	Specifică ce text să fie afișat când obiectul este convertit la șir de caractere: print(obiect)
__eq__(self, alt)	Definește operatorul == (equal): obiect1 == obiect2
__lt__(self, alt)	Definește operatorul < (less than): obiect1 < obiect2
__le__(self, alt)	Definește operatorul <= (less than or equal to): obiect1 <= obiect2
__gt__(self, alt)	Definește operatorul >(greater than): obiect1 > obiect2
__ge__(self, alt)	Definește operatorul >=(greater than or equal to): obiect1 >= obiect2

```
class Animal:
    def __init__(self, nume, varsta):
        self.nume   = nume
        self.varsta = varsta
    def __str__(self):              # apelat de str()
        return f'{self.nume} are {self.varsta} ani.'
    def __eq__(self, alt_animal):         # apelat de ==
        # Considerăm că două animale sunt la fel dacă au același nume
        return self.nume == alt_animal.nume
    def __lt__(self, alt_animal):         # apelat de <
        # Comparăm animalele după vârstă
        return self.varsta < alt_animal.varsta
# Creăm trei obiecte (instanțe ale clasei)
pufu = Animal('Pufu', 3)
pufi = Animal('Pufu', 5)
bobi = Animal('Bobi', 2)
# Afișăm obiectele (print apelează automat funcția str())
print(pufu)       # Pufu are 3 ani.     (apelează __str__)
print(pufi)       # Pufu are 5 ani.     (apelează __str__)
print(bobi)       # Bobi are 2 ani.     (apelează __str__)
# Testăm egalitatea (__eq__)
print(pufu == pufi)      # True                (apelează __eq__)
# Testăm ordinea (__lt__)
print(bobi < pufu)       # True  (2 < 3)       (apelează __lt__)
print(pufi < bobi)# False (5 < 2)       (apelează __lt__)
```

Funcțiile pot fi apelate cu denumirea lor, dar și direct, folosind simbolul respectiv (*de exemplu*, *print(obiect)* apelează automat *metoda __str__*).

12.5. Obiectul



Obiectul este o instanță concretă creată pe baza unei clase. Clasa este șablonul, iar obiectul este produsul creat după acel șablon – cu propriile valori ale datelor, dar aceleași metode ca toate celelalte obiecte din clasa respectivă.

Caracteristici ale unui obiect:

- Are propriile valori ale atributelor, independente de alte obiecte din aceeași clasă;
- Împărtășește metodele definite în clasă cu toate celelalte obiecte;
- Poate fi stocat în variabile, liste, sau transmis ca argument unui subprogram;
- Analogie: clasa ***Elev*** este tiparul, iar ***elev1*** și ***elev2*** sunt elevi concreți, fiecare cu numele și nota lui.

```
elev1 = Elev('Ana', 8.5)    # obiect - instanță concretă
elev2 = Elev('Mihai', 4.2)  # alt obiect - instanță diferită
# elev1 și elev2 au aceleași metode (saluta, este_promovat)
# dar valori diferite ale atributelor (nume, nota_medie)
```

Studiu de caz: Gestionarea unui cabinet veterinar

Imaginează-ți că ești la recepția unei clinici veterinare. În loc să scrii pe foi volante datele fiecărui cățel, ai un formular standard. Acest formular este **Clasa**. Când vine un stăpân cu un animal, tu completezi un formular nou. Acea foaie completată cu datele lui „Grivei" este **Obiectul**.

În loc să folosim zeci de variabile separate, creăm un șablon:

```
class Animal:
    nume = 'Grivei'
    tip = 'caine'
    stapan = 'Ana'
    varsta = 4
# Creăm un pește numit Bobi
bobi = Animal()
bobi.nume = 'Bobi'
bobi.tip = 'Peste'
print(bobi.nume, bobi.tip, bobi.stapan)
```

12.6. Biblioteci



Bibliotecile (sau modulele) sunt colecții de clase, subprograme și constante predefinite, grupate tematic, care extind funcționalitățile de bază ale limbajului Python. Ele permit reutilizarea codului scris de alți programatori.

Caracteristici ale bibliotecilor:

- Se importă cu instrucțiunea `import` sau `from ... import ...`
- Python vine cu mai multe biblioteci standard bogate (math, datetime, random, os etc.)
- Există și biblioteci externe care se instalează separat (numpy, pandas, tkinter etc.)

```
import math                          # importă întreaga bibliotecă math
from datetime import date            # importă doar clasa date din datetime
# Acum putem folosi clasa date și subprogramele din math:
azi = date.today()                   # folosim clasa date
r = math.sqrt(16)                    # folosim subprogramul sqrt din math
```

12.7. Instanțierea unei clase predefinite

Instanțierea este procesul prin care se creează un obiect concret pe baza unei clase. Se realizează prin apelul clasei ca și cum ar fi un subprogram, transmițând argumentele necesare constructorului:

Pentru a crea un obiect dintr-o clasă, realizăm un proces numit ***instanțiere***. În Python, folosim paranteze după numele clasei pentru a crea instanța.

Metodele de instanță au, prin convenție, primul parametru numit self – referință către instanța respectivă. Referința poate avea orice denumire, self este doar o convenție.

```
# Sintaxa generală:
# obiect = NumeClasa(argument1, argument2, ...)
# Instanțierea clasei Elev (definită de noi):
elev1 = Elev('Ana', 8.5)
# Instanțierea clasei date din biblioteca datetime:
from datetime import date
azi = date.today()                   # metodă de clasă care creează un obiect
nastere = date(2008, 3, 15)          # instanțiere cu argumente: an, lună, zi
# Instanțierea clasei list (predefinită):
note = list()                        # sau: note = []
siruri = list('abc')                 # → ['a', 'b', 'c']
```

12.8. Accesul la membrii unui obiect

Accesul la membrii unui obiect se realizează prin operatorul punct (.), care permite atât citirea/modificarea atributelor, cât și apelul metodelor:

Accesul la date (atribute):

```
elev1 = Elev('Ana', 8.5)
# Citirea unui atribut:
print(elev1.nume)           #  Ana
print(elev1.nota_medie)     #  8.5
# Modificarea unui atribut:
elev1.nota_medie = 9.0
print(elev1.nota_medie)     #  9.0
```

Accesul la metode:

```
# Apelul unei metode:
elev1.saluta()                      # Buna! Sunt Ana.
rezultat = elev1.este_promovat()    # True
# Accesul la membrii clasei datetime.date:
from datetime import date
azi = date.today()
print(azi.year)        # atribut - anul curent
print(azi.month)             # atribut - luna curentă
print(azi.strftime('%d/%m/%Y'))   # metodă - formatare dată
```

Sintaxă	Descriere
obiect.atribut	Citirea valorii unui atribut al obiectului
obiect.atribut = val	Modificarea valorii unui atribut al obiectului
obiect.metoda()	Apelul unei metode fără argumente suplimentare
obiect.metoda(arg1, arg2)	Apelul unei metode cu argumente

Reține!

- ✔ **Clasa** – șablonul care definește structura și comportamentul unui tip de obiect
- ✔ **Membrii clasei**: date (atribute) și metode (subprograme)
- ✔ **Obiectul** – instanța concretă creată pe baza clasei, cu propriile valori ale atributelor
- ✔ **Biblioteci** – colecții de clase și subprograme predefinite (import / from ... import)
- ✔ **Instanțierea** – crearea unui obiect: obiect = NumeClasa(argumente)
- ✔ **Accesul la membri** – prin operatorul punct: obiect.atribut și obiect.metoda()

Fișa de lucru 12

Creați dosarul numit *Fisa12*. Rezolvați exercițiile următoare, notând răspunsurile corecte într-un fișier *raspunsuri.docx*.

1. Care este diferența principală dintre o clasă și un obiect? Folosește o analogie din viața reală.

2. Ce rol are parametrul self în interiorul unei metode?

a. Reprezintă o variabilă globală accesibilă de oriunde.

b. Este o referință către instanța curentă a clasei.

c. Este un cuvânt-cheie care oprește execuția codului.

3. Dacă definim o clasă Caine cu un atribut nume, scrie codul necesar pentru a crea două obiecte diferite (doi câini cu nume diferite).

4. Identifică eroarea în următorul cod și explică de ce nu va rula:

```
class Masina:
    def __init__(nume):
        self.nume = nume
        auto = Masina('Dacia')
```

5. Definiți o clasă numită Elev care să aibă atributele nume și medie. Adăugați o metodă numită este_premiant() care să returneze *True* dacă media este mai mare sau egală cu 9, altfel *False*.

6. Definiți o clasă Masina cu datele: marca, model, an_fabricatie. Adăugați o metodă varsta() și o metodă descriere() care afișează toate datele.

7. Creați 3 obiecte de tip Masina. Accesați atributele și metodele fiecăruia. Modificați an_fabricatie al unui obiect și verificați că celelalte nu s-au schimbat.

8. ***Mini-proiect:*** Extindeți clasa Catalog cu metodele nota_minima(), nota_maxima() și este_promovat(prag=5). Creați două obiecte Catalog pentru materii diferite.



Autoevaluare

Completați răspunsurile într-un fișier *FisaAutoevaluare12.docx*, folosind notații precum *1 – da*, *2 – nu*, *3 – nu sunt sigur*. Salvați fișierul în dosarul *Fisa12*.

	La finalul lecției, știu...	***Da***	***Nu***	***Nu sunt sigur(ă)***
1.	ce este o clasă.			
2.	ce sunt membrii clasei.			
3.	ce este constructorul (__init__).			
4.	ce este self.			
5.	ce sunt metodele speciale (dunder methods) și cum funcționează.			
6.	ce este un obiect.			
7.	ce sunt bibliotecile.			
8.	ce este instanțierea și cum se realizează.			
9.	cum se accesează membrii unui obiect.			
10.	Am întâmpinat dificultăți în: …			

Nu uitați să vă evaluați comportamentul și activitatea din timpul lecției!
(cu ajutorul fișei de observare a comportamentului și a activității, de la pagina 7)

La finalul lecției, m-am simțit:



Evaluare sumativă - Capitolul 3

Verifică-ți cunoștințele!

(Punctaj: 90 de puncte + 10 puncte din oficiu)

Cerință: Rezolvați cerințele de mai jos într-un fișier ***evaluare_sumativa3.docx***. Salvați toate fișierele pe care le creați, în dosarul ***Capitol3***.

1. Alegeți situația care descrie cel mai bine diferența dintre o variabilă locală și una globală în Python:

a. Variabila locală funcționează peste tot, cea globală doar în funcții.

b. Variabila globală e declarată în afara funcțiilor, cea locală e declarată în funcție și dispare după apel.

c. Diferă doar prin scrierea cu majuscule, variabilele locale necesitând scrierea cu litere mici.

d. Cea locală reține doar valori întregi, cea globală poate reține orice tip de valoare.

2. Analizați codul din imaginea alăturată. Ce se va afișa și de ce?

```
class Fractie:
    def __init__(self, numarator, numitor):
        self.numarator = numarator
        self.numitor = numitor
f1 = Fractie(1, 2)
f2 = f1
f2.numarator = 3
f2.numitor = 4
print(f1.numarator, f1.numitor)
```

3. Asociați fiecare funcție/concept din coloana **A** cu descrierea/efectul corespunzător din coloana **B**.

Funcție/concept	Descriere/efect
1. math.ceil(4.1)	a. Procesul de creare a unui obiect concret pe baza unei clase.
2. int(9.8)	b. Determină valoarea absolută a unui număr (elimină semnul minus).
3. Instanțiere	c. Transformă numărul în întreg prin tăierea zecimalelor, returnând 9.
4. abs(x)	d. Calculează radicalul (rădăcina pătrată) a unui număr pozitiv.
5. math.sqrt(x)	e. Rotunjește numărul prin adaos la întregul imediat următor, returnând 5.

4. a. Scrieți definiția completă a unui subprogram numit **sub**, care primește doi parametri, **a** și **b** (numere naturale). Subprogramul extrage cifra unităților din **a**, o adaugă la sfârșitul lui **b** și returnează valoarea modificată a lui b.

b. Scrieți un program care citește de la tastatură un număr natural **n** cu un număr par de cifre. Folosind exclusiv variabile simple și apeluri ale subprogramului **sub**, programul trebuie să verifice dacă prima jumătate a numărului și a doua jumătate a sa sunt identice. Programul va afișa **DA** sau **NU**.

- Exemplul 1: Pentru n = 12051205, prima jumătate este 1205, iar a doua este 1205. Cele două sunt identice, deci se va afișa DA.
- Exemplul 2: Pentru n = 12344321, prima jumătate este 1234, a doua este 4321, se va afișa **NU**.

Barem de evaluare (100 puncte) oficiu – **10p**, **SUBIECTUL 1-b (10p)**, **SUBIECTUL 2 (15p)**: afișare valori – **8p**, justificare - **7p, SUBIECTUL 3 – (15p = 5x3p)** 1-e, 2-c, 3-a, 4-b, 5-d, **SUBIECTUL 4 – (50p)**: **Cerința a)**: **20p** (**6p** antet corect, **10p** algoritmul de mutare a cifrei, **4p** returnarea rezultatului corect), **Cerința b) (30p)**: Citire date, utilizarea exclusivă a variabilelor simple, apelarea corectă a funcției sub pentru formarea numărului, corectitudinea prelucrării jumătăților, afișarea corectă a mesajului DA/NU, sintaxă corectă, corectitudine globală).



Proiect

(100 puncte)

„Calculator Modular pentru Conversii”

Dezvoltă un program Python ***calcule.py***, structurat modular, care afișează un meniu interactiv pentru realizarea conversiilor de mai jos. Reflecție: Justifică în scris rolul variabilelor locale utilizate în funcții.

MENIU	CERINȚE TEHNICE
1. **EURO → RON**: (cursul zilei va fi citit de la tastatură). Rezultatul va fi rotunjit la 2 zecimale. 2. **Celsius → Fahrenheit**. 3. **Centimetri → Inchi**: Distanța va fi procesată în valoare absolută. 4. **Ieșire** (oprește execuția).	Fiecare conversie va fi calculată într-un **subprogram (funcție) propriu**, cu parametri și returnare de rezultat. Meniul va rula într-o buclă while până la selectarea opțiunii 4. Se utilizează funcțiile de conversie a tipurilor de date (float(), int()).

Barem de evaluare (100 puncte): I. Implementare Tehnică (30p): 3×10p fiecare conversie care funcționează corect. **II. Cod (40p):** 3 subprograme × **5p = 15p, 20p** – Crearea meniului, funcționare corectă, **5p** – Calitatea codului. **III. Teorie & Prezentare (30p): 10p** – Întrebarea de reflecție, **10p** – Claritatea discursului și utilizarea riguroasă a termenilor, **10p** – Creativitate / Atenție la detalii (personalizarea meniului, tratarea opțiunilor invalide introduse de utilizator).

(**Recomandări:** se lucrează în echipe de 2-3 elevi. Fiecare echipă acordă celorlalte echipe, pe baza unei fișe de evaluare, punctaj, iar punctajul final va fi media aritmetică a punctajelor obținute.)



Investighează!

(100 puncte)

Studiu de caz: Stația Meteo

Documentați, în fișierul **timp.docx**, următoarea situație: codul de mai jos prelucrează datele unei stații meteo folosind subprograme și funcții predefinite. Sarcini de investigat:

- Justifică diferența care apare între round() și math.floor() pentru valoarea -5.71? **(30p)**
- Identifică: antet, corp funcție, un parametru. **(15p)**
- Analizează de ce modificările aplicate variabilei *temperatura* în interiorul funcției nu afectează variabila *temp_globala*? **(15p)**
- Identifică biblioteca folosită și metoda apelată. Investighează și folosește alte două funcții din această bibliotecă. (ex: pentru radical și rotunjire prin adaos). **(40p)**

```
import math
temp_globala = 21.65
def proceseaza_date(temperatura):
    temp_abs = abs(temperatura)
    print(f"Valoare absolută: {temp_abs}")
    print(f"Rotunjiri: round={round(temperatura)}, "
          f"floor={math.floor(temperatura)}")
proceseaza_date(-5.71)
print(f"Temperatura globală a rămas: {temp_globala}")
```



Portofoliu

(100 puncte)

La finalul acestui capitol, portofoliul personal al elevului trebuie să cuprindă, în dosarul ***Capitol3***, următoarele materiale: dosarele ***Fisa9***, ***Fisa10***, ***Fisa11***, ***Fisa12*** (fiecare cu fișierele ***raspunsuri.docx***, ***autoevaluare.docx***), dosarul ***Proiecte3***, cu fișierele ***timp.docx*** și ***calcule.py*** și fișierul ***evaluare_sumativa3.docx***.

Modalitate de evaluare: Există dosarul ***Capitol3*** cu structura de directoare specificată – **20p**

Există în dosare:
- Peste 95% din fișiere – **80p**
- Între 75% și 94% din fișiere – **60p**
- Între 50% și 74% din fișiere – **40p**
- Între 25% și 49% din fișiere – **20p**
- Sub 25% – **0p**

Capitolul 4 – Interfețe grafice și fișiere text

Lecția 13 – Biblioteca Tkinter: crearea ferestrelor și managementul așezării

Introducere în Tkinter și realizarea primelor ferestre grafice

„Programele bune sunt scrise pentru oameni care le citesc și doar accidental pentru calculatoare care le execută.” – ***Harold Abelson***

13.1. Interfața grafică și biblioteca Tkinter

<def/>

În programare, aplicațiile pot funcționa în două moduri principale:

- *în linie de comandă (CLI – Command Line Interface)*;
- *cu interfață grafică (GUI – Graphical User Interface)*.

Interfața grafică permite utilizatorului să interacționeze cu programul prin ferestre, butoane, casete de text, meniuri și alte elemente vizuale.

În limbajul ***Python***, una dintre cele mai utilizate biblioteci pentru realizarea interfețelor grafice este ***Tkinter***.

Tkinter este modulul din bliblioteca utilizată pentru construirea interfețelor grafice, oferind toate instrumentele necesare pentru crearea ferestrelor, organizarea elementelor vizuale și gestionarea evenimentelor.

Importarea bibliotecii Tkinter

<def/>

Pentru utilizarea bibliotecii Tkinter este necesară importarea acesteia.
Sintaxa utilizată de obicei este: `import tkinter as tk`

Prin utilizarea aliasului ***`tk`***, componentele bibliotecii vor fi apelate mai ușor.

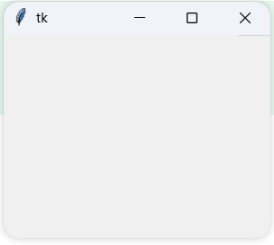
EX.

```
import tkinter as tk
```

Crearea unei ferestre principale

<def/>

Orice aplicație Tkinter începe prin crearea unei ferestre principale.
Fereastra principală reprezintă spațiul în care vor fi plasate toate componentele grafice ale aplicației.



EX.

```
import tkinter as tk
fereastra = tk.Tk()
fereastra.mainloop()
```

Explicații:

- `tk.Tk()` creează fereastra principală;
- `mainloop()` pornește bucla principală a aplicației și menține fereastra activă.

Fără metoda `mainloop()`*, fereastra s-ar închide imediat după executarea programului.*

Personalizarea ferestrei

<def/>

Fereastra principală poate fi personalizată prin: stabilirea titlului; modificarea dimensiunii; schimbarea culorii de fundal; blocarea redimensionării.

Stabilirea titlului

<def/>

Titlul unei ferestre se setează cu metoda `title()`.

EX.

```
fereastra.title("Prima aplicație Tkinter")
```

Stabilirea dimensiunii ferestrei

<def/> *Dimensiunea* se stabilește cu metoda ***geometry()***.

EX.
```
fereastra.geometry("400x300")
```

Valoarea este exprimată în pixeli: 400 = lățimea; 300 = înălțimea.

Schimbarea culorii de fundal

<def/> *Culoarea de fundal* poate fi modificată cu metoda ***config()***.

EX.
```
fereastra.config(bg="lightblue")
```

Blocarea redimensionării

<def/> *Redimensionarea* poate fi permisă sau interzisă.

EX.
```
fereastra.resizable(False, False)
```

Parametrii reprezintă: redimensionarea pe orizontală; redimensionarea pe verticală.
Valoarea ***False*** interzice redimensionarea.

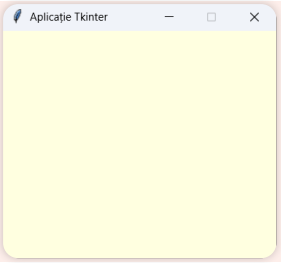
EX. **Exemplu complet de creare a unei ferestre**

```
import tkinter as tk

fereastra = tk.Tk()

fereastra.title("Aplicație Tkinter")
fereastra.geometry("300x250")
fereastra.config(bg="lightyellow")
fereastra.resizable(False, False)

fereastra.mainloop()
```



13.2. Widgeturi în Tkinter

<def/> *Elementele grafice* adăugate într-o fereastră se numesc ***widgeturi***.

Cele mai utilizate widgeturi sunt:

- `Label` – afișează text;
- `Button` – creează butoane;
- `Entry` – permite introducerea datelor;
- `Text` – afișează texte mai lungi;
- `Frame` – organizează alte componente;
- `Checkbutton` – creează casete de selectare;
- `Radiobutton` – permite alegerea unei opțiuni.

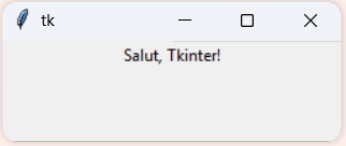
Widgetul ***Label*** – este utilizat pentru afișarea textului.

EX.
```
import tkinter as tk

fereastra = tk.Tk()

eticheta = tk.Label(fereastra, text="Salut, Tkinter!")
eticheta.pack()

fereastra.mainloop()
```

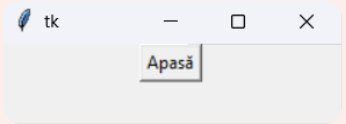


Explicații: • primul parametru reprezintă fereastra în care este plasat widgetul;
• parametrul text stabilește mesajul afișat.

Widgetul ***Button*** – creează butoane interactive.

EX.
```
import tkinter as tk
fereastra = tk.Tk()

buton = tk.Button(fereastra, text="Apasă")
buton.pack()
fereastra.mainloop()
```



13.3. Managementul așezării componentelor

<def/> În ***Tkinter***, componentele grafice trebuie organizate în interiorul ferestrei.
Procesul de poziționare al widgeturilor se numește ***managementul așezării***.

Tkinter oferă trei metode principale: ***pack()***; ***grid()***; ***place()***.

Metoda *pack()* – organizează automat widgeturile.

Aceasta este cea mai simplă metodă de așezare și poziționează elementele unul sub altul sau unul lângă altul.

EX.

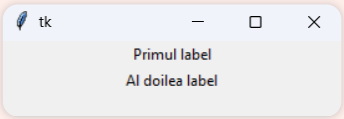
```
import tkinter as tk

fereastra = tk.Tk()

label1 = tk.Label(fereastra, text="Primul label")
label1.pack()

label2 = tk.Label(fereastra, text="Al doilea label")
label2.pack()

fereastra.mainloop()
```



Opțiuni importante pentru pack()

side – *Permite poziționarea widgetului.*

EX.

```
buton.pack(side="left")
```

Poziții posibile: left; right; top; bottom.

padx și **pady** – *Adaugă spațiu exterior în jurul widgetului.*

EX.

```
buton.pack(padx=10, pady=10)
```

Metoda *grid()* – organizează widgeturile într-un tabel format din linii și coloane.

Este foarte utilă pentru formulare și interfețe ordonate.

EX.

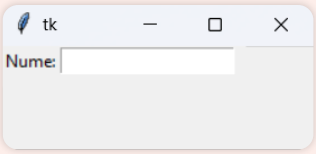
```
import tkinter as tk

fereastra = tk.Tk()

label_nume = tk.Label(fereastra, text="Nume:")
label_nume.grid(row=0, column=0)

caseta_nume = tk.Entry(fereastra)
caseta_nume.grid(row=0, column=1)

fereastra.mainloop()
```



Explicații: ***row*** reprezintă linia; ***column*** reprezintă coloana.

Spațierea cu grid()

EX.

```
label.grid(row=0, column=0, padx=10, pady=10)
```

Îmbinarea celulelor

Parametrul ***columnspan*** permite extinderea widgetului pe mai multe coloane (număr specificat).

EX.

```
titlu.grid(row=0, column=0, columnspan=2)
```

Metoda *place()* – poziționează widgeturile folosind coordonate exacte.

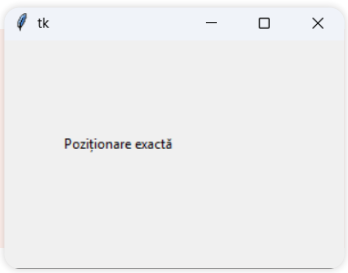
EX.

```
import tkinter as tk

fereastra = tk.Tk()
fereastra.geometry("300x200")

label = tk.Label(fereastra, text="Poziționare exactă")
label.place(x=50, y=80)

fereastra.mainloop()
```



Explicații: ***x*** *reprezintă poziția orizontală;* ***y*** *reprezintă poziția verticală.*

Diferențe între pack(), grid() și place()

Metodă	Caracteristică principală	Utilizare
pack()	Organizare automată	Interfețe simple
grid()	Organizare pe linii și coloane	Formulare și aplicații ordonate
place()	Poziționare exactă	Jocuri și interfețe speciale

Reguli importante

În aceeași fereastră nu este recomandată combinarea metodelor `pack()` și `grid()`. Acest lucru poate produce erori sau comportamente neașteptate. Totuși, metode diferite pot fi utilizate în frame-uri diferite.

Avantajele utilizării Tkinter sunt: este gratuită și inclusă în ***Python***; este ușor de învățat; permite dezvoltarea rapidă a aplicațiilor ***GUI***; funcționează pe Windows, Linux și macOS; este potrivită pentru proiecte educaționale și aplicații simple.

Limitările utilizării Tkinter sunt: designul componentelor poate părea învechit; nu oferă efecte grafice complexe; este mai puțin utilizată pentru aplicații comerciale moderne.

Reține!

- ✔ ***Tkinter*** este biblioteca standard ***Python*** pentru interfețe grafice.
- ✔ Orice aplicație începe prin crearea unei ferestre principale folosind ***Tk()***.
- ✔ ***Widgeturile*** reprezintă componentele grafice ale aplicației. Poziționarea ***widgeturilor*** se realizează prin: `pack()`; `grid()`; `place()`. Metoda `grid()` este cea mai utilizată pentru interfețe ordonate.

Concluzii: **Tkinter** este cea mai accesibilă bibliotecă **Python** pentru crearea de aplicații grafice intuitive. Realizarea unor interfețe bine organizate depinde direct de stăpânirea widgeturilor și a metodelor de așezare a acestora în fereastră, aceste concepte fiind baza pentru dezvoltarea proiectelor vizuale complexe.

Fișa de lucru 13

Creați dosarul numit *Fisa13*. Rezolvați exercițiile următoare, în limbajul Python, utilizând biblioteca Tkinter și salvați fișierele în dosarul de lucru.

1. ***Realizați o fereastră cu:*** dimensiunea 400x200; titlul „Exercițiu Tkinter”; fundal verde deschis.
2. ***Adăugați două etichete și un buton folosind metoda*** `pack()`.
3. ***Creați un formular cu:*** nume; prenume; vârstă. Folosiți metoda `grid()`.
4. ***Poziționați trei butoane în locuri diferite folosind metoda*** `place()`.

Autoevaluare

Completați răspunsurile într-un fișier *FisaAutoevaluare13.docx*, folosind notații precum *1 – da*, *2 – nu*, *3 – nu sunt sigur*. Salvați fișierul în dosarul *Fisa13*.

	La finalul lecției, știu să...	***Da***	***Nu***	***Nu sunt sigur(ă)***
1.	explic ce este Tkinter.			
2.	import biblioteca Tkinter în program și să creez o fereastră principală.			
3.	personalizez fereastra.			
4.	enumăr principalele widgeturi disponibile în Tkinter.			
5.	organizez widgeturile în fereastră folosind pack().			
6.	organizez widgeturile sub formă de tabel cu grid().			

7. Am întâmpinat dificultăți în: …

Nu uitați să vă evaluați comportamentul și activitatea din timpul lecției!
(cu ajutorul fișei de observare a comportamentului și a activității, de la pagina 7)

La finalul lecției, m-am simțit:

Lecția 14 – Widget-uri interactive: butoane, etichete și casete text

14.1. Utilizarea widgeturilor interactive în Tkinter

„Orice aplicație bună începe cu o interfață clară și ușor de utilizat."

Interacțiunea utilizatorului cu aplicațiile grafice

Aplicațiile moderne permit utilizatorului să interacționeze direct cu elementele afișate pe ecran.
În interfețele grafice, această interacțiune se realizează prin intermediul ***widgeturilor***.

<def/>

Widgeturile *sunt componente grafice* care permit:

- afișarea informațiilor;
- introducerea datelor;
- executarea comenzilor;
- comunicarea dintre utilizator și aplicație.

În biblioteca ***Tkinter****, cele mai utilizate* ***widgeturi*** *interactive sunt:*

- `Label` (etichetă);
- `Button` (buton);
- `Entry` (casetă text);
- `Text` (zonă text extinsă).

Aceste componente reprezintă baza oricărei aplicații grafice.

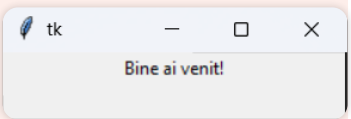
<def/>

Widgetul `Label` – este utilizat pentru afișarea informațiilor text sau grafice.
Etichetele sunt folosite frecvent pentru: titluri; instrucțiuni; mesaje informative; descrieri ale altor componente.

Crearea unei etichete

EX.

```
import tkinter as tk
fereastra = tk.Tk()
eticheta = tk.Label(fereastra, text="Bine ai venit!")
eticheta.pack()
fereastra.mainloop()
```



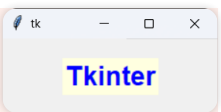
Explicații: `Label()` *creează eticheta*; *parametrul* `text` *stabilește mesajul afișat*; `pack()` *poziționează widgetul în fereastră.*

Personalizarea etichetelor

Etichetele pot fi personalizate prin: culoare; font; dimensiune; fundal; aliniere.

EX.

```
import tkinter as tk
fereastra = tk.Tk()
eticheta = tk.Label(fereastra,text="Tkinter",
          font=("Arial", 20, "bold"), fg="blue", bg="lightyellow")
eticheta.pack(pady=20)
fereastra.mainloop()
```



Parametri importanți pentru Label

Parametru	*Rol*
text	Textul afișat
font	Fontul și dimensiunea
fg	Culoarea textului

Parametru	*Rol*
bg	Culoarea fundalului
width	Lățimea widgetului
height	Înălțimea widgetului

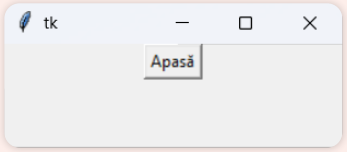
<def/>

Widgetul `Button` – permite executarea unei acțiuni atunci când utilizatorul apasă un buton.
Butoanele sunt componente esențiale în aplicațiile interactive.
Prin intermediul lor se pot: deschide ferestre; afișa mesaje; procesa date; executa funcții.

Crearea unui buton

EX.

```
import tkinter as tk
fereastra = tk.Tk()
buton = tk.Button(fereastra, text="Apasă")
buton.pack()
fereastra.mainloop()
```

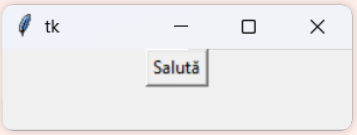


Asocierea unei funcții unui buton

Butonul devine interactiv atunci când este asociat unei funcții prin parametrul `command`.

EX.

```
import tkinter as tk
fereastra = tk.Tk()
def salut():
    print("Salut!")
buton = tk.Button(fereastra, text="Salută", command=salut)
buton.pack()
fereastra.mainloop()
```



Explicații: • funcția `salut()` va fi executată la apăsarea butonului;
• numele funcției se transmite fără paranteze.

Modificarea unei etichete prin buton

EX.

```
import tkinter as tk
fereastra = tk.Tk()
eticheta = tk.Label(fereastra, text= "Mesaj inițial" )
eticheta.pack(pady=10)
def modifica_text():
    eticheta.config(text= "Text modificat" )
buton = tk.Button(fereastra, text="Modifică", command=modifica_text)
buton.pack()
fereastra.mainloop()
```

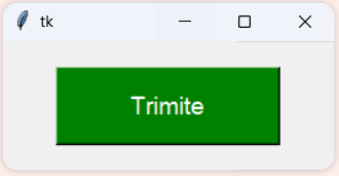


Explicații: metoda ***config()*** permite modificarea proprietăților unui **widget**.

Personalizarea butoanelor

EX.

```
import tkinter as tk
fereastra = tk.Tk()
buton = tk.Button(
    fereastra,
    text="Trimite",
    font=("Arial", 14),
    bg="green",
    fg="white",
    width=15,
    height=2
)
buton.pack(pady= 20)
fereastra.mainloop()
```



Parametri importanți pentru Button

Parametru	*Rol*
text	Textul afișat pe buton
command	Funcția executată
bg	Culoarea fundalului
fg	Culoarea textului
font	Stilul textului

Parametru	*Rol*
width	Lățimea butonului, specifică numărul de caractere care încap pe lățime
height	Înălțimea butonului, specifică numărul de linii de text care încap pe înălțime

Widgetul `Entry` – permite introducerea datelor pe o singură linie.
Acesta este utilizat pentru: nume; parole; adrese; valori numerice.

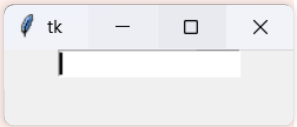
Crearea unei casete text

```
import tkinter as tk

fereastra = tk.Tk()

caseta = tk.Entry(fereastra)
caseta.pack()

fereastra.mainloop()
```



Citirea textului introdus

Conținutul unei casete text poate fi citit cu metoda `get()`.

```
import tkinter as tk

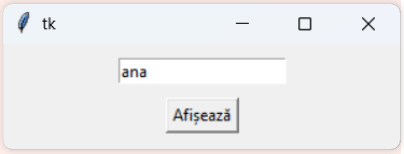
fereastra = tk.Tk()

caseta = tk.Entry(fereastra)
caseta.pack(pady=10)

def afiseaza():
    text = caseta.get()
    print(text)

buton = tk.Button(fereastra, text="Afișează", command=afiseaza)
buton.pack()

fereastra.mainloop()
```



Introducerea și ștergerea textului

Tkinter *permite*: inserarea textului începând cu o anumită poziție; ștergerea conținutului. Dacă în casetă există text în poziția indicată, acesta este deplasat spre dreapta, după textul inserat.

- ***Inserarea textului***

```
caseta.insert(0, "Salut")
```

Primul parametru reprezintă poziția.

- ***Ștergerea textului***

```
caseta.delete(0, tk.END)
```

Casete pentru parole

Parolele pot fi ascunse folosind parametrul `show`.

```
entry_parola = tk.Entry(fereastra, show="*")
```

Widgetul `Text` – permite introducerea textelor pe mai multe linii.
Este utilizat pentru: comentarii; descrieri; note; editoare simple de text.

Crearea unui widget Text

```
import tkinter as tk

fereastra = tk.Tk()

zona_text = tk.Text(fereastra, width=30, height=10) # 30 de caractere maximum,
# 10 linii de text
zona_text.pack()

fereastra.mainloop()
```

Citirea textului din widgetul Text

```
continut = zona_text.get("1.0", tk.END)
```

Explicații: • *`1.0`* reprezintă începutul textului (linia 1, caracterul 0);
• *`tk.END`* reprezintă finalul textului.

Inserarea textului într-un widget Text (la final)

EX.

```
zona_text.insert(tk.END, "Acesta este un text.")
```

Ștergerea textului

EX.

```
zona_text.delete("1.0", tk.END)
```

Exemplu practic – Aplicație interactivă

EX.

```
import tkinter as tk

fereastra = tk.Tk()
fereastra.title("Aplicație interactivă")
fereastra.geometry("400x250")

label = tk.Label(fereastra, text="Introdu numele tău")
label.pack(pady=10)

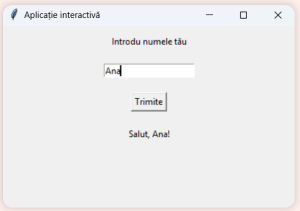
entry = tk.Entry(fereastra)
entry.pack(pady=10)

def afiseaza_mesaj():
    nume = entry.get()
    rezultat.config(text="Salut, " + nume + "!")

buton = tk.Button(fereastra, text="Trimite", command=afiseaza_mesaj)
buton.pack(pady=10)

rezultat = tk.Label(fereastra, text="")
rezultat.pack(pady=10)

fereastra.mainloop()
```



Explicația metodei get() pentru widgetul Text

Conținutul unui **widget** `Text` se citește folosind metoda:

```
zona_text.get("1.0", tk.END)
```

Explicații:

- **`"1.0"`** reprezintă poziția de început:
  - **`1`** = linia 1;
  - **`0`** = coloana 0 (primul caracter).
- **`tk.END`** reprezintă finalul textului.

Sintaxa generală este:

```
widget_text.get(linie.coloana, pozitie_finala)
```

EX.

```
continut = zona_text.get("1.0", tk.END)
print(continut)
```

Widgetul `Text` utilizează coordonate de tip: linie; coloană.
Astfel, "2.5" înseamnă: linia 2; coloana 5.

Diferența dintre Entry și Text

Widget	*Caracteristică*
Entry	Introducere pe o singură linie
Text	Introducere pe mai multe linii

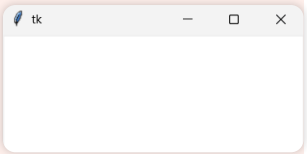
Widgetul `Canvas` – este utilizat pentru desen și reprezentări grafice.

Canvas permite: desenarea formelor geometrice; realizarea jocurilor simple; afișarea imaginilor; crearea animațiilor; reprezentări grafice și simulări. **Canvas** transformă fereastra într-o suprafață de desen.

Crearea unui Canvas

EX.

```
import tkinter as tk
fereastra = tk.Tk()
canvas = tk.Canvas(fereastra, width=300, height=200, bg="white")
canvas.pack()
fereastra.mainloop()
```



Explicații:

- ***width*** stabilește lățimea suprafeței de desen (în pixeli);
- ***height*** stabilește înălțimea (în pixeli);
- ***bg*** stabilește culoarea fundalului.

Desenarea unei linii

EX.

```
import tkinter as tk
fereastra = tk.Tk()
canvas = tk.Canvas(fereastra, width=300, height=200)
canvas.pack()
canvas.create_line(20, 20, 200, 100)
fereastra.mainloop()
```



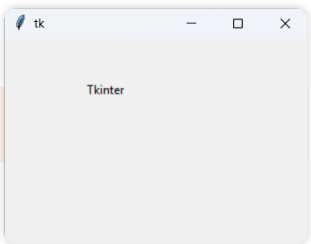
Canvas() creează suprafața de desen; `create_line()` desenează o linie. Coordonatele reprezintă: punctul de început; punctul de sfârșit.

OBS Se poate adăuga și fundal cu comanda ex. `bg="white"`

Adăugarea textului pe Canvas

EX.

```
canvas.create_text(100, 50, text="Tkinter")
```

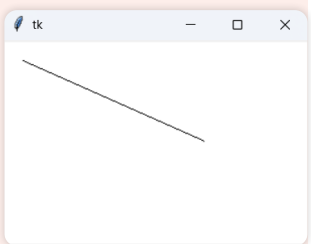


Utilizări ale widgetului Canvas

Canvas este utilizat pentru: jocuri 2D; aplicații de desen; simulări grafice; animații; reprezentări vizuale ale datelor.

EX.

```
import tkinter as tk
fereastra = tk.Tk()
canvas = tk.Canvas(fereastra, width=300, height=200, bg="white")
canvas.pack()
canvas.create_line(20, 20, 200, 100)
fereastra.mainloop()
```



Activitate practică – Comparație între aplicație CLI și aplicație GUI

Aceeași funcționalitate poate fi realizată: în linie de comandă (**CLI**); cu interfață grafică (**GUI**).

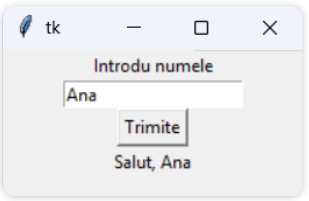
Varianta CLI – folosind input()

```
nume = input("Introdu numele: ")
print("Salut,", nume)
```

```
>>> nume = input("Introdu numele: ")
... print("Salut,", nume)
...
Introdu numele: Ana
Salut, Ana
```

Caracteristici: interacțiune prin tastatură; rulare în consolă; interfață simplă.

Varianta GUI – folosind Entry și Button




```
import tkinter as tk

fereastra = tk.Tk()

label = tk.Label(fereastra, text="Introdu numele")
label.pack()

entry = tk.Entry(fereastra)
entry.pack()

def saluta():
    nume = entry.get()
    rezultat.config(text="Salut, " + nume)

buton = tk.Button(fereastra, text="Trimite", command=saluta)
buton.pack()

rezultat = tk.Label(fereastra, text="")
rezultat.pack()

fereastra.mainloop()
```

Caracteristici: interfață vizuală; utilizare mai intuitivă; interacțiune prin widgeturi.

Comparație între CLI și GUI

Caracteristică	CLI	GUI
Interacțiune	Tastatură	Mouse și tastatură
Aspect vizual	Simplu	Grafic și intuitiv
Complexitate	Mai redusă	Mai mare
Ușurință pentru utilizator	Mai redusă	Mai ridicată

OBS

Widget-ul **Canvas** extinde posibilitățile de dezvoltare ale aplicațiilor **Tkinter**. **Canvas** permite desenul și reprezentările grafice.

Acest widget este esențial pentru realizarea aplicațiilor grafice complexe și bine structurate.

Reține!

✔ Widgeturi: ***Label*** (afișare), ***Button*** (acțiuni), ***Entry*** (text o linie), ***Text*** (text multilinie).

✔ Interacțiune: ***command*** (asociază funcții butoanelor), ***get()*** (citește textul), ***config()*** (modifică proprietățile).

Concluzii

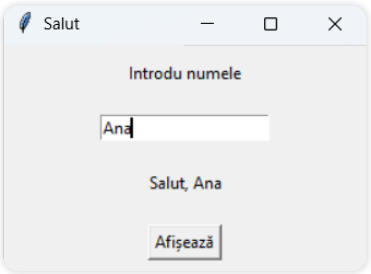
Widgeturile interactive (etichete, butoane, casete text) sunt elementele fundamentele din **Tkinter** care permit schimbul de informații între utilizator și program, fiind esențiale pentru dezvoltarea aplicațiilor grafice moderne.

Fișa de lucru 14

Creați dosarul numit *Fisa14*. Rezolvați exercițiile următoare, în limbajul Python, utilizând biblioteca Tkinter și salvați fișierele în dosarul de lucru.

1. Creați o fereastră care conține:
- o etichetă;
- un buton;
- o casetă text.

2. Realizați un buton care schimbă textul unei etichete.



3. Construiți un formular de autentificare cu:
- utilizator;
- parolă;
- buton de conectare.

4. Creați o aplicație care citește textul introdus și îl afișează într-o etichetă.

5. ***Citirea textului din widgetul Text***

Realizați o aplicație care:
- conține un widget `Text`;
- citește conținutul folosind `get()`;
- afișează textul într-un `Label`.

Autoevaluare

Completați răspunsurile într-un fișier *FisaAutoevaluare14.docx*, folosind notații precum *1 – da*, *2 – nu*, *3 – nu sunt sigur*. Salvați fișierul în dosarul *Fisa14*.

	La finalul lecției, știu...	***Da***	***Nu***	***Nu sunt sigur(ă)***
1.	să creez o etichetă (Label) și îi setez proprietățile.			
2.	să creez un buton (Button) și îl personalizez cu parametri.			
3.	să asociez o funcție unui buton prin parametrul command.			
4.	să modific proprietățile unui widget existent.			
5.	să creez/inserez o casetă text.			
6.	să construiesc o aplicație simplă care combină Label, Entry și Button pentru a prelua și afișa date de la utilizator.			

7. Am întâmpinat dificultăți în: …

Nu uitați să vă evaluați comportamentul și activitatea din timpul lecției!
(cu ajutorul fișei de observare a comportamentului și a activității, de la pagina 7)

La finalul lecției, m-am simțit:

Lecția 15 – Gestionarea evenimentelor în interfețe grafice (afișarea mesajelor la clic)

Gestionarea evenimentelor în Tkinter și afișarea mesajelor la clic

„Un program interactiv răspunde acțiunilor utilizatorului asemenea unei conversații dintre om și calculator."

15.1. Evenimentele în aplicațiile grafice

Aplicațiile grafice moderne funcționează pe baza interacțiunii dintre utilizator și program.

Orice acțiune realizată de utilizator produce un eveniment.

EX. Exemple de *evenimente*: *apăsarea unui buton*; *deplasarea mouse-ului*; *apăsarea unei taste*; *închiderea unei ferestre*; *selectarea unei opțiuni*.

<def/> Gestionarea evenimentelor reprezintă procesul prin care programul detectează și răspunde la aceste acțiuni.

În ***Tkinter****, evenimentele* sunt gestionate prin: funcții; parametrul `command`; metoda `bind()`.

15.2. Programarea orientată pe evenimente

Tkinter utilizează *programarea orientată pe evenimente*.

În acest model:

- aplicația așteaptă acțiuni din partea utilizatorului;
- la apariția unui eveniment, este executată o funcție;
- funcția asociată se numește funcție de tratare a evenimentului.

La interacțiunea utilizatorului cu aplicația, se produce următoarea succesiune de pași:

1. utilizatorul execută o acțiune;
2. apare un eveniment;
3. evenimentul este detectat;
4. este executată funcția asociată.

Funcții utilizate în gestionarea evenimentelor

În **Python**, funcțiile permit organizarea codului și executarea unor instrucțiuni atunci când sunt apelate.

EX.
```
def salut():
    print("Salut!")
```

În ***Tkinter***, o funcție poate fi asociată unui eveniment și este apelată automat atunci când evenimentul respectiv are loc.

Evenimentul de clic pe buton

Cel mai utilizat eveniment este apăsarea unui buton. Acesta poate fi gestionat prin parametrul `command`.

Utilizarea parametrului command

EX.
```
import tkinter as tk

fereastra = tk.Tk()

def mesaj():
    print("Buton apăsat")

buton = tk.Button(fereastra, text="Clic", command=mesaj)
buton.pack()

fereastra.mainloop()
```

Explicații:

- funcția mesaj() este executată la apăsarea butonului;
- numele funcției se transmite fără paranteze;
- parametrul command stabilește acțiunea butonului.

Afișarea mesajelor într-o etichetă

În aplicațiile grafice, mesajele pot fi afișate direct în fereastră folosind ***widgetul*** `Label`.

EX.

```
import tkinter as tk

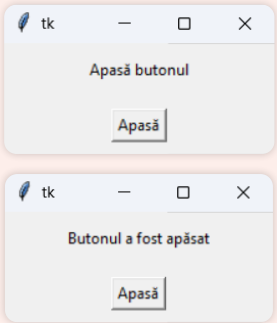
fereastra = tk.Tk()

eticheta = tk.Label(fereastra, text="Apasă butonul")
eticheta.pack(pady=10)

def modifica_mesaj():
    eticheta.config(text="Butonul a fost apăsat")

buton = tk.Button(fereastra, text="Apasă",
command=modifica_mesaj)
buton.pack(pady=10)

fereastra.mainloop()
```



Metoda ***config()*** modifică proprietățile unui ***widget*** existent.

Afișarea mesajelor în ferestre speciale

Tkinter oferă modulul ***messagebox*** pentru afișarea ferestrelor de dialog.
Aceste ferestre pot afișa: informații; avertismente; erori; întrebări.

Importarea modulului messagebox se face folosind:

```
from tkinter import messagebox
```

Afișarea unui mesaj informativ

EX.

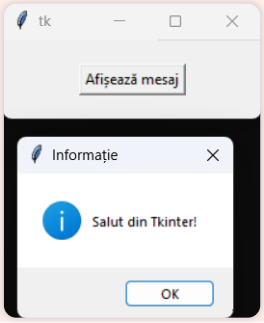
```
import tkinter as tk
from tkinter import messagebox

fereastra = tk.Tk()

def afiseaza_mesaj():
    messagebox.showinfo("Informație","Salut din Tkinter!")

buton = tk.Button(fereastra, text="Afișează mesaj",
command=afiseaza_mesaj)
buton.pack(pady=20)

fereastra.mainloop()
```



Explicații:

- Primul parametru reprezintă titlul ferestrei;
- Al doilea parametru reprezintă mesajul afișat.

Tipuri de ferestre messagebox

Funcție	*Rol*
`showinfo()`	Afișează informații
`showwarning()`	Afișează avertismente
`showerror()`	Afișează erori
`askquestion()`	Pune întrebări
`askyesno()`	Returnează True sau False

EX.

Avertisment

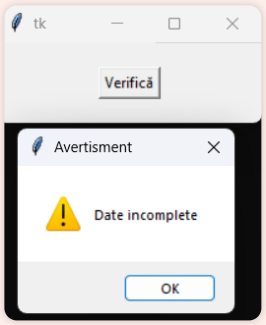
```
import tkinter as tk
from tkinter import messagebox

fereastra = tk.Tk()

def avertisment():
    messagebox.showwarning("Avertisment", "Date incomplete")

buton = tk.Button(fereastra, text="Verifică")
buton.config(command=avertisment)
buton.pack(pady=20)

fereastra.mainloop()
```



EX.

Mesaj de eroare

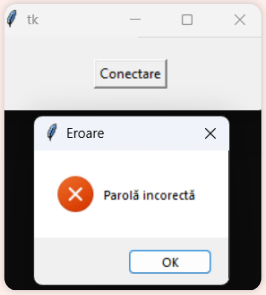
```
import tkinter as tk
from tkinter import messagebox

fereastra = tk.Tk()

def eroare():
    messagebox.showerror("Eroare","Parolă incorectă")

buton = tk.Button(fereastra, text="Conectare", command=eroare)
buton.pack(pady=20)

fereastra.mainloop()
```



Schimbarea culorii la clic

EX.

```
import tkinter as tk

fereastra = tk.Tk()
fereastra.geometry("300x200")

def schimba_culoare():
    fereastra.config(bg="lightblue")

buton = tk.Button(fereastra, text="Schimbă culoarea", command=schimba_culoare)
buton.pack(pady=50)

fereastra.mainloop()
```

EX.

Calculator de click-uri

```
import tkinter as tk

fereastra = tk.Tk()
fereastra.title("Contor")

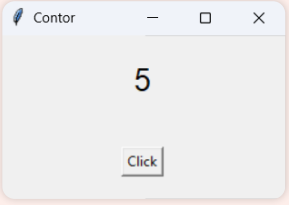
numar = 0

eticheta = tk.Label(fereastra, text="0", font=("Arial", 20))
eticheta.pack(pady=20)

def creste():
    global numar
    numar += 1
    eticheta.config(text=str(numar))

buton = tk.Button(fereastra, text="Click", command=creste)
buton.pack(pady=20)

fereastra.mainloop()
```



Evenimente pentru taste specifice

Tkinter permite detectarea tastelor speciale utilizând metoda `bind()`.

Taste utilizate frecvent:

- `<Return>` → tasta Enter;
- `<Escape>` → tasta Escape;
- `<Key>` → orice tastă.

```
# Detectarea tastelor Enter, Escape și orice tastă

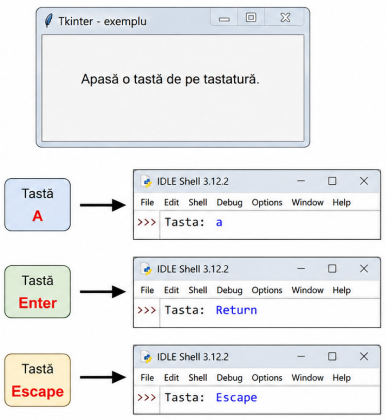
import tkinter as tk

def afiseaza(event):
    print("Tasta:", event.keysym)

fereastra = tk.Tk()

fereastra.bind("<Key>", afiseaza)
fereastra.bind("<Return>", afiseaza)
fereastra.bind("<Escape>", afiseaza)

fereastra.mainloop()
```



Organizarea codului în aplicațiile interactive

În aplicațiile mai complexe este recomandată: separarea funcțiilor; utilizarea variabilelor clare; comentarea codului; organizarea **widgeturilor** în frame-uri. Aceste practici îmbunătățesc lizibilitatea și întreținerea programului.

Erori frecvente

Utilizarea parantezelor la command

Greșit: `command=functie()`

Corect: `command=functie`

Parantezele execută funcția imediat.

Lipsa parametrului event la bind()

Greșit:
```
def functie():
    pass
```

Corect:
```
def functie(event):
    pass
```

Reține!

- ✔ Evenimentele sunt acțiuni ale utilizatorului care controlează aplicația (folosind programarea orientată pe evenimente).
- ✔ Interacțiunea cu interfața grafică se face prin parametrul command (pentru butoane) și modulul messagebox (pentru ferestre de dialog).

Concluzii: Gestionarea evenimentelor face aplicațiile Tkinter dinamice, legând acțiunile utilizatorului de funcții specifice. Stăpânirea elementelor `command` și `messagebox` este esențială pentru dezvoltarea interfețelor grafice interactive.

Fișa de lucru 15

Creați dosarul numit *Fisa15*. Rezolvați exercițiile următoare, în limbajul Python, utilizând biblioteca Tkinter și salvați fișierele în dosarul de lucru.

1. Creați un buton care afișează un mesaj într-o etichetă.

2. Realizați o aplicație care schimbă culoarea ferestrei la apăsarea unui buton.

3. Construiți un formular care afișează un mesaj de eroare dacă utilizatorul nu introduce date.

4. Găsiți și gestionați greșeala, analizând codul de mai jos, apoi corectați instrucțiunea.

```
buton = tk.Button(fereastra, text="Click", command=functie())
```

5. ***Mini-proiect***

Realizați o aplicație care conține: widget-ul Entry; validarea datelor cu messagebox; afișarea rezultatului într-un Label; confirmarea introducerii datelor cu Enter (<Return>).

Aplicația trebuie:

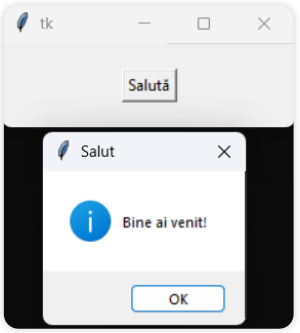
- să afișeze un avertisment dacă utilizatorul nu introduce date;
- să afișeze un mesaj de salut dacă datele sunt corecte.

Probleme rezolvate

Problema 1 – Afișarea unui mesaj de salut

Să se realizeze o aplicație care afișează un mesaj de salut la apăsarea unui buton.

Rezolvare

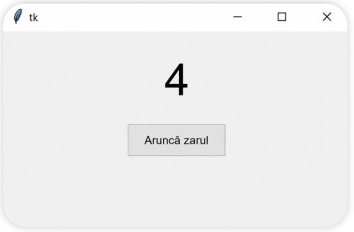



```
import tkinter as tk
from tkinter import messagebox
fereastra = tk.Tk()
def salut():
    messagebox.showinfo("Salut", "Bine ai venit!")
buton = tk.Button(fereastra, text="Salută", command=salut)
buton.pack(pady=20)
fereastra.mainloop()
```

Problema 2 – Zar virtual

Să se realizeze o aplicație care afișează un număr aleator între 1 și 6 la apăsarea unui buton.

Rezolvare




```
import tkinter as tk
import random
fereastra = tk.Tk()
eticheta = tk.Label(fereastra, text="-", font=("Arial", 20))
eticheta.pack(pady=10)
def arunca_zarul():
    eticheta.config(text=str(random.randint(1, 6)))
buton = tk.Button(fereastra, text="Aruncă zarul", command=arunca_zarul)
buton.pack(pady=10)
fereastra.mainloop()
```

Autoevaluare

Completați răspunsurile într-un fișier *FisaAutoevaluare15.docx*, folosind notații precum *1 – da*, *2 – nu*, *3 – nu sunt sigur*. Salvați fișierul în dosarul *Fisa15*.

	La finalul lecției, știu...	*Da*	*Nu*	*Nu sunt sigur(ă)*
1.	ce este un eveniment într-o aplicație grafică.			
2.	să asociez o funcție unui buton prin parametrul command.			
3.	importarea modulului messagebox și afișarea unei ferestre de dialog.			
4.	să construiesc o aplicație completă cu buton, etichetă actualizată la clic și fereastră messagebox.			

5. Am întâmpinat dificultăți în: …

Nu uitați să vă evaluați comportamentul și activitatea din timpul lecției!
(cu ajutorul fișei de observare a comportamentului și a activității, de la pagina 7)

La finalul lecției, m-am simțit:

Lecția 16 – Fișiere text: deschidere, închidere și transfer de date

Lucrul cu fișiere text în Python

„Datele devin informații valoroase atunci când pot fi stocate, accesate și prelucrate eficient."

Fișierele și rolul lor în informatică

În informatică, fișierele sunt utilizate pentru stocarea permanentă a informațiilor.

Spre deosebire de variabile, care păstrează datele doar pe durata executării programului, fișierele permit memorarea informațiilor pentru utilizări ulterioare.

Fișierele sunt folosite pentru: salvarea documentelor; stocarea datelor utilizatorilor; memorarea rezultatelor; schimbul de informații între programe.

Tipuri de fișiere

Fișierele pot fi clasificate în două categorii principale: fișiere text; fișiere binare.

Fișiere text – sunt fișiere care stochează informațiile sub formă de caractere.

EX. .txt; .csv; .py; .html.

Aceste fișiere pot fi deschise și citite cu editoare de text.

Fișiere binare – sunt fișiere în care informațiile sunt stocate într-un format codificat, sub formă de date binare și nu sunt, în general, direct inteligibile pentru utilizatorii umani.

EX. .jpg, .png, .gif (imagini), .mp3, .wav (fișiere audio), .mp4, .avi (fișiere video), .exe, .com, .bat (executabile) ș.a.

Lucrul cu fișiere în Python

În **Python**, lucrul cu fișiere implică: deschiderea fișierului; citirea sau scrierea datelor; închiderea fișierului. Procesul este important deoarece resursele calculatorului trebuie gestionate corect.

Deschiderea unui fișier

Fișierele sunt deschise cu funcția *open()*.

Sintaxa generală: `fisier = open("cale\nume_fisier", "mod")`

Parametri: `nume_fisier` – numele fișierului;
`mod` – modul de utilizare.
`calea` – poate fi relativă sau absolută.

Moduri de deschidere a fișierelor

Mod	*Semnificație*
`r`	Citire (read)
`w`	Scriere (write)
`a`	Adăugare la final (append)

Mod	*Semnificație*
`x`	Creare fișier nou (create)
`rb`	Citire binară (read binary)
`wb`	Scriere binară (write binary)

Modul `r` – citire, permite citirea conținutului unui fișier.

EX.
```
fisier = open("date.txt", "r") # Cale relativă - fișierul text e în același folder
```

Dacă fișierul nu există, programul va produce eroare.

Citirea datelor din fișier

Python oferă mai multe metode de citire.

Metoda `read()` – citește întregul conținut al fișierului.

EX.
```
fisier = open("c:\\date.txt", "r")
continut = fisier.read()
print(continut)
fisier.close()
```

Fișierul date.txt

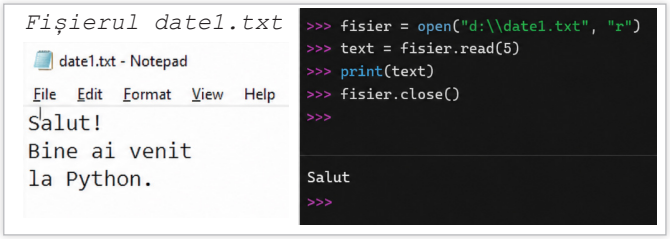


Citirea unui număr de caractere

EX.
```
fisier = open("d:\\date1.txt", "r")
text = fisier.read(5)
print(text)
fisier.close()
```

Programul citește primele 5 caractere.

Fișierul date1.txt



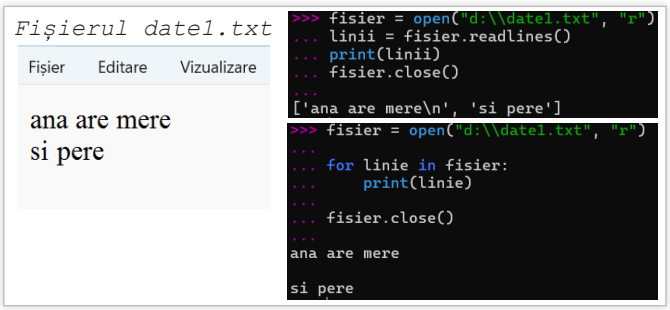
Metoda `readline()` – citește o singură linie.

EX.
```
fisier = open("d:\\date1.txt", "r")
linie = fisier.readline()
print(linie)
fisier.close()
```

Metoda `readlines()` – citește toate liniile și le memorează într-o listă.

EX.
```
fisier = open("d:\\date1.txt", "r")
linii = fisier.readlines()
print(linii)
fisier.close()
```

Fișierul date1.txt



Parcurgerea liniilor unui fișier

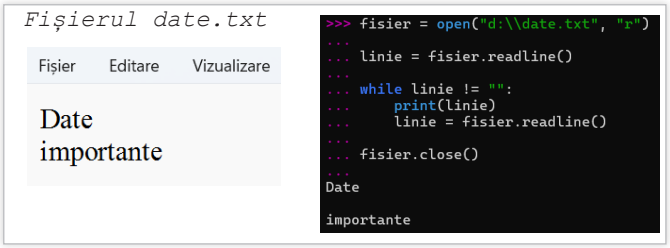
Fișierele pot fi parcurse cu instrucțiunea `for`.

EX.
```
fisier = open("d:\\date1.txt", "r")
for linie in fisier:
    print(linie)
fisier.close()
```

Notă: Când metoda read() returnează șirul vid "" sau readline() returnează "", înseamnă că am ajuns la sfârșitul fișierului (EOF = End Of File).

EX.
```
fisier = open("d:\\date.txt", "r")
linie = fisier.readline()
while linie != "":
    print(linie)
    linie = fisier.readline()
fisier.close()
```

Fișierul date.txt



Închiderea fișierelor

După utilizare, fișierele trebuie închise. Acest lucru se realizează cu metoda `close()`.

EX.
```
fisier.close()
```

Închiderea fișierelor: eliberează memoria; salvează modificările; evită erorile.

Scrierea în fișiere

Pentru scriere se utilizează modul `w`.

EX.
```
fisier = open("d:\\date1.txt", "w")
fisier.write("Salut!")
fisier.close()
```

Fișierul date1.txt



⚠ **Atenție!** Modul `w` șterge conținutul existent înainte de scriere.

Scrierea pe mai multe linii

EX.
```
fisier = open("d:\\date1.txt", "w")
fisier.write("Ana\n")
fisier.write("Maria\n")
fisier.write("Ioan")
fisier.close()
```



Caracterul **\n** produce trecerea la o linie nouă.

Modul *a* – adăugare, permite adăugarea datelor la finalul fișierului fără ștergerea conținutului existent.

EX.
```
fisier = open("d:\\date1.txt", "a")
fisier.write("\nText nou")
fisier.close()
```

Fișierul date1.txt

Fișier Editare

Ana
Maria
Ioan

```
>>> fisier = open("d:\\date1.txt", "a")
... fisier.write("\nText nou")
... fisier.close()
```

Fișier Editare

Ana
Maria
Ioan
Text nou

Modul *x* – creare fișier nou, creează un fișier nou.

EX.
```
fisier = open("d:\\nou.txt", "x")
fisier.close()
```

Dacă fișierul există deja, apare eroare.

Transferul de date între fișiere și program

Transferul de date reprezintă: citirea informațiilor din fișiere; prelucrarea lor; salvarea rezultatelor.

EX.
```
fisier = open("d:\\numere.txt", "r")
suma = 0
for linie in fisier:
    suma += int(linie)
fisier.close()
print("Suma este:", suma)
```

Fișierul numere.txt

Fișier Editare Vizualizare

3
5
7
9
11

```
>>> fisier = open("d:\\numere.txt", "r")
...
... suma = 0
...
... for linie in fisier:
...     suma += int(linie)
...
... fisier.close()
...
... print("Suma este:", suma)
...
Suma este: 35
```

Utilizarea instrucțiunii with

Python oferă instrucțiunea ***with***, care închide automat fișierul.

EX.
```
with open("d:\\numere.txt", "r") as fisier:
    continut = fisier.read()
print(continut)
```

```
>>> with open("d:\\numere.txt", "r") as fisier:
...     continut = fisier.read()
...     print(continut)
...
3
5
7
9
11
```

Avantaje: cod mai scurt; închidere automată; gestionare sigură a erorilor.

Citirea și scrierea simultană

EX.
```
with open("d:\\intrare.txt", "r") as sursa:
    text = sursa.read()

with open("d:\\iesire.txt", "w") as destinatie:
    destinatie.write(text)
```

Fișierul intrare.txt

Fișier Editare Vizualizare

ana

Fișierul iesire.txt

Fișier Editare Vizualizare

ana

Programul copiază conținutul unui fișier în altul.

Gestionarea erorilor

La lucrul cu fișiere pot apărea erori: fișier inexistent; lipsa permisiunilor; nume incorect. Aceste situații pot fi tratate cu ***try*** și ***except***.

Avantajele utilizării fișierelor

Fișierele permit: stocarea permanentă a datelor; transferul informațiilor; organizarea datelor; realizarea aplicațiilor complexe.

Erori frecvente

Uitarea închiderii fișierului

Greșit:
```
fisier = open("d:\\date.txt","r")
print(fisier.read())
```

Corect:
```
fisier = open("date.txt","r")
print(fisier.read())
fisier.close()
```

Utilizarea modului greșit de lucru cu fișierul

EX.
```
open("date.txt","w")
```

Acest mod șterge conținutul existent.

Reține!

✔ Fișierele sunt deschise cu funcția `open()` și închise cu `close()`. Instrucțiunea `with` permite gestionarea automată a închiderii fișierelor, contribuind la scrierea unui cod mai sigur și mai eficient.

✔ Citirea/scrierea se fac cu `read()`/`readline()`/`readlines()`, respectiv `write()`.

✔ Siguranță: Gestionează erorile posibile, pentru a preveni blocarea programului în cazul unor probleme cu fișierele.

Concluzii: Fișierele text sunt esențiale pentru stocarea permanentă și prelucrarea eficientă a datelor. Înțelegerea fluxului de date (deschidere, citire, scriere, închidere) stă la baza dezvoltării aplicațiilor software care gestionează volume mari de informații.

Fișa de lucru 16

Creați dosarul numit *Fisa16*. Rezolvați exercițiile următoare, în limbajul Python și salvați fișierele în dosarul de lucru.

Exerciții propuse

1. Creați un fișier text și scrieți în el trei nume.
2. Citiți conținutul unui fișier și afișați-l pe ecran.
3. Realizați un program care adaugă o propoziție nouă la finalul unui fișier.
4. Calculați suma numerelor memorate într-un fișier text.

Probleme rezolvate

Problema 1 – Citirea și afișarea datelor

Cerință: Să se citească și să se afișeze conținutul fișierului ***medii.txt***.

Rezolvare:

```
with open("d:\\medii.txt", "r") as fisier:
    continut = fisier.read()
print(continut)
```

Problema 2 – Copierea conținutului unui fișier

Cerință: Să se copieze conținutul fișierului ***date1.txt*** în fișierul ***copie.txt***.

Rezolvare:

```
with open("d:\\date1.txt", "r") as sursa:
    text = sursa.read()
with open("d:\\copie.txt", "w") as destinatie:
    destinatie.write(text)
```

Problema 3 – Gestionarea erorilor

Cerință: Scrieți un program care încearcă să deschidă un fișier numit `date5.txt`. Dacă fișierul nu există, programul va afișa mesajul: `Fisierul nu exista!`

Rezolvare:

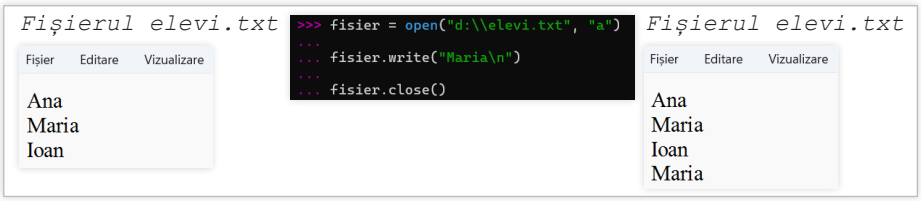
```
try:
    fisier = open("d:\\date5.txt", "r")
    print(fisier.read())
    fisier.close()
except FileNotFoundError:
    print("Fisierul nu exista!")
```

Problema 4 – Adăugare date în fișier

Cerință: Adăugați un nou nume într-un fișier fără a șterge conținutul existent.

Rezolvare:

```
fisier = open("d:\\elevi.txt", "a")
fisier.write("Maria\n")
fisier.close()
```

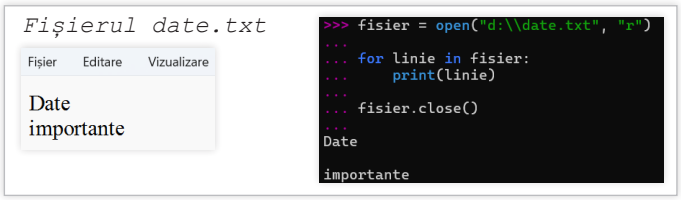


Problema 5 – Parcurgere directă a fișierului

Cerință: Afișați toate liniile unui fișier folosind instrucțiunea for.

Rezolvare:

```
fisier = open("d:\\date.txt", "r")
for linie in fisier:
    print(linie)
fisier.close()
```



Notă – Comparația fișier vs input()

Citirea datelor dintr-un fișier este mai rapidă și mai practică decât introducerea repetată de la tastatură (`input()`), mai ales atunci când lucrăm cu volume mari de date.

EX.

Introduceți 20 de numere:
- o dată folosind `input()`
- o dată citindu-le din fișier.

Observați care metodă este mai rapidă și mai comodă.

Autoevaluare

Completați răspunsurile într-un fișier *FisaAutoevaluare16.docx*, folosind notații precum *1 – da*, *2 – nu*, *3 – nu sunt sigur*. Salvați fișierul în dosarul *Fisa16*.

	La finalul lecției, știu...	*Da*	*Nu*	*Nu sunt sigur(ă)*
1.	să explic diferența dintre fișierele text și fișierele binare.			
2.	să deschid un fișier în Python cu open() și aleg modul corect (r, w, a, x) în funcție de operația dorită.			
3.	să parcurg un fișier linie cu linie.			
4.	să scriu date într-un fișier.			
5.	să adaug date la finalul unui fișier.			
6.	să închid corect un fișier.			

7. Am întâmpinat dificultăți în: …

Nu uitați să vă evaluați comportamentul și activitatea din timpul lecției!
(cu ajutorul fișei de observare a comportamentului și a activității, de la pagina 7)

La finalul lecției, m-am simțit:

Lecția 17 – Clasa TextIOWrapper: metode de citire și scriere eficientă

Utilizarea clasei TextIOWrapper în Python

„Performanța unui program depinde nu doar de algoritmi, ci și de modul eficient în care sunt gestionate datele."

Fluxurile de date și fișierele text

În programare, datele circulă între: memorie; fișiere; utilizator; dispozitive externe.

Acest transfer de informații se realizează prin intermediul fluxurilor de date.

Un flux de date este un mecanism prin care un program citește sau scrie informații una după alta (secvențial).

În **Python**, lucrul cu fișiere text este realizat prin obiecte speciale create de funcția ***open()***.

Aceste obiecte aparțin clasei ***TextIOWrapper***.

Clasa *TextIOWrapper* – face parte din modulul ***io*** și este utilizată pentru: citirea fișierelor text; scrierea fișierelor text; gestionarea fluxurilor de caractere.

Atunci când un fișier text este deschis cu funcția ***open()***, ***Python*** creează automat un obiect de tip ***TextIOWrapper***.

EX.

```
fisier = open("date.txt", "r")
print(type(fisier))
```

```
<class '_io.TextIOWrapper'>
```

Rezultat: `<class '_io.TextIOWrapper'>`

Caracteristicile clasei TextIOWrapper

Clasa TextIOWrapper: lucrează cu caractere Unicode; gestionează automat codificarea textului; oferă metode eficiente de citire și scriere; permite lucrul cu fișiere mari.

Codificarea caracterelor

Fișierele text sunt salvate folosind diferite codificări. Cele mai utilizate sunt: UTF-8; ASCII; UTF-16.

Python utilizează frecvent codificarea UTF-8.

EX.

```
fisier = open("date.txt", "r", encoding="utf-8")
```

Parametrul ***encoding*** stabilește modul de interpretare al caracterelor.

Deschiderea unui fișier

Funcția ***open()*** returnează un obiect ***TextIOWrapper***.

Sintaxa generală:

```
fisier = open(nume_fisier, mod, encoding="utf-8")
```

EX.

```
fisier = open("date.txt", "r", encoding="utf-8")
```

Citirea eficientă a fișierelor

Pentru fișiere foarte mari, utilizarea metodei ***read()*** poate consuma multă memorie. O metodă mai eficientă este citirea linie cu linie, care nu încarcă întregul fișier în memorie.

EX.

```
with open("d:\\date1.txt", "r") as fisier:
    for linie in fisier:
        print(linie)
```

```
>>> with open("d:\\date1.txt", "r", encoding="utf-8") as fisier:
...     continut = fisier.read()
...
... print(continut)
...
Intreprinderea
Electronica
```

```
>>> with open("d:\\date1.txt","r") as fisier:
...     for linie in fisier:
...         print(linie)
...
Intreprinderea

Electronica
```

Avantaje: consum redus de memorie; potrivit pentru fișiere mari.

Pregătirea datelor citite din fișiere

Eliminarea caracterului \n cu strip()

```
with open("d:\\date.txt", "r") as f:
    for linie in f:
        print(linie.strip())
```

```
>>> with open("d:\\date.txt", "r") as f:
...     for linie in f:
...         print(linie.strip())
...
Date
importante
```

Explicație: metoda ***strip()*** elimină spațiile și caracterul special **\n** de la sfârșitul liniei.

Conversia șirurilor în numere

```
with open("d:\\note.txt", "r") as f:
    for linie in f:
        nota = float(linie.strip())
        print(nota)
```

```
>>> with open("d:\\note.txt", "r") as f:
...     for linie in f:
...         nota = float(linie.strip())
...         print(nota)
...
10.0
9.0
8.0
6.0
4.0
7.0
```

Explicație: datele citite din fișier sunt șiruri de caractere. Pentru calcule trebuie convertite folosind **int()** sau **float()**.

Separarea valorilor cu split() – este foarte importantă pentru fișiere de tip CSV simplificat.

```
linie = "Ana,10"
date = linie.split(",")
print(date)
```

Rezultat:

```
['Ana', '10']
```

```
>>> linie = "Ana,10"
...
... date = linie.split(",")
...
... print(date)
...
['Ana', '10']
```

```
with open("d:\\elevi9.txt", "r") as f:
    for linie in f:
        nume, nota= linie.strip().split(",")
        nota = int(nota)
        print(nume, nota)
```

```
>>> with open("d:\\elevi9.txt", "r") as f:
...     for linie in f:
...         nume, nota = linie.strip().split(",")
...
...         nota = int(nota)
...
...         print(nume, nota)
...
ion 8
ana 10
tudor 9
```

Metoda *write()* – permite scrierea textului într-un fișier.

```
with open("d:\\date2.txt","w") as fisier:
    fisier.write("Salut din Python")
```

Scrierea mai multor linii

```
with open("d:\\elevi.txt","w") as fisier:
    fisier.write("Ana\n")
    fisier.write("Maria\n")
    fisier.write("Ioan")
```

Metoda *writelines()* – scrie mai multe linii simultan.

```
linii = ["Ana\n", "Maria\n", "Ioan\n"]
with open("d:\\elevi1.txt","w") as fisier:
    fisier.writelines(linii)
```

```
>>> with open("d:\\date2.txt","w") as fisier:
...     fisier.write("Salut din Python")
```

Fișierul date2.txt

Fișier Editare Vizualizare

Salut din Python

```
>>> with open("d:\\elevi.txt","w") as fisier:
...     fisier.write("Ana\n")
...     fisier.write("Maria\n")
...     fisier.write("Ioan")
```

Fișierul elevi.txt

Fișier Editare Vizualizare

Ana
Maria
Ioan

```
>>> linii = ["Ana\n", "Maria\n", "Ioan\n"]
...
... with open("d:\\elevi1.txt","w") as fisier:
...     fisier.writelines(linii)
```

Fișierul elevi1.txt

Fișier Editare Vizualizare

Ana
Maria
Ioan

Diferența dintre *write()* și *writelines()*

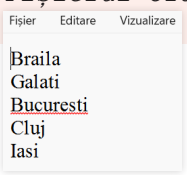
Metodă	*write()*	*writelines()*
Caracteristică	Scrie un singur șir	Scrie o listă de șiruri

EX.

Salvați o listă de 5 orașe într-un fișier folosind metoda ***writelines()***.

```
orase = ["Braila\n", "Galati\n", "Bucuresti\n", "Cluj\n", "Iasi\n" ]
with open("d:\\orase.txt", "w") as f:
    f.writelines(orase)
```

Fișierul orase.txt



OBS

`writelines()` **NU** adaugă automat `\n`.

Poziția cursorului în fișier

Clasa ***TextIOWrapper*** utilizează un cursor intern. Cursorul indică poziția curentă de citire sau scriere.

Închiderea fișierelor

Fișierele trebuie închise după utilizare.

EX.

```
fisier.close()
```

Metoda *close()*: eliberează resursele; salvează modificările; închide fluxul de date.

Utilizarea instrucțiunii *with* – este cea mai recomandată metodă de lucru.

EX.

```
with open("d:\\date.txt","r") as fisier:
    continut = fisier.read()
```

Avantaje: închidere automată; cod mai sigur; reducerea erorilor.

Transferul eficient de date între fișiere

EX.

```
with open("d:\\date.txt","r") as sursa:
    continut = sursa.read()
with open("d:\\destinatie.txt", "w") as destinatie:
    destinatie.write(continut)
```

Fișierul date.txt




```
>>> with open("d:\\date.txt", "r") as sursa:
...     continut = sursa.read()
...
... with open("d:\\destinatie.txt", "w") as destinatie:
...     destinatie.write(continut)
```

Fișierul destinatie.txt



Copierea eficientă linie cu linie

Pentru fișiere mari este recomandată copierea linie cu linie.

EX.

```
with open("d:\\date.txt","r") as sursa:
    with open("d:\\destinatie.txt", "w") as destinatie:
        for linie in sursa:
            destinatie.write(linie)
```

Această metodă consumă mai puțină memorie.

Gestionarea erorilor

EX.

```
try:
    with open("d:\\date3.txt", "r") as fisier:
        print(fisier.read())

except FileNotFoundError:
    print("Fișierul nu există")
```

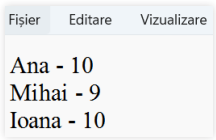
```
>>> try:
...     with open("d:\\date3.txt", "r") as fisier:
...         print(fisier.read())
...
... except FileNotFoundError:
...     print("Fișierul nu există")
...
Fișierul nu există
```

Proprietăți:

Proprietate	*Rol*
name	Numele fișierului
mode	Modul de deschidere
closed	Verifică dacă fișierul este închis
encoding	Codificarea utilizată

Avantajele clasei TextIOWrapper

Clasa ***TextIOWrapper*** oferă: gestionarea eficientă a fișierelor text; suport Unicode; bufferizare automată; metode rapide de citire și scriere; integrare simplă în programe.



Erori frecvente

Uitarea închiderii fișierului

Greșit:

```
fisier = open("date.txt", "r")
print(fisier.read())
```

Corect:

```
with open("date.txt", "r") as fisier:
    print(fisier.read())
```

Citirea completă a fișierelor foarte mari

Greșit:

```
continut = fisier.read()
```

Reține!

- ✔ Tip fișier: ***open()*** returnează un obiect ***TextIOWrapper*** care gestionează fișierele text prin bufferizare (optimizarea transferului).
- ✔ Scriere: ***write()*** / ***writelines()*** pentru scriere.
- ✔ Siguranță: Se recomandă utilizarea instrucțiunii ***with*** pentru închiderea automată și sigură.

Concluzii

Clasa ***TextIOWrapper*** este mecanismul principal din Python pentru gestionarea eficientă și sigură a fișierelor text. Înțelegerea metodelor sale de citire, scriere și control al fluxului de date este esențială pentru optimizarea transferului de informații între memorie și fișiere, în special când se lucrează cu volume mari de date.

Fișa de lucru 17

Creați dosarul numit *Fisa17*. Rezolvați exercițiile următoare, în limbajul Python și salvați fișierele în dosarul de lucru.

Exerciții propuse

1. Creați un fișier text și scrieți în el denumirile a cinci orașe.
2. Citiți un fișier linie cu linie și afișați conținutul.
3. Numărați caracterele de pe prima linie a unui fișier text.

Probleme rezolvate

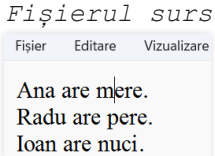
Problema 1 – Copierea eficientă a unui fișier

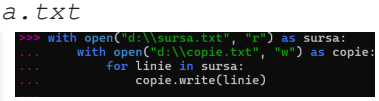
Cerință: Să se copieze conținutul unui fișier text folosind citirea linie cu linie.

Rezolvare:

```
with open("d:\\sursa.txt", "r") as sursa:
    with open("d:\\copie.txt", "w") as copie:
        for linie in sursa:
            copie.write(linie)
```

Fișierul sursa.txt





Problema 2 – Numărarea caracterelor

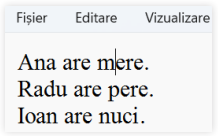
Cerință: Să se determine numărul total de caractere dintr-un fișier.

Rezolvare:

```
with open("d:\\copie.txt", "r") as fisier:
    continut = fisier.read()

print("Număr caractere:", len(continut))
```

Fișierul copie.txt




```
>>> with open("d:\\copie.txt", "r") as fisier:
...     continut = fisier.read()
...
... print("Număr caractere:", len(continut))
...
Număr caractere: 43
```

Problema 3 – Conversie int()

Cerință: Fișierul `numere.txt` conține câte un număr natural pe fiecare linie. Calculați suma numerelor.

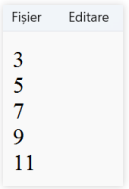
Rezolvare:

```
suma = 0

with open("d:\\numere.txt", "r") as f:
    for linie in f:
        suma += int(linie.strip())

print(suma)
```

Fișierul numere.txt




```
>>> suma = 0
...
... with open("d:\\numere.txt", "r") as f:
...     for linie in f:
...         suma += int(linie.strip())
...
... print(suma)
...
35
```

Problema 4 – Utilizarea split()

Cerință: Fișierul `elevi9.txt` conține următoarele date:

```
ion,8
ana,10
tudor,9
```

Afișați pe linii separate numele și nota fiecărui elev.

Rezolvare:

```
with open("d:\\elevi9.txt", "r") as f:
    for linie in f:
        nume, nota = linie.strip().split(",")

        print(nume)
        print(nota)
```

```
>>> with open("d:\\elevi9.txt", "r") as f:
...     for linie in f:
...         nume, nota = linie.strip().split(",")
...
...         print(nume)
...         print(nota)
...
ion
8
ana
10
tudor
9
```

Problema 5 – Verificare fișier gol

Cerință: Verificați dacă fișierul `date.txt` este gol.

Rezolvare:

```
with open("d:\\date.txt", "r") as f:
    continut = f.read()
    if continut == "":
        print("Fisier gol")
    else:
        print("Fisierul contine date")
```

Fișierul date.txt




```
>>> with open("d:\\date.txt", "r") as f:
...     continut = f.read()
...
...     if continut == "":
...         print("Fisier gol")
...     else:
...         print("Fisierul contine date")
...
Fisierul contine date
```

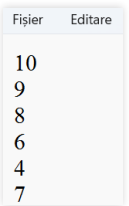
Problema 6 – Calcul medie note

Cerință: Fișierul `note.txt` conține câte o notă pe fiecare linie. Calculați media notelor.

Rezolvare:

```
suma = 0
nr = 0
with open("d:\\note.txt", "r") as f:
    for linie in f:
        suma += float(linie.strip())
        nr += 1
print(suma / nr)
```

Fișierul note.txt




```
>>> suma = 0
... nr = 0
...
... with open("d:\\note.txt", "r") as f:
...     for linie in f:
...         suma += float(linie.strip())
...         nr += 1
...
... print(suma / nr)
...
7.333333333333333
```

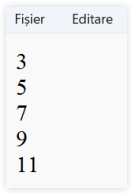
Problema 7 – Statistică pe numere

Cerință: Determinați suma, minimul și maximul numerelor dintr-un fișier care conține un număr natural pe fiecare linie.

Rezolvare:

```
numere = []
with open("d:\\numere.txt", "r") as f:
    for linie in f:
        numere.append(int(linie.strip()))
print(sum(numere))
print(min(numere))
print(max(numere))
```

Fișierul numere.txt




```
>>> numere = []
... 
... with open("d:\\numere.txt", "r") as f:
...     for linie in f:
...         numere.append(int(linie.strip()))
... 
... print(sum(numere))
... print(min(numere))
... print(max(numere))
... 
35
3
11
```

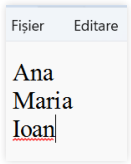
Problema 8 – Căutare într-un fișier

Cerință: Verificați dacă un nume introdus de la tastatură există într-un fișier.

Rezolvare:

```
nume_cautat = input("Nume: ")
gasit = False
with open("d:\\nume.txt", "r") as f:
    for linie in f:
        if linie.strip() == nume_cautat:
            gasit = True
if gasit:
    print("Numele exista")
else:
    print("Numele nu exista")
```

Fișierul nume.txt




```
>>> nume_cautat = input("Nume: ")
... 
... gasit = False
... 
... with open("d:\\nume.txt", "r") as f:
...     for linie in f:
...         if linie.strip() == nume_cautat:
...             gasit = True
... 
... if gasit:
...     print("Numele exista")
... else:
...     print("Numele nu exista")
... 
Nume: Maria
Numele exista
```



Autoevaluare

Completați răspunsurile într-un fișier *FisaAutoevaluare17.docx*, folosind notații precum *1 – da*, *2 – nu*, *3 – nu sunt sigur*. Salvați fișierul în dosarul *Fisa17*.

	La finalul lecției, știu...	*Da*	*Nu*	*Nu sunt sigur(ă)*
1.	să explic ce este clasa TextIOWrapper.			
2.	să deschid un fișier cu codificare specificată folosind parametrul encoding='utf-8'.			
3.	să citesc eficient un fișier mare linie cu linie, fără a încărca tot conținutul în memorie.			
4.	să accesez proprietățile unui obiect TextIOWrapper.			
5.	să identific și tratez cazurile limită: fișier gol, fișier inexistent, date invalide.			

6. Am întâmpinat dificultăți în: …

Nu uitați să vă evaluați comportamentul și activitatea din timpul lecției!
(cu ajutorul fișei de observare a comportamentului și a activității, de la pagina 7)

La finalul lecției, m-am simțit:

Evaluare sumativă - Capitolul 4

Verifică-ți cunoștințele!

(Punctaj: 90 de puncte + 10 puncte din oficiu)

Cerință: Rezolvați cerințele de mai jos într-un fișier ***evaluare_sumativa4.docx***. Salvați toate fișierele pe care le creați, în dosarul ***Capitol4***.

1. Alegeți, din lista următoare, situația sau descrierea care definește cel mai bine biblioteca Tkinter în limbajul Python: a. O bibliotecă utilizată exclusiv pentru efectuarea calculelor matematice complexe. **b.** O bibliotecă standard Python utilizată pentru crearea interfețelor grafice (GUI), care oferă ferestre, butoane, câmpuri de introducere și alte controale vizuale. **c.** O funcție specială care permite doar afișarea textului în consolă. **d.** O structură de date folosită pentru memorarea imaginilor și fișierelor multimedia.

2. Mai jos sunt prezentate două fragmente de cod scrise de un elev. Fiecare fragment conține o eroare care blochează execuția programului (FileNotFoundError sau ValueError).

Cerință: Pentru fiecare caz: **a.** Identificați linia de cod care generează eroarea. **b.** Numiți tipul de eroare produsă. **c.** Scrieți codul corect.

Cazul 1	Cazul 2
`f = open("datee.txt", "r")` `continut = f.read()` `print(continut)` `f.close()` *(În director există doar fișierul „date.txt”.)*	`f = open("numere.txt", "r")` `linie = f.readline()` `n = int(linie)` `print(n)` `f.close()` *(Prima linie din fișier conține textul: „douăzeci”.)*

3. Asociați situația descrisă în coloana din stânga cu elementul corespunzător din coloana din dreapta.

Situație	Element Tkinter
1. Afișează un text într-o fereastră	a. Entry
2. Permite introducerea datelor de la tastatură	b. Button
3. Execută o acțiune la apăsare	c. Label
4. Creează fereastra principală	d. Tk()

4. Creați o mini-aplicație Tkinter care să conțină: o fereastră principală; o etichetă (Label) cu textul „Calculator simplu”; două câmpuri de introducere (Entry); un buton „Adună”; afișarea rezultatului sumei numerelor introduse într-o etichetă. Programul trebuie să funcționeze corect și să permită efectuarea repetată a operației.

5. Se consideră o aplicație pentru autentificarea utilizatorilor. Ce componente Tkinter sunt cele mai potrivite pentru realizarea unei ferestre de autentificare care conține: nume utilizator; parolă; buton de conectare. Argumentați răspunsul și realizați o mini-aplicație Python care să ilustreze funcționarea acesteia.

6. Se dorește realizarea unei aplicații grafice pentru evidența temperaturilor zilnice.

Aplicația trebuie să permită introducerea temperaturilor pentru **n** zile folosind un câmp Entry și un buton „Adaugă”. Scrieți un program Tkinter care să realizeze următoarele sarcini: citește și memorează temperaturile introduse într-o listă; identifică și afișează temperaturile eronate (mai mici de -40°C sau mai mari de 60°C); dacă nu există erori, afișează mesajul „Date corecte”; calculează și afișează temperatura medie folosind doar valorile valide; afișează numărul zilelor în care temperatura a depășit media calculată.

7. O librărie dorește să monitorizeze vânzările de cărți printr-o aplicație cu interfață grafică.

Scrieți un program Tkinter care citește: numărul **n** de cărți vândute; cele **n** coduri ale cărților vândute (numere naturale cu valori între 1 și 1000). Programul trebuie să permită, prin intermediul unor butoane, efectuarea următoarelor operații: afișarea numărului de coduri distincte; căutarea unui cod **x** introdus de utilizator; afișarea primei poziții la care apare codul **x**; afișarea numărului de apariții ale codului **x**; afișarea codurilor care nu au fost vândute; afișarea codurilor în ordinea descrescătoare a frecvenței de apariție.

Interfața trebuie să conțină cel puțin: Label; Entry; Button; Listbox sau Text pentru afișarea rezultatelor.

Punctaj acordat: cerința 1 – **10 puncte**, cerința 2 – **15 puncte**, cerința 3 – **15 puncte**, cerința 4 – **10 puncte**, cerința 5 – **10 puncte**, cerința 6 – **10 puncte**, cerința 7 – **20 puncte**.



Proiect

(100 puncte)

„Aplicații Grafice pentru Viața de Zi cu Zi”

Realizați în Python o aplicație grafică denumită **aplicatie.py**, utilizând biblioteca Tkinter. Aplicația trebuie să rezolve o problemă practică la alegere (de exemplu: gestionarea unei biblioteci, catalog de note, agendă telefonică, calculator de buget, organizator de activități, evidența produselor etc.).

Aplicația trebuie să conțină cel puțin: o fereastră principală personalizată; etichete (Label); câmpuri de introducere a datelor (Entry); cel puțin trei butoane (Button); o zonă de afișare a rezultatelor (Label, Text sau Listbox); validarea datelor introduse de utilizator; utilizarea funcțiilor și a evenimentelor Tkinter. Proiectul va fi prezentat în fața clasei.

Barem de evaluare (100 puncte): 10p – Proiectarea și organizarea interfeței grafice, **10p** – Utilizarea corectă a componentelor Tkinter, **10p** – Funcționarea corectă a butoanelor și evenimentelor, **10p** – Gestionarea datelor introduse de utilizator, **10p** – Validarea datelor și tratarea erorilor, **10p** – Calitatea și organizarea codului sursă, **10p** – Corectitudinea explicațiilor teoretice privind componentele utilizate, **10p** – Claritatea prezentării și utilizarea terminologiei informatice, **10p** – Implicarea egală a membrilor echipei în prezentare, **10p** – Creativitate și aspect vizual.

(**Recomandări:** se lucrează în echipe de 2–3 elevi. Fiecare echipă acordă celorlalte echipe, pe baza unei fișe de evaluare, un punctaj, iar punctajul final va fi media aritmetică a punctajelor obținute.)



Investighează!

(100 puncte)

Studiu de caz: Cum construim o interfață prietenoasă?

Documentați, într-un fișier **tkinter.docx**, următoarea situație: O firmă dorește dezvoltarea unei aplicații desktop pentru gestionarea programărilor clienților. Interfața trebuie să fie intuitivă și ușor de utilizat.

Folosiți surse diverse și realizați sarcinile de mai jos:

- Descoperiți ce este o interfață grafică și care sunt avantajele acesteia față de aplicațiile în consolă. **(20p)**
- Documentați principalele componente Tkinter: Tk(), Label, Entry, Button, Text, Listbox și Frame. Prezentați rolul fiecăreia, într-un tabel. **(20p)**
- Realizați o fereastră care utilizează cel puțin cinci componente diferite. Includeți capturi de ecran și explicați-i modul de lucru. **(20p)**
- Scrieți în 4–6 rânduri ce recomandări ați face pentru realizarea unei interfețe grafice eficiente și ușor de utilizat. **(10p)**
- Căutați în documentația Tkinter o componentă sau metodă neprezentată la clasă (de exemplu: Checkbutton, Radiobutton, Scale, Menu, Spinbox, Combobox). **(10p)**
- Descrieți funcționarea acesteia și realizați un exemplu practic de utilizare. **(10p)**
- Integrați componenta studiată într-o mini-aplicație proprie și explicați avantajele utilizării sale. **(10p)**



Portofoliu

(100 puncte)

La finalul acestui capitol, portofoliul personal al elevului trebuie să cuprindă, în dosarul ***Capitol4*** următoarele materiale: dosarul ***Fisa13***, dosarul ***Fisa14***, dosarul ***Fisa15***, dosarul ***Fisa16***, dosarul ***Fisa17*** fiecare cu fișierele corespunzătoare, dosarul ***Proiecte4***, cu fișierele ***aplicatie.py*** și ***tkinter.docx*** și fișierul ***evaluare_sumativa4.docx.***

Modalitate de evaluare: Există dosarul ***Capitol4*** cu structura de directoare specificată – **20p**

Există în dosare:
- Peste 95% din fișiere – **80p**
- Între 75% și 94% din fișiere – **60p**
- Între 50% și 74% din fișiere – **40p**
- Între 25% și 49% din fișiere – **20p**
- Sub 25% – **0p**

Capitolul 5 – Organizarea conceptuală a datelor (liste)

Lecția 18 – Modelul conceptual liniar: caracteristici, acces direct vs secvențial

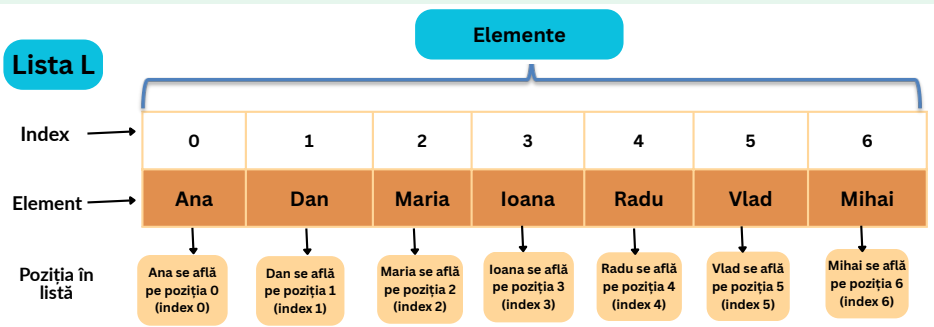
Procesul de rezolvare a unei probleme cu ajutorul calculatorului cuprinde mai multe etape. Prima etapă este ***etapa de analiză*** prin care sunt identificate datele problemei. Ea este urmată de cea de-a doua etapă, ***etapa de proiectare***. Aceasta constă în găsirea unui algoritm adecvat care include și identificarea unei modalități de organizare a datelor. În programare, ***procesul de organizare a datelor*** este util pentru a stoca, a gestiona și a prelucra datele într-un mod eficient și logic. Practic, organizarea datelor ajută la înțelegerea modului în care sunt păstrate datele și cum pot fi acestea folosite. Orice program lucrează cu date: nume, numere, note, mesaje, produse și altele. Organizarea datelor ne ajută să le păstrăm într-o formă clară și ordonată, astfel încât programul să le poată găsi și folosi cât mai repede. *De exemplu*, aplicația ***Catalog Școlar*** trebuie să stocheze, printre altele, numele elevilor și notele lor.

OBS În informatică, datele pot fi organizate sub formă de date simple sau date compuse. ***Datele simple*** reprezintă cele mai mici unități de informație și nu pot fi descompuse în alte componente cu sens, în timp ce ***datele compuse*** se obțin prin combinarea mai multor date simple.

Date simple – exemple	Date compuse – exemple
un număr: 23	o listă de note: [8, 9, 10]
un caracter: 'A'	un text de forma "Popa Ana", format din nume și prenume
o valoare logică: True	o dată calendaristică sub forma *zi*, *lună*, *an*: 12 ianuarie 2026

Modelul conceptual liniar este adesea reprezentat prin structuri de date de tip ***listă*** sau ***tablouri unidimensionale (șiruri de valori)***. Este o formă de organizare secvențială a informațiilor, în care elementele sunt dispuse unul după celălalt.

<def/> **Lista** reprezintă o structură compusă de date care stochează, într-o anumită ordine, un număr finit de elemente. Ordinea elementelor este dată de poziția fiecărui element în listă, această poziție putând fi identificată printr-un ***index*** (***index*** = poziția unui element în listă, numărată de la 0 pentru primul element).

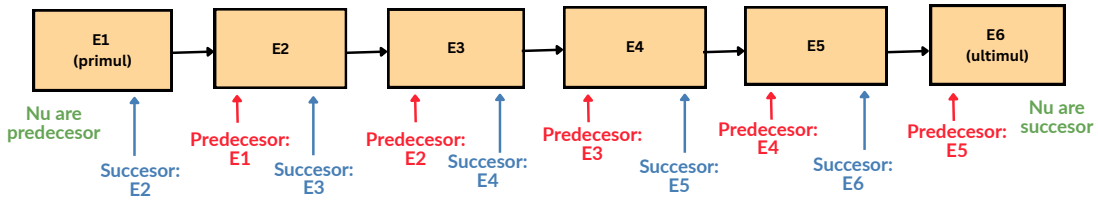


În limbajul de programare C++, listele/tablourile cuprind elemente de același tip. În limbajul de programare Python, listele pot cuprinde elemente de tipuri diferite, ca în exemplul de mai jos.

Limbajul Python	Limbajul C++
0 1 2 3	0 1 2 3 4 5
`lista=[1, 3.50, "a", "este luni"]`	`v=[12, 2, 3, 6, 9, 1]`

O listă are următoarele caracteristici conceptuale:

- ***structură de date compusă***: lista conține mai multe elemente reprezentând un grup de valori organizate împreună;
- ***organizare secvențială (liniaritate)***: elementele unei liste sunt aranjate astfel încât ele respectă relații simple de succesiune: fiecare element are un predecesor (cu excepția primului) și un succesor (cu excepția ultimului);



- ***ordine bine definită***: fiecare element ocupă o poziție clară (indexată) care permite identificarea și accesarea sa directă, de la primul până la ultimul element;
- ***acces controlat***: lista permite atât accesul direct, cât și cel secvențial, în funcție de necesități dar și de suportul fizic de stocare;
- ***dimensiune variabilă***: într-o listă se pot adăuga elemente sau se pot șterge elemente.

Exemplu:

0 1 2 3 4 5

Fie lista L=[4, 8, 15, 2, 32, 23]. În această listă:

- 4 este primul element
- 23 este ultimul element
- 8 are predecesorul 4 și succesorul 15

Elementul 15 are indexul 2, deci L[2]=15.

Exemple de liste din viața reală:

- ✓ lista cărților de la bibliotecă
- ✓ lista invitaților la un eveniment
- ✓ lista notelor unui elev la informatică
- ✓ lista comenzilor dintr-un magazin online
- ✓ lista elevilor dintr-o clasă
- ✓ lista melodiilor unui artist

Operațiile de bază care se pot realiza cu o listă sunt:

Operație	Descriere
Creare	Alocarea spațiului în memorie și inițializarea structurii pentru o nouă listă (inițial vidă).
Parcurgere	Accesarea secvențială a fiecărui element din listă, de la început până la sfârșit, pentru a le procesa sau afișa.
Inserare	Adăugarea unui element nou într-o anumită poziție din listă (la început, la sfârșit sau în interior).
Ștergere	Eliminarea unui element existent din listă și refacerea legăturilor dintre elementele rămase.
Căutare	Identificarea unui element specific în listă, pe baza unei valori sau a unei condiții date.
Acces	Citirea sau modificarea valorii unui element de pe o poziție specifică, fără a schimba structura listei.

Tipuri particulare de liste în funcție de acces

Listele se pot clasifica după tipul de acces, dar și după modul în care sunt realizate operațiile de bază. **Accesul la elementele unei liste** reprezintă modul în care elementele sunt găsite și obținute.

După tipul de acces, listele sunt: cu ***acces direct*** și cu ***acces secvențial***. Există și cazuri particulare de liste date de anumite restricții de acces la elemente: ***stivele*** și ***cozile***. Ele sunt particularizate datorită modului în care se realizează operațiile de adăugare și ștergere.

Accesul direct oferă posibilitatea de a localiza imediat un element pe baza indexului său sau a unei adrese, fără a fi nevoie de parcurgerea elementelor precedente. Fiecare element este accesat direct, pe baza poziției sale.

EX. Fie lista de numere L=[7, 14, 21, 28]. Putem accesa direct elementul cu indexul 0 (L[0]=7) sau elementul cu indexul 2 (L[2]=21).

Accesul secvențial presupune parcurgerea elementelor în ordine, pornind de la începutul listei și avansând succesiv. În acest fel, elementele sunt accesate unul după celălalt. Pentru a ajunge la un element, trebuie parcurse cele anterioare acestuia.

	Acces direct	Acces secvențial
Avantaje	✓ acces rapid la un element	✓ eficiente de cele mai multe ori dacă poziția este deja cunoscută
Dezavantaje	× inserările/ștergerile pot fi ineficiente	× acces lent la un element oarecare din listă

OBS Când prelucrăm o listă trebuie să avem în vedere următoarele:

- ***Modul de parcurgere a listei***: indexul este cel care ne spune unde ne aflăm, condiția de oprire pentru a ști dacă am ajuns la sfârșitul listei (limita superioară a listei), sensul de parcurgere a listei, alegerea unor indici corespunzători.
- ***Eficiența spațiului utilizat***: există situații în care vom memora elementele listei pentru a le putea prelucra (*de exemplu* în sortarea unui șir) sau memorăm doar câte un element la un moment dat, variantă care este mai eficientă din punctul de vedere al memoriei utilizate.

APLICAȚII

1) ***Prelucrarea fără memorare:*** parcurgem lista memorând elementul curent într-o variabilă. În felul acesta economisim resurse (mai ales spațiu de memorie și, uneori, timp de execuție). Algoritmii de bază care presupun prelucrarea elementelor dintr-o listă fără memorare: calculul sumei/produsul elementelor, găsirea valorii maxime/minime, numărarea elementelor care respectă o condiție etc.

EX. Se dă un șir de n elemente numere naturale (n număr natural nenul). Câte elemente din șir au cel puțin 2 cifre?

Algoritm	Cod Python
citește n (număr natural nenul) ct←0 pentru i←1,n execută citește x dacă x>9 atunci ct←ct+1 sfârșit dacă sfârșit pentru scrie ct	`n = int(input("Introduceti n: "))` `ct = 0` `for i in range(1, n + 1):` `  x = int(input(f"Introduceti elementul {i}:"))` `  if x > 9:` `  ct = ct + 1` `print("Rezultat ct: ", ct)`

2) ***Prelucrarea cu memorare:*** memorăm toate elementele într-o listă atunci când algoritmul necesită acces repetat sau non-liniar la date. Algoritmii de bază: sortarea elementelor, căutarea binară a unei valori într-o listă, inversarea ordinii elementelor, verificarea unor proprietăți, numărarea perechilor de elemente din lista nu neapărat consecutive care îndeplinesc o anumită condiție etc.

EX. Se dă un șir de n elemente numere naturale (n număr natural nenul). Câte perechi de elemente din șir au aceeași cifră a unităților? *Spre exemplu*, dacă n=7, L=[7, 12, 17, 51, 412 ,67, 3] perechile din șir care au aceeași cifră a unităților sunt: (7, 17), (7, 67), (12, 412), (17, 67), deci numărul lor este 4.

```
citește n (număr natural nenul)
pentru i←1,n execută
    citește v_i
sfârșit pentru
ct←0
pentru i←1,n execută
    pentru j←i+1,n execută
        dacă v_i %10==v_j %10  atunci
            ct←ct+1
        sfârșit dacă
    sfârșit pentru
sfârșit pentru
scrie ct
```

Reține!

- ✔ Se alege varianta de **prelucrare fără memorare** atunci când este suficientă o singură parcurgere a elementelor din listă, acestea fiind procesate pe rând, atunci când sunt accesate, fără revenire la elementele anterioare.
- ✔ Se alege varianta de **prelucrare cu memorare** (în tablouri/liste în memorie) când algoritmul necesită accesul de mai multe ori la aceleași date sau accesul non-liniar.

Fișa de lucru 18

Creați dosarul numit *Fisa18*. Rezolvați exercițiile următoare, notând răspunsurile corecte într-un fișier *raspunsuri.docx*.

Pentru cerințele 1 și 2 de mai jos, alegeți răspunsul corect.

1. Modelul conceptual liniar se caracterizează prin:

a. relații complexe între elemente; **b.** organizare ierarhică;
c. organizare secvențială; **d.** număr infinit de elemente.

2. Accesul direct la un element se referă la:

a. acces aleator lent; **b.** utilizarea unui index;
c. parcurgerea tuturor elementelor; **d.** existența unui succesor.

3. Completați fraza de mai jos cu răspunsurile potrivite:
Accesul la elementele unei liste poate fi: ..... și ..... .
Dimensiunea variabilă a unei liste se referă la: ..... .

4. Scrieți 3 argumente prin care justifici afirmația *"Lista reprezintă o structură compusă de date care stochează, într-o anumită ordine, un număr finit de elemente."*, plecând de la caracteristicile conceptuale ale listei.

5. Care dintre următoarele scenarii necesită în mod obligatoriu utilizarea unei liste în locul unor variabile simple și de ce?

a. înregistrarea temperaturilor zi/an într-un anumit oraș pe parcursul unei luni pentru a răspunde unor întrebări ulterioare (*de exemplu*, afișarea tuturor zilelor în care temperatura a fost strict mai mare decât media);
b. verificarea dacă un număr este impar sau par;
c. contorizarea numerelor impare dintr-un șir de n numere date;
d. contorizarea perechilor de numere impare consecutive dintr-un șir de n numere date.

6. Scrieți un algoritm și un program Python care să rezolve problema: *"Se dă un șir de n elemente numere naturale (n număr natural nenul). Câte perechi de elemente consecutive egale sunt în șir?"*. Justificați alegerea prelucrării fără memorare sau cea a prelucrării cu memorare.

7. Persoanele participante la un eveniment sportiv sunt înregistrate pe baza CNP-ului personal. Scrieți un algoritm și un program Python care să determine câte persoane sunt născute în luna Noiembrie.

8. Creați un proiect de tip eseu argumentativ cu titlul ***proiect***, în dosarul ***Fisa18***, despre importanța listelor în organizarea datelor. Proiectul trebuie să facă trecerea de la modul în care folosim listele în activitățile zilnice la proprietățile lor tehnice, argumentând de ce această structură este fundamentală în informatică dar și în organizarea informațiilor în viața de zi cu zi.



Autoevaluare

Completați răspunsurile într-un fișier *FisaAutoevaluare18.docx*, folosind notații precum *1 – da*, *2 – nu*, *3 – nu sunt sigur*. Salvați fișierul în dosarul *Fisa18*.

	La finalul lecției, știu...	***Da*** 	***Nu*** 	***Nu sunt sigur(ă)*** 
1.	să definesc o listă și să enumăr caracteristicile ei conceptuale.			
2.	diferența dintre accesul direct și cel secvențial.			
3.	când este utilă prelucrarea fără memorare și când cu memorare.			
4.	să scriu un algoritm simplu de prelucrare a unei liste fără memorare.			

5. Am întâmpinat dificultăți în: …

Nu uitați să vă evaluați comportamentul și activitatea din timpul lecției!
(cu ajutorul fișei de observare a comportamentului și a activității, de la pagina 7)

La finalul lecției, m-am simțit: 

Lecția 19 – Liste de frecvență, stiva (LIFO) și coada (FIFO)

O ***listă de frecvență*** (sau vectorul de frecvență) este o structură de date care asociază fiecărei valori distincte numărul său de apariții într-un șir. Folosește indicele listei drept element numărat, transformând date brute într-o reprezentare compactă, utilă în statistică și prelucrarea datelor.

EX. Profesorul de informatică dorește să știe cum s-au descurcat cei 30 de elevi dintr-o clasă la primul test. Notele primite de elevi sunt: 7, 5, 4, 8, 9, 9, 10, 7, 8, 7, 4, 5, 7, 9, 10, 4, 6, 9, 8, 9, 3, 6, 7, 8, 9, 9, 7, 8, 10, 9. Pentru a afla anumite valori, va contoriza numărul de apariții ale fiecărei posibile note în lista de frecvență, astfel:

Notă posibilă	1	2	3	4	5	6	7	8	9	10
Frecvența	0	0	1	3	2	2	6	5	8	3

Pe baza datelor de mai sus, profesorul poate trage diverse concluzii, cum ar fi: nota care a fost obținută de cele mai multe ori, câți elevi din clasă necesită măsuri remediale urgente, rata de promovare, numărul elevilor cu notă maximă.

Algoritmul de construire a unei liste de frecvență:

Pași	**Pasul 1:** Se creează o listă (vector) de dimensiune fixă, egală cu intervalul valorilor pentru care se dorește calculul frecvenței. (*De exemplu*, de la 0 la 10 pentru note.) **Pasul 2:** Se inițializează fiecare element al listei cu 0. **Pasul 3:** Se parcurge setul de date de intrare, element cu element. **Pasul 4:** Pentru fiecare element, se incrementează cu 1 frecvența din lista la indexul egal cu valoarea elementului.

Pseudocod:

```
Creează o listă de frecvență L unde L[i] ← 0 pentru fiecare valoare unică din setul de date de intrare
Pentru fiecare element x din setul de date de intrare:
    L[x] ← L[x] + 1
sfârșit pentru
```

Avantaje

- ✓ Organizare clară a datelor de același fel
- ✓ Eficiența din punct de vedere al memoriei
- ✓ Calcul rapid al frecvenței datelor (%)
- ✓ Identificarea ușoară a unor valori (valoarea cu frecvența maximă, cu frecvența minimă, media ponderată)
- ✓ Bază pentru reprezentări grafice (histogramă, diagramă circulară)
- ✓ Viteză de procesare a unui volum mare de date
- ✓ Util în analize statistice și rapoarte.

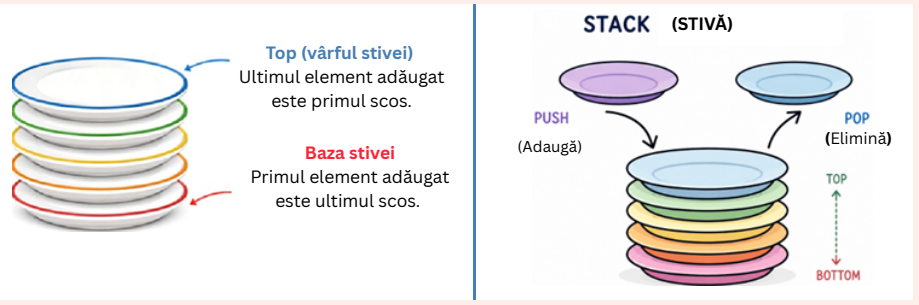
Dezavantaje

- × Nu păstrează ordinea originală a datelor
- × Datele individuale dispar, deci se poate pierde contextul (nu mai știm ce notă a luat un anumit elev)
- × Ineficiență pentru valori unice

Stiva (Stack) – Principiul LIFO

Stiva este o structură de date liniară în care elementele sunt inserate și șterse doar dintr-un singur capăt, numit ***vârf*** **(top)**. Principiul de funcționare al acesteia este: ***LIFO – Last In, First Out*** (ultimul intrat, primul ieșit).

EX. În bucătăria unui restaurant, farfuriile sunt așezate una peste cealaltă, formând o stivă. Când este nevoie de o farfurie, este luată cea mai de sus (din vârf). O farfurie nouă este așezată peste farfuriile existente (în vârf).



Operații fundamentale:

- **Push** – adaugă un element în vârful stivei;
- **Pop** – elimină un element din vârful stivei;
- **Top** – vizualizează elementul din vârful stivei fără a-l elimina;
- **IsEmpty** – verifică dacă stiva este goală.

Avantaje

- ✓ Implementare simplă
- ✓ Acces rapid la ultimul element
- ✓ Viteză de execuție: Operațiile Push și Pop sunt extrem de rapide (au complexitate O(1)[1]).

Dezavantaje

- × Acces limitat (doar vârful)
- × Nu permite acces direct

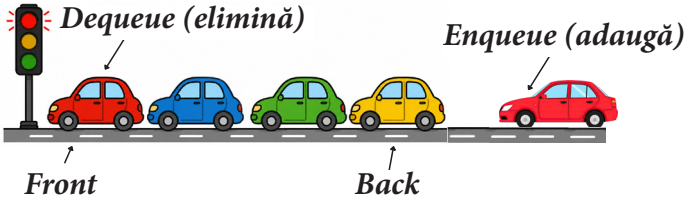
EX. **O stivă** poate fi formată din:

- Un teanc de lucrări de pe biroul unui profesor (vor fi corectate în ordinea în care se află în stivă);
- Mai multe cărți, puse una peste cealaltă.

Coada (Queue) – Principiul FIFO

Coada este o structură de date liniară în care elementele sunt inserate la un capăt numit ***spate*** (**rear/tail**) și eliminate de la celălalt capăt, numit ***față*** (**front/head**). Principiul de funcționare al acesteia este: ***FIFO – First In, First Out*** (primul intrat, primul ieșit).

EX. La semafor, mașinile sunt oprite în ordinea în care ajung și vor pleca în aceeași ordine.



Operații fundamentale:

- **Enqueue** – adaugă un element în coadă (mereu la sfârșitul cozii);
- **Dequeue** – elimină un element din coadă (mereu elementul de la începutul cozii);
- **Front/Peek** – permite vizualizarea primului element din coadă fără a-l elimina;
- **IsEmpty** – verifică dacă este goală coada.

[1] Operațiile Push și Pop au complexitate O(1) – timp constant, neinfluențat de dimensiunea stivei.

Avantaje
- ✓ Implementare simplă
- ✓ Acces rapid la primul element și la ultimul
- ✓ Viteză de execuție: Operațiile de adăugare (Enqueue) și eliminare (Dequeue)[2] sunt extrem de rapide (au complexitate **O(1)**).

Dezavantaje
- × Acces limitat la alte elemente decât primul și ultimul.
- × Nu permite acces direct (dacă se dorește accesarea unui element de pe poziția x, trebuie eliminate elementele aflate înaintea lui în coadă)

EX. Documentele trimise la tipar sunt gestionate prin operații de tip ***Enqueue*** (când se dă "Print") și ***Dequeue*** (când imprimanta termină de tipărit o pagină).

OBS În implementarea unui algoritm, dacă va conta ordinea adăugării elementelor în listă, se alege o coadă, dacă se dorește ca ultimul element adăugat să aibă prioritate se alege o stivă.

Criteriu	Listă	Stivă (Stack)	Coadă (Queue)
Principiu de funcționare	Acces liber: Elementele pot fi accesate, inserate sau șterse de pe orice poziție.	LIFO (Last In, First Out): Ultimul element introdus este primul extras.	FIFO (First In, First Out): Primul element introdus este primul extras.
Puncte de acces	Oricare element (prin parcurgere de la capăt/capete).	Un singur capăt, numit Vârf (Top).	Două capete: Spate - pentru inserare și Față - pentru ștergere.
Operație de inserare	Inserare la început, la sfârșit sau în interior.	Push (adaugă în vârful stivei).	Enqueue / Add (adaugă la sfârșitul cozii).
Operație de ștergere	Ștergere de pe orice poziție.	Pop (extrage din vârful stivei).	Dequeue / Remove (extrage de la începutul cozii).
Operație de vizualizare	Acces/Citire pe orice poziție.	Peek / Top (vede elementul din vârf, fără a-l șterge).	Peek / Front (vede primul element, fără a-l șterge).
Exemple din viața de zi cu zi	O listă de cumpărături sau un playlist de melodii.	Un set de farfurii spălate puse una peste alta sau istoricul din browser (Undo).	Oameni care așteaptă la casă, într-un magazin sau imprimanta care printează documente.

Fișa de lucru 19

Creați dosarul numit *Fisa19*. Rezolvați exercițiile următoare, notând răspunsurile corecte într-un fișier *raspunsuri.docx*.

Pentru cerințele 1 și 2 de mai jos, alegeți răspunsul corect.

1. O stivă este o structură de date gestionată după principiul:

a. Primul intrat – primul servit (F.I.F.O); **b.** Ultimul intrat – primul servit (L.I.F.O);
c. Nu are principiu de gestionare; **d.** Acces direct la orice element.

2. Asociați scenariile de mai jos cu structura de date corespunzătoare (Stivă/LIFO sau Coadă/FIFO) și explicați alegerea variantei.

a. Istoricul de navigare dintr-un browser (butonul "Back"): se apasă butonul Back pentru a te întoarce la ultima pagină vizitată.

[2] Operația dequeue pe liste mari este ineficientă. Pentru liste mari se recomandă deque cu popleft().

b. Imprimanta de la birou: 5 colegi trimit documente la printat într-o perioadă limitată.
c. Drive-thru la fast-food: mașinile intră pe o bandă și sunt servite în ordinea sosirii.
d. Gestionarea apelurilor la un centru de suport tehnic.

3. O companie de servicii IT a colectat răspunsurile a 40 de clienți la întrebarea: "Cât de mulțumit ești de serviciile noastre?". Variantele de răspuns au fost: *Foarte mulțumit (FM)*, *Mulțumit (M)*, *Neutru (N)*, *Nemulțumit (NM)*, *Foarte nemulțumit (FNM)*. Se cere o analiză a datelor colectate pentru a îmbunătăți serviciile oferite de firmă, răspunsurile colectate fiind: *FM, FM, N, M, FM, NM, FNM, M, FM, M, N, FM, M, FM, M, FM, M, N, M, FM, NM, FM, M, FM, N, M, FM, FM, M, FM, N, M, NM, FM, M, FM, M, NM, FNM, M*. Pentru a analiza datele în scopul îmbunătățirii serviciilor, calculați frecvența răspunsurilor și completați tabelul de mai jos în fișierul *raspunsuri.docx*:

Calificativ	Frecvența	Procent	Interpretare
Foarte mulțumit	15	37.5%	Cel mai frecvent răspuns – serviciu apreciat
Mulțumit			
Neutru			
Nemulțumit			
Foarte nemulțumit			
TOTAL			

4. ***Proiect:*** Creați un **model conceptual** al unui sistem de gestiune pentru un centru de livrări modern. Provocarea constă în organizarea „drumului" unui colet folosind inteligent trei structuri de date diferite, astfel încât fluxul de lucru să fie optimizat și să nu existe blocaje: o listă (depozitul general), o coadă / queue (lista cu comenzile), o stivă pentru modul în care curierul așează produsele pentru a fi livrate.

Autoevaluare

Completați răspunsurile într-un fișier *FisaAutoevaluare19.docx*, folosind notații precum *1 – da*, *2 – nu*, *3 – nu sunt sigur*. Salvați fișierul în dosarul *Fisa19*.

	La finalul lecției, știu...	*Da*	*Nu*	*Nu sunt sigur(ă)*
1.	să definesc o listă de frecvență.			
2.	să descriu principiul LIFO al stivei și să dau exemple din viața reală.			
3.	să descriu principiul FIFO al cozii și să dau exemple din viața reală.			
4.	să identific operațiile fundamentale ale stivei.			
5.	să identific operațiile fundamentale ale cozii.			

6. Am întâmpinat dificultăți în: …

Nu uitați să vă evaluați comportamentul și activitatea din timpul lecției!
(cu ajutorul fișei de observare a comportamentului și a activității, de la pagina 7)

La finalul lecției, m-am simțit:

Lecția 20 – Clasa list din Python: operatori specifici și metode de bază

În Python, **lista** este o structură de date versatilă, folosită pentru a stoca o *colecție ordonată* și *modificabilă* de elemente. Lista este ***ordonată*** deoarece se păstrează ordinea în care au fost adăugate elementele și ***modificabilă (mutable)*** deoarece conținutul ei poate fi schimbat prin adăugare, ștergere sau înlocuire de elemente. Deși listele pot avea elemente de tipuri diferite, în practică acestea sunt folosite cel mai des pentru a stoca date de același tip. Listele sunt definite în Python prin plasarea elementelor între paranteze pătrate ***[ ]***, separate prin virgulă, sau prin utilizarea constructorului ***list()***.

```
Listă vidă: L=[]
Listă omogenă L=['galben', 'rosu', 'alb']
Listă neomogenă L=['banana', 23, 130, True, [1,2,3]]
```

```
lista_mea = list(('rosu', 'verde', 'alb'))    # ['rosu', 'verde', 'alb']
print(lista_mea)
```

```
lista_mea = ['Ana', 'Maria', 'Victor']        # ['Ana', 'Maria', 'Victor']
print(lista_mea)
```

Pentru a accesa elemente, se folosește ***operatorul de indexare*** sau ***operatorul de feliere*** (slicing).

Operatorul de indexare este reprezentat prin paranteze pătrate, sub forma **ID_lista[index]**.

Indexarea poate fi: ***pozitivă*** (de la stânga la dreapta, începând cu indexul primului element care este **0**) și ***negativă*** (de la dreapta la stânga, începând cu **-1**, care este indexul ultimului element, **-2** indexul penultimului element etc.). **L** = [**'Ana'**, **'Maria'**, **'Matei'**, **'Victor'**, **'Tudor'**]

-5	-4	-3	-2	-1
0	1	2	3	4
'Ana'	'Maria'	'Matei'	'Victor'	'Tudor'

Indexare negativă
Indexare pozitivă

L[2]
L[-3] → 'Matei'

Operatorul de feliere **(slicing)** este reprezentat prin semnul **:**, fiind folosit pentru a accesa un grup de elemente consecutive din listă sub forma **[start:stop:pas]**, unde:

- **start** este indexul de unde începe extragerea (elementul de la acest index este inclus). Dacă **start** lipsește, este considerat implicit **0**.
- **stop** este indexul unde se termină extragerea (elementul de la acest index este exclus). Dacă **stop** lipsește, se parcurge lista până la final.
- **pas** reprezintă intervalul dintre elementele extrase. Dacă lipsește, pasul implicit este **1**. Dacă **pas** are valoare negativă, lista este parcursă de la dreapta la stânga.

```
L = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']
print(L[2:4])    # ['C','D'] afișează elementele de la indicele 2 și indicele 3
print(L[:4])     # ['A','B','C','D']   afișează primele 4 elemente
print(L[2:])     # ['C','D','E','F','G','H']afișează toate elementele începând de la
                 # indicele 2
print(L[0:8:2])  # ['A','C','E','G']afișează elementele cu indicii pari, începând cu 0
print(L[0:8:3])  # ['A','D','G'] afișează elementele începând de la indicele 0, din 3 în 3
print(L[::-1])   # ['H','G','F','E','D','C','B','A'] lista în ordine inversă
print(L[5:2:-1]) # ['F','E','D'] afișarea elementelor se face de la dr. la st.
                 # (elementele cu indicii 5, 4 și 3)
print(L[:])      #['A','B','C','D','E','F','G','H'] afișează toate elementele listei
print(L[::])     #['A','B','C','D','E','F','G','H'] afișează toate elementele listei
```

Sintaxă	Elemente accesate
L[0]	primul element din listă (cel mai din stânga)
L[-1]	ultimul element din listă (cel mai din dreapta)
L[i:j]	elementele cu indexul în intervalul ***[i,j)***
L[i:]	toate elementele din listă, începând cu indexul ***i***, până la sfârșitul listei
L[::] L[:]	toate elementele listei, de la cel cu indexul ***0***, până la ultimul

Sintaxă	Elemente accesate
L[:i]	toate elementele din listă cu indexul în intervalul ***[0,i)***
L[-x:], x > 0	ultimele ***x*** elemente ale listei
L[:-x], x > 0	toate elementele listei, cu excepția ultimelor ***x***
L[::-1]	toate elementele listei, în ordine inversă

OBS Pentru a accesa elemente relativ la sfârșitul listei, se vor utiliza indecși negativi.

```
L = ['Ana', 'Maria', 'Victor', 'Matei', 'Tudor']
print(L[-4:-2])          # ['Maria', 'Victor']
print(L[:-2])            # ['Ana', 'Maria', 'Victor']
print(L[-2:])            # ['Matei', 'Tudor']
```

OBS Comportamentul Python la depășirea limitelor listei:

- Indexare individuală: Accesarea unui index care nu există va genera eroarea ***IndexError: list index out of range*** (*de exemplu*, L[10] într-o listă cu 5 elemente).
- Slicing cu **start** în afara limitelor (indexul de început depășește lungimea listei): va returna o listă vidă, fără eroare.
- Slicing cu **stop** în afara limitelor listei (indexul de sfârșit depășește lungimea listei): va lua lista până la sfârșit, fără eroare.

```
L=['Ana','Maria', 'Ion','Ema', 'Victor']
print('Ema' in L)          # True
print('Andrei' in L)       # False
print('Andrei' not in L)   # True
if('Victor' in L):
      print("Victor este in lista")
```

Pentru a verifica apartenența unor elemente la listă, se folosește **operatorul de apartenență *in***. Acesta returnează valoarea *True* dacă elementul se află în listă și *False* în caz contrar. De asemenea, există și posibilitatea verificării non-apartenenței la o listă prin utilizarea operatorului negat ***not in***.

Concatenarea a două sau mai multe liste reprezintă procesul de unire a acestora, pentru a forma o listă nouă. **Operatorul de concatenare** este **+**. Ambii operanzi trebuie să fie de tip listă.

```
L1=[1,2,3]
L2=[4,5,6]
rezultat=L1+L2
print(rezultat) # [1, 2, 3, 4, 5, 6]
```

Multiplicarea unei liste cu un număr întreg n are ca rezultat o listă nouă obținută prin repetarea listei inițiale de **n** ori. Operatorul de multiplicare este *.

OBS
- Multiplicarea poate fi utilizată ca o metodă de a inițializa o listă cu o valoare implicită.
- Dacă se încearcă multiplicarea cu un număr **n≤0**, rezultatul va fi o listă vidă.

```
L1 = [1,2,3]
L2=L1*2    # L2=[1, 2, 3, 1, 2, 3]
L3=L1*0    # L3=[]
L4=[0]*5   # L4=[0, 0, 0, 0, 0]
```

Compararea listelor se realizează cu **operatorii relaționali**: ==, !=, <, >, <=, >=. Python compară listele element cu element, în ordine lexicografică, pornind de la elementele aflate la indexul 0.

Egalitatea: două liste sunt egale dacă au aceeași lungime și toate elementele din perechile cu aceiași indici.

Inegalitatea: se compară perechile de elemente de la același index, începând cu indexul **0**. Dacă elementele comparate sunt egale, se trece la următoarea pereche. Dacă sunt diferite, rezultatul comparației dintre ele devine rezultatul final. Dacă toate elementele sunt egale dar listele au lungimi diferite, lista mai scurtă este considerată mai mică.

```
print([3,2,1]==[3,2,1])          # True
print([1,2,3]>[3,2,1])           # False
print([3,2,1]!=['mere',2,1])     # True
```

OBS Pentru a folosi operatorii <, >, <=, >= în compararea listelor, elementele care vor fi comparate trebuie să fie de același tip.

Modificarea elementelor unei liste se poate realiza în două moduri: modificarea valorii unui singur element, prin specificarea indexului său (*de exemplu*, L[3] = 7), sau înlocuirea unui grup de elemente consecutive, prin slicing. În cazul modificării prin slicing, secțiunea selectată poate fi înlocuită cu mai multe elemente, de obicei dintr-o altă listă.

```
L = [1,2,3,4,5,6]
L[2:4]=[9,10]      # L=[1,2,9,10,5,6]
L[2:4]=[11,12,13] # L=[1,2,11,12,13,5,6]
L[2:4]=[9]         # L=[1,2,9,13,5,6]
```

Înlocuirea poate fi:

- **egală** (un grup de elemente este înlocuit cu alt grup cu același număr de elemente)
- **prin extindere** – apare când lista de înlocuit are mai puține elemente decât lista cu care se realizează înlocuirea. (L1[start:stop]=L2 va înlocui elementele cu indecșii în lista **L1** specificată, cu valorile din lista **L2**).
- **prin contracție** – apare când lista de înlocuit are mai multe elemente decât lista cu care se realizează înlocuirea. (L1[start:stop]=L2 va înlocui elementele cu indecșii în lista **L1** specificată, cu valorile din lista **L2**).

Obiecte de tip listă au **metode de bază, predefinite pentru analizare și manipulare**: adăugare/inserare, ștergere/eliminare elemente din listă, extinderea listei (au ca efect modificarea lungimii listei), numărare.

Adăugarea unui element într-o listă se poate realiza la finalul listei prin **numeLista.append(element)**.

Inserarea unui element în interiorul unei liste se realizează prin **numeLista.insert(index, element)**.

EX. La sfârșitul listei L=[1,2,3,4,5], vom adăuga elementul cu valoarea 10. Pe poziția cu indexul 3 vom insera elementul cu valoarea 13.

```
L=[1,2,3,4,5,6]
L.append(10)    # L=[1,2,3,4,5,6,10]
L.insert(3,13) # L=[1,2,3,13,4,5,6,10]
```

Extinderea unei liste presupune adăugarea elementelor unei liste la sfârșitul unei alte liste. Se realizează prin metoda **extend(): numeLista.extend (obiect_iterabil)**. Un obiect este iterabil dacă poate fi parcurs element cu element.

```
L1=['Ana', 'Teo']
L2=['Ion', 'Dan']
L1.extend(L2)  # L1=['Ana','Teo','Ion','Dan']
```

Ștergerea elementelor unei liste se poate realiza prin:

- metoda **remove(element)**, șterge prima apariție a elementului: **numeLista.remove(element)**;
- metoda **clear()**, șterge toate elementele listei: **numeLista.clear()**;
- metoda **pop([index])**, care va extrage elementul de la indexul specificat din listă și îl returnează pentru utilizare. Fără index, metoda elimină și returnează ultimul element din listă.

OBS
- Dacă se utilizează metoda ***pop()*** pe o listă goală sau pentru un index care nu există, Python va genera eroare de tip IndexError.
- Dacă se utilizează metoda ***remove()*** pe o listă vidă sau dacă elementul care se dorește a fi eliminat nu există, Python va genera eroare de tip ValueError.

Determinarea primei poziții a unui element cu valoarea specificată se realizează prin metoda **index(valoare,[start,stop]): numeLista.index(valoare)**. Dacă valoarea căutată nu există, Python va genera eroare de tip ValueError, de aceea trebuie verificată existența valorii în listă înainte de utilizare.

Numărarea aparițiilor unei valori într-o listă se realizează prin metoda **count(valoare)**.

Sortarea listei presupune aranjarea elementelor într-o anumită ordine, dată de un criteriu (ordine crescătoare, ordine lexicografică etc.). Aceasta se realizează cu metoda ***sort()***, care modifică lista originală, sau cu funcția ***sorted()***, care creează și returnează o copie sortată a acesteia.

```
L = [7,6,3,8,9]
L.sort()            # L=[3,6,7,8,9]
L.insert(4,5)       # L=[3,6,7,8,5,9]
L2=sorted(L)        # L2=[3,5,6,7,8,9]
```

Pentru a fi sortată, toate elementele unei liste trebuie să fie comparabile între ele, altfel Python generează o eroare TypeError.

OBS Funcția **len()** returnează numărul de elemente dintr-o listă: **len(lista)**.

```
L = [7,6,3,8,9]
print(len(L))    #5
```

Implementarea în Python a unei stive

O variantă de implementare a unei stive în Python constă în utilizarea unei liste standard. Deoarece stiva respectă principiul LIFO (Last In, First Out), avem nevoie de două operații principale:

- **Adăugare (Push)**: prin metoda.append() (adaugă în vârf).
- **Ștergere/Extragere (Pop)**: prin metoda.pop() (șterge și returnează ultimul element).

Aplicație rezolvată: Să se implementeze o structură de date de tip stivă folosind o listă în Python.

```
stiva = []              # stiva implementata ca lista Python
stiva.append(5)         # Push - adaugare in varf
stiva.append(3)
stiva.append(7)
print(stiva[-1])        # Top - vizualizare varf: 7
stiva.pop()             # Pop - eliminare din varf: returneaza 7
print(stiva)            # [5, 3]
```

Implementarea în Python a unei cozi

O variantă simplă de implementare a unei cozi în Python constă în utilizarea unei liste standard. Deoarece coada respectă principiul FIFO (First In, First Out), avem nevoie de două operații principale:

- **Adăugare (Enqueue)**: prin metoda.append() (adaugă elementul la sfârșitul listei).
- **Ștergere/Extragere (Dequeue)**: prin metoda.pop(0), unde 0 este indicele primului element. Această metodă are complexitate O(n), deoarece deplasează toate elementele. Aceasta șterge și returnează elementul care a fost cel mai mult timp în coadă.

EX. Să se implementeze o structură de date de tip coadă folosind o listă în Python.

```
coada = []              # coada implementata ca lista Python
coada.append('Ana')     # Enqueue - adaugare la sfarsit
coada.append('Ion')
print(coada[0])         # Front - primul element: 'Ana'
coada.pop(0)            # Dequeue - eliminare din fata: returneaza 'Ana'
print(coada)            # ['Ion']
```

Reține!

Metoda	Ce face	Modifică lista originală?	Returnează valoare?
append(x)	Adaugă elementul x la finalul listei.	DA	NU
insert(i, x)	Inserează elementul x la poziția/indicele i.	DA	NU
extend(L)	Adaugă toate elementele din lista L la final.	DA	NU
remove(x)	Șterge prima apariție a elementului x.	DA	NU
clear()	Șterge toate elementele (lista devine goală).	DA	NU
pop(i)	Extrage (șterge și returnează) elementul de la poziția i. Dacă i lipsește, extrage ultimul element.	DA	DA (elementul extras)
index(x)	Returnează prima poziție/indice unde se află elementul x.	NU	DA (indicele)
count(x)	Numără de câte ori apare elementul x în listă.	NU	DA (numărul)
sort()	Sortează elementele direct în lista originală	DA	NU

Fișa de lucru 20

Creați dosarul numit *Fisa20*. Rezolvați exercițiile următoare, notând răspunsurile corecte într-un fișier *raspunsuri.docx*.

Pentru cerințele 1 și 2 de mai jos, alegeți răspunsul corect (un singur răspuns)

1. Se dă lista de numere: `L=[10, 20, 30, 40, 50]`. Ce va afișa comanda: `print(L[1:4])`?

a. `[10, 20, 30, 40]`; **b.** `[20, 30, 40, 50]`; **c.** `[10, 20, 30]`; **d.** `[20, 30, 40]`.

2. Dacă avem lista `L=['ananas', 'mere', 'pere', 'struguri', 'nuci']`, care este rezultatul operației `L[::2]`?

a. `L=['ananas', 'pere' , 'nuci'];` **b.** `L=['ananas', 'mere', 'pere'];`
c. `L=['ananas'];` **d.** `L=['ananas', 'pere', 'nuci' , 'pere', 'ananas'].`

3. Fără a rula codul Python de mai jos, scrieți ce se va afișa în fiecare dintre cele 2 situații:

```
culori=['alb', 'verde']
print(culori*2)
```

```
numere=[10, 20, 30, 40]
numere.pop(1)
print(numere)
```

4. Fie lista `L =['Teo', 'Ema', 'Ana', 'Alin', 'Mara']`. Care sunt instrucțiunile necesare pentru a efectua în ordine: a) adaugă numele Sara la sfârșitul listei; b) inserează numele Victor la începutul listei c) șterge numele Ema din listă d) extinde lista cu primele 3 persoane din lista inițială e) afișează lista finală.

Creați programul Python mini-proiect (în dosarul Fisa20) corespunzător exercițiului următor:

5. Creați un ***mini-proiect*** care să gestioneze o listă de cumpărături (lista este creată la începutul proiectului) folosind un meniu cu următoarele opțiuni: **a.** *afișare listă*; **b.** *utilizatorul este întrebat ce dorește să cumpere*; **c.** *dacă produsul există în listă afișează mesajul "Produs găsit" și îl șterge din listă, dacă nu există afișează mesajul "Produsul nu există"*; **d.** *Afișează lista actualizată*; **e.** *Exit.* Meniul va rula repetitiv până la apăsarea tastei 0.

6. Proiectați o ***mini-aplicație*** în Python care să fie asistentul tău personal, prin gestionarea sarcinilor pe care le ai de rezolvat. Trebuie să gestionați sarcinile prin adăugarea unei sarcini noi, eliminarea celor îndeplinite, prioritizarea unora dintre ele, afișarea sarcinilor urgente, afișarea sarcinilor care mai trebuie rezolvate etc. Aplicația trebuie să ruleze până când utilizatorul decide închiderea acesteia.

Autoevaluare

Completați răspunsurile într-un fișier *FisaAutoevaluare20.docx*, folosind notații precum *1 – da*, *2 – nu*, *3 – nu sunt sigur*. Salvați fișierul în dosarul *Fisa20*.

	La finalul lecției, știu...	***Da***	***Nu***	***Nu sunt sigur(ă)***
1.	să accesez elemente dintr-o listă prin indexare pozitivă și negativă.			
2.	să extrag subliste cu operatorul de slicing.			
3.	să verific apartenența unui element la o listă cu in și not in.			
4.	să folosesc operatorii +, * și operatorii relaționali pe liste.			
5.	cum se aplică metode de bază asupra listelor.			
6.	să găsesc poziția unui element cu index() și să număr aparițiile cu count().			
7.	să sortez o listă.			

8. Am întâmpinat dificultăți în: …

Nu uitați să vă evaluați comportamentul și activitatea din timpul lecției!
(cu ajutorul fișei de observare a comportamentului și a activității, de la pagina 7)

La finalul lecției, m-am simțit:

Lecția 21 – Metode avansate ale clasei list și parcurgerea elementelor

În Python, există și metode avansate ale clasei list. ***Metoda sort()*** devine o metodă avansată dacă se utilizează parametrul **key: ID_lista.sort(key=functie)**. Acesta permite sortări personalizate, prin care se specifică după ce criteriu vor fi sortate elementele. *De exemplu*, o listă de cuvinte poate fi sortată nu doar alfabetic, ci și după lungimea cuvintelor care o compun ***(Lista.sort(key=len))***, o listă de numere întregi poate fi sortată după valoarea lor absolută ***(Lista.sort(key=abs))*** etc.

```
L = ["ananas","nuci","pepene","mere","struguri"]
L.sort(key=len)  # L=['nuci', 'mere', 'ananas', 'pepene', 'struguri']
L2=[3,9,-6,7,-8,5]
L2.sort(key=abs) # L2=[3,5,-6,7,-8,9]
```

Metoda ***sort()*** este stabilă: dacă două elemente sunt egale după criteriul ***key***, ele își păstrează ordinea.

De asemenea, metoda ***sort()*** poate fi utilizată și cu parametrul ***reverse*** care, pentru valoarea **True**, are ca efect sortarea listei în ordine inversă (descrescătoare). Valoarea implicită a parametrului este **False**. Parametrii pot fi combinați, pentru a obține un criteriu complex de sortare. Iată un *exemplu* în care lista L este sortată în ordine descrescătoare după lungimea cuvintelor care o compun.

```
L1=[7,6,5,8,3,4]
L1.sort(reverse=True)          # L1=[8, 7, 6, 5, 4, 3]
L = ["ananas","nuci","pepene","mere","struguri"]
L.sort(key=len, reverse=True) # L=['struguri', 'ananas', 'pepene', 'nuci', 'mere']
```

Copierea unei liste se face prin:

- ***atribuire*** (lista1=lista2). În această situație nu se creează o listă nouă, ci doar o a doua variabilă (un alias) care pointează[3] către (face referire la) aceeași adresă de memorie ca și prima listă. Astfel, orice modificare adusă ***listei1*** se va reflecta și asupra ***listei2*** și invers.
- ***metoda copy()***. În această situație se creează o copie a primei liste, independentă de aceasta.

Copiere prin atribuire	Copiere prin metoda copy()
`L1 = [1, 2, 3]` `L2 = L1  # creez un alias listei L1` `L2.append(4) # adaug 4 la sfârșitul obiectului` `  # unic` `print(L1)  # L1=[1,2,3,4]` `print(L2)  # L2=[1,2,3,4]`	`L1 = [1, 2, 3]` `L2 = L1.copy()` `L2.append(4) # adaug 4 la sfârșitul listei L2` `print(L1)  # L1=[1,2,3]` `print(L2)  # L2=[1,2,3,4]`

- ***copiere prin slicing***. Se creează o listă nouă, cu elementele din prima listă.

Sintaxă: *lista_noua=lista_veche[start:stop]*.

```
L1=[10,20,30]
L2=L1[:]
L2.append(40)
print(L1) # [10, 20, 30]
print(L2) # [10, 20, 30, 40]
```

OBS Cu operatorul ***identitate (is)*** se poate verifica dacă două variabile indică același obiect. Acesta are comportament diferit față de ***operatorul de egalitate (==)***. Primul verifică dacă cele două variabile indică spre aceeași zonă de memorie, iar cel de-al doilea verifică dacă au același conținut. Operatorul ***is*** este mai rapid, deoarece compară două adrese de memorie, pe când operatorul de egalitate parcurge listele element cu element.

```
L1=[1,2,3]
L2=L1
L3=L1.copy()
print(L1 is L2) # True
print(L1 is L3) # False
```

[3] A pointa către o adresă de memorie înseamnă că variabila nu stochează datele în sine, ci doar adresa locației din memorie unde acestea sunt stocate. Cele două variabile „privesc" astfel spre același obiect din memorie.

```
L1 = [1, 2, 3]
L2 = L1
       # Cazul 1: Alias (referință către același obiect)
print(L1 == L2)  # True (au același conținut)
print(L1 is L2)  # True (sunt exact același obiect în memorie)
       # Cazul 2: Copie (obiecte diferite cu același conținut)
L3 = [1, 2, 3]
print(L1 == L3)  # True (conținutul este identic)
print(L1 is L3)  # False (obiecte diferite, cu același conținut)
```

Inversarea elementelor unei liste se face prin metoda ***reverse()***. Prin această metodă lista nu este sortată, doar îi sunt puse elementele în ordine inversă.

```
L = [3, 1, 4, 2]
L.sort(reverse=True)
print(L)   # [4, 3, 2, 1]
```

```
L = [3, 1, 4, 2]
L.reverse()
print(L)      # [2, 4, 1, 3]
```

Parcurgerea elementelor unei liste se poate face în mai multe moduri:

- ***parcurgerea element cu element***, o folosim atunci când ne interesează valoarea elementelor, nu și indexul lor. **Sintaxă:** *for element in lista*:
- ***parcurgerea prin index***: o folosim atunci când avem nevoie și de poziția elementului în listă. **Sintaxă**: *for contor in range(len(lista))*:
- ***parcurgerea*** avansată cu ***funcția enumerate()***: la fiecare pas al buclei for, aceasta returnează o pereche (*tuplu*) de forma (*număr_ordine, element*), adăugând un număr de ordine fiecărui element din listă, fără a schimba valoarea indexului. Iată un *exemplu* comparativ pentru o listă cu invitați la un eveniment:

```
numere = [5,2,3,4,1]
s=0
for i in numere:
    s=s+i
print(s)
```

Parcurgere element cu element	Parcurgere prin index	Parcurgere cu enumerate()
`invitati = ["Ana","Ion","Dan"]` `for i in invitati:` `  print(i)`	`invitati = ["Ana","Ion","Dan"]` `for i in range(len(invitati)):` `  print(i, invitati[i])`	`invitati = ["Ana","Ion","Dan"]` `for i,nume in enumerate(invitati):` `  print(i,nume)`
Se afișează: **Ana** **Ion** **Dan**	**Se afișează:** **0 Ana** **1 Ion** **2 Dan**	**Se afișează:** **0 Ana** **1 Ion** **2 Dan**

OBS Funcția ***enumerate()*** are sintaxa **enumerate(lista, start=valoare)**, unde start reprezintă valoarea inițială a contorului, implicit start are valoarea 0 și poate lipsi în acest caz.

```
for i, elem in enumerate(["A", "B"]):
    print(i, elem)
# Afișează: 0 A
#           1 B
```

```
for i, elem in enumerate(["A", "B"], start=1):
    print(i, elem)
# Afișează: 1 A
#           2 B
```

Crearea unei liste se poate realiza și prin **secvențe de inițializare (list comprehensions)** cu sintaxa: noua_lista = [expresie for element in iterabil], unde:

- *expresie* este valoarea care va fi adăugată în listă
- *element* este variabila care ia pe rând valorile din iterabil
- *iterabil* este o listă sau orice obiect care se poate itera

```
# crearea unei liste cu primele
# 5 numere naturale nenule
L = [x + 1 for x in range(5)]
print(L) # L=[1,2,3,4,5]
```

```
patrate = []     # Varianta clasică
for x in range(5):
    patrate.append(x**2)
# patrate=[0,1,4,9,16]
```

```
# Varianta cu List Comprehension
patrate = [x**2 for x in range(5)]
```

EX. Crearea unei liste cu primele 5 pătrate perfecte.

OBS **Adăugarea unei Condiții (if)** poate filtra elementele care să fie incluse în listă: *[expresie for element in iterabil if conditie].*

EX. Extragerea numerelor pare dintr-o listă:

```
numere = [1, 2, 3, 4, 5, 6, 7, 8]
pare = [n for n in numere if n % 2 == 0]
print(pare) # Rezultat: [2, 4, 6, 8]
```

Modificarea valorii elementului în funcție de o condiție, se face utilizând *if-else* înaintea buclei for.

EX. Modificarea elementelor dintr-o listă cu "par" sau "impar", în funcție de valoarea lor.

```
numere = [1, 2, 3, 4, 5]
numere = ["par" if numere[i] % 2 == 0 else "impar" for i in range(5)]
print(numere) # numere= ['impar', 'par', 'impar', 'par', 'impar']
```

OBS Operatorul **+=** combină **concatenarea** cu **atribuirea**, modificând direct lista existentă și evită crearea explicită a unei liste rezultat. Spre deosebire de operatorul **+** (concatenare), care creează o listă complet nouă prin combinarea celor două liste, operatorul += nu creează o listă nouă și nu necesită o listă intermediară completă, dar poate necesita realocarea memoriei interne a listei pentru a-i mări capacitatea.

```
numere = [1, 2, 3]
numere += [4, 5]
print(numere) # Rezultat: [1, 2, 3, 4, 5]
```

```
# Exemplul 1: L = L + [1, 2] (creează listă nouă)
L = [10, 20]
M = L
L = L + [1, 2]
print(L)  # [10, 20, 1, 2]
print(M)  # [10, 20]
```

```
# Exemplul 2: L += [1, 2] (modifică lista existentă)
L = [10, 20]
M = L
L += [1, 2]
print(L)  # [10, 20, 1, 2]
print(M)  # [10, 20, 1, 2]
```

Alte funcții predefinite pentru liste:

- ***sum(lista):*** Calculează suma elementelor (doar dacă sunt numerice).
- ***max(lista) / min(lista):*** Returnează cel mai mare, respectiv cel mai mic element.

Implementarea în Python a unei liste de frecvență

Implementarea unei liste de frecvență în Python se face folosind o listă standard, în care pozițiile elementelor în listă sunt chiar elementele numărate. Metoda funcționează perfect atunci când elementele numărate sunt numere întregi, pozitive și relativ mici (cum ar fi notele de la 1 la 10).

EX. Se citesc de la tastatură cele 30 de note ale elevilor dintr-o clasă, la un test. Profesorul dorește să știe cum s-au descurcat elevii, aflând care notă a fost luată de cei mai mulți elevi.

OBS Pentru rezolvare, se citesc notele una câte una și se contorizează direct într-o listă de frecvență. Nu este nevoie de salvarea tuturor notelor într-o listă, este mult mai eficientă actualizarea contorului fiecărei note pe măsură ce profesorul o introduce de la tastatură. La final, se parcurge lista de frecvență în căutarea notei (notelor) cu cele mai multe apariții.

```
# Creăm lista de frecvență pentru note (pot fi de la 1 la 10)
# Folosim 11 elemente ca să putem folosi direct indicii de la 1 la 10
frecventa = [0] * 11
print("Introduceți cele 30 de note, una câte una:")
for i in range(30):
    nota = int(input())
    frecventa[nota] += 1   # Incrementăm frecvența notei citite
# Găsim nota care a apărut de cele mai multe ori (frecvența maximă)
frecventa_maxima = max(frecventa)
# Identificăm ce notă corespunde acestei frecvențe maxime
print("Cu frecvență maximă")
for nota in range(1, 11):
    if frecventa[nota] == frecventa_maxima:
        print(nota)
```

Aplicații rezolvate:

1. Să se citească de la tastatură un număr natural nenul **n** și apoi cele **n** elemente numere naturale ale unei liste.

Varianta 1

```
n = int(input("Introduceți n: "))
# Citim elementele de pe n linii
L = []
for i in range(n):
    x = int(input())
    # se apasă ENTER după fiecare element
    L.append(x)
print(L)
```

Varianta 2

```
n = int(input("Introduceți n: ")) # utilizatorul introduce exact n valori
L = [int(x) for x in input("Introduceți elemente:").split()]
if len(L)==n: print(L)
else: print ("Nu sunt", n, "elemente")
# Elementele sunt introduse pe o singură linie, cu spațiu între ele, șirul introdus este
# tratat ca un șir de caractere care apoi este separat în entități transformate în numere
```

2. Să se afișeze elementele listei în ordinea citirii lor.

```
print(L)
```

3. Să se afișeze elementele listei în ordinea inversă citirii lor.

Varianta 1

```
print(L[::-1])
# Creează o copie completă a listei în memorie, dar inversată, apoi o afișează.
```

Varianta 2

```
for i in range(n - 1, -1, -1):
    print(L[i], end=" ")
# Lista este parcursă element cu element, de la dreapta la stânga.
```

Varianta 3

```
for element in reversed(L):
    print(element, end=" ")
# Lista este parcursă element cu element, de la dreapta la stânga, fără gestionarea
# manuală a indicilor.
```

4. Să se afișeze elementele de pe pozițiile pare ale listei.

```
print("Elemente de pe poziții pare:", L[0::2])
# Folosim slicing cu pasul 2: L[start:stop:pas]
```

5. Să se afișeze elementele pare de pe poziții impare.

Varianta 1

```
L2 = [L[i] for i in range(1, n, 2) if L[i] % 2 == 0]
print("Elemente pare de pe poziții impare:", L2)
# Folosim List Comprehension
```

Varianta 2

```
L2=[]
for i in range(1, n, 2):
    if L[i] % 2 == 0:
        L2.append(L[i])
print("Elemente pare de pe poziții impare:", L2)
# Se pornește de la indexul 1, până la n-1, cu pasul 2.
```

6. Numărul elementelor impare din listă.

```
L3 = [x for x in L if x % 2 != 0]
print("Numărul elementelor impare:", len(L3))
# Creăm o listă nouă, cu elementele impare din prima listă.
```

7. Se citește un număr natural **x**. Să se afișeze numărul elementelor din listă egale cu **x**.

```
# Folosim metoda count()
x = int(input("x= "))
nr_aparitii = L.count(x)
print(f"Elementul {x} apare de {nr_aparitii} ori în listă.")
```

8. Să se sorteze crescător doar numerele impare, cele pare rămânând pe pozițiile lor inițiale.

EX.

Input: [8, 3, 2, 9, 7, 1, 6] Output: [8, 1, 2, 3, 7, 9, 6]

Idee de rezolvare: **Pas 1:** Se creează o nouă listă cu toate numerele impare; **Pas 2:** Se sortează lista numerelor impare; **Pas 3:** Se parcurge lista originală și, ori de câte ori se identifică un loc ocupat de un număr impar, se pune în locul lui următorul număr din lista sortată.

```
L=[8,3,2,9,7,1,6]
impare = [x for x in L if x % 2 != 0]
impare.sort()
# Reintroducem elementele sortate în lista originală L
index_impar = 0
for i in range(len(L)):
    if L[i] % 2 != 0:
        L[i] = impare[index_impar]
        index_impar += 1
print(L)
```

9. Să se scrie o funcție care primește ca parametru o listă și care calculează și returnează suma tuturor numerelor impare din acea listă. Dacă nu există elemente impare, returnează valoarea **-1**.

```
def suma_numere_impare(Lista):
    suma = 0
    gasit = False              # Presupunem inițial că nu avem numere impare
    for numar in Lista:
        if numar % 2 != 0:
            suma += numar
            gasit = True       # Am găsit cel puțin unul!
    if not gasit:
        return -1
    return suma
# Apelul funcției
L1 = [2, 4, 5, 6, 7]
print("Suma valorilor impare:", suma_numere_impare(L1))
```

Fișa de lucru 21

Creați dosarul numit *Fisa21*. Rezolvați exercițiile următoare, notând răspunsurile corecte într-un fișier *raspunsuri.docx*.

Pentru cerințele 1 și 2 de mai jos, alegeți răspunsul corect.

1. Care este variabila care ia valoarea fiecărui element din listă pe rând în următoarea structură?

a. `print`; **b.** `in`; **c.** `x`; **d.** `numere`.

```
numere = [10, 20, 30]
for x in numere:
    print(x)
```

2. Ce se afișează în urma executării programului Python de mai jos?

a. `['mere', 'pere', 'struguri']` și `True`;

b. `['ananas', 'mere', 'pere']` și `False`;

c. `['mere', 'pere', 'struguri']` și `False`;

d. `['mere', 'banane', 'struguri']` și `True`.

```
L=['ananas', 'mere', 'pere' , 'struguri', 'nuci']
copie_L = L[1:4]
L[2] = 'banane'
print(copie_L)
print(L[2] == copie_L[1])
```

3. Fără a rula codul Python de mai jos, scrieți ce se va afișa în fiecare dintre cele 2 situații:

```
A = [1, 2, 3]
B = A[:]
print(A is B)
print(A == B)
```

```
L1 = [0]
L2 = L1 * 2
print(L2 == [0, 0])
```

4. Se dă lista `fructe = ["mar", "para"]` și instrucțiunea `lista_noua = fructe`. Se modifică apoi `lista_noua[0] = "banana"`. Ce fruct va fi pe prima poziție în lista *fructe*?

5. Fie lista persoanelor participante la o petrecere, în ordinea sosirii lor: `L=['Teo', 'Ema', 'Ana', 'Alin', 'Mara']`. Scrieți programul Python ***petrecere.py*** (în dosarul *Fisa21*) care, pe baza listei de mai sus, afișează:

a. *invitații în ordinea sosirii lor*;
b. *invitații în ordinea inversă sosirii lor*;
c. *invitații în ordine alfabetică, fără a modifica lista inițială*;
d. *primii doi și ultimii doi invitați sosiți.*

6. Se citește de la tastatură o listă cu n numere naturale (***n*** număr natural nenul). Să se scrie un program Python cu numele ***numere.py*** pentru a afla: **a.** *suma cifrelor unităților elementelor de pe poziții pare*; **b.** *numărul elementelor divizibile cu x, unde x este un număr natural nenul citit de la tastatură*; **c.** *produsul elementelor impare de pe poziții impare*; **d.** *numărul elementelor din listă, mai mari decât ultimul element*; **e.** *poziția primului element par și poziția ultimului element impar.*

7. Se citește de la tastatură o listă cu ***n*** numere naturale (***n*** număr natural nenul), reprezentând notele obținute de elevii unei clase la un test, transmise de profesor în ordinea numerelor de identificare ale elevilor în clasă (1, 2, ..., n). Care sunt numerele de identificare ale elevilor care au nota peste media clasei? Să se scrie un program Python cu numele ***note.py***.



Autoevaluare

Completați răspunsurile într-un fișier *FisaAutoevaluare21.docx*, folosind notații precum *1 – da, 2 – nu, 3 – nu sunt sigur*. Salvați fișierul în dosarul *Fisa21*.

	La finalul lecției, știu...	***Da***	***Nu***	***Nu sunt sigur(ă)***
1.	cum se parcurge o listă în Python.			
2.	să utilizez metode avansate pentru sortare, copiere, inversare, creare liste.			
3.	diferența dintre copierea prin atribuire (alias) și copierea independentă cu copy() sau slicing.			
4.	să inversez o listă cu reverse().			
5.	să rezolv probleme simple/complexe cu liste.			

6. Am întâmpinat dificultăți în: …

Nu uitați să vă evaluați comportamentul și activitatea din timpul lecției!
(cu ajutorul fișei de observare a comportamentului și a activității, de la pagina 7)

La finalul lecției, m-am simțit:

Evaluare sumativă - Capitolul 5

Verifică-ți cunoștințele!

(Punctaj: 90 de puncte + 10 puncte din oficiu)

Cerință: Rezolvați cerințele de mai jos într-un fișier ***evaluare_sumativa5.docx***. Salvați toate fișierele pe care le creați, în dosarul ***Capitol5***.

I. Alegeți situația/descrierea care definește cel mai bine o listă standard în limbajul Python:

a. O colecție fixă de elemente de același tip, nemodificabilă ulterior.

b. O structură liniară, ordonată, mutabilă, cu elemente identificate prin index.

c. O stivă cu un număr fix de elemente.

d. O funcție specială pentru adăugarea și eliminarea numerelor invers introduse.

II. Mai jos sunt prezentate două fragmente de cod scrise de un elev, conținând câte o eroare care blochează execuția programului. Pentru fiecare caz: **a.** Identificați linia de cod care generează eroarea; **b.** Numiți tipul de eroare produsă (**IndexError** sau **ValueError**); **c.** Scrieți codul corectat, adăugând „Mihai" în listă pe orice poziție.

Cazul 1: Catalogul clasei

Elevul dorește să afișeze ultimul element din lista cu notele unui elev, folosindu-se de lungimea listei (n).

```
note = [9, 10, 8, 7, 10]
n = len(note) # n va fi 5
# Elevul vrea să afișeze ultima notă (adică 10)
print("Ultima notă din catalog este:", note[n])
```

Cazul 2: Eliminarea unui invitat

Elevul are o listă cu invitații participanți dintr-un proiect și vrea să îl elimine pe „Mihai" din listă, folosind metoda .remove().

```
invitati_proiect=["Ana","Dan","Elena"]
invitati_proiect.remove("Mihai")
print("Lista rămasă:", invitati_proiect)
```

III. Asociați situația descrisă în coloana din stânga cu elementul de sintaxă din coloana din dreapta:

Situație / Descriere	Element de sintaxă în Python
1. Andrei verifică dacă două liste ocupă aceeași zonă RAM.	a. Va folosi metoda.sort(key=...)
2. Maria afișează elementele listei împreună cu indexul lor.	b. Va folosi funcția predefinită sorted()
3. Rareș sortează o listă după valoarea absolută, modificând-o direct.	c. Va folosi operatorul de identitate is
4. Teodor obține o listă sortată, fără a modifica lista inițială.	d. Va folosi operatorul +=
5. Ioana adaugă elementele unei liste la finalul altei liste.	e. Va folosi funcția enumerate()

IV. Se citesc un număr natural **n** și **n** numere întregi (listă cu consumul de apă înregistrat de-a lungul a n zile). Valorile negative sau >100 sunt date eronate. Scrieți un program ***apa.py*** care: **a.** afișează zilele cu erori sau mesajul *„Date corecte"*; **b.** creează o listă cu datele corecte, calculând, pe baza ei, consumul mediu sau mesajul *"Fara date corecte"* și numărul zilelor în care consumul a depășit media.

V. La un magazin, produsele sunt înregistrate într-o listă în ordinea vânzării lor. Se citesc numărul **n** și o listă cu **n** coduri de produse (între 1 și 1000), precum și un cod **x**. Scrieți un program ***magazin.py*** pentru a afla: **a.** numărul de produse distincte din listă; **b.** prima poziție a lui **x** (sau „Nu este cumpărat!"); **c.** de câte ori a fost cumpărat produsul **x**; **d.** codurile produselor nevândute, ordonate crescător; **e.** codurile produselor unice, afișate în ordinea descrescătoare a numărului de bucăți vândute din fiecare, justificând alegerea structurilor de date și a metodelor folosite.

Barem de evaluare (100 puncte) oficiu – **10p**, **subiectul I – 5p**, **subiectul II – 30p** (**caz 1 – 15p**, **caz 2 – 15p**, pentru fiecare caz: **a. 5p, b. 5p, c. 5p**), **subiectul III** – 5 × 3p = **15p** (**1-c, 2-e, 3-a, 4-b, 5-d**), **subiectul IV – 20p** (**a – 8p, b –12p**), **subiectul V** – 5 × 4p = **20p**.



Proiect

(100 puncte)

„Algoritmi la Rollercoaster”

Realizați în Python un proiect ***parc.py*** pentru un parc de distracții, folosind clasa list. Gestionați atracțiile prin acces direct și secvențial, ordonați îmbarcarea la rollercoaster folosind o structură de tip coadă, organizați depozitul de suveniruri folosind o structură de tip stivă și implementați o listă de frecvență pentru analiza recenziilor. Prezentați proiectul în fața clasei.

Barem de evaluare (100 puncte): 10p – Gestionarea listei de atracții, **10p** – Funcționarea cozii, **10p** – Funcționarea stivei, **10p** – Listă de frecvență, **10p** – Calitatea codului, **10p** – Corectitudinea teoretică a modelelor alese, **10p** – Analiza avantajelor/dezavantajelor clasei list, **10p** – Claritatea discursului și utilizarea termenilor informatici riguroși. **10p** – Implicarea egală a membrilor echipei în prezentare, **10p** – din oficiu.

(**Recomandări:** se lucrează în echipe de 2-3 elevi. Fiecare echipă acordă celorlalte echipe, pe baza unei fișe de evaluare, punctaj, iar punctajul final va fi media aritmetică a punctajelor obținute).



Investighează!

(100 puncte)

Studiu de caz: Cum stăm cu timpul?

Documentați, în fișierul ***timp.docx***, următoarea situație: o platformă de livrări primește zilnic zeci de mii de comenzi. Fiecare comandă nouă se adaugă în listă, iar comenzile finalizate se elimină. Viteza acestor operații contează. Folosiți surse diverse și realizați sarcinile de mai jos:

- Descoperiți cum se poate măsura timpul de execuție al unui cod Python. **(10p)**
- Folosind modulul descoperit, măsurați și comparați timpii pentru ***append() vs insert(0,x)*** și ***pop() vs pop(0)*** pe liste de 100, 10 000 și 100 000 de comenzi. Prezentați rezultatele într-un tabel. **(20p)**
- Simulați două structuri cu aceleași 100 000 de ID-uri de comenzi: o listă și un dicționar. Măsurați cât durează găsirea unui ID aflat la final în fiecare structură (cu operatorul ***in***). Repetați pentru 100 și 10 000 de elemente, notați în tabel datele obținute. Explicați diferența observată. **(20p)**
- Scrieți în 4-6 rânduri ce structură și operații ați recomanda platformei. **(10p)**
- Căutați în documentația Python o metodă sau funcție neutilizată la clasă, utilă pentru lista de comenzi (Exemplu: *filter()*, *zip()*). **(10p)** Descrieți ce face și demonstrați cu un exemplu concret. **(10p)** Rezolvați cu ea o problemă pentru lista de comenzi de mai sus (Exemplu: filtrare după zonă). **(10p)**

(10p - din oficiu)



Portofoliu

(100 puncte)

La finalul acestui capitol, portofoliul personal al elevului trebuie să cuprindă, în dosarul ***Capitol5***, următoarele materiale: dosarul ***Fisa18*** (cu fișierele ***raspunsuri.docx***, ***p6.py***, ***p7.py***, ***proiect.docx***, ***FisaAutoevaluare18.docx***), dosarul ***Fisa19*** (cu fișierele ***raspunsuri.docx***, ***FisaAutoevaluare19.docx***), dosarul ***Fisa20*** (cu fișierele ***raspunsuri.docx***, ***miniproiect.py***, ***miniaplicatie.py***, ***FisaAutoevaluare20.docx***), dosarul ***Fisa21*** (cu fișierele ***raspunsuri.docx***, ***petrecere.py***, ***numere.py***, ***note.py***, ***FisaAutoevaluare21.docx***), dosarul ***Proiecte5***, cu fișierele ***parc.py*** și ***timp.docx*** și fișierul ***evaluare_sumativa5.docx***.

Modalitate de evaluare:
- Există dosarul ***Capitol5*** cu structura de directoare specificată – **20p**
- Există în dosare:
  - Peste 95% din fișiere – **80p**
  - Între 75% și 94% din fișiere – **60p**
  - Între 50% și 74% din fișiere – **40p**
  - Între 25% și 49% din fișiere – **20p**
  - Sub 25% – **0p**

Capitolul 6 – Strategii avansate de rezolvare. Generare sistematică, sortare și eficiență algoritmică

Lecția 22 – Eficiența algoritmilor: timp de execuție, memorie și notația O(n)

„Un program bun nu este doar corect, ci și eficient."

OBS Importanța eficienței algoritmilor. În informatică, rezolvarea corectă a unei probleme reprezintă doar primul pas. La fel de important este modul în care este obținut rezultatul, din punct de vedere al timpului și al resurselor utilizate.

<def/> Un ***algoritm eficient*** permite prelucrarea rapidă a unui volum mare de date și contribuie la obținerea unor aplicații performante. În practică, diferențele dintre doi algoritmi pot deveni semnificative atunci când dimensiunea datelor crește. Pentru a analiza performanța unui algoritm, se iau în considerare două aspecte principale:



- **Timpul de execuție:** Reprezintă durata necesară pentru rularea completă a algoritmului. Aceasta este influențată de numărul de operații efectuate și de dimensiunea datelor de intrare.
- **Memoria utilizată:** Reprezintă spațiul de memorie ocupat în timpul execuției algoritmului. Aceasta include variabilele, structurile de date utilizate și eventualele spații auxiliare.

<def/> ***Măsurarea eficienței.*** Evaluarea eficienței unui algoritm nu se realizează în funcție de timpul măsurat în secunde, deoarece acesta depinde de performanța sistemului pe care rulează programul. În schimb, analiza se face în funcție de ***numărul de operații efectuate***, raportat la ***dimensiunea datelor de intrare***, notată de obicei cu **n**.

Notația O mare (Big O)

<def/> Pentru a descrie modul în care crește timpul de execuție al unui algoritm, se utilizează notația **O (Big O)**. Aceasta indică ritmul de creștere al numărului de operații în funcție de dimensiunea datelor de intrare. Astfel, putem compara diferiți algoritmi și putem alege varianta cea mai eficientă.

Tipuri uzuale de complexitate:

Notație	Denumire	Semnificație
O(1)	timp constant	Timpul de execuție nu depinde de n
O(n)	timp liniar	Timpul de execuție crește proporțional cu n
O(n^2)	timp pătratic	crește rapid pentru valori mari ale lui n
O(n^3)	timp cubic	crește foarte rapid, pentru valori mari ale lui n
O(a^n), a > 1	timp superexponențial	crește exploziv chiar și pentru valori mici ale lui n
O(log n)	timp logaritmic	crește lent, fiind specific algoritmilor eficienți

Notă: *Un exemplu tipic de algoritm cu complexitate* ***O(log n)*** *este* ***căutarea binară****, care se aplică pe liste sortate și elimină jumătate din spațiul de căutare la fiecare comparație. Pentru o listă de 1024 de elemente, sunt necesare cel mult 10 comparații ($\log_2 1024 = 10$), față de 1024 în căutarea liniară.*

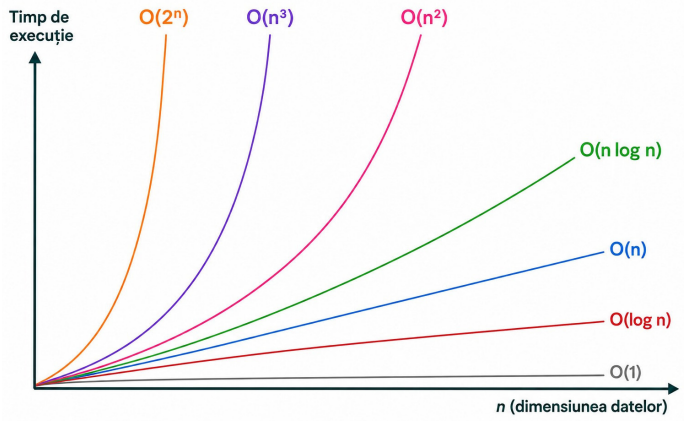
Ordinea complexităților. Din punct de vedere al eficienței, complexitățile pot fi ordonate astfel (de la cel mai eficient la cel mai lent):

$$O(1) < O(\log n) < O(n) < O(n^2) < O(n^3) < O(a^n)$$

OBS Se observă că, pe măsură ce complexitatea crește, algoritmul devine mai lent și mai dificil de utilizat pentru volume mari de date.

Graficul complexităților O(n): Reprezentarea grafică a acestor funcții evidențiază diferențele dintre ele: unele cresc lent (logaritmic), în timp ce altele cresc foarte rapid (pătratic sau exponențial).

Se observă că diferențele dintre complexități devin semnificative pentru valori mari ale lui n.



Exemple în Python

EX. În continuare sunt prezentate câteva exemple de algoritmi implementați în limbajul Python, în vederea evidențierii modului de determinare a complexității în funcție de structura instrucțiunilor utilizate.

Parcurgerea unei liste: Enunț: Se dă o listă de numere naturale. Să se afișeze toate elementele listei.	Bucle imbricate: Enunț: Se dă un număr natural ***n***. Să se afișeze toate perechile de forma ***(i,j)***, unde ***i*** și ***j*** iau valori de la ***0*** la ***n−1***.	Acces direct la element: Enunț: Se dă o listă de numere și o poziție ***k***. Să se afișeze elementul aflat pe poziția ***k***.
`v = [1, 2, 3, 4, 5]` `for x in v:` `  print(x)`	`n = 5` `for i in range(n):` `  for j in range(n):` `  print(i, j)`	`# citim poziția k` `k=int(input())` `v = [10, 20, 30]` `print(v[k])`
Complexitate: **O(n)** Fiecare element este parcurs o singură dată.	Complexitate: **O(n^2)** Numărul de operații crește foarte rapid.	Complexitate: **O(1)** Accesul se face instant.

<def/> *Cazuri de analiză.* Un algoritm poate avea comportamente diferite în funcție de datele de intrare. Astfel se analizează trei cazuri:

- **caz favorabil** – cel mai rapid scenariu posibil
- **caz mediu** – comportament obișnuit
- **caz nefavorabil** – cel mai lent scenariu posibil

EX. **Analiza cazurilor pentru căutare liniară:** *Se consideră o listă de numere întregi și o valoare dată x. Să se determine dacă valoarea x apare în listă.*

Caz favorabil	Caz mediu	Caz nefavorabil
• v = [7, 2, 9, 1, 4] • **x=7** • Element pe prima poziție → 1 comparație	• v = [7, 2, 1,9, 4] • **x=1** • Elementul este la mijloc → ~ n/2 comparații	• v = [7, 2, 9, 1, 4] • **x=5** • Elementul nu există → n comparații

Caz favorabil. Situația în care elementul căutat se găsește pe prima poziție. Exemplu: ***x = 7***. Se compară direct cu primul element. Elementul este găsit imediat. Numărul de comparații este egal cu **1**. Algoritmul se oprește foarte repede.

Caz mediu. Situația în care elementul se află undeva în interiorul listei. Exemplu: ***x = 1***. Se verifică mai multe elemente până la găsire. În medie, elementul este găsit la jumătatea listei. Numărul de comparații este aproximativ **n / 2**. Timpul de execuție este moderat.

Caz nefavorabil. Situația în care elementul nu există în listă sau se află pe ultima poziție. Exemplu: ***x = 5***. Se parcurge întreaga listă. Nu se găsește elementul. Numărul de comparații este egal cu **n**, iar algoritmul execută toate verificările posibile.

Concluzie: Complexitatea căutării liniare este **O(n)** — în cazul nefavorabil, algoritmul parcurge întreaga listă. În general, analiza eficienței se face pentru **cazul nefavorabil**, deoarece acesta oferă o estimare sigură a performanței.

Se dă o listă de numere întregi. Să se verifice dacă există valori duplicate, adică două sau mai multe elemente care au aceeași valoare.

Varianta ineficientă – O(n²)

```
v = [1, 2, 3, 2]
# presupunem că nu există duplicate
ok = False
for i in range(len(v)):
    # comparăm cu elementele următoare
    for j in range(i + 1, len(v)):
        if v[i] == v[j]:
        # am găsit două valori egale
            ok = True
# afișează True dacă există duplicate
print(ok)
```

Comparăm fiecare element cu toate celelalte

Varianta eficientă – O(n) în medie

```
v = [1, 2, 3, 2]
# mulțime pentru valorile deja întâlnite
s = set()
# presupunem că nu există duplicate
ok = False
for x in v:
    if x in s:
        # am găsit un duplicat
        ok = True
        break
        # adăugăm elementul în set
    s.add(x)
# afișează True dacă există duplicate
print(ok)
```

Folosim un set pentru a reține valorile văzute deja

Notă: set() este o colecție Python care reține valori unice și permite verificarea apartenenței (x in s) foarte rapid, în timp constant în medie. Va fi studiată în clasa a X-a

Compararea metodelor. În prima variantă, fiecare element este comparat cu ***toate celelalte → multe operații***. În a doua variantă, fiecare element este verificat ***o singură dată → mai puține operații***. De aceea, a doua metodă este mult mai eficientă, mai ales pentru liste mari.

Deși ambele metode rezolvă corect problema, diferența de performanță devine semnificativă atunci când numărul de elemente este mare.

Exerciții rezolvate

Exercițiul 1. *Determină complexitatea următoarelor fragmente de cod:*

a)
```
v = [10, 20, 30, 40, 50]
print(v[len(v) - 1])
```

Secvența **a)** afișează ultimul element dintr-o listă.

Complexitate: **O(1)** – accesul se face instant, indiferent de lungimea listei.

b)
```
v = [3, 1, 7, 2, 5]
maxim = v[0]
for x in v:
    if x > maxim:
        maxim = x
print(maxim)
```

Secvența **b)** afișează elementul maxim dintr-o listă.

Complexitate: **O(n)** – fiecare element este parcurs o singură dată.

c)
```
n = 5
for i in range(1, n+1):
    for j in range(1, n+1):
        print(i * j)
```

Secvența **c)** afișează toate produsele **i×j** pentru **i** și **j** de la **1** la **5**.

Complexitate: **O(n²)** – pentru fiecare valoare a lui **i**, se parcurg toate valorile lui **j**.

Exercițiul 2. *Se dă o listă de numere întregi. Sunt propuse două variante pentru calculul sumei elementelor. Compară cele două variante din punct de vedere al complexității și al clarității codului. Argumentează.*

Varianta A	**Varianta B**
un ciclu **for** care parcurge lista și adună fiecare element	folosirea funcției predefinite **sum(lista)**
`v = [1, 2, 3, 4, 5]` `suma = 0` `for x in v:` `    suma += x` `print(suma)`	`v = [1, 2, 3, 4, 5]` `suma = sum(v)` `print(suma)`
Funcția **sum()** parcurge intern toate elementele, exact ca ciclul **for**, deci complexitatea este **O(n)** în ambele cazuri. ***Varianta B*** este preferată pentru claritatea codului, nu pentru eficiență.	



De reținut!

- ✔ Un algoritm eficient utilizează optim timpul și memoria;
- ✔ Complexitatea descrie performanța algoritmului;
- ✔ Diferențele dintre algoritmi devin semnificative pentru date mari;
- ✔ Ne interesează în special cazul nefavorabil (cel mai lent);
- ✔ Alegerea metodei potrivite este esențială.



Fișa de lucru 22

Creați dosarul numit *Fisa22*. Rezolvați exercițiile următoare, notând răspunsurile corecte într-un fișier *raspunsuri.docx*.



I. *Alegeți varianta corectă:*

1. Complexitatea O(n) înseamnă:
- **a.** timpul nu depinde de n;
- **b.** timpul crește proporțional cu n;
- **c.** timpul crește foarte rapid.

2. Care dintre următoarele este mai eficientă?
- **a.** $O(n^2)$;
- **b.** O(log n);
- **c.** $O(n^3)$.

3. Notația Big O reprezintă:
- **a.** timpul exact de execuție;
- **b.** numărul de operații în funcție de n;
- **c.** memoria utilizată.

4. Relația corectă este:
- **a.** $O(n^2)$ este mai eficient decât O(n);
- **b.** O(n) este mai eficient decât $O(n^2)$;
- **c.** $O(n^3)$ este mai eficient decât O(log n).

II. *Completați:*

- Complexitatea O(1) reprezintă timp ..... .
- Complexitatea $O(n^2)$ apare când avem ..... .
- Analiza se face în general pentru cazul ..... .

III. *Determinați complexitatea:*

```
for i in range(n):
    print(i)
```

Răspuns: .....

```
for i in range(n):
    for j in range(n):
        print(i, j)
```

Răspuns: .....

IV. *Răspundeți la următoarele cerințe*

1. Explică: Ce înseamnă eficiența unui algoritm?

2. Argumentați:

De ce $O(n^2)$ devine ineficient pentru n mare?

3. Analizați:

Care este diferența dintre metoda cu două bucle și metoda cu set()? Comparați complexitățile.

V. *Aplicație*

Se dă lista:

v = [1, 2, 3, 2]

Cerință:

- verificați dacă există elemente duplicate;
- scrieți două metode;
- precizați complexitatea.

TEMĂ – <u>Studiu de caz – Compararea eficienței algoritmilor</u>

Enunț: Realizați un studiu de caz în care comparați două metode de rezolvare a aceleiași probleme: o metodă cu complexitate $O(n^2)$ și o metodă cu complexitate O(n).

Sarcini de lucru:

Alegeți o problemă simplă (de exemplu: verificarea existenței valorilor duplicate într-o listă).

Scrieți două variante de rezolvare:

- varianta ineficientă (cu două bucle – $O(n^2)$)
- varianta eficientă (cu set() – O(n) în medie)

Testați algoritmii pentru:

- n=10
- n=100
- n=1000

Notați: numărul aproximativ de operații sau timpul de execuție (dacă știți să îl măsurați).

Autoevaluare

Completați răspunsurile într-un fișier *FisaAutoevaluare22.docx*, folosind notații precum *1 – da*, *2 – nu*, *3 – nu sunt sigur*. Salvați fișierul în dosarul *Fisa22*.

	La finalul lecției, știu...	*Da*	*Nu*	*Nu sunt sigur(ă)*
1.	ce este eficiența unui algoritm.			
2.	să identific complexitatea unui fragment de cod Python.			
3.	ce reprezintă notația Big O și ce descrie ea.			
4.	să ordonez complexitățile de la cea mai eficientă la cea mai lentă.			
5.	să explic diferența dintre cazul favorabil, mediu și nefavorabil.			
6.	să aleg între două variante de rezolvare, pe baza complexității lor.			

7. Am întâmpinat dificultăți în: …

Nu uitați să vă evaluați comportamentul și activitatea din timpul lecției!
(cu ajutorul fișei de observare a comportamentului și a activității, de la pagina 7)

La finalul lecției, m-am simțit:

Lecția 23 – Generare sistematică: șiruri recurente și secvențe cu proprietăți date

OBS În Informatică, apar frecvent situații în care trebuie să construim pas cu pas o listă de valori respectând o anumită regulă. Acest proces poartă numele de ***generare sistematică***. Spre deosebire de calculul izolat al unui singur element, generarea produce **întreaga secvență** de la primul termen până la cel dorit.

<def/> O ***secvență (șir)*** este o listă ordonată de valori: a_1, a_2, a_3, …, a_n. Fiecare valoare din secvență se numește **termen**. *Exemple* de secvențe frecvente:

- 1, 2, 3, 4, 5 (numerele naturale consecutive)
- 2, 4, 6, 8, 10 (numerele pare)
- 0, 1, 1, 2, 3, 5, 8 (șirul Fibonacci)
- 1, 4, 9, 16, 25 (pătratele perfecte)



Metode de generare sistematică. Există trei metode principale de generare a elementelor unei liste, fiecare adaptată unui tip specific de problemă:

Metoda	Descriere	Exemplu
1. Prin formulă	Fiecare termen se calculează direct dintr-o expresie matematică ce depinde de indice.	$a_i = 2 \cdot i$, i = 0, 1, ..., n-1 → 0, 2, 4, 6…
2. Cu proprietăți	Se testează fiecare valoare; se include în secvență doar dacă satisface o condiție dată.	numere prime, pătrate perfecte
3. Recurentă	Fiecare termen depinde de termenii anteriori ai secvenței. Nu se poate calcula direct.	Fibonacci, dobânda compusă

<def/> ***Metoda 1. Generare prin formulă directă.*** Atunci când există o relație matematică directă între indicele ***n*** și valoarea termenului, fiecare element poate fi determinat ***independent***, fără a fi necesară cunoașterea termenilor anteriori. În acest caz, bucla parcurge indicii și calculează fiecare termen folosind formula corespunzătoare.

Multiplii lui 2 $a_i = 2 \cdot i$	Pătrate perfecte $a_n = n^2$	Suma primilor n termeni $S_n = n \cdot (n+1)/2$
`n = int(input())` `for i in range(n):` `  print(2 * i)`	`n = int(input())` `for i in range(1, n+1):` `  print(i * i)`	`n = int(input())` `for i in range(1, n+1):` `  print(i*(i+1)//2)`

<def/> ***Metoda 2. Generarea cu proprietăți date.*** În anumite situații, dorim să construim o secvență formată doar din numerele naturale care îndeplinesc o anumită condiție. Pentru aceasta, algoritmul parcurge numerele candidate posibile și le selectează doar pe cele care satisfac proprietatea impusă.

Numere pare Condiția: i % 2 == 0	Numere prime Condiția: dacă nu are divizori	Multiplii de 3 sau 5 Două condiții legate cu or
`n = int(input())` `for i in range(n):` `  if i % 2 == 0:` `  print(i)`	`import math` `n = int(input())` `for i in range(n):` `  if i >= 2:` `  este_prim = True` `  for d in range(2, int(math.sqrt(i)) + 1):` `  if i % d == 0:` `  este_prim = False` `  break` `  if este_prim:` `  print(i)`	`n = int(input())` `for i in range(n):` `  if i%3==0 or i%5==0:` `  print(i)`

Metoda 3. Șiruri recurente. Un ***șir recurent*** este un șir în care fiecare termen depinde de unul sau mai mulți termeni anteriori. Spre deosebire de formula directă, fiecare termen depinde de termenii anteriori ai secvenței și, de regulă, nu poate fi calculat direct fără a cunoaște valorile precedente. Pentru a descrie complet un șir recurent, sunt necesare:

- ***Termenii inițiali*** (condiții de inițializare) – valori date explicit (a_1, a_2, …).
- ***Relația de recurență*** – formula care leagă termenul curent de cei anteriori.

Recurența de ordinul 1. Fiecare termen depinde de imediat precedentul. Este necesară o singură variabilă pentru starea curentă.

EX. Se consideră un șir de numere în care primul termen este 3, iar fiecare termen se obține prin înmulțirea termenului anterior cu 2. Să se afișeze primii n termeni ai șirului.

Notăm: primul termen $\mathbf{a_1 = 3}$, cu relația de recurență $\mathbf{a_n = 2 \cdot a_{(n-1)}}$ pentru n > 1 → Șirul generat: 3, 6, 12, 24, 48, …

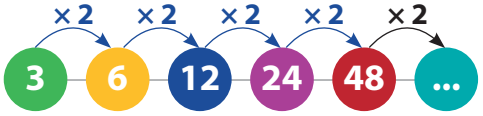
```
# citim numărul de termeni
n = int(input("Nr. termeni: "))
# primul termen al șirului (a0 = 3)
a = 3  # afișăm termenul inițial
print(a, end=" ")
for i in range(1, n):
        # aplicăm relația de recurență:
        a = 2 * a
        # afișăm termenul obținut
        print(a, end=" ")
```

```
Nr. termeni: 5
3 6 12 24 48
```

$\mathbf{a_1 = 3}$
$\mathbf{a_n = 2 \cdot a_{n-1}\ (n \geq 1)}$

⚠ **Atenție!** În șirurile recurente termenii inițiali se afișează înaintea termenilor calculați.

3
$2 \cdot 3 = \mathbf{6}$
$2 \cdot 6 = \mathbf{12}$
$2 \cdot 12 = \mathbf{24}$
$2 \cdot 24 = \mathbf{48}$



Recurența de ordinul 2 – **Șirul Fibonacci.** Fiecare termen depinde de cei doi termeni anteriori. Sunt necesare **două variabile de stare**.

EX. Se consideră șirul Fibonacci, în care fiecare termen se obține adunând cei doi termeni anteriori. Să se genereze primii ***n*** termeni ai șirului și să se observe modul de formare al acestuia.

$\mathbf{F_0 = 0,\ F_1 = 1,}$
$\mathbf{F_n = F_{n-1} + F_{n-2}\ (n \geq 2)}$

$0 + 1 = \mathbf{1}$
$1 + 1 = \mathbf{2}$
$1 + 2 = \mathbf{3}$
$2 + 3 = \mathbf{5}$...



Semnificație. Șirul Fibonacci apare în natură (spirale de scoici, aranjamente de petale, filotaxie), în artă (proporția de aur: raportul F(n+1)/F(n)=φ ≈ 1,618) și în algoritmi de căutare și optimizare.

Algoritm Pseudocod	Implementare Python
citește n **p ← 0** **d ← 1** **scrie p** **dacă n > 1 atunci** **scrie d** **pentru i ← 3, n execută** **c ← p + d** **scrie c** **p ← d** **d ← c** **sfârșitPentru** **sfârșitDacă**	`n = int(input("Nr. termeni: "))` `# F0=0, F1=1` `a, b = 0, 1` `# se afișează cel puțin un termen` `print(a, end=" ")` `# verificăm dacă există al doilea termen` `if n > 1:` `  # se afișează al doilea termen` `  print(b, end=" ")` `  for i in range(2, n):` `  # se aplică recurența` `  a, b = b, a + b  # calcul dreapta apoi` `  # schimb simultan` `  print(b, end=" ")`

```
Nr. termeni: 10
0 1 1 2 3 5 8 13 21 34
```

⚠ **Atenție!** În cazurile speciale în care numărul de termeni este mai mic decât **2** (n < 2), verificăm dacă există al doilea termen (**if n > 1**).

⚠ **Atenție!** Instrucțiunea a, b = b, a realizează ***interschimbarea*** valorilor.

În unele limbaje, interschimbarea valorilor a două variabile se realiza, de obicei, folosind o variabilă auxiliară. ***Python*** permite atribuirea simultană a două variabile într-o singură instrucțiune, fără a fi necesară o variabilă auxiliară.

EX. Un cont bancar are un sold inițial S_0. La sfârșitul fiecărei luni, soldul este actualizat prin adăugarea unei dobânzi de ***d%***. Soldul după fiecare lună depinde de soldul lunii precedente. Să se calculeze și să se afișeze soldul contului după ***n*** luni.

Aplicație practică – soldul unui cont bancar. Dacă soldul inițial este S_0 și lunar crește cu dobânda ***d%***, atunci: $S_n = S_{n-1} \cdot (1 + d/100)$. Aceasta este o recurență de ordinul 1.

Implementare Python	Evoluție sold (S_0 = 1000 RON, d = 2%) în 12 luni

```
# Citim soldul, dobânda și numărul de luni
S0 = float(input("Sold inițial: "))
d = float(input("Dobândă (%): "))
n = int(input("Număr luni: "))
# Inițializăm soldul curent cu valoarea inițială
sold = S0 # Afișăm soldul pentru luna 0 (inițial)
print("Luna", 0, ":", round(sold, 2), "RON")
# Parcurgem fiecare lună de la 1 la n
for i in range(1, n + 1):
    # Actualizăm soldul aplicând dobânda
    # Formula: Sn = Sn-1 * (1 + d/100)
    sold = sold * (1 + d / 100)
    # Afișăm soldul pentru luna curentă
    # round(sold,2) = valoarea cu 2 zecimale
    print("Luna", i, ":", round(sold, 2), "RON")
```

```
Sold inițial: 1000
Dobândă (%): 2
Număr luni: 12
Luna 0: 1000.00 RON
Luna 1: 1020.00 RON
Luna 2: 1040.40 RON
Luna 3: 1061.21 RON
Luna 4: 1082.43 RON
Luna 5: 1104.08 RON
Luna 6: 1126.16 RON
Luna 7: 1148.69 RON
Luna 8: 1171.66 RON
Luna 9: 1195.09 RON
Luna 10: 1218.99 RON
Luna 11: 1243.37 RON
Luna 12: 1268.24 RON
```

Eficiența algoritmilor de generare

<def/> Algoritmii de generare pot avea eficiențe diferite, în funcție de metoda utilizată.

- Generarea prin formulă directă parcurge secvența o singură dată → eficiență ***O(n)***.
- Generarea cu proprietăți poate varia de la ***O(n)*** (ex: numere pare) până la $O(n \cdot \sqrt{n})$ (ex: numere prime).
- În șirurile recurente, fiecare termen se calculează pe baza termenilor anteriori, iar generarea primilor ***n*** termeni are, de regulă, eficiență ***O(n)***.

Alegerea unei metode eficiente poate reduce semnificativ timpul de execuție al programului.

De reținut!

✔ Generarea sistematică produce o secvență de valori după o regulă.

✔ Metode de generare:

- formulă (calcul direct)
- proprietăți (verificarea unei condiții)
- recurență (dependență de termenii anteriori)
  - ✔ Un șir recurent este definit prin: termeni inițiali și relație de recurență
- Recurența de ordinul 1: fiecare termen depinde de termenul anterior → se folosește o singură variabilă.
- Recurența de ordinul 2: fiecare termen depinde de doi termeni anteriori → se folosesc două variabile (Exemplu: șirul lui Fibonacci).
  - ✔ Instrucțiunea: a, b = b, a + b realizează **atribuirea simultană**: noul a primește vechiul b, iar noul b primește suma valorilor vechi.
  - ✔ Eficiența algoritmilor depinde de metoda utilizată.

Fișa de lucru 23

Creați dosarul numit *Fisa23*. Rezolvați exercițiile următoare, notând răspunsurile corecte într-un fișier *raspunsuri.docx*.

I. *Completați cu răspunsurile corecte:*

1. Completați cu următorii 3 termeni șirurile:

a. 2, 5, 8, ___, ___, ___

b. 3, 6, 12, ___, ___, ___

c. 1, 1, 2, 3, 5, ___, ___, ___

d. 4, 9, 16, 25, ___, ___, ___

2. Completați următorii 3 termeni:

a. $a_0 = 2, a_n = a_{n-1} + 5 \rightarrow$

b. $a_0 = 1, a_n = 2 \cdot a_{n-1} \rightarrow$

c. $a_0 = 0, a_1 = 1, a_n = a_{n-1} + a_{n-2} \rightarrow$

II. *Alegeți varianta corectă:*

3. Un șir recurent:

a. depinde de termenii anteriori;

b. este calculat direct; **c.** nu are regulă.

4. Șirul Fibonacci este:

a. de ordin 1; **b.** de ordin 2;

c. fără recurență.

III. *Observați și formulați regula:*

5. Aplicați principiul de generare recurentă și construiți următoarele șiruri:

a. Primul termen este 2, iar fiecare termen se obține adunând 3 la termenul anterior.

b. Primul termen este 5, iar fiecare termen se obține înmulțind termenul anterior cu 2.

c. Primii doi termeni sunt 1 și 4, iar fiecare termen este suma celor doi termeni anteriori.

6. Se consideră șirul: 1, 3, 6, 10, 15, …

a. Calculați următorii 3 termeni

b. Precizați regula de formare

c. Indicați tipul (recurent / formulă)

7. Explicați:

Care este diferența dintre:

- generarea prin formulă
- generarea recurentă

8. Aplicație: Un cont bancar are soldul inițial 500 RON și crește lunar cu 10%.

a. Calculați soldul după 3 luni. **b.** Scrieți relația de recurență. **c.** Precizați tipul recurenței.

9. Analiză: Se consideră șirul: 2, 4, 7, 11, 16, …

a. Determinați regula. **b.** Calculați termenul următor. **c.** Stabiliți cum s-a obținut șirul.

10. Problemă: Se consideră șirul:

$T_n = 1^2 + 2^2 + 3^2 \ldots + n^2$

a. Calculați primii 5 termeni **b.** Ce observați?

c. Este șir recurent sau generat prin formulă?

11. ***Analizați diferența dintre tipurile de șiruri:*** un șir generat prin formulă (Exemplu: pătrate perfecte) și un șir recurent. (Exemplu: Fibonacci).

Scrieți observațiile. Comparați modul de calcul al termenilor, dependența față de valorile anterioare și ușurința determinării unui termen oarecare

IV. Realizați un program în Python pentru fiecare enunț în parte:

12. Numere impare consecutive. Se dă un număr natural **n**. Să se afișeze primele **n** numere impare pozitive.

Date de intrare:	Date de ieșire:
Se citește **n**.	Se afișează primele **n** numere impare.
Exemplu:	
5	**1 3 5 7 9**

13. Șir cu diferență constantă. Se consideră șirul în care primul termen este **100**, iar fiecare termen următor se obține scăzând 7 din termenul anterior. Să se afișeze primii **n** termeni.

Date de intrare:	Date de ieșire:
Se citește **n**.	Se afișează primii **n** termeni ai șirului.
Exemplu:	
5	**100 93 86 79 72**

14. Puteri ale lui 3. Se dă un număr natural **n**. Să se afișeze primele **n** puteri ale lui **3**, începând cu 3^0.

Date de intrare:	Date de ieșire:
Se citește n.	Se afișează valorile generate.
Exemplu:	
5	**1 3 9 27 81**

15. Creștere zilnică. O plantă are inițial înălțimea **h** centimetri. În fiecare zi crește cu **p** centimetri. Să se afișeze înălțimea plantei în fiecare dintre următoarele **n** zile.

Date de intrare:	Date de ieșire:
Se citesc **h**, **p**, **n**.	Se afișează cele **n** valori ale înălțimii.
Exemplu:	
10 3 5	**13 16 19 22 25**

16. Consum de energie. Un dispozitiv consumă în prima oră **c** unități de energie. În fiecare oră următoare consumul crește cu **2** unități. Să se determine consumul total după **n** ore.

Date de intrare:	Date de ieșire:
Se citesc **c** și **n**.	Se afișează consumul total.
Exemplu:	
5 4	**32**

Explicație: consumurile sunt 5, 7, 9, 11, iar suma este 32

17. Cod de acces. Un cod numeric începe cu valoarea **a**. La fiecare pas, noua valoare se obține după regula: **a = a * 2 + 1**. Să se afișeze primele **n** valori ale codului.

Date de intrare:	Date de ieșire:
Se citesc **a** și **n**.	Se afișează valorile generate.
Exemplu:	
2 5	**2 5 11 23 47**

18. Se dă un număr natural **n**. Să se afișeze toate numerele din intervalul [**1**, **n**] care sunt în același timp pare și pătrate perfecte.

Date de intrare:	Date de ieșire:
Se citește n.	Se afișează numerele care îndeplinesc ambele condiții.
Exemplu:	
30	**4 16**
Condiții: i % 2 == 0 și i este pătrat perfect	

19. Analizează programul Fibonacci din lecție și răspunde:

a. Ce afișează programul pentru ***n = 0***?
b. Ce afișează pentru ***n = 1***?
c. Ce afișează pentru ***n = 2***?
d. De ce este necesară instrucțiunea if ***n > 1*** în program?

Autoevaluare

Completați răspunsurile într-un fișier *FisaAutoevaluare23.docx*, folosind notații precum *1 – da*, *2 – nu*, *3 – nu sunt sigur*. Salvați fișierul în dosarul *Fisa23*.

	La finalul lecției, știu...	***Da***	***Nu***	***Nu sunt sigur(ă)***
1.	recunosc un șir recurent.			
2.	calculez termeni ai unui șir.			
3.	diferențiez formulă vs recurență.			
4.	scriu un algoritm simplu.			
5.	să identific eficiența (complexitatea) algoritmilor de generare.			

6. Am întâmpinat dificultăți în: …

Nu uitați să vă evaluați comportamentul și activitatea din timpul lecției!
(cu ajutorul fișei de observare a comportamentului și a activității, de la pagina 7)

La finalul lecției, m-am simțit:

Lecția 24 – Metode de sortare: selecția minimului

Sortarea reprezintă procesul de aranjare a elementelor unei liste într-o ordine specificată – de obicei crescătoare sau descrescătoare. Este una dintre operațiile fundamentale în informatică: datele ordonate pot fi căutate și prelucrate mult mai eficient.



Metode de sortare. În lecțiile care urmează veți studia trei metode diferite de sortare a elementelor unei liste. Fiecare are principii, avantaje și cazuri de utilizare specifice:

Metoda	Cuvânt-cheie	Idee principală
1. Selecția minimului	Minimul	Caută minimul din subșirul nesortat, **îl mută la locul său**, repetă.
2. Metoda bulelor	Adiacent	Compară elementele **adiacente**/ alăturate, interschimbă pe cele în ordine greșită, repetă.
3. Listă de frecvențe	Numărare	Numără aparițiile fiecărei valori, **reconstituie lista** în ordine. Eficientă pentru valori cu interval mic.

⚠ **Atenție!** Nu există o metodă de sortare universală optimă. Alegerea depinde de: **dimensiunea listei**, **intervalul valorilor**, **costul operațiilor** (comparare vs. interschimbare) și dacă lista este **parțial sortată**.

Metode de sortare: selecția minimului

Scenariul lecției – Concursul de talente. *Organizatorii unui concurs de talente trebuie să aranjeze 7 finaliști în ordinea punctajelor, de la cel mai mic la cel mai mare. Cum procedezi? Cauți scorul cel mai mic din toată lista, îl plasezi pe primul loc, apoi repeți procedeul pentru restul listei. Aceasta este exact ideea din spatele* ***metodei selecției minimului****.*

Cum funcționează selecția minimului? Algoritmul efectuează ***n−1 iterații*** externe. La fiecare iterație i (de la 0 la n−2), caută ***elementul minim*** din subșirul nesortat (de la poziția i la n−1) și îl aduce pe poziția i printr-o singură interschimbare.

Inițial	5	2	8	1	9	3	7	
Pas 1	1	2	8	5	9	3	7	*min=1 (poz.3) ↔ poz.0*
Pas 2	1	2	8	5	9	3	7	*min=2 (poz.1), loc corect*
Pas 3	1	2	3	5	9	8	7	*min=3 (poz.5) ↔ poz.2*
Pas 4	1	2	3	5	9	8	7	*min=5 (poz.3), loc corect*
Pas 5	1	2	3	5	7	8	9	*min=7 (poz.6) ↔ poz.4*
Pas 6	1	2	3	5	7	8	9	*min=8 (poz.5), loc corect*
Final ✓	1	2	3	5	7	8	9	*Listă sortată complet*

La finalul fiecărei iterații, un element este plasat pe poziția sa definitivă. **Subșirul sortat crește** de la stânga la dreapta, iar **subșirul nesortat se micșorează**.

Selecția minimului este un algoritm de sortare care, la fiecare pas ***i***, găsește elementul minim din subșirul ***a[i..n−1]*** și îl plasează pe poziția ***i*** prin interschimbare. Algoritmul efectuează exact ***n−1*** iterații externe și cel mult ***n−1 interschimbări***, indiferent de ordinea inițială a elementelor.

Pașii algoritmului

Pas 1. **Pornire** – întreaga listă este nesortată.

Pas 2. **Căutare** – găsim poziția elementului minim din subșirul nesortat.

Pas 3. **Plasare** – interschimbăm minimul cu primul element din subșirul nesortat.

Pas 4. **Avansare** – mutăm limita spre dreapta; elementul plasat este pe poziția definitivă.

Pas 5. **Repetare** – reluăm pașii 2–4 până când lista este complet sortată.

Algoritm Pseudocod

```
citește n
pentru i ← 0 până la n-1 execută
    citește a[i]
sfârșit pentru
pentru i ← 0 până la n-2 execută
    pos_min ← i
    pentru j ← i+1 până la n-1 execută
        dacă a[j] < a[pos_min] atunci
            pos_min ← j
        sfârșit dacă
    sfârșit pentru
    dacă pos_min ≠ i atunci
        aux ← a[i]
        a[i] ← a[pos_min]
        a[pos_min] ← aux
    sfârșit dacă
sfârșit pentru
pentru i ← 0 până la n-1 execută
    scrie a[i]
sfârșit pentru
```

Implementare Python

```
n = int(input("Număr de elemente: "))
# citire listă
a = []
for i in range(n):
    x = int(input("a[" + str(i) + "] = "))
    a.append(x)
# sortare prin selecția minimului
for i in range(n - 1):
    # presupunem că minimul e la poz. i
    pos_min = i
    for j in range(i + 1, n):
        if a[j] < a[pos_min]:
            # reține poziția noului minim
            pos_min = j
    if pos_min != i:
        # interschimbare directă
        a[i], a[pos_min] = a[pos_min], a[i]
print("Lista sortată:", a)
```

```
Număr de elemente: 7
a[0] = 5   a[1] = 2   a[2] = 8   a[3] = 1   a[4] = 9   a[5] = 3   a[6] = 7
Lista sortată: [1, 2, 3, 5, 7, 8, 9]
```

⚠ **Atenție!** Variabila ***pos_min*** reține ***poziția minimului***, nu valoarea lui. Interschimbarea are loc ***o singură dată*** la finalul fiecărei iterații externe, nu la fiecare comparare.

Criteriu	Caz favorabil	Caz mediu	Caz nefavorabil
Comparări	n(n−1)/2	n(n−1)/2	n(n−1)/2
Interschimbări	0 (deja sortată)	~n/2	n−1
Complexitate timp	$O(n^2)$	$O(n^2)$	$O(n^2)$
Memorie suplimentară	O(1)	O(1)	O(1)

De reținut! Numărul de comparări este ***identic*** în toate cazurile: $O(n^2)$. Numărul de interschimbări variază: minim ***0*** (listă deja sortată), maxim ***n−1*** (listă sortată descrescător). Algoritmul nu profită de o ordine parțială existentă.

Avantaje

- ✓ Maxim n−1 interschimbări — eficient când mutarea datelor este costisitoare
- ✓ Algoritm simplu, ușor de implementat și de înțeles
- ✓ Nu necesită memorie suplimentară — sortare în loc, O(1)
- ✓ Comportament predictibil indiferent de ordinea inițială

Dezavantaje

- × $O(n^2)$ comparări chiar dacă lista e aproape sortată
- × Nu este un algoritm stabil (ordinea egalilor poate fi modificată)
- × Ineficient pentru liste mari (n > 500–1000 elemente)
- × Nu profită de o ordine parțială existentă în listă

APLICABILITATE PRACTICĂ: Selecția minimului este potrivită pentru liste cu **număr mic de elemente** ($n \leq 50–100$) sau când interschimbarea elementelor este costisitoare. **Nu se recomandă** pentru seturi mari de date — metodele mai eficiente vor fi studiate în lecțiile următoare.

Problemă rezolvată. *Un magazin online dorește să afișeze comenzile clienților în ordine descrescătoare a valorii, pentru a evidenția cele mai importante vânzări. Valorile celor 7 comenzi sunt: [320, 85, 210, 55, 140, 390, 170]. Urmăriți sortarea listei în ordine descrescătoare folosind metoda selecției.*

Rezolvare: La fiecare pas alegem valoarea maximă din partea nesortată și o plasăm pe poziția corectă.

```
a = [320, 85, 210, 55, 140, 390, 170]
n = len(a)
for i in range(n - 1):
    pos_max = i
    for j in range(i + 1, n):
        # pentru sortarea descrescătoare doar schimbăm semnul
        if a[j] > a[pos_max]:
            pos_max = j
    a[i], a[pos_max] = a[pos_max], a[i]
print("Lista sortată:", a)
```

```
Lista sortată : [390, 320, 210, 170, 140, 85, 55]
```

Pentru sortarea descrescătoare, algoritmul rămâne același. Se modifică doar condiția de comparare: în loc să căutăm minimul, căutăm maximul. **a[j]>a[pos_max]** în loc de: **a[j]<a[pos_min]**

De reținut!

- ✔ Sortarea ordonează elementele unei liste într-o ordine dată; selecția minimului fixează la fiecare pas cel mai mic element pe poziția sa corectă.
- ✔ După fiecare iterație, partea sortată crește cu un element; algoritmul efectuează exact $n–1$ pași externi, $O(n^2)$ comparări și cel mult $n–1$ interschimbări.
- ✔ Metoda este simplă, potrivită pentru liste mici; ineficientă pentru volume mari de date.

Fișa de lucru 24

Creați dosarul numit *Fisa24*. Rezolvați exercițiile următoare, notând răspunsurile corecte într-un fișier *raspunsuri.docx*.

Pentru cerințele 1 și 2 de mai jos, alegeți răspunsul corect.

1. Care este numărul de comparări efectuat de algoritmul selecției minimului pentru o listă cu *5 elemente*?

a. 4; **b.** 10; **c.** 5; **d.** 20.

2. Se aplică selecția minimului pe lista `[6, 2, 8, 1, 4]`. Care este starea listei după `Pasul 2 (i=1)`?

a. `[1, 2, 8, 6, 4]`; **b.** `[1, 2, 6, 8, 4]`; **c.** `[2, 1, 8, 6, 4]`; **d.** `[1, 8, 2, 6, 4]`.

3. Fără a rula codul Python de mai jos, scrie ce va afișa programul în cele 2 situații:

a.

```
nre = [5, 3, 8, 1, 6]
n = len(nre)
for i in range(n - 1):
    pos_min = i
    for j in range(i + 1, n):
        if nre[j] < nre[pos_min]:
            pos_min = j
    if pos_min != i:
        nre[i], nre[pos_min] = nre[pos_min], nre[i]
print(nre)
```

Răspuns: .....

b.

```
a = [4, 4, 4, 4]
n = len(a)
nr_schimbari = 0
for i in range(n - 1):
    pos_min = i
    for j in range(i + 1, n):
        if a[j] < a[pos_min]:
            pos_min = j
    if pos_min != i:
        a[i], a[pos_min] = a[pos_min], a[i]
        nr_schimbari += 1
print(a, nr_schimbari)
```

Răspuns: .....

4. De ce selecția minimului nu este eficientă pentru liste mari?

5. Fie lista *L = [320, 85, 210, 55, 140, 390, 170]*. Completați tabelul de trasare pentru sortarea descrescătoare folosind selecția maximului, indicând pentru fiecare pas elementul selectat și acțiunea efectuată:

Pas	poz.0	poz.1	poz.2	poz.3	poz.4	poz.5	poz.6	Acțiune
Start	320	85	210	55	140	390	170	
Pas 1								
Pas 2								
Pas 3								
Pas 4								
Pas 5								
Pas 6								
Final ✓								

6. Analizați comportamentul algoritmului selecției minimului pentru următoarele cazuri limită. Pentru fiecare, precizează câte interschimbări se efectuează și de ce:

a. [1, 2, 3, 4, 5] – listă deja sortată crescător;
b. [5, 4, 3, 2, 1] – listă sortată descrescător;
c. [7] – listă cu un singur element;
d. [3, 3, 3, 3] – toate elementele egale.

7. Se dă lista de perechi (nume, notă): [("Ana", 8), ("Bobe", 9), ("Cara", 8), ("Dan", 9)]
Algoritmul selecției minimului sortează lista după notă. Răspundeți:

a. Ana și Cara au aceeași notă (8). După sortare, care dintre ele apare prima?

b. Bobe și Dan au aceeași notă (9). După sortare, care apare primul?

c. Ce înseamnă că un algoritm de sortare este stabil? Selecția minimului este stabilă?

Indicație: Un algoritm este stabil dacă elementele cu valori egale își păstrează ordinea relativă din lista inițială. Selecția minimului nu este stabilă – interschimbarea directă poate modifica ordinea elementelor egale.



Autoevaluare

Completați răspunsurile într-un fișier *FisaAutoevaluare24.docx*, folosind notații precum *1 – da*, *2 – nu*, *3 – nu sunt sigur*. Salvați fișierul în dosarul *Fisa24*.

	La finalul lecției, știu...	*Da*	*Nu*	*Nu sunt sigur(ă)*
1.	ce este sortarea și de ce este importantă în Informatică.			
2.	cum funcționează algoritmul de sortare prin selecția minimului.			
3.	cum se calculează numărul de comparări: formula n(n–1)/2.			
4.	care este complexitatea timpului algoritmului: $O(n^2)$.			
5.	care sunt avantajele și dezavantajele selecției minimului.			
6.	cum implementez algoritmul în Python fără funcții predefinite de sortare.			

7. Am întâmpinat dificultăți în: …

Nu uitați să vă evaluați comportamentul și activitatea din timpul lecției!
(cu ajutorul fișei de observare a comportamentului și a activității, de la pagina 7)

La finalul lecției, m-am simțit:

Lecția 25 – Metode de sortare: metoda bulelor

OBS Am învățat că **selecția minimului** caută, la fiecare pas ***i***, elementul minim din subșirul ***a[i..n−1]*** și îl plasează pe poziția corectă printr-o singură interschimbare. Algoritmul efectuează mereu **n(n−1)/2 comparări**, indiferent de ordinea inițială – nu poate beneficia de o ordine parțială existentă în listă.

De ce avem nevoie de altă metodă? Selecția minimului este simplă și predictibilă, dar nu profită de o ordine parțială existentă: chiar dacă lista este deja sortată, efectuează tot ***n(n−1)/2*** comparări. **Metoda bulelor** poate reduce această limitare prin variabila ***schimbat***, care permite detectarea anticipată a unei liste deja sortate, reducând numărul de comparări la doar n−1 în cazul favorabil.

Metode de sortare: metoda bulelor

<def/> Metoda bulelor ***(Bubble Sort)*** este un algoritm de sortare care ordonează elementele unui șir prin comparări și interschimbări repetate ale elementelor ***adiacente*** (alăturate). Numele provine din analogia cu bulele de aer care urcă treptat spre suprafața apei: elementele mari „plutesc” spre sfârșitul șirului, iar cele mici „coboară” spre început, la fiecare trecere prin șir.

EX. **Scenariul lecției – Clasamentul de înălțimi.** *Profesorul de sport vrea să aranjeze elevii în ordinea înălțimii. Procedează astfel: compară fiecare elev cu cel din dreapta — dacă cel din stânga e mai înalt, interschimbă locurile. Repetă procedeul pentru întreaga clasă de **n−1 ori**. Elementul cel mai mare ajunge mereu la capătul din dreapta al listei, ca o bulă care plutește la suprafață. Aceasta este exact ideea din spatele **metodei bulelor**.*

? **Cum funcționează „metoda bulelor” /„Bubble Sort”?** Algoritmul efectuează cel mult ***n−1*** treceri externe. La fiecare trecere i (de la 0 la n−2), parcurge subșirul nesortat și compară perechile de elemente ***alăturate*** a[j] și a[j+1]. Dacă a[j] > a[j+1], le interschimbă. Variabila ***schimbat*** permite oprirea timpurie: dacă o trecere nu produce nicio interschimbare, lista este deja sortată.

Urmărește starea listei de înălțimi [170, 155, 182, 163, 175, 168] cm după fiecare trecere:

Inițial	170	155	182	163	175	168	
Trecere 1	155	170	163	175	168	182	*4 interschimbări: (170↔155), (182↔163), (182↔175), (182↔168) →* ***182*** *pe poziție finală*
Trecere 2	155	163	170	168	175	182	*2 interschimbări: (170↔163), (175↔168) →* ***175*** *pe poziție finală*
Trecere 3	155	163	168	170	175	182	*1 interschimbare: (170↔168) →* ***170*** *pe poziție finală*
Trecere 4	155	163	168	170	175	182	*0 interschimbări →* ***Oprire anticipată!***
Final ✓	155	163	168	170	175	182	***Listă sortată complet***

La finalul fiecărei treceri, cel mai mare element din subșirul nesortat ajunge pe poziția finală – **plutind spre dreapta**. Variabila ***schimbat*** permite oprirea algoritmului dacă o trecere nu produce nicio interschimbare.

<def/> ***Metoda bulelor*** este un algoritm de sortare care compară, la fiecare pas ***j***, perechea de elemente alăturate ***a[j]*** și ***a[j+1]*** și le interschimbă dacă ***a[j] > a[j+1]***. Algoritmul efectuează cel mult ***n−1 treceri***. Varianta optimizată oprește algoritmul dacă o trecere nu produce nicio interschimbare (**schimbat = fals**).

<def/> **Pașii algoritmului:**

Pas 1. **Pornire** – întreaga listă este nesortată.

Pas 2. **Comparare** – comparăm fiecare pereche de elemente alăturate a[j] și a[j+1].

Pas 3. **Interschimbare** – dacă a[j] > a[j+1], le interschimbăm și marcăm schimbat ← adevărat.
Pas 4. **Avansare** – ultimul element al trecerii curente este pe locul definitiv.
Pas 5. **Repetare** – reluăm pașii 2–4 sau oprim dacă schimbat = fals.

Algoritm Pseudocod	Implementare Python

```
citeste n
pentru i <- 0 pana la n-1 executa
    citeste a[i]
sfarsit pentru
// metoda bulelor (optimizata)
pentru i <- 0 pana la n-2 executa
  schimbat <- fals
  pentru j <- 0 pana la n-2-i executa
      daca a[j] > a[j+1] atunci
          aux <- a[j]
          a[j] <- a[j+1]
          a[j+1] <- aux
          schimbat <- adevarat
      sfârșit daca
  daca schimbat = fals atunci
      opreste
  sfarsit daca
sfarsit pentru
pentru i <- 0 pana la n-1 executa
    scrie a[i]
sfarsit pentru
```

```python
n = int(input("Nr. elemente: "))
# citire lista
a = []
for i in range(n):
    x = int(input("a["+str(i)+"] = "))
    a.append(x)
# sortare metoda bulelor
schimbat = True
while schimbat:
    schimbat = False
    for j in range(n - 1):
        if a[j] > a[j + 1]:
            a[j], a[j+1] = a[j+1], a[j]
            schimbat = True
print("Lista sortata:", a)
```

```
Număr de elemente: 5
a[0] = 5   a[1] = 2   a[2] = 8   a[3] = 1   a[4] = 4
Lista sortata: [1, 2, 4, 5, 8]
```

⚠ **Atenție!** Variabila ***schimbat*** este cheia pentru varianta optimizată prezentată: dacă după o trecere completă nu s-a efectuat nicio interschimbare, lista este sortată și algoritmul se oprește. Astfel, în cazul favorabil (deja sortată), se efectuează doar **n−1 comparări — O(n)**.

Criteriu	Caz favorabil	Caz nefavorabil	Observație
Comparări	n−1	n(n−1)/2	depind de ordinea inițială
Interschimbări	0 (deja sortată)	n(n−1)/2	maxim pentru lista invers sortată
Complexitate timp	O(n)	$O(n^2)$	în cazul favorabil se oprește după prima parcurgere
Memorie suplimentară	O(1)	O(1)	nu necesită spațiu suplimentar

De reținut! Metoda bulelor *se oprește timpuriu dacă lista este sortată* – avantaj față de selecția minimului care efectuează mereu n(n−1)/2 comparări. Totuși, în caz nefavorabil, metoda bulelor efectuează mai multe ***interschimbări*** (maxim n(n−1)/2 față de n−1). Metoda bulelor este ***stabilă***: elementele egale nu își schimbă ordinea relativă.

Când folosim **metoda bulelor**?

Avantaje
- ✓ Algoritm stabil – ordinea elementelor egale se păstrează
- ✓ Oprire anticipată – eficient dacă lista e aproape sortată
- ✓ Simplu de înțeles și implementat
- ✓ Nu necesită memorie suplimentară – sortare în loc, O(1)

Dezavantaje

- × $O(n^2)$ comparări și interschimbări în caz nefavorabil
- × Multe interschimbări față de selecția minimului (n(n−1)/2 vs. n−1)
- × Ineficient pentru liste mari (n > 500–1000 elemente)
- × Nu este recomandată când interschimbarea este costisitoare

APLICABILITATE PRACTICĂ: Metoda bulelor este potrivită pentru liste ***mici sau aproape sortate*** datorită opririi timpurii. Exemplu: ordonarea programărilor zilnice ale unei clinici medicale după durata consultațiilor. **Nu se recomandă** pentru seturi mari de date.

EX.

Problemă rezolvată. *O stație meteo înregistrează temperaturile (°C) din 7 zile consecutive: [3, -1, 5, 2, 0, 4, -2]. Sortați lista în ordine crescătoare folosind metoda bulelor, identificând trecerile efectuate.*

Rezolvare: La fiecare trecere, cel mai mare element nesortat ajunge pe poziția finală.

Inițial	**3**	**-1**	**5**	**2**	**0**	**4**	**-2**	
Trecere 1	-1	3	2	0	4	-2	**5**	*5 interschimbări; 5 la loc*
Trecere 2	-1	2	0	3	-2	**4**	5	*3 interschimbări; 4 la loc*
Trecere 3	-1	0	2	-2	**3**	4	5	*2 interschimbări; 3 la loc*
Trecere 4	-1	0	-2	**2**	3	4	5	*1 interschimbare; 2 la loc*
Trecere 5	-1	-2	**0**	2	3	4	5	*1 interschimbare; 0 la loc*
Trecere 6	-2	**-1**	0	2	3	4	5	*1 interschimbare; -1 la loc*
Final ✓	**-2**	**-1**	**0**	**2**	**3**	**4**	**5**	*Listă sortată complet*

Implementare Python

```
a = [3, -1, 5, 2, 0, 4, -2]
for i in range(len(a) - 1):
     schimbat = False
     for j in range(len(a) - 1 - i):
         if a[j] > a[j+1]:
             a[j],a[j+1]=a[j+1],a[j];
             schimbat=True
     if not schimbat: break
print("Lista sortata:", a)
```

```
Lista sortata:  [-2,  -1,  0,  2,  3,  4,  5]
```

Notă – Sortare descrescătoare. Pentru a sorta o listă în **ordine descrescătoare**, este suficientă o singură modificare față de varianta crescătoare: condiția de interschimbare se inversează din **a[j] > a[j+1]** în **a[j] < a[j+1]**. Astfel, elementul minim plutește spre capătul din dreapta la fiecare trecere, iar lista se ordonează de la mare la mic.

Implementare Python

```
a = [3, -1, 5, 2, 0, 4, -2]
for i in range(len(a) - 1):
    schimbat = False
    for j in range(len(a) - 1 - i):
        if a[j] < a[j + 1]:          # condiție inversată
            a[j], a[j+1] = a[j+1], a[j]
            schimbat = True
    if not schimbat:
        break
print("Lista sortata descrescator:", a)
```

```
Lista sortata descrescator: [5, 4, 3, 2, 0, -1, -2]
```

OBS Metoda bulelor se aseamănă cu metoda selecției minimului prin complexitatea **O(n²)**, însă diferă prin mecanismul de plasare a elementelor: la metoda bulelor, interschimbările sunt locale (elemente adiacente), iar elementul **maxim** se deplasează treptat **spre capătul șirului**. La metoda selecției minimului, se selectează direct **minimul** și se plasează pe **poziția corectă**.

De reținut!

- ✔ Metoda bulelor compară elementele alăturate și le interschimbă dacă sunt în ordine greșită.
- ✔ La finalul fiecărei treceri, cel mai mare element nesortat ajunge pe poziția finală.
- ✔ Variabila schimbat permite oprirea anticipată — complexitate O(n) în cazul favorabil. Stabil: ordinea elementelor egale se păstrează.
- ✔ Cel mult n−1 treceri, complexitate O(n²) în cazul nefavorabil. Simplu, potrivit pentru liste mici sau aproape sortate (n ≤ 500).

Fișa de lucru 25

Creați dosarul numit *Fisa25*. Rezolvați exercițiile următoare, notând răspunsurile corecte într-un fișier *raspunsuri.docx*.

I. *Alegeți răspunsul corect.*

1. La finalul fiecărei treceri, în metoda bulelor:
a. cel mai mic element ajunge pe prima poziție;
b. elementele sunt complet sortate;
c. cel mai mare element nesortat ajunge pe poziția finală;
d. lista se inversează.

2. Care este rolul variabilei schimbat?
a. memorează elementul minim;
b. numără comparările;
c. permite oprirea anticipată a algoritmului;
d. sortează lista descrescător.

3. Care este complexitatea metodei bulelor în cazul nefavorabil?
a. O(1);
b. O(log n);
c. O(n);
d. O(n²).

4. Dacă lista este deja sortată crescător, algoritmul:
a. efectuează toate trecerile;
b. se oprește după prima trecere;
c. produce eroare;
d. sortează descrescător.

5. Completați spațiile libere
a. Metoda bulelor compară elemente ..... .
b. Dacă **a[j] > a[j+1]**, elementele se ..... .
c. În cazul favorabil, complexitatea timpului este ..... .

6. Asociați corect:

Situație	Comportamentul algoritmului
A. Listă deja sortată	**a.** n(n−1)/2 interschimbări
B. Listă descrescătoare	**b.** oprire după prima trecere
C. Toate elementele egale	**c.** 0 interschimbări

II. *Răspundeți complet*

7. Explicați de ce metoda bulelor este potrivită pentru liste mici sau aproape sortate.

8. Care este diferența principală dintre metoda bulelor și selecția minimului?

9. De ce metoda bulelor este considerată o metodă stabilă?

10. Urmăriți sortarea folosind metoda bulelor. Fie lista: [4, -2, 7, 1, 0, 5] Completați tabelul pentru sortarea crescătoare:

Pas	Lista obținută	Nr. interschimbări
Inițial	**4 -2 7 1 0 5**	-
Trecere 1		
Trecere 2		
Trecere 3		
Trecere 4		
Trecere 5		
Final ✓		

11. Fără a rula programul, scrieți ce se afișează:

```
a = [5, 1, 4, 2]
for i in range(len(a)-1):
    schimbat = False
    for j in range(len(a)-1-i):
        if a[j] > a[j+1]:
            a[j], a[j+1] = a[j+1],a[j]
            schimbat = True
    if not schimbat:
        break
print(a)
```

Răspuns: .....

III. *Mini-proiect Python*

12. Realizați o aplicație Python pentru analiza vizualizărilor unor videoclipuri publicate pe o platformă online. Programul va citi de la tastatură un număr n de valori reprezentând numărul de vizualizări ale videoclipurilor și le va sorta în ordine descrescătoare folosind metoda bulelor, astfel încât videoclipurile cele mai populare să apară primele.

Aplicația trebuie să afișeze:

- lista vizualizărilor sortate descrescător;
- numărul total de comparări efectuate;
- numărul total de interschimbări realizate;
- un mesaj care să indice dacă lista era deja sortată înainte de aplicarea algoritmului.

Salvați proiectul cu numele ***vizualizari.py*** în dosarul ***Fisa25*** și o captură de ecran cu programul rulat cu numele ***captură.jpg*** în ***Fisa25***.

Autoevaluare

Completați răspunsurile într-un fișier *FisaAutoevaluare25.docx*, folosind notații precum *1 – da*, *2 – nu*, *3 – nu sunt sigur*. Salvați fișierul în dosarul *Fisa25*.

	La finalul lecției, știu...	***Da***	***Nu***	***Nu sunt sigur(ă)***
1.	să explic cum funcționează metoda bulelor.			
2.	să identific rolul variabilei schimbat.			
3.	să trasez pașii algoritmului.			
4.	să calculez complexitatea metodei.			
5.	să implementez metoda bulelor în Python.			
6.	să explic avantajele și dezavantajele metodei.			

7. Am întâmpinat dificultăți în: …

Nu uitați să vă evaluați comportamentul și activitatea din timpul lecției!
(cu ajutorul fișei de observare a comportamentului și a activității, de la pagina 7)

La finalul lecției, m-am simțit:

Lecția 26 – Metode de sortare: sortarea cu listă de frecvențe

OBS Am învățat că ***metoda bulelor*** compară elementele alăturate și le interschimbă, având complexitate $O(n^2)$. Variabila ***schimbat*** permite oprire timpurie în cazul favorabil – **O(n)**. Totuși, ambele metode studiate (selecția minimului și metoda bulelor) necesită ***compararea elementelor*** și sunt limitate la $O(n^2)$ în cazul nefavorabil.

De ce avem nevoie de altă metodă? Metodele de sortare studiate până acum compară elementele între ele și, pentru liste mari, pot deveni lente. Dacă valorile sunt ***numere întregi dintr-un interval cunoscut și redus***, putem sorta fără comparații, numărând de câte ori apare fiecare valoare. Astfel, sortarea poate fi realizată mai rapid, în timp **O(n+k)** (n - numărul de elemente, k - numărul de valori posibile).

Metode de sortare: Sortarea cu listă de frecvențe (Counting Sort)

<def/> ***Sortarea cu listă de frecvențe (Counting Sort)*** este un algoritm de sortare care ***numără aparițiile fiecărei valori*** din șir într-un tablou auxiliar și reconstruiește șirul sortat fără nicio ***comparație*** între elemente. Funcționează exclusiv pentru ***valori întregi*** cu domeniu finit și este deosebit de eficient când **k** este mic comparativ cu **n**.

EX. ***Scenariul lecției – Catalogul de note.*** *Profesorul vrea să aranjeze notele a 10 elevi (numere întregi între 5 și 10) în ordine crescătoare. În loc să compare notele între ele, procedează altfel: face un tabel cu 6 căsuțe (una pentru fiecare notă posibilă 5–10) și trece prin catalog o singură dată – bifând căsuța corespunzătoare fiecărei note. Apoi rescrie catalogul, menționând fiecare notă de atâtea ori cât indică căsuța. Aceasta este exact ideea din spatele* ***sortării cu listă de frecvențe****.*

Cum funcționează? Se parcurge șirul o singură dată și se construiește tabloul **freq**, unde **freq[v – min]** reține numărul de apariții ale valorii ***v***. Apoi se reconstruiește șirul sortat parcurgând **freq** de la stânga la dreapta și adăugând fiecare valoare de atâtea ori cât indică frecvența sa. ***Nicio comparație între elemente!***

Urmărește construcția listei de frecvențe pentru notele [7, 5, 8, 5, 9, 7, 6, 8, 5, 9]:

Etapă	notă=5	notă=6	notă=7	notă=8	notă=9	Observație
Șir inițial	3 apariții	1 apariție	2 apariții	2 apariții	2 apariții	**[7,5,8,5,9,7,6,8,5,9]**
freq[ ]	**3**	**1**	**2**	**2**	**2**	freq[0..4] = [3,1,2,2,2]
Șir final ✓	5,5,5	6	7,7	8,8	9,9	[5,5,5,6,7,7,8,8,9,9]

La pasul 1 se numără aparițiile fiecărei valori în tabloul **freq**. La pasul 2 se rescrie șirul parcurgând **freq** de la ***v = min*** până la ***v = max***, adăugând ***v*** de **freq[v – min]** ori. Nicio comparație între elemente!

<def/> **Sortarea cu listă de frecvențe** (Counting Sort) este un algoritm de sortare care, pentru un șir de ***n*** numere întregi cu valori în intervalul ***[min, max]***, construiește un tablou auxiliar **freq[0..k–1]** (k = max – min + 1) unde **freq[v – min]** = numărul de apariții ale valorii ***v***, apoi reconstruiește șirul sortat fără nicio comparație. Complexitate: **O(n + k)** timp, **O(k)** memorie suplimentară.

<def/> Pașii algoritmului:

1. **Inițializare** – determină valorile min și max din șir; creează tabloul freq de dimensiune k = max – min + 1, inițializat cu 0.
2. **Numărare** – parcurge șirul a[0..n–1]; pentru fiecare element a[i] execută freq[a[i] – min] ← freq[a[i] – min] + 1.

3. **Reconstruire** – parcurge tabloul freq de la index 0 la k–1; adaugă valoarea (index + min) de freq[index] ori în șirul rezultat.
4. **Finalizare** – șirul rezultat este sortat crescător.

Algoritm Pseudocod	Implementare Python

```
citeste n
pentru i <- 0 pana la n-1 executa
    citeste a[i]
sfârșit pentru
// Pas 1: init freq
min_v <- a[0];  max_v <- a[0]
pentru i <- 1, n-1 executa
    dacă a[i]<min_v atunci min_v<-a[i]
    sfârșit dacă
    dacă a[i]>max_v atunci max_v<-a[i]
    sfârșit dacă
sfârșit pentru
k <- max_v - min_v + 1
pentru i <- 0, k-1 executa
    freq[i] <- 0
sfârșit pentru
// Pas 2: numarare
pentru i <- 0, n-1 executa
    freq[a[i]-min_v] <- freq[a[i]-min_v]+1
sfârșit pentru
// Pas 3: reconstruire
idx <- 0
pentru v <- 0, k-1 executa
    pentru j <- 1, freq[v] executa
        a[idx] <- v + min_v; idx<-idx+1
    sfârșit pentru
sfârșit pentru
pentru i <- 0, n-1 executa
    scrie a[i]
sfârșit pentru
```

```
# n>0
n = int(input("Nr. elemente: "))
# citire lista
a = []
for i in range(n):
    x = int(input("a["+str(i)+"] = "))
    a.append(x)
if len(a) == 0:
    print("Lista sortata: []")
else:
    # Pasul 1 — determinarea intervalului
    min_v = min(a)
    max_v = max(a)
    k = max_v - min_v + 1
    freq = [0] * k
    # Pas 2: numarare
    for x in a:
        freq[x - min_v] += 1
    # Pas 3: reconstruire
    idx = 0
    for v in range(k):
        for _ in range(freq[v]):
            a[idx] = v + min_v
            idx += 1
    print("Lista sortata:", a)
```

```
Numar de elemente: 10
a[0]=7  a[1]=5  a[2]=8  a[3]=5  a[4]=9
a[5]=7  a[6]=6 a[7]=8  a[8]=5  a[9]=9
Lista sortata: [5, 5, 5, 6, 7, 7, 8, 8, 9, 9]
```

⚠ **Atenție!** Funcțiile Python **min(a)** și **max(a)** parcurg șirul o dată fiecare (O(n)). Alternativ, putem inițializa min/max manual în aceeași parcurgere, rămânând tot O(n). Tabloul **freq** ocupă O(k) memorie suplimentară.

Criteriu	Caz favorabil	Caz nefavorabil	Observație
Timp numărare	O(n)	O(n)	Întotdeauna O(n) — un singur parcurs
Timp iniț./reconstr.	O(k)	O(k)	depinde de domeniu, nu de date
Total timp	O(n+k)	O(n+k)	constant față de ordinea inițială!
Comparații	0	0	Nicio comparație între elemente!
Memorie	O(k)	O(k)	tablou freq suplimentar

De reținut! Spre deosebire de metodele anterioare, complexitatea nu depinde de ordinea inițială – caz favorabil = caz nefavorabil = **O(n+k)**. Este crucial ca **k să fie mic** comparativ cu **n**.

Când folosim **sortarea cu liste de frecvențe**?

Avantaje

- ✓ O(n+k) — cel mai rapid pentru întregi în domeniu redus
- ✓ Nicio comparație între elemente — nu este supus limitei inferioare de timp specifică algoritmilor de sortare bazați pe comparații.
- ✓ Implementare simplă și intuitivă

Dezavantaje

- × Funcționează doar pentru numere întregi (nu float, nu string)
- × Ineficient dacă k >> n: O(k) memorie și inițializare inutile
- × Nu sortează obiecte compuse direct (necesită adaptare)
- × Consum ridicat de memorie pentru domenii mari (ex. $[0, 10^6]$)

Ce se întâmplă în cazurile limită?
Un algoritm corect trebuie să funcționeze nu doar pe exemple obișnuite, ci și pe **cazuri limită**.

Caz limită	Comportamentul algoritmului
Toate elementele egale	freq[0]=n, restul=0; un singur parcurs freq; O(n+1)=O(n) ✓
Un singur element	k=1, freq[0]=1; loop intern se execută o dată — corect ✓
Valori negative	min_v<0; offset=|min_v|; freq[a[i]+offset] funcționează corect ✓
k >> n (ex. n=5, k=10^6)	O(k) memorie irosită; în această situație preferăm alt algoritm ×
Listă deja sortată	Același timp O(n+k) — nu beneficiază de ordinea inițială ×

EX. **Problemă rezolvată.** *O aplicație de monitorizare a traficului reține vitezele (km/h, numere întregi) ale mașinilor detectate într-o oră: [72, 55, 88, 55, 100, 72, 61, 88, 55, 100, 72, 61]. Sortați șirul folosind lista de frecvențe.*

Rezolvare: min=55, max=100, k=46. Se construiește freq[0..45] (doar valorile prezente sunt nenule):

Etapă	55 km/h	61 km/h	72 km/h	88 km/h	100 km/h	Rezultat
Șir inițial	3	2	3	2	2	**[72,55,88,55,100,72,61,88, 55,100,72,61]**
freq[ ]	freq[0]=3	freq[6]=2	freq[17]=3	freq[33]=2	freq[45]=2	k=46, restul=0
Șir final ✓	55,55,55	61,61	72,72,72	88,88	100,100	**[55,55,55,61,61,72,72,72,88, 88,100,100]**

Implementare Python

```
# Implementare compacta
a = [72,55,88,55,100,72,61,88,55,100,72,61]
min_v, max_v = min(a), max(a)
freq = [0] * (max_v - min_v + 1)
for x in a:
  freq[x - min_v] += 1
idx = 0
for v, cnt in enumerate(freq):
     a[idx:idx+cnt] = [v + min_v] * cnt;  idx += cnt
print("Lista sortata:", a)
```

```
Lista sortata: [55, 55, 55, 61, 61, 72, 72, 72, 88, 88, 100, 100]
```

OBS Sortarea prin frecvențe are complexitatea ***O(n+k)***, iar metoda bulelor are complexitatea ***O(n²)***. Pentru algoritmul anterior, avantajul sortării prin frecvențe este redus, dar acesta crește pe măsură ce numărul elementelor devine mai mare.

Algoritm	Timp (worst)	Comparații	Stabil	Tip date
Selecția minim.	$O(n^2)$	$n(n-1)/2$	Nu	Orice
Metoda bulelor	$O(n^2)$	$n(n-1)/2$	Da	Orice
Listă frecvențe	**O(n+k)**	**0**	Nu	Întregi

De reținut!

- ✔ Sortarea cu listă de frecvențe numără aparițiile fiecărei valori fără nicio comparație între elemente.
- ✔ Complexitate O(n+k) timp și O(k) memorie suplimentară, unde k = max−min+1.
- ✔ Nefiind bazat pe comparații, nu i se aplică limita inferioară O(n log n) — funcționează exclusiv pentru numere întregi cu domeniu redus.
- ✔ Algoritm stabil: elementele egale își păstrează ordinea relativă. Potrivit pentru note, scoruri, coduri etc.
- ✔ În multe aplicații reale se recomandă funcția **sort()** din Python (Timsort, O(n log n)), care funcționează pentru orice tip de date.

Fișa de lucru 26

Creați dosarul numit *Fisa26*. Rezolvați exercițiile următoare, notând răspunsurile corecte într-un fișier *raspunsuri.docx*.

I. *Alegere multiplă – alegeți un singur răspuns corect*

1. Care este complexitatea sortării cu listă de frecvențe pentru ***n*** elemente cu valori în ***[min,max]***, ***k=max−min+1***?

a. $O(n^2)$; **b.** $O(n \log n)$; **c.** $O(n + k)$; **d.** $O(k^2)$.

2. Se aplică sortarea pe lista [3, 1, 4, 1, 3, 2]. Ce valoare are **freq[0]** (min = 1)?

a. freq[0]=3; **b.** freq[0]=2; **c.** freq[0]=1; **d.** freq[0]=0.

II. *Adevărat / Fals – încercuiți răspunsul corect.*

1. Sortarea cu listă de frecvențe nu efectuează nicio comparație între elementele listei. A / F

2. Dacă ***k*** este mult mai mare decât ***n*** (*Exemplu:* n=5, $k=10^6$), sortarea cu frecvențe este mai eficientă decât metoda bulelor. A / F

3. Complexitatea O(n+k) este aceeași indiferent de ordinea inițială a elementelor. A / F

III. *Urmărire cod – fără a rula programul, scrieți ce afișează fiecare print()*

```
a = [2, 5, 2, 1, 5, 1, 2]
min_v, max_v = min(a), max(a)
freq = [0] * (max_v - min_v + 1)
for x in a:
  freq[x - min_v] += 1
print("freq:", freq)
print("sortat:", [v + min_v for v, c in enumerate(freq) for _ in range(c)])
```

freq: ..... . sortat: ..... .

IV. ***Completați tabelul de frecvențe pentru lista [4, 2, 4, 1, 3, 2, 1] – min=1, max=4, k=4***

	val = 1	val = 2	val = 3	val = 4	Șir sortat final
freq[]	freq[0]=	freq[1]=	freq[2]=	freq[3]=	[..... , ... , ... , ... , ... , ... , ...]

V. ***Răspuns scurt***

a. De ce sortarea cu frecvențe **nu poate fi aplicată** direct pe lista [3.5, 1.2, 4.8]? Răspuns: .....

b. Două domenii reale unde sortarea cu frecvențe este ideală: 1. ..... . 2. ..... .

VI. ***Mini-program Python*** – creați fișierul ***miniproiect.py*** în dosarul ***Fisa26***, pentru programul: Citește ***n*** numere din intervalul [0,100], sortează-le cu **lista de frecvențe** (fără **sort()**), apoi afișează:

a. lista sortată crescător;

b. numărul de valori distincte (freq[i]>0);

c. mesaj dacă lista era deja sortată.

VII. ***Mini-aplicație Python*** (***miniaplicatie.py*** în ***Fisa26***):

Proiectați o aplicație pentru un concurs în care se pot obține punctaje valori întregi din intervalul [0,50], cu meniu repetat (0 - Exit) cu opțiunile:

a. adăugare punctaj;

b. clasament sortat cu lista de frecvențe;

c. punctajul maxim și câți participanți l-au obținut;

d. numărul de punctaje distincte.

Autoevaluare

Completați răspunsurile într-un fișier *FisaAutoevaluare26.docx*, folosind notații precum *1 – da*, *2 – nu*, *3 – nu sunt sigur*. Salvați fișierul în dosarul *Fisa26*.

	La finalul lecției, știu...	***Da***	***Nu***	***Nu sunt sigur(ă)***
1.	ce este sortarea cu listă de frecvențe și cum funcționează cei 3 pași ai algoritmului.			
2.	cum se construiește tabloul freq și ce semnificație are freq[v–min].			
3.	complexitatea O(n+k) și când este eficient algoritmul comparativ cu metodele prin comparații.			
4.	avantajele și dezavantajele sortării cu frecvențe (tip de date, domeniu, memorie).			
5.	cum implementez algoritmul în Python fără funcții predefinite de sortare.			

6. Am întâmpinat dificultăți în: …

Nu uitați să vă evaluați comportamentul și activitatea din timpul lecției!
(cu ajutorul fișei de observare a comportamentului și a activității, de la pagina 7)

La finalul lecției, m-am simțit:

Evaluare sumativă - Capitolul 6

Verifică-ți cunoștințele!

(Punctaj: 90 de puncte + 10 puncte din oficiu)

Cerință: Rezolvați cerințele de mai jos într-un fișier ***evaluare_sumativa6.docx***. Salvați toate fișierele pe care le creați, în dosarul ***Capitol6***.

I. Alegere multiplă – *Încercuiți litera corectă* **[3 × 5p = 15p]**

1. Care dintre complexitățile de mai jos reprezintă cel mai eficient algoritm (creșterea cea mai lentă)?

a. $O(n^2)$; **b.** $O(n)$; **c.** $O(\log n)$; **d.** $O(n^3)$.

2. Metoda de sortare care poate opri execuția anticipat, dacă lista este deja sortată, folosind o variabilă suplimentară, este:

a. Selecția minimului; **b.** Metoda bulelor; **c.** Lista de frecvențe; **d.**Toate trei.

3. Sortarea cu listă de frecvențe are complexitatea de timp:

a. $O(n^2)$; **b.** $O(n \log n)$; **c.** $O(n+k)$, unde k = max−min+1; **d.** $O(k)$.

II. Adevărat / Fals – *[A] = Adevărat • [F] = Fals* **[3 × 3p = 9p]**

[A] [F] 1. Selecția minimului efectuează întotdeauna exact n(n−1)/2 comparări, indiferent de ordinea inițială a listei.

[A] [F] 2. Metoda bulelor este stabilă: elementele egale nu își schimbă ordinea relativă.

[A] [F] 3. Eficiența unui algoritm se măsoară în secunde, deoarece aceasta este independentă de hardware.

III. Analiză de cod Python **[4p + 6p + 8p = 18p]**

```
n = int(input("Nr. termeni: "))
a, b = 0, 1
print(a, end=" ")
if n > 1:
    print(b, end=" ")
    for i in range(2, n):
        a, b = b, a + b  # (X)
        print(b, end=" ")
```

a. Ce afișează pentru **n = 8**? Scrieți toți cei **8** termeni. .....

b. Ce rol are instrucțiunea (X)? De ce nu e necesară variabilă auxiliară?

c. Care este complexitatea? Justificați în maximum 2 propoziții.

IV. Completați celulele goale **[30p]**

1. Metoda bulelor. Sortați crescător **[6, 1, 4, 8, 3]**. Completați starea după fiecare trecere și nr. interschimbări. **[4 × 3p + 2p + 4p]**

Pas	[6, 1, 4, 8, 3]					Nr. interschimbări
Inițial	**6**	**1**	**4**	**8**	**3**	.....
Trecere 1						.....
Trecere 2						.....
Trecere 3						.....
Trecere 4						.....
Final ✓						**Total: .....**

Oprire anticipată? Justificați: .....

2. Lista de frecvențe: **[8,6,9,6,10,8,7,9,6,8]** – Completați **freq[]** și scrieți șirul sortat. **[6p + 4p + 2p]**

Notă / freq[]	6	7	8	9	10
freq[ ]					

Șir sortat: .....

min_v= .....

max_v= .....

k= .....

V. Scrieți un program Python: **[18p]**

Citiți ***n*** și generați primii n termeni ai șirului $a_0 = 5$ $a_1 = 2$, $a_n = a_{n-1} + 2 \cdot a_{n-2}$ **(n≥2)**.

Exemplu: n=6 → 5 2 12 16 40 72. Atenție la **n=2**.

Punctaj: I - 15p + II - 9p + III - 18p + IV - 30p + V - 18p + Oficiu - 10p = 100p.



Proiect

(100 puncte)

„Comparatorul de sortare” – aplicație Python pentru analiza metodelor de sortare

Cerințe:

- (1) implementați cei 3 algoritmi;
- (2) meniu interactiv cu cel puțin 6 opțiuni:
- (3) afișați comparări/interschimbări/complexitate la fiecare sortare;
- (4) testați n=10, 50, 100;
- (5) tabel comparativ eficiență;
- (6) comentarii cod;
- (7) raport ***evaluare.docx***.

Fișiere: ***proiect_sortare.py*** și ***evaluare.docx***

Barem de evaluare (100 puncte): Cerința 1 – 30p, Cerința 2 – 15p, Cerința 3 – 20p, Cerința 4 – 10p, Cerința 5 – 10p, Cerința 6 – 5p, Cerința 7 – 10p.



Investighează!

(100 puncte)

Întrebare: Cum variază eficiența celor 3 metode în funcție de tipul, dimensiunea și ordinea inițială a datelor? Ipoteză: .....

Etape:

- (1) generați 3 seuri de date (aleator/↑/↓) pentru n=10, 50, 100;
- (2) scrieți ***investigheaza.py*** — cei 3 algoritmi + înregistrare operații;
- (3) completați tabelul (9 seuri);
- (4) analizați rezultatele, verificați ipoteza;
- (5) redactați ***investigheaza6.docx*** cu concluzii și recomandare.

Barem de evaluare (100 puncte): 10p – ipoteză, **25p** – program, **25p** – tabel date, **20p** – analiză, **15p** – concluzii, **5p** – raport.



Portofoliu

(100 puncte)

La finalul acestui capitol, portofoliul personal al elevului trebuie să cuprindă, în dosarul ***Capitol6***, următoarele materiale:

- Dosarul ***Fisa22***, cu fișierele: ***raspunsuri.docx***, ***aplicatie.py***, ***studiu_de_caz.py***, ***FisaAutoevaluare22.docx***;
- Dosarul ***Fisa23***, cu fișierele: ***raspunsuri.docx***, ***aplicatie1.py*** … ***aplicatie9.py***, ***FisaAutoevaluare23.docx***;
- Dosarul ***Fisa24***, cu fișierele: ***raspunsuri.docx***, ***FisaAutoevaluare24.docx***;
- Dosarul ***Fisa25***, cu fișierele: ***raspunsuri.docx***, ***vizualizari.py***, ***captura.jpg***, ***FisaAutoevaluare25.docx***;
- Dosarul ***Fisa26***, cu fișierele: ***raspunsuri.docx***, ***miniproiect.py***, ***miniaplicatie.py***, ***FisaAutoevaluare26.docx***;
- Dosarul ***Proiecte6***, cu fișierele: ***proiect_sortare.py***, ***evaluare.docx***, ***investigheaza.py*** și ***investigheaza6.docx***;
- Fișierul ***evaluare_sumativa6.docx***.

Modalitate de evaluare:
- Există dosarul ***Capitol6*** cu structura de directoare specificată – **20p**
- Există în dosare:
  - Peste 95% din fișiere – **80p**
  - Între 75% și 94% din fișiere – **60p**
  - Între 50% și 74% din fișiere – **40p**
  - Între 25% și 49% din fișiere – **20p**
  - Sub 25% – **0p**

Evaluare finală

Creați dosarul numit *evaluare_finala*. Rezolvați ***Subiectul I*** de mai jos, notând răspunsurile într-un fișier: ***raspunsuri.docx***. Acest fișier și programele Python cerute mai jos trebuie salvate în dosarul *evaluare_finala*.

SUBIECTUL I **(20 puncte)**

A. Alegeți varianta corectă: **(6 puncte)**

1. Structură repetitivă în Python este:
a) while; **b)** def; **c)** if; **d)** abs.

2. Returnează numărul de elemente dintr-o listă funcția Python:
a) count(); **b)** len(); **c)** size(); **d)** sum().

3. Principiul de funcționare FIFO caracterizează:
a) stiva; **b)** lista de frecvențe; **c)** coada; **d)** variabila.

4. Biblioteca Python utilizată pentru interfețe grafice este:
a) random; **b)** math; **c)** tkinter; **d)** os.

5. Complexitatea metodei bulelor în cazul cel mai defavorabil este:
a) O(1); **b)** O(log n); **c)** O(n); **d)** $O(n^2)$.

6. Variabilele definite în interiorul unui subprogram sunt:
a) globale; **b)** locale; **c)** constante; **d)** recursive.

B. Completați spațiile libere **(8 puncte)**

1. Reprezentarea grafică a unui algoritm se realizează prin ..... .

2. Funcția/mecanismul utilizat pentru deschiderea fișierelor text în Python este ..... .

3. Metoda care ordonează elementele unei liste în mod direct (in-place) este ..... .

4. Notația care exprimă limita superioară a complexității algoritmilor este ..... .

5. Instrucțiunea utilizată pentru definirea unui subprogram în Python este ..... .

6. Componenta Tkinter utilizată pentru introducerea textului pe o singură linie este ..... .

7. Numărul de iterații ale buclei **for i in range(5)** este ..... .

8. Funcția Python utilizată pentru valoarea absolută este ..... .

C. Stabiliți valoarea de adevăr **(6 puncte)**

1. O listă Python este modificabilă.

2. Metoda pop() elimină ultimul element al unei liste dacă nu primește parametri.

3. Tkinter permite realizarea interfețelor grafice în Python.

4. Variabilele locale pot fi utilizate în orice subprogram.

5. Metoda bulelor compară elemente aflate pe poziții alăturate.

6. Funcția round() realizează rotunjirea numerelor reale.

SUBIECTUL al II-lea **(70 puncte)**

A. La un magazin online se dorește monitorizarea vânzărilor. Astfel, managerul știe valoarea vânzărilor din prima zi de monitorizare, apoi, de-a lungul unor zile consecutive, managerul notează și câte un număr întreg reprezentând diferența dintre valoarea vânzărilor din ziua curentă și ziua precedentă. El dorește să știe numărul de zile în care vânzările au crescut față de ziua precedentă precum și numărul de zile în care vânzările au fost mai mari decât în prima zi. Scrieți un program ***magazin.py*** care să îl ajute pe manager. Fișierul de intrare **magazin.in** conține, pe o singură linie, vânzările din prima zi (număr natural), apoi valorile reprezentând diferențele zilnice consecutive (numere întregi separate prin spațiu). Să se afișeze pe ecran, numerele cerute, pe o linie, separate printr-un spațiu. De exemplu, dacă managerul notează valorile 100 -10 20 -5, se afișează pe ecran 1 2. Justificați în 2-3 rânduri tipurile de date utilizate și eficiența algoritmului.

B. Scrieți un program Python numit ***numeric.py***, care să conțină următoarele funcții:

- CMMDC(a, b) – calculează și returnează cel mai mare divizor comun al două numere naturale a și b, folosind algoritmul lui Euclid.
- OGLINDIT(n) – calculează și returnează oglinditul unui număr natural **n**.
- PRIM(n) – verifică dacă un număr natural n este prim, returnând valoarea **1** sau **0**.
- CONVERSIE(n) – calculează și returnează reprezentarea în baza **2** a numărului natural **n**.

Programul va include o interfață grafică realizată cu biblioteca **tkinter** (care va conține și un *Menu bar* cu opțiuni). Prin intermediul acesteia, utilizatorul va introduce o listă de **n** numere naturale (n≥3). La apăsarea unui buton, utilizând funcțiile definite mai sus, programul va realiza următoarele operații:

- calculează CMMDC-ul tuturor numerelor din listă;
- determină oglinditul sumei elementelor din listă;
- afișează cel mai mare număr din listă care are toate cifrele distincte (dacă nu există, afișează **-1**);
- verifică dacă oglinditul sumei este număr prim, afișând mesajul „prim” sau „neprim”;
- afișează conversia în baza **2** a acestui oglindit;
- sortează lista crescător după suma cifrelor fiecărui element (la egalitate se păstrează ordinea inițială);
- elimină elementele egale cu **0** din listă și afișează lista rezultată.

Rezultatele obținute vor fi afișate într-o zonă de text a ferestrei (widget de tip Text) și vor fi scrise în fișierul **numeric.out**, fiecare rezultat pe o linie separată.

Barem de corectare: Total – **100 puncte** (**10 puncte din oficiu**)
Subiectul I – 20 puncte (fiecare răspuns corect primește 1 punct)
I. A: **1**-a, **2**-b, **3**-c, **4**-c, **5**-d, **6**-b
I. B: 1-schemă logică/flowchart, 2-open(), 3-sort(), 4-notația O( ) sau similară, 5-def, 6-Entry, 7-5, 8-abs()
I. C: **1**-A, **2**-A, **3**-A, **4**-F, **5**-A, **6**-A
Subiectul II – 70 puncte (II.A – 20p, II.B – 50p)
II. A: Citirea corectă din fișier – **4p**, numărarea corectă a diferențelor strict pozitive – **5p**, reconstituirea vânzărilor zilnice – **3p**, compararea cu prima zi – **3p**, afișarea rezultatelor – **3p**, justificare – **2p**.
II. B: Cele 4 funcții – **16p** (**4p**/funcție), prelucrările din main – **18p**, scriere în fișier – **4p**, interfață tkinter + meniu – **12p**.

INDICAȚII ȘI RĂSPUNSURI

Capitolul 1

Fișa de lucru 1:

1. ***a)*** adevărat; ***b)*** fals; ***c)*** fals; ***d)*** adevărat. **2.** 1 - (Umple paharul albastru cu suc), 2 - (Umple paharul galben cu apă), 3 - (Ia un pahar gol de culoare mov), 4 - (Toarnă sucul din paharul albastru în paharul mov), 5 - (Toarnă apa din paharul galben în paharul albastru), 6 - (Toarnă sucul din paharul mov în paharul galben).

Fișa de lucru 2:

1. ***a)*** 175 și 2; ***b)*** algoritmul determină cel mai mare număr din cele 3 numere citite. **2.** ***a)*** fals; ***b)*** adevărat; ***c)*** fals; ***d)*** adevărat.

Fișa de lucru 3:

1. Extrage cele trei cifre ale unui număr de 3 cifre (ultima, a doua și prima) și formează un număr nou cu cifrele în ordine inversă. Practic, inversează un număr cu exact 3 cifre.

Capitolul 3

Fișa de lucru 12:

1. Clasa este un ***șablon/tipar***, iar ***obiectul*** este o ***instanță*** concretă creată după acel tipar. ***Analogie***: Clasa este ca o ***rețetă de prăjitură*** – descrie ingredientele și pașii, dar nu este ea însăși o prăjitură. Obiectul este ***prăjitura concretă*** făcută după acea rețetă. Din aceeași rețetă poți face multe prăjituri diferite (obiecte diferite). **2.** b).

Capitolul 5

Fișa de lucru 18:

1. c. **2.** b. **3.** direct și secvențial; posibilitatea de a adăuga sau elimina elemente din listă. **4.** Lista este **structură compusă de date** deoarece conține mai multe elemente de același tip; elementele unei liste sunt memorate **într-o anumită ordine**, fiecare element având o poziție (index); lista conține un **număr finit de elemente**, deoarece la un moment dat ea are o dimensiune determinată, chiar dacă aceasta poate fi modificată prin adăugare sau eliminare de elemente. **5.** a. necesită listă, deoarece se precizează că vor exista întrebări ulterioare asupra datelor, deci toate valorile trebuie memorate.

Fișa de lucru 19:

1. b. **2.** a - stivă, b - coadă, c - coadă, d - coadă.

Fișa de lucru 20:

1. d. **2.** a. **3.** Caz 1: se afișează ['alb', 'verde', 'alb', 'verde'] Caz 2: se elimină elementul cu indexul 1, lista rămasă și afișată este [10, 30, 40]. **4. a)** L.append('Sara'); **b)** L.insert(0, 'Victor'); **c)** L.remove('Ema'); **d)** L.extend(['Teo', 'Ema', 'Ana']); **e)** print(L).

Fișa de lucru 21:

1. c. **2.** c. **3.** Caz 1: **False, True** Caz 2: **True** **4.** banana **5.** a. print(L) b. print(L[::-1]) c. print(sorted(L)) d. primii 2: print(L[:2]) ultimii 2: print(L[-2:])

Capitolul 6

Fișa de lucru 22:

I: 1. b. **2.** b. **3.** b. **4.** b. **II:** O(1) – timp constant; $O(n^2)$ – bucle imbricate; – cazul cel mai defavorabil. **III:** un for range(n)→**O(n)**; două for imbricate range(n)→**$O(n^2)$** **IV: 1.** nr.operații/dim.date. **2.** crește pătratic. **3.** bucle→$O(n^2)$; set→O(n). **V:** bucle dublu→$O(n^2)$; set→O(n).

Fișa de lucru 23:

I.1: a. 11,14,17. **b.** 24,48,96. **c.** 8,13,21. **d.** 36,49,64. **I.2: a.** 7,12,17. **b.** 2,4,8. **c.** 0,1,1,2,3,5. **II: 3.** depinde de termenii anteriori. **4.** ord.2. **III: 5. a.** 2,5,8,11. **b.** 5,10,20,40. **c.** 1,4,5,9,14. **6: a.** 21,28,36. **b.** Δ crește cu 1. **c.** recurent. **7:** formulă→direct; recurență→din precedenți. **8: a.** 665,50 RON. **b.** $a_n=a_{n-1}\times 1.1$. **c.** recurent. **9: a.** Δ cresc cu 1. **b.** 22. **c.** recurent ord. 1. **10: a.** 1,5,14,30,55. **b.** $T_n=T_{n-1}+n^2$. **c.** recurent (și $n(n+1)(2n+1)/6$).

Fișa de lucru 24:

I: 1. b(10). **2.** a [1,2,8,6,4]. **3.** cod1→ [1,3,5,6,8]; cod2→ [4, 4, 4, 4] 0. **4.** $O(n^2)$–ineficient. **5:** P1: P1:390↔320; P2:320↔85; P3:210 fix; P4:170↔55; P5:140 fix; P6:85 fix → [390,320,210,170,140,85,55].

Fișa de lucru 25:

I: 1. c. **2.** c. **3.** d. **4.** b. **5.** elemente alaturate; interschimbă; O(n). **6:** A↔b, B↔a, C↔c. **II. 7:** mică/aproape sortată→eficient. **8:** Bubble–stabil; Selection–nestabil. **9:** stabil. **10:** T2 [-2, 1, 0, 4, 5, 7]/2, T3 [-2, 0, 1, 4, 5, 7]/1, T4 [-2, 0, 1, 4, 5, 7/0, T5 [-2, 0, 1, 4, 5, 7]/0, oprire. **11:** [1,2,4,5].

Fișa de lucru 26:

I: 1. c. **2.** b. **II: 1.** A. **2.** F. **3.** A. **III:** a=[2,5,2,1,5,1,2]: **freq:[2,3,0,0,2]**; sortat: **1 1 2 2 2 5 5**. **IV:** [4,2,4,1,3,2,1]: **freq[0..3]=[2,2,1,2]**; sortat: **1 1 2 2 3 4 4**. **V: a.** sortarea cu frecvențe se aplică direct doar pe numere întregi, iar valorile 3.5, 1.2 și 4.8 sunt numere reale. **b.** note școlare, vârste copii, punctaje 0..100.


AI
0.7
0.4
0.2
0.2
75%
30%
50%
20%