

CURS 1

Introducere în limbajul Python

Python este un limbaj de programare creat de către Guido van Rossum în 1991 ([https://en.wikipedia.org/wiki/Python_\(programming_language\)](https://en.wikipedia.org/wiki/Python_(programming_language))). Ultima sa versiune stabilă este 3.9, iar kitul de instalare poate fi descărcat de pe site-ul oficial al limbajului, <https://www.python.org/>. Un mediu integrat de dezvoltare foarte popular pentru Python este PyCharm și poate fi descărcat de pe site-ul companiei producătoare JetBrains: <https://www.jetbrains.com/pycharm/>.

Spre deosebire de limbajele C/C++, care sunt *limbaje compilate*, limbajul Python este un *limbaj interpretat*. Acest lucru înseamnă faptul că programele Python nu sunt transformate în *cod mașină/executabil* neportabil, care poate fi executat doar de un anumit sistem de operare (așa cum se întâmplă în cazul limbajelor compilate), ci este transformat într-un *cod intermediar bytecode* portabil, care poate fi executat de orice sistem de operare în care a fost instalată o mașină virtuală Python (Fig. 1 – sursa: <https://www.c-sharpcorner.com/article/why-learn-python-an-introduction-to-python/>).

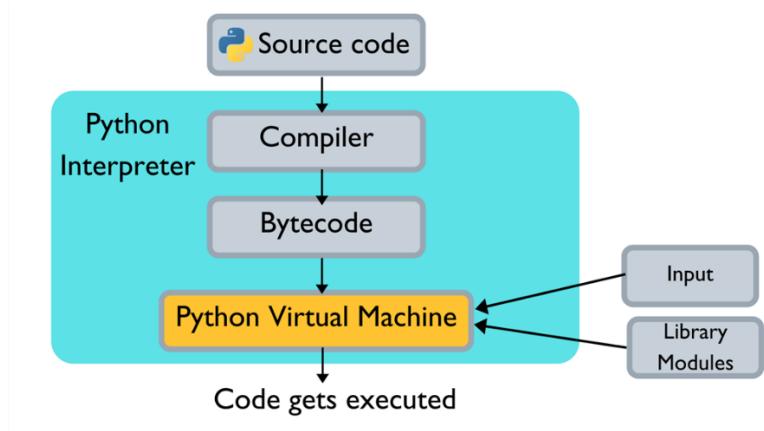


Fig. 1: Etapele executării unui program Python

Limbajul Python este un limbaj complex, care permite utilizarea mai multe paradigmă de programare: *programare procedurală*, *programare orientată pe obiecte* și *programare funcțională*. De asemenea, limbajul Python conține multe librării standard dedicate unor domenii diverse (matematică, sisteme de operare, programare concurrentă, programare distribuită etc.): <https://docs.python.org/3/library/>.

Tipuri de date

În orice limbaj de programare, un tip de date reprezintă un set de valori având o reprezentare binară internă unitară pentru care s-au definit anumite operații. De exemplu, în limbajele C/C++ valorile de tip `int` sunt reprezentate intern prin complement față de 2 pe 4 octeți și asupra lor se pot efectua diverse operații aritmetice cum ar fi adunarea, scăderea, înmulțirea și împărțirea.

În limbajul Python, fiecărui tip de date îi corespunde o anumită clasă predefinită, iar constantele și variabilele de tipul respectiv sunt instanțe ale clasei respective sau, altfel spus, obiecte. Practic, datele membre ale clasei modelează în mod unitar valorile de tipul respectiv, iar metodele sale implementează operațiile care pot fi efectuate cu valorile respective.

Principalele tipuri de date predefinite în limbajul Python sunt:

- *tipul NoneType* (clasa `NoneType`) conține o singură valoare, `None`, utilizată, de obicei, pentru a indica faptul că o funcție nu a întors nicio valoare sau pentru a inițializa un parametru al unei funcții cu o valoare implicită.
- *tipuri numerice* care permit memorarea valorilor numerice întregi, reale sau complexe:
 - *tipul întreg* (clasa `int`) permite memorarea valorilor întregi cu semn (în limbajul Python nu există tipuri de date întregi fără semn!). Literalii de tip întreg pot fi scriși în baza 10, în baza 2 folosind prefixele `0b` și `0B`, în baza 16 folosind prefixele `0x` și `0X` sau în baza 8 în folosind prefixele `0o` și `0O`.
 - *tipul real* (clasa `float`) permite memorarea valorilor de tip real ("cu virgulă") folosind reprezentarea în virgulă mobilă cu dublă precizie din standardul IEEE-754 (https://en.wikipedia.org/wiki/IEEE_754), fiind echivalent tipului de date `double` din limbajele C/C++.
 - *tipul complex* (clasa `complex`) permite memorarea numerelor complexe sub forma `a+bj`, unde `a` și `b` sunt două numere reale reprezentând partea reală și partea imaginată.
- *tipul boolean* (clasa `bool`) conține două valori, `True` și `False`. Valorile având alt tip de date sunt asimilate cu `False` dacă sunt nule (i.e., valorile numerice `0`, `0.0` și `0+0j`, precum și listele, tuplurile, multimile, dicționarele sau sirurile de caractere vide), respectiv cu `True` în caz contrar.
- *tipuri secvențiale* care permit memorarea unor siruri de valori, indexate de la 0:
 - *siruri de caractere* (clasa `str`) care permit memorarea secvențelor de caractere Unicode. Literalii de tip sir de caractere pot fi scriși folosind apostrofuri (e.g., `'Limbajul Python'`), ghilimele (e.g., `"Limbajul Python"`) sau triplu apostrof (e.g., `'''Limbajul Python'''`).
 - *liste* (clasa `list`) permit memorarea unei secvențe mutabile de valori (i.e., elementele listei pot fi modificate după ce lista a fost creată) care pot avea tipuri de date diferite (e.g., `[12, -3.14, 'Python', -20]`).
 - *tupluri* (clasa `tuple`) permit memorarea unei secvențe imutabile de valori (i.e., elementele tuplului nu mai pot fi modificate după ce tuplul a fost creat) care pot avea tipuri de date diferite (e.g., `(12, -3.14, 'Python', -20)`).
- *tipuri multime* (clasa mutabilă `set` și clasa imutabilă `frozenset`) care permit memorarea unor valori fără duplicate (*multimi*) și efectuarea operațiilor specifice multimilor (e.g., reunire, intersecție etc.). Multimile nu sunt indexate!
- *tablouri asociative* (clasa `dict`) care permit memorarea unor perechi de forma *cheie:valoare*.

Variabile

În limbajul Python o variabilă nu se declară explicit, deci nu are asociat un tip de date static (i.e., care nu mai poate fi modificat ulterior). Practic, tipul de date al unei variabile se stabilește dinamic, în funcție de valoarea pe care aceasta o primește la un moment dat. Din acest motiv, orice variabilă trebuie inițializată înainte de a fi utilizată într-un program (altfel, va fi generată o eroare)!

Pentru a determina tipul de date asociat unei variabile la un moment dat se folosește funcția `type(variabilă)`:

The screenshot shows a Python script named 'rs.py' in the left pane and its execution output in the right pane. The script contains the following code:

```

rs.py
x = 100
print("x =", x)
print("Tipul de date al variabilei x:", type(x), "\n")

x = x / 3
print("x =", x)
print("Tipul de date al variabilei x:", type(x), "\n")

x = "Ana are mere!"
print("x =", x)
print("Tipul de date al variabilei x:", type(x))

```

The output window shows the results of each print statement:

```

Run: Test_curs x
C:\Users\Ours\Desktop\Test_Python\venv\Scripts\python.exe
x = 100
Tipul de date al variabilei x: <class 'int'>
x = 33.333333333333336
Tipul de date al variabilei x: <class 'float'>

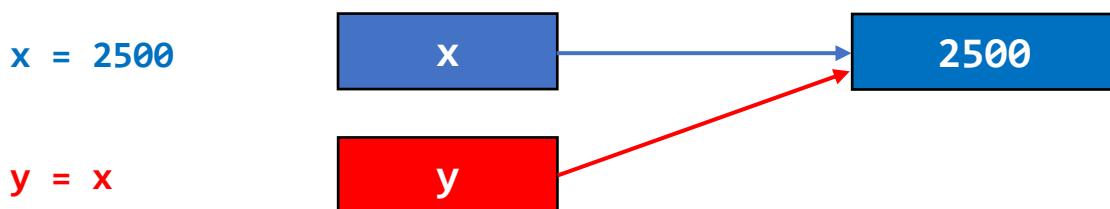
x = Ana are mere!
Tipul de date al variabilei x: <class 'str'>

Process finished with exit code 0

```

În limbajul Python, o variabilă nu conține o valoare, ci o *referință* spre un obiect care conține valoarea respectivă. Astfel, printr-o instrucțiune de atribuire nu se copiază valoarea respectivă, ci doar referința sa!

Exemplu:



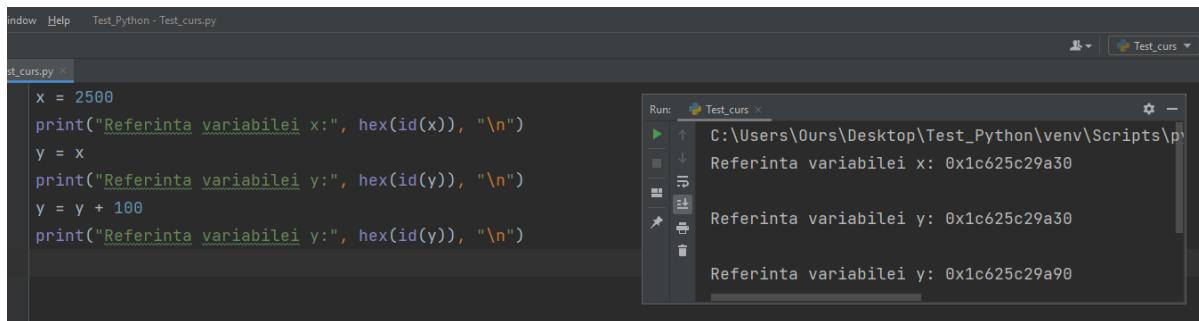
După efectuarea instrucțiunii `y = y + 100`, variabilele `x` și `y` vor conține următoarele referințe:



Mașina virtuală Python realizează automat managementul memoriei, respectiv un obiect care nu mai este utilizat (i.e., referința sa nu mai este stocată în nicio variabilă) va

fi șters de *Garbage Collector*. În exemplul anterior, dacă s-ar efectua instrucțiunea de atribuire **x = y**, atunci referința obiectului cu valoarea 2500 nu ar mai fi păstrată în nicio variabilă și acesta va fi șters, la un moment dat, de Garbage Collector.

Intern, referința unui obiect este chiar adresa sa de memorie și poate fi aflată folosind funcția **id(variabilă)**:



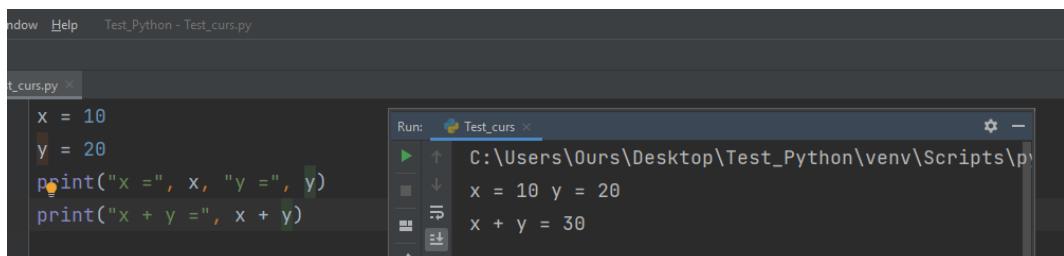
```
Test_Python - Test_curs.py
t_curs.py
x = 2500
print("Referinta variabilei x:", hex(id(x)), "\n")
y = x
print("Referinta variabilei y:", hex(id(y)), "\n")
y = y + 100
print("Referinta variabilei y:", hex(id(y)), "\n")
```

Run: Test_curs x
C:\Users\Ours\Desktop\Test_Python\venv\Scripts\p
Referinta variabilei x: 0x1c625c29a30
Referinta variabilei y: 0x1c625c29a30
Referinta variabilei y: 0x1c625c29a90

Identifierul unui obiect este unic pe parcursul ciclului său de viață, dar pot exista obiecte cu același identifier dacă ele au cicluri de viață disjuncte.

Afișarea datelor pe monitor

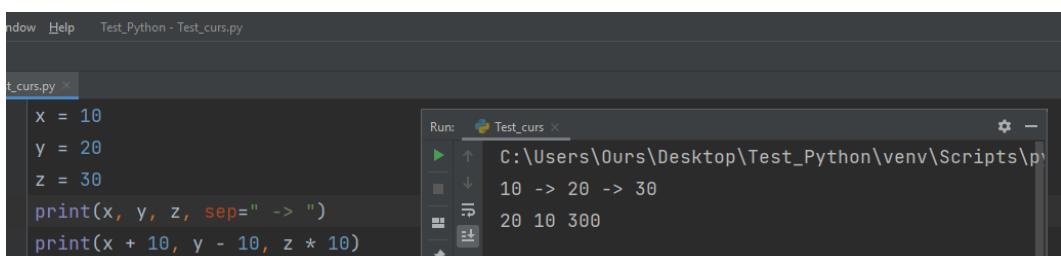
Pentru afișarea datele pe monitor se utilizează funcția **print(argumente)**. Această funcție are un număr variabil de argumente care pot fi constante, variabile sau expresii. În mod implicit, argumentele sunt afișate cu un spațiu între ele, iar la sfârșit se va afișa o linie nouă, vidă:



```
Test_Python - Test_curs.py
t_curs.py
x = 10
y = 20
print("x =", x, "y =", y)
print("x + y =", x + y)
```

Run: Test_curs x
C:\Users\Ours\Desktop\Test_Python\venv\Scripts\p
x = 10 y = 20
x + y = 30

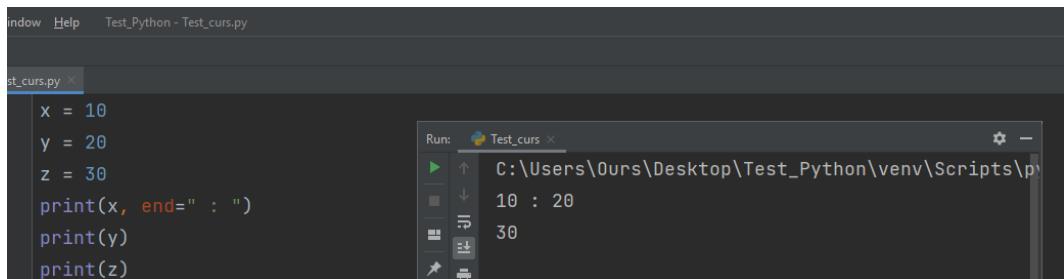
Funcția **print** are parametrul optional **sep**, de tip sir de caractere, prin intermediu căruia se poate stabili un alt separator pentru valorile afișate, însă doar în cadrul apelului respectiv:



```
Test_Python - Test_curs.py
t_curs.py
x = 10
y = 20
z = 30
print(x, y, z, sep="-> ")
print(x + 10, y - 10, z * 10)
```

Run: Test_curs x
C:\Users\Ours\Desktop\Test_Python\venv\Scripts\p
10 -> 20 -> 30
20 10 300

De asemenea, funcția `print` are parametrul optional `end`, de tip sir de caractere, prin intermediul căruia se poate modifica linia nouă afișată la sfârșitul apelului respectiv:



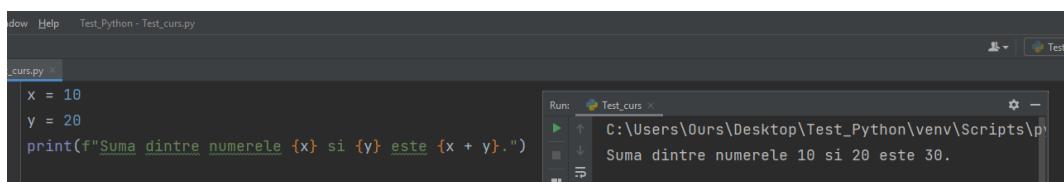
The screenshot shows a Python script named `st_curs.py` in PyCharm. The code contains three `print` statements: `x = 10`, `y = 20`, and `print(x, end=" : ")`. The output window shows the results: `10 : 20` followed by a new line, and then `30`.

```

x = 10
y = 20
z = 30
print(x, end=" : ")
print(y)
print(z)

```

O altă posibilitate de afișare a datelor o constituie utilizarea *șirurilor de caractere formatație (f-strings)*. Un astfel de șir se indică folosind litera `f` sau `F` înaintea sa, iar în interiorul său se precizează, între acolade, valorile expresiilor care trebuie afișate:



The screenshot shows a Python script named `_curs.py` in PyCharm. It contains a single `print` statement: `print(f"Suma dintre numerele {x} si {y} este {x + y}.")`. The output window shows the result: `Suma dintre numerele 10 si 20 este 30.`

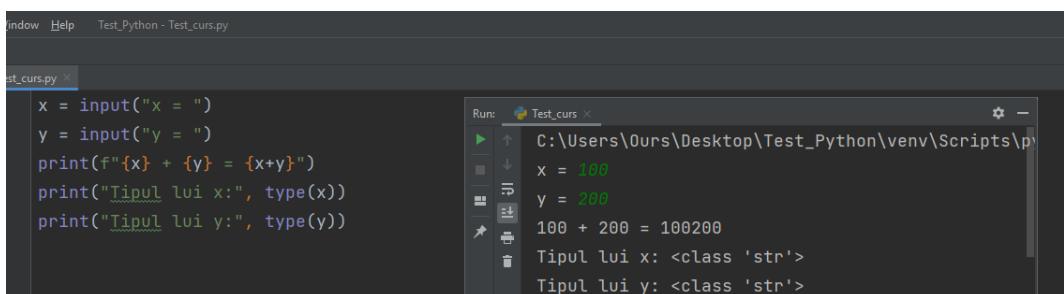
```

x = 10
y = 20
print(f"Suma dintre numerele {x} si {y} este {x + y}.")

```

Citirea datelor de la tastatură

Pentru citirea datele de la tastatură se utilizează funcția `input(mesaj)`. Parametrul `mesaj` este optional, de tip sir de caractere, iar în cazul în care este utilizat se va afișa mesajul respectiv pe monitor, înainte de citirea datelor. Funcția `input` furnizează întotdeauna valoarea citită de la tastatură sub forma unui sir de caractere:



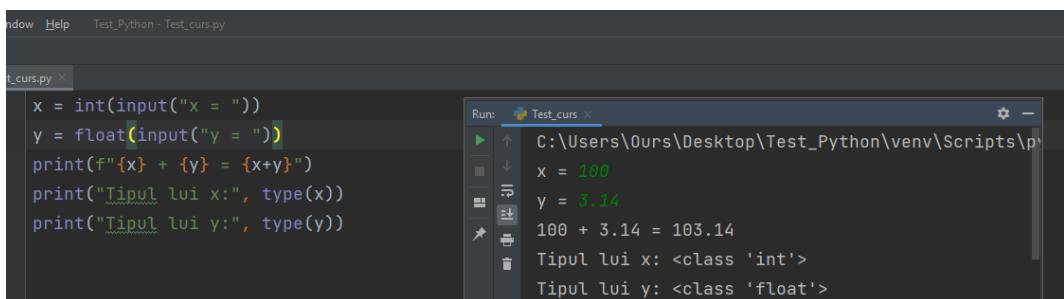
The screenshot shows a Python script named `st_curs.py` in PyCharm. It uses `input` to read values for `x` and `y`, then prints their sum and types. The output window shows the input values `100` and `200`, the calculation `100 + 200 = 100200`, and the type information for `x` and `y`.

```

x = input("x = ")
y = input("y = ")
print(f"{x} + {y} = {x+y}")
print("Tipul lui x:", type(x))
print("Tipul lui y:", type(y))

```

Pentru a transforma șirurile de caractere citite în valori de alte tipuri primitive se folosesc funcțiile de conversie `int(șir)`, `float(șir)`, `complex(șir)` sau `bool(șir)`:



The screenshot shows a Python script named `t_curs.py` in PyCharm. It reads `x` and `y` as integers and `y` as a float, then prints the sum and types. The output window shows the integer `100`, the float `3.14`, the calculation `100 + 3.14 = 103.14`, and the type information for `x` and `y`.

```

x = int(input("x = "))
y = float(input("y = "))
print(f"{x} + {y} = {x+y}")
print("Tipul lui x:", type(x))
print("Tipul lui y:", type(y))

```

Expresii. Operatori. Instrucțiuni

O expresie poate fi formată din *operanzi* (constante sau variabile), *operatori* (simboluri corespunzătoarea unor operații) și *paranteze rotunde* (pentru modificarea ordinii implicite de efectuare a operațiilor). De exemplu, expresia $(x+3)*y$ conține operatorii aritmetici + și *, operanzii sunt variabilele x, y și constanta (literalul) 3, iar parantezele sunt utilizate pentru a forța efectuarea adunării înainte înmulțirii.

Operatori

Operatorii sunt simboluri corespunzătoarea anumitor operații. Un operator poate avea mai multe semnificații, în funcție de context. De exemplu, operatorul – poate fi utilizat și pentru a schimba semnul unei variabile și pentru a efectua o scădere.

Un operator se caracterizează prin:

- *aritate*: numărul de operanzi asupra căruia poate acționa operatorul respectiv (de exemplu, în expresia -3 operatorul – are aritatea 1, iar în expresia $7-3$ acesta are aritatea 2);
- *prioritate*: stabilește ordinea de evaluare a operatorilor dintr-o expresie (de exemplu, expresia $2+3*4$ se evaluează în ordinea $2+3*4 = 2+12 = 14$, deoarece operatorul * are prioritate mai mare decât operatorul +);
- *asociativitate*: stabilește ordinea de evaluare a unor operatori cu priorități egale dintr-o expresie (de exemplu, expresia $x+y+z$ se evaluează de la stânga la dreapta, respectiv $(x+y)+z$, deoarece operatorul + are asociativitate de la stânga la dreapta).

În limbajul Python sunt definiți mai mulți operatori, pe care îi putem grupa în următoarele categorii:

1. *operatori aritmetici*: + (adunare sau semnul plus), - (scădere sau semnul minus), * (înmulțire), / (împărțire reală), // (împărțire întreagă), % (modulo), ** (exponențiere)
- Operatorul / efectuează întotdeauna o împărțire reală ("cu virgulă"), indiferent de tipul operanzilor (de exemplu, $7 / 2 = 3.5$).
- Expresia $a//b$ furnizează cel mai mare întreg mai mic sau egal decât a/b , iar expresia $a\b$ se calculează folosind formula $a\b = a-b*(a//b)$.

Exemple:

$7 // 2 = 3$

$-7 // 2 = -4$

$-7.12 // 3.213 = -3.0$

$-7.12 \% 3.213 = 2.519$

$11 // -3 = -4$

$11 \% -3 = 11 - (-3) * (11//(-3)) = 11 + 3*(-4) = -1$

- Operatorul ** are asociativitate de la dreapta la stânga.

Exemple:

```
0**0 = 1
3**4**2 = 3**(4**2) = 3**16 = 43046721
2**-3 = 0.125
-2**4 = -16
(-2)**4 = 16
31.44**0.788 = 15.136178456437747
```

2. *operatori relationali*: < (strict mai mic), <= (mai mic sau egal), > (strict mai mare), >= (mai mare sau egal), == (egal), != (diferit), **is** și **is not** (testarea identității), **in** și **not in** (testarea apartenenței)

- Operatorii **is/is not** testează dacă două variabile/expresii sunt identice sau nu, comparând referințele asociate valorilor lor. Practic, expresia `x is y` este True dacă și numai dacă `id(x) == id(y)`.

Exemple:

```
x = 3
y = 5
print(x is y)           #False
print(x+2 is y)         #True
print(y-2 is not x)     #False
```

- Operatorii **in/not in** testează apartenența unei valori la o colecție.

Exemple:

```
sir = "exemplu"
print("m" in sir)          #True
print("emp" not in sir)    #False

lista = [11, -3, 7, 5, -10, 8]
x = 7
print(x not in lista)      #False
print(x+4 in lista)        #True
```

- Deoarece numerele reale nu pot fi reprezentate exact în memorie, pot să apară erori în momentul comparării lor. De exemplu, expresia `1.1 + 2.2 == 3.3` va furniza valoarea `False`!

```
print(1.1 + 2.2 == 3.3)      #False
print(1.1 + 2.2, "==", 3.3)  #3.3000000000000003 == 3.3
```

Pentru a evita astfel de erori, se recomandă înlocuirea expresiei `x == y` (în care `x` și `y` sunt numere reale) cu o expresie de tipul `abs(x-y) <= 1e-9`, verificându-se astfel faptul că primele 9 zecimale ale numerelor reale `x` și `y` sunt identice.

- Spre deosebire de alte limbaje de programare, de exemplu C/C++, operatorii relationali pot fi înlăntuiți!

Exemple:

```
a = 1
b = 10
x = 2
if a <= x <= b:
    print("Da")
else:
    print("Nu")
```

```
x = 1
y = 2
z = 4
if x + 2 == y + 1 > z:
    print("Da")
else:
    print("Nu")
```

În exemplul din partea stângă se va afișa mesajul "Da", iar în cel din dreapta "Nu".

3. operatori logici: **not** (negare), **and** (și), **or** (sau)

- Valorile nule corespunzătoare tipurilor de date (de exemplu, valorile **0**, **0.0**, **0+0j**, **" "**, **[]** etc.) sunt considerate ca fiind echivalente cu **False**, iar orice altă valoare este considerată echivalentă cu **True**.
- În urma evaluării unor expresii care conțin operatori logici, în limbajul Python se pot obține și alte valori în afară de **True** sau **False**, astfel:

$$\text{not } x = \begin{cases} \text{False}, & \text{dacă } x \text{ este } \text{True} \\ \text{True}, & \text{dacă } x \text{ este } \text{False} \end{cases}$$

$$x \text{ and } y = \begin{cases} y, & \text{dacă } x \text{ este } \text{True} \\ x, & \text{dacă } x \text{ este } \text{False} \end{cases}$$

$$x \text{ or } y = \begin{cases} x, & \text{dacă } x \text{ este } \text{True} \\ y, & \text{dacă } x \text{ este } \text{False} \end{cases}$$

Exemple:

<code>-100 and "test"</code>	<code>=></code>	<code>'test'</code>
<code>-100 or "test"</code>	<code>=></code>	<code>-100</code>
<code>not 3.14</code>	<code>=></code>	<code>False</code>
<code>-100 or "" and 3.14</code>	<code>=></code>	<code>-100</code> (se evaluează mai întâi operatorul and)
<code>"" or 10 and 3.14</code>	<code>=></code>	<code>3.14</code>
<code>not (0 and 123.45)</code>	<code>=></code>	<code>True</code>

- Operatorul **not** este singurul operator logic care furnizează întotdeauna doar valorile **True** sau **False**!
- Expresiile care conțin operatori logici se evaluează prin scurtcircuitare, astfel:
 - într-o expresie de forma `expr_1 and expr_2 and ... and expr_n` evaluarea se oprește la prima expresie a cărei valoare este **False**, deoarece, oricum, valoarea întregii expresii va fi **False**;

- într-o expresie de forma `expr_1 or expr_2 or ... or expr_n` evaluarea se oprește la prima expresie a cărei valoare este `True`, deoarece, oricum, valoarea întregii expresii va fi `True`.
- 4. operatori pe biți:** `~` (negare pe biți / bitwise not), `&` (și pe biți / bitwise and), `|` (sau pe biți / bitwise or), `^` (sau exclusiv / xor), `<<` (deplasare la stânga pe biți / left shift), `>>` (deplasare la dreapta pe biți / right shift)
- Operatorii pe biți acționează asupra reprezentărilor binare ale valorilor de tip întreg.
 - În limbajul Python toate numerele întregi sunt considerate cu semn (nu există tipuri de date asemănătoare celor de tipul `unsigned` din limbajele C/C++) și sunt reprezentate intern în complement față de 2, astfel:

Numere pozitive	Numere negative
$x = 23 = 00..0\textcolor{red}{1}0111$	$x = -24$ $ x = 24 = 00..0\textcolor{red}{1}1000$ $\sim x = \sim24 = 11..100111$ $\sim x +1 = \sim24 + 1 = 11..101000$ $x = -24 = 11..101000$

Se observă faptul că numerele întregi pozitive se reprezintă binar direct prin scrierea lor în baza 2, în timp ce un număr întreg negativ x se reprezintă astfel:

- se reprezintă în baza 2 valoarea absolută a lui x ;
 - se calculează complementul față de 1 a valorii obținute anterior, respectiv toți biții egali cu 0 devin 1 și invers;
 - reprezentarea binară a lui x se obține adunând 1 la valoarea obținută anterior.
- Operatorul `~` (negare pe biți / bitwise not) este un operator unar care calculează numărul obținut prin negarea fiecărui bit al operandului său (complementul față de 1):

$$\begin{array}{c|c|c} \sim & 0 & 1 \\ \hline & 1 & 0 \end{array}$$

$$\sim b = 1 - b$$

$$\begin{aligned} x &= 23 = 00..0\textcolor{red}{1}0111 \\ \sim x &= -24 = 11..101000 \\ \sim x &= -(x + 1) = -x - 1 \end{aligned}$$

- Operatorii `&` (și pe biți / bitwise and), `|` (sau pe biți / bitwise or) și `^` (sau exclusiv / bitwise xor) sunt operatori binari care acționează asupra perechilor de biți aflați pe aceeași poziție în cei doi operanzi, astfel:

$$\begin{array}{c|c|c} \& 0 & 1 \\ \hline 0 & 0 & 0 \\ 1 & 0 & 1 \end{array}$$

$$b_1 \& b_2 = 1 \Leftrightarrow b_1 = b_2 = 1$$

$$\begin{aligned} x &= 349 = 00101011\textcolor{green}{1}01 \\ y &= 2006 = 11111010\textcolor{green}{1}10 \\ x \& y &= 340 = 00101010\textcolor{green}{1}00 \end{aligned}$$

	0	1
0	0	1
1	1	1
b_1	$b_2 = 0 \Leftrightarrow b_1 = b_2 = 0$	

	0	1
0	0	1
1	1	0
$b_1 \wedge b_2 = 1 \Leftrightarrow b_1 \neq b_2$		

$x = 349 = 00101011\textcolor{red}{1}01$
 $y = 2006 = 11111010\textcolor{red}{1}10$
 $x | y = 2015 = 11111011\textcolor{red}{1}11$

$x = 349 = 00101011\textcolor{red}{1}01$
 $y = 2006 = 11111010\textcolor{red}{1}10$
 $x \wedge y = 1675 = 11010001\textcolor{red}{0}11$

- Operatorul \ll (left shift) este un operator binar care deplasează spre stânga biții unui număr întreg cu un număr dat de poziții, inserând la sfârșitul reprezentării binare a numărului respectiv un număr de biți nuli egal cu numărul de biți deplasați.

Exemplu:

$x = 2006 = 11111010110$
 $x \ll 3 = 11111010110\textcolor{blue}{000} = 16048 = 2006 * (2^{**3})$

În general, expresia $x = x \ll b$ este echivalentă cu expresia $x = x * (2^{**b})$.

- Operatorul \gg (right shift) este un operator binar care deplasează spre dreapta biții unui număr întreg cu un număr dat de poziții, eliminându-i efectiv.

Exemplu:

$x = 2006 = 11111010110$
 $x \gg 3 = 11111010110\textcolor{blue}{0} = 11111010 = 250 = 2006 // (2^{**3})$

În general, expresia $x = x \gg b$ este echivalentă cu expresia $x = x // (2^{**b})$.

5. operatorul condițional: $expr_1$ if $expr_logică$ else $expr_2$

- Operatorul condițional este un operator ternar care furnizează valoarea expresiei $expr_1$ dacă $expr_logică$ este **True** sau valoarea expresiei $expr_2$ în caz contrar.

Exemple:

`max = x if x > y else y` (calculul maximului dintre două numere)
`print("Nr. par") if x % 2 == 0 else print("Nr. impar")` (testarea parității unui număr întreg)

Prioritățile și asociativitățile operatorilor

Evaluarea unei expresii se realizează ținând cont de *prioritățile* și *asociativitățile* operatorilor utilizati, așa cum am menționat anterior.

În limbajul Python, aproape toți operatorii au *asociativitate de la stânga la dreapta* (mai puțin operatorul de exponențiere care are asociativitate de la dreapta la stânga), iar prioritățile lor sunt indicate în tabelul următor, în ordine descrescătoare:

Prioritate	Operatori	Descriere
1 (maximă)	()	Parenteze (grupare)
2	f(args...)	Apel de funcție
	x[index_1:index_2]	Accesare unei secvență (slicing)
	x[index]	Accesare unei element (indexare)
	x.dată_membră	Accesare unei date membre (obiecte)
3	**	Exponențiere
4	~x	Negare pe biți (bitwise NOT)
	+x, -x	Operatorii de semn (unari)
5	*, /, //, %	Înmulțire și împărțiri
6	+, -	Adunare și scădere (binari)
7	<<, >>	Deplasări pe biți (bitwise shifts)
8	&	ȘI pe biți (bitwise AND)
9	^	SAU EXCLUSIV pe biți (bitwise XOR)
10		SAU pe biți (bitwise OR)
11	<, <=, >, >=, !=, ==, in, not in, is, is not	Operatorii relaționali
12	not	Negare logică (boolean NOT)
13	and	ȘI logic (boolean AND)
14	or	SAU logic (boolean OR)
15 (minimă)	if...else	Operatorul condițional

În general, prioritățile operatorilor sunt "naturale" (de exemplu, operațiile de exponențiere, înmulțire și împărțire au o prioritate mai mare decât cele de adunare și scădere, operatorii logici și relaționali au priorități mici deoarece trebuie ca înaintea evaluării lor să fie evaluate restul expresiilor etc.) și expresiile pot fi evaluate de către un

programator fără a cunoaște în detaliu priorităților operatorilor. Totuși, există și câteva cazuri în care evaluarea corectă a unei expresii se poate efectua de către un programator doar cunoscând exact aceste priorități:

- valoarea expresiei $2 + 3 \ll 4$ este 80, deoarece operatorul + are prioritate mai mare decât operatorul \ll , deci este echivalentă cu expresia $(2 + 3) \ll 4 = 5 \ll 4 = 5 * 2^{**4} = 5 * 16 = 80$ (de multe ori, se consideră în mod eronat faptul că operatorul \ll are prioritate mai mare decât operatorul +, deci expresia s-ar evalua prin $2 + (3 \ll 4) = 2 + 3 \ll 4 = 2 + 3 * 2^{**4} = 2 + 3 * 16 = 50$);
- expresia $x == \text{not } y$ este incorrectă sintactic, deoarece operatorul == are prioritate mai mare decât operatorul not și expresia este considerată echivalentă cu $(x == \text{not}) y$, ceea ce evident nu are niciun sens! În acest caz, suntem obligați să utilizăm paranteze, deci expresia corectă este $x == (\text{not } y)$. Atenție, există multe alte expresii de acest tip, de exemplu $\text{True} == \text{not } y, x \& \text{not } y == \text{True}$ etc.!
- o expresie de forma $a^{**}-b$ se evaluează corect prin a^{-b} , fiind considerată o excepție (operatorul de expoziție are prioritate mai mare decât operatorul unar de semn -), deci expresia ar trebui să fie considerată ca fiind echivalentă cu $(a^{**}-)b$, dar aceasta nu are niciun sens);
- secvența de cod de mai jos va afișa eronat mesajul "Bursier din grupa 131 sau 132!", deoarece operatorul and are prioritate mai mare decât operatorul or, deci expresia va fi considerată echivalentă cu $131 == 131 \text{ or } (131 == 132 \text{ and } 5 >= 9)$, deci va fi evaluată prin True or False și se va obține valoarea True!

```
grupa = 131
media = 5
if grupa == 131 or grupa == 132 and media >= 9:
    print("Bursier din grupa 131 sau 132!")
else:
    print("Nu este bursier din grupa 131 sau 132!")
```

Evident, modificarea expresiei logice în `(grupa == 131 or grupa == 132) and media >= 9` va elimina această eroare.

CURS 2

Instrucțiuni

Pentru a putea controla fluxul unui program (ordinea în care se vor executa operațiile dorite), majoritatea limbajelor de programare folosesc *instrucțiuni de control*. Aceste instrucțiuni pot fi, de exemplu, *instrucțiuni de decizie* (cu ajutorul cărora se stabilește dacă o anumită operație se efectuează sau nu în funcție de o anumită condiție), *instrucțiuni repetitive* (cu ajutorul cărora se efectuează de mai multe ori o anumită operație) etc.

În limbajul Python nu există delimitatori pentru *blocurile de instrucțiuni* (cum sunt accoladele în limbajele C/C++), ci gruparea mai multor instrucțiuni se realizează prin indentarea lor în raport de instrucțiunea căreia i se subordonează.

În limbajul Python sunt definite următoarele instrucțiuni de control:

1. *instrucțiunea de atribuire*

- Spre deosebire de limbajele C/C++, atribuirea nu este un operator, ci este o instrucțiune!
- Instrucțiunea de atribuirea poate avea următoarele forme:
 - *atribuire simplă* (`x = 100`);
 - *atribuire multiplă* (`x = y = 100`);
 - *atribuire compusă* (`x, y, z = 100, 200, 300`).
- Două variabile se pot interzimba prin atribuirea compusă `x, y = y, x!`
- O atribuire de forma `x = x operator expresie` poate fi scrisă prescurtat sub forma `x operator= expresie`, unde operator este un operator aritmetic sau pe biți binar. De exemplu, instrucțiunea `x = x + y*10` poate fi scrisă prescurtat sub forma `x += y*10`.
- În limbajul Python nu sunt definiți operatorii `++/-` din limbajele C/C++!

2. *instrucțiunea de decizie / alternativă if*

- Instrucțiunea de decizie este utilizată pentru a executa o instrucțiune (sau un bloc de instrucțiuni) doar în cazul în care o expresie logică este adevărată:

```
if expresie_logică:  
    instrucțiune
```

Exemplu (maximul dintre două numere):

```
a = int(input("a = "))  
b = int(input("b = "))  
maxim = a  
if b > maxim:  
    maxim = b  
print("Maximul dintre", a, "și", b, "este", maxim)
```

- Instrucțiunea alternativă este utilizată pentru a alege executarea unei singure instrucțiuni (sau a unui bloc de instrucțiuni) dintre două posibile, în funcție de valoarea de adevăr a unei expresii logice:

```
if expresie_logică:
    instrucțiune_1
else:
    instrucțiune_2
```

Exemplu (maximul dintre două numere):

```
a = int(input("a = "))
b = int(input("b = "))
if a > b:
    maxim = a
else:
    maxim = b
print("Maximul dintre", a, "si", b, "este", maxim)
```

- Instrucțiunile alternative imbricate se pot scrie mai concis folosind instrucțiunea **elif**, aşa cum se poate observa din exemplul următor:

<pre>a = int(input("a = ")) if a < 0: print("Strict negativ") else: if a == 0: print("Zero") else: print("Strict pozitiv")</pre>	<pre>a = int(input("a = ")) if a < 0: print("Strict negativ") elif a == 0: print("Zero") else: print("Strict pozitiv")</pre>
---	---

- În limbajul Python nu este definită o instrucțiune alternativă multiplă, cum este, de exemplu, instrucțiunea **switch** din limbajele C/C++!

3. *instrucțiunea repetitivă while*

- Instrucțiunea **while** este o instrucțiune repetitivă cu test inițial, fiind utilizată pentru a executa o instrucțiune (sau un bloc de instrucțiuni) cât timp o expresie logică este adevărată:

```
while expresie_logică:
    instrucțiune
```

Exemplu (suma cifrelor unui număr natural):

```
n = int(input("n = "))
aux = n
sc = 0
while aux != 0:
    sc = sc + aux % 10
    aux = aux // 10
print("Suma cifrelor numarului", n, "este", sc)
```

- În limbajul Python nu este definită o instrucțiune repetitivă cu test final, cum este, de exemplu, instrucțiunea **do...while** din limbajele C/C++!

4. instrucțiunea repetitivă for

- Instrucțiunea **for** este utilizată pentru a accesa, pe rând, fiecare element dintr-o secvență (de exemplu, un sir de caractere, o listă etc.), elementele fiind considerate în ordinea în care apar în secvență:

```
for variabilă in secvență:
    instrucțiune
```

Exemple:

<pre>sir = "test" for c in sir: print(c, end=" ")</pre> <p>#Se va afișa: t e s t</p>	<pre>lista = [1, 2, 3] for x in lista: print(x, end=" ")</pre> <p>#Se va afișa: 1 2 3</p>
--	---

- Pentru a genera secvențe numerice de numere întregi asemănătoare unor progresii aritmetice se poate utiliza funcția **range([min], max, [pas])**, care va genera, pe rând, numerele întregi cuprinse între valorile **min** (inclusiv) și **max** (exclusiv!!!) cu rația **pas**. Parametrii scriși între paranteze drepte sunt optionali, iar parametrul optional **pas** se poate specifica doar dacă se specifică și parametrul optional **min**. Dacă pentru parametrul **min** nu se specifică nicio valoare, atunci el va fi considerat în mod implicit ca fiind **0**.

Exemple:

<pre>range(6) => 0, 1, 2, 3, 4, 5 range(2, 6) => 2, 3, 4, 5 range(2, 11, 3) => 2, 5, 8 range(2, 12, 3) => 2, 5, 8, 11 range(7, 2) => secvență vidă (deoarece 7 > 2) range(7, 2, -1) => 7, 6, 5, 4, 3</pre>
--

5. instrucțiunea continue

- Instrucțiunea **continue** este utilizată în cadrul unei instrucțiuni repetitive pentru a termina forțat iterația curentă (dar nu și instrucțiunea repetitivă!), continuându-se direct cu următoarea iterație.

Exemplu:

<pre>for i in range(1, 11): if i%2 == 0: continue print(i, end=" ")</pre> <p>#Se va afișa: 1 3 5 7 9</p>
--

6. instrucțiunea break

- Instrucțiunea **break** este utilizată în cadrul unei instrucțiuni repetitive pentru a termina forțat executarea instrucțiunii respective.

Exemplu: Se citește un sir de numere care se termină cu valoarea 0 (care se consideră că nu face parte din sir, ci este doar un marcat al sfârșitului său). Să se afișeze suma numerelor citite.

```
s = 0
while True:
    x = int(input("x = "))
    if x == 0:
        break
    s += x
print("Suma numerelor citite: ", s)
```

Atenție, în acest program instrucțiunea `s += x` ar fi putut fi scrisă și înaintea instrucțiunii `if` fără a-i afecta corectitudinea! Totuși, în alte cazuri (de exemplu, dacă s-ar fi cerut produsul numerelor citite), acest lucru ar fi dus la afișarea unui rezultat eronat!

7. instrucțiunea else

- Instrucțiunea `else` poate fi adăugată la sfârșitul unei instrucțiuni repetitive, instrucțiunile subordonate ei fiind executate doar în cazul în care instrucțiunea repetitivă se termină natural (condiția dintr-o instrucțiune `while` devine falsă sau o instrucțiune `for` a parcurs toate elementele unei secvențe), ci nu din cauza intreruperii sale forțate (utilizând o instrucțiune `break`).

Exemplu:

- a) Se citește un sir format din n numere întregi. Să se verifice dacă toate numerele citite au fost pozitive sau nu.

```
n = int(input("n = "))
for i in range(n):
    x = int(input("x = "))
    if x < 0:
        print("A fost citit un număr negativ!")
        break
else:
    print("Toate numerele citite au fost pozitive!")
```

- b) Să se afișeze cel mai mic număr prim cuprins între două numere naturale a și b sau un mesaj corespunzător în cazul în care nu există niciun număr prim cuprins între a și b .

```
a = int(input("a = "))
b = int(input("b = "))

for x in range(a, b+1):
    for d in range(2, x//2+1):
        if x % d == 0:
            break
```

```

else:
    #instructiunea for d in ... s-a terminat natural,
    #deci numărul x nu are divizori proprii,
    #ceea ce însemană că x este un număr prim
    print("Cel mai mic numar prim cuprins intre", a,
          "si", b, "este", x)

    #numărul x este cel mai mic număr prim cuprins între
    #a și b, deci nu are rost să mai continuăm căutarea
    #altuia și oprim forțat instructiunea for x in ...
    break

else:
    #instructiunea for x in ... s-a terminat natural, deci
    # nu a fost găsit niciun număr prim cuprins între a și b
    print("Nu exista niciun numar prim cuprins intre", a,
          "si", b)

```

8. *instructiunea pass*

- Instrucțiunea `pass` este o instrucțiune care nu are niciun efect în program (este similară unei instrucțiuni vide). Această instrucțiune se utilizează în cazurile în care sintactic ar fi necesată o instrucțiune vidă, deoarece în limbajul Python aceasta nu este definită.

Exemplu:

```

varsta = 10
if varsta <= 18:
    print("Junior")
elif varsta < 60:
    #nu prelucrăm informațiile despre persoanele
    #cu vârste cuprinse între 19 și 59 de ani
    pass
else:
    print("Senior")

```

CURS 3

Şiruri de caractere

Un *şir de caractere* este o secvență de caractere indexată de la 0, memorată folosind un obiect de tipul clasei `str`.

Limbajul Python folosește setul de caractere Unicode (<https://home.unicode.org/>), ceea ce permite utilizarea într-un program a caracterelor din aproape orice alfabet existent pe Glob (*internationalizare*) și a simbolurilor specifice multor domenii (pentru mai multe detalii consultați pagina <https://docs.python.org/3/howto/unicode.html>).

Constantele de tip șir de caractere (*literali*) se reprezintă în două moduri:

- prin '`şir`' sau "`şir`", dacă dorim ca șirul respectiv să fie considerat ca fiind scris pe o singură linie (sunt ignorate caracterele `newline`);
- prin '`'''şir'''`' sau "`"""şir"""`", dacă dorim ca șirul respectiv să fie considerat ca fiind scris pe mai multe linii (nu sunt ignorate caracterele `newline`).

Exemplu:

```

Factor Run Tools VCS Window Help Test_Python [C:\Users\Ours\OneDrive - unibuc.ro\Desktop\Test_Python] - ...\\Test_curs.py - PyCharm
Test_curs.py
1     sir_1 = "Acesta este, de fapt, "
2         "un şir scris" \
3         "" \
4         "pe o singura linie!"
5
6     print(sir_1)
7
8     sir_2 = """Acesta este
9         un şir scris
10
11        pe mai multe linii linie!"""
12
13     print(sir_2)
14
Run Test_curs
"C:\Program Files (x86)\Python38-32\python.exe" "C:/Users/Ours/OneDrive - unibuc.ro/Desktop/Test_Python/Test_curs.py"
Acesta este, de fapt, un şir scris pe o singura linie!
Acesta este
    un şir scris
    pe mai multe linii linie!
Process finished with exit code 0

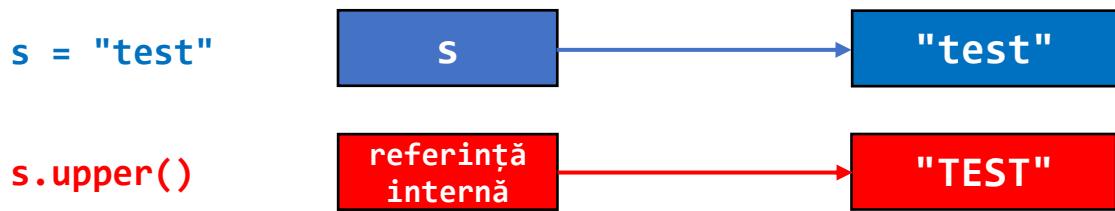
```

Într-un șir de caractere se pot insera *secvențe escape*, la fel ca în limbajele C/C++: `\n` (linie nouă), `\t` (tab), `\\` (backslash) etc. (pentru mai multe detalii consultați pagina <http://www.python-ds.com/python-3-escape-sequences>).

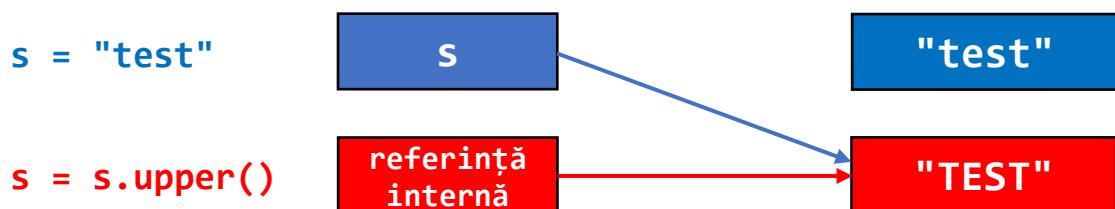
O proprietate foarte importantă a șirurilor de caractere o constituie faptul că sunt *immutable*, respectiv valoarea unui obiect de tip șir de caractere nu mai poate fi modificată după crearea sa, ci se poate modifica doar o referință spre el. Din acest motiv, nicio metoda din clasa `str` nu va modifica efectiv șirul curent (cel care apelează metoda respectivă), ci va crea un nou șir! De exemplu, să considerăm următoarea secvență de cod, care va afișa șirul `test`:

```
s = "test"
s.upper()
print(s)    #test
```

Practic, metoda `upper()` nu va transforma literele mici ale șirului `s` în litere mari, ci va genera un nou șir, obținut prin transformarea literelor mici ale șirului `s` în litere mari:

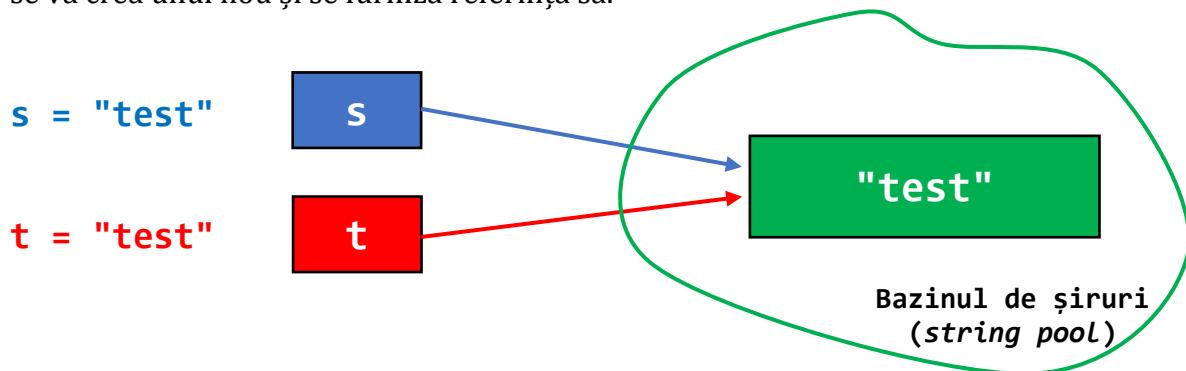


Pentru a modifica efectiv șirul `s`, trebuie să modificăm referința conținută de el în referință internă asociată șirului generat prin apelul `s.upper()`:



Obiectul conținând șirul "test" va fi șters automat din memorie de *Garbage Collector* după ce nu va mai exista nicio referință activă spre el.

Pentru stocarea șirurilor de caractere în memoria internă, limbajul Python utilizează mecanismul *string interning*, prin care se stochează în memorie o singură copie a fiecărui șir de caractere distinct utilizat în cadrul unui program. Astfel, obiectele de tip șir de caractere sunt păstrate într-un bazin de șiruri (*string pool*), iar în momentul în care o variabilă este inițializată cu un șir constant se verifică dacă acesta există deja în bazin sau nu. În caz afirmativ, nu se mai creează un nou obiect de tip șir, ci variabila de tip șir va primi direct referința șirului existent din bazin, iar în cazul în care șirul nu există în bazin se va crea unul nou și se furniza referința sa.



Funcționarea corectă a mecanismul *string interning* se bazează pe imutabilitatea șirurilor de caractere, respectiv pe faptul că nu putem să modificăm conținutul unui șir după crearea sa. În caz contrar, dacă șirurile de caractere ar fi fost *mutable*, modificare unui obiect de tip șir din bazinul de șiruri ar fi condus la modificarea implicită a tuturor șirurilor care conțin referința sa!

Utilizarea mecanismului *string interning* prezintă mai multe avantaje, cele mai importante fiind micșorarea spațiului de memorie utilizat pentru stocarea sirurilor și creșterea vitezei de comparare a lor. Fără utilizarea bazinului de siruri, compararea a două siruri de caractere *s* și *t* prin *s == t* necesită compararea caracterelor aflate pe aceleași poziții, deci complexitatea computațională va fi $\mathcal{O}(\text{len}(s))$. În cazul utilizării bazinului de siruri, compararea *s == t* a două siruri de caractere poate fi înlocuită cu compararea *s is t* a referințelor obiectelor asociate, ceea ce revine la compararea a două numere întregi (i.e., *id(s) == id(t)*), deci complexitatea computațională va fi doar $\mathcal{O}(1)!!!$

În *mod implicit*, un sir este salvat în bazinul de siruri dacă face parte din sintaxa limbajului Python (i.e., sirul este un cuvânt cheie, denumirea unei funcții/clase/variabile etc.) sau are lungimea cel mult 1 (i.e., este sirul vid sau este format dintr-un singur caracter). Orice alt sir de caractere este salvat în bazinul de siruri dacă îndeplinește următoarele condiții:

- lungimea sa este cel mult 4096;

```
s = "a" * 4096
t = "a" * 4096
print(s == t)    #True
print(s is t)    #True

s = "a" * 4097
t = "a" * 4097
print(s == t)    #True
print(s is t)    #False
```

- valoarea sa poate fi determinată în momentul compilării;

```
s = "test"
t = "te" + "st"
print(s == t)        #True
print(s is t)        #True

s = "test"
x = "st"
t = "te" + x
print(s == t)        #True
print(s is t)        #False
```

- toate caracterele sunt caractere ASCII (condiția nu este obligatorie în toate implementările Python).

În *mod explicit*, un sir poate fi salvat în bazinul de siruri dacă se utilizează metoda *intern(sir)* din modulul *sys*:

```
import sys

s = "test"
x = "st"
t = "te" + x
print(s is t)          #False
```

```
s = "test"
x = "st"
t = sys.intern("te" + x)
print(s is t) #True
```

Atenție, metoda `sys.intern(șir)` este lentă, deci trebuie utilizată doar dacă acest lucru este absolut necesar (de exemplu, dacă în programul respectiv se vor efectua multe comparări de șiruri care nu sunt implicit supuse procesului de *string interning*)!

Mai multe detalii referitoare la mecanismul *string interning* din limbajul Python găsiți în pagina <https://stackabuse.com/guide-to-string-interning-in-python/>.

Accesarea elementelor unui șir de caractere

Elementele unui șir de caractere pot fi accesate în mai multe moduri, astfel:

- a) prin indici pozitivi sau negativi

În limbajul Python, oricarei secvențe (*multime iterabilă*) de lungime n îi sunt asociati atât indici pozitivi, cuprinși între 0 și $n - 1$ de la stânga spre dreapta, cât și indici negativi, cuprinși între $-n$ și -1 de la stânga la dreapta.

Exemplu: pentru șirul `s = "programare"` avem asociati următorii indici:

	0	1	2	3	4	5	6	7	8	9
s	'p'	'r'	'o'	'g'	'r'	'a'	'm'	'a'	'r'	'e'
	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Astfel, al patrulea caracter din șir (litera 'g'), poate fi accesat atât prin `s[3]`, cât și prin `s[-7]`. Atenție, accesarea unui caracter va fi de tip *read-only* (doar în citire), deoarece șirurile sunt imutabile! De exemplu, instrucțiunea `s[6] = 'd'` o să genereze, în momentul executării programului, eroarea "TypeError: 'str' object does not support item assignment". Totuși, caracterul aflat într-un șir `s` pe o poziție `p` validă (cuprinsă între 0 și `len(s)-1`) poate fi modificat sau șters construind un nou șir a cărui referință va înlocui referința inițială:

`s = s[:p] + "caracter_nou" + s[p+1:]` (modificare)
`s = s[:p] + s[p+1:]` (ștergere)

- b) prin secvențe de indici pozitivi sau negativi (slice)

Expresia `șir[st:dr]` extrage din șirul dat subșirul cuprins între pozițiile `st` și `dr-1`, dacă `st ≤ dr`, sau șirul vid în caz contrar.

Exemplu:

`s[1:4] = s[-9:-6] = "rog"`
`s[:4] = s[0:4] = "prog"`
`s[4:] = "ramare"`
`s[5:2] = ""` (deoarece $5 > 2$)

```
s[5:2:-1] = "arg"
s[:] = "programare" (întregul sir)
s[::-1] = "eramargorp" (sirul oglindit)
s[-9:4] = "rog"
```

Operatori pentru siruri de caractere

În limbajul Python sunt definiți următorii operatori pentru manipularea sirurilor de caractere:

- a) *operatorul de concatenare*: +

Exemplu: "un" + " " + "exemplu" = "un exemplu"
- b) *operatorul de multiplicare* (concatenare repetată): *

Exemplu: "test" * 3 = 3 * "test" = "testtesttest"
- c) *operatorii pentru testarea apartenenței*: in, not in

Exemplu: expresia "est" in "atestat" va avea valoarea True
- d) *operatorii relaționali* (comparări lexicografice): <, <=, >, >=, ==, !=

Exemplu: expresia "Popa" <= "Popescu" va avea valoarea True, iar expresia "POPA" == "Popa" va avea valoarea True

Funcții predefinite pentru siruri de caractere

În limbajul Python sunt predefinite mai multe funcții (*built-in functions* – <https://docs.python.org/3/library/functions.html>), dintre care unele pot fi utilizate pentru mai multe tipuri de date. De exemplu, funcția `len(secvență)` va furniza numărul de elemente dintr-o secvență (iterabil), indiferent dacă aceasta este o listă sau un sir de caractere. Funcțiile predefinite care se pot utiliza pentru siruri de caractere sunt următoarele:

- a) `len(sir)`: furnizează numărul de caractere unui sir (lungimea sirului)

Exemplu: `len("test")` = 4
- b) `str(expresie)`: transformă valoarea expresiei într-un sir de caractere

Exemplu: `str(123)` = "123", `str(1+2==3)` = "True"
- c) `min(sir)` / `max(sir)`: furnizează caracterul minim/maxim în sens lexicografic (alfabetic) din sirul respectiv (atenție, literele mari sunt mai mici, din punct de vedere lexicografic, decât literele mici!)

Exemplu: `min("Examene")` = "E", `max("examene")` = "x"
- d) `ord(caracter)`: furnizează codul Unicode asociat caracterului respectiv

Exemplu: `ord("A")` = 65
- e) `chr(număr)`: furnizează caracterul având codul Unicode respectiv

Exemplu: `chr(65)` = "A"

Metode pentru prelucrarea sirurilor de caractere

Metodele pentru prelucrarea sirurilor de caractere sunt, de fapt, metodele încapsulate în clasa `str`. Aşa cum am precizat anterior, sirurile de caractere sunt *immutable*, deci metodele respective nu vor modifica sirul curent, ci vor genera un nou sir care va conţine rezultatul prelucrării sirului iniţial. Din acest motiv, metodele clasei `str` se vor apela, de obicei, în cadrul unei expresii de forma următoare: `sir = sir.metodă(parametri)`. De exemplu, pentru a transforma toate literele mici ale unui sir în litere mari, vom utiliza expresia `sir = sir.upper()`.

În continuare, vom prezenta mai multe metode pentru prelucrarea sirurilor de caractere, cu observaţia că parametrii scrişi între paranteze drepte sunt optionali:

a) metode pentru formatare

- `strip([caractere])`: furnizează sirul obţinut din sirul curent prin eliminarea celui mai lung prefix şi celui mai lung sufix formate doar din caracterele indicate prin parametrul optional de tip sir.

Exemplu:

```
s = "www.example.org"
s = s.strip("...misgrown?!")
print(s)                                #example
```

Dacă metoda este apelată fără parametru, atunci metoda va furniza sirul obţinut din sirul curent prin eliminarea celui mai lung prefix şi celui mai lung sufix formate doar din spaţii albe:

```
s = "      www.python.org    "
s = s.strip()
print(s)                                #www.python.org
```

- `center(lăţime, [caracter])`: furnizează sirul obţinut prin centrarea sirului iniţial folosind caracterul indicat.

Exemplu:

```
s = "www.python.org"
s = s.center(30, ".")
print(s)                                #.....www.python.org.....
```

Dacă metoda este apelată fără parametru, atunci metoda va furniza sirul obţinut prin centrarea sirului iniţial folosind spaţii:

```
s = "www.python.org"
s = s.center(30)
print(s)                                #      www.python.org
```

- `format(număr variabil de parametri)`: furnizează sirul obţinut prin înlocuirea câmpurilor variabile din sirul curent (marcate prin {}) cu parametrii metodei.

Parametrii metodei `format(...)` pot fi accesati de câmpurile variabile din sirul curent prin mai multe modalități:

- *accesare secvențială* (primul câmp variabil din sir va fi înlocuit cu primul parametru al metodei, al doilea câmp va fi înlocuit cu al doilea parametru etc.):

```
s = "Ana are {} mere {}!"  
s = s.format(5, "roșii")  
print(s) #Ana are 5 mere roșii!
```

- *accesare pozitională* (fiecare câmp variabil din sir va contine numărul de ordine al parametrului metodei cu care va fi înlocuit):

```
s = "Ana are {1} mere {0}!"  
s = s.format("roșii", 3+4)  
print(s) #Ana are 7 mere roșii!
```

- *accesare prin parametrii cu nume* (fiecare câmp variabil din sir va contine numele parametrului metodei cu care va fi înlocuit):

```
s = "Ana are {numar} mere {culoare}!"  
s = s.format(culoare="roșii", numar=3)  
print(s) #Ana are 3 mere roșii!
```

Metoda `format(...)` permite realizarea unor formatari complexe ale sirurilor de caractere, descrise în mod detaliat în pagina <https://docs.python.org/3.8/library/string.html#formatstrings>.

b) metode pentru transformări la nivel de caracter

- `lower()`: furnizează sirul obținut din sirul inițial prin transformarea tuturor literelor mari în litere mici;
- `upper()`: furnizează sirul obținut din sirul inițial prin transformarea tuturor literelor mici în litere mari;
- `swapcase()`: furnizează sirul obținut din sirul inițial prin transformarea tuturor literelor mici în litere mari și invers;
- `title()`: furnizează sirul obținut din sirul inițial prin transformarea primei litere a fiecarui cuvânt în literă mare, restul literelor fiind transformate în litere mici;
- `capitalize()`: furnizează sirul obținut din sirul inițial prin transformarea primei sale litere în literă mare, restul literelor fiind transformate în litere mici.

Exemple:

```
s = "amINTiri DIN CopilăRIE..."  
print(s) #amINTiri DIN CopilăRIE...  
print(s.lower()) #amintiri din copilărie...  
print(s.upper()) #AMINTIRI DIN COPILĂRIE...  
print(s.swapcase()) #AMinTIRI din cOPILĂrie...  
print(s.title()) #Amintiri Din Copilărie...  
print(s.capitalize()) #Amintiri din copilărie...
```

c) metode pentru clasificare

Metodele din această categorie verifică dacă toate caracterele sirului curent sunt de un anumit tip și returnează `True` sau `False`:

- `isascii()`: verifică dacă toate caracterele sirului sunt caractere ASCII (au codul cuprins între 0 și 127);
- `isalpha()`: verifică dacă toate caracterele sirului sunt litere;
- `isdigit()`: verifică dacă toate caracterele sirului sunt cifre;
- `isnumeric()`: verifică dacă toate caracterele sirului sunt caractere numerice;
- `isalnum()`: verifică dacă toate caracterele sirului sunt alfanumerice (i.e., sunt litere sau cifre);
- `isspace()`: verifică dacă toate literele din sir sunt spații albe (blank, tab etc.);
- `islower()`: verifică dacă toate literele din sir sunt litere mici, iar sirul poate să mai conțină și alte caractere care nu sunt litere (de exemplu, cifre sau semne de punctuație);
- `isupper()`: verifică dacă toate literele din sir sunt litere mari, iar sirul poate să mai conțină și alte caractere care nu sunt litere;
- `istitle()`: verifică dacă prima literă a fiecărui cuvânt este literă mare, iar restul literelor sunt mici.

Exemplu:

```
s = "Ana are 123 de mere!!!"
print(s.isascii())                      #True
print(s.isalpha())                       #False

s = "Anul2020"                          #True
print(s.isalnum())

s = "Ana Are 123 De Mere!!!"
print(s.istitle())                      #True

s = "123\u00BD"                           #s = 123½
print(s.isdigit())                       #False
print(s.isnumeric())                     #True

s = "ANA ARE mere și pere!"
print(s[3:10].islower())                  #False
print(s[0:7].isupper())                  #True
print(s[7:9].isspace())                  #True
```

d) *metode pentru căutare*

- `count(subsir, [start], [stop])`: furnizează numărul aparițiilor disjuncte ale subșirului indicat în sirul curent. Parametrii optionali `start` și `stop` pot fi utilizati pentru a specifica poziții din sir între care să fie căutat subșirul respectiv (se va considera inclusiv poziția `start` și exclusiv poziția `stop`).

Exemplu:

```
s = "Hahahahahahaha..."
print(s.count("haha"))                   #3

s = "Ana are mere în camerele casei sale."
```

```
print(s.count("mere"))                      #2
print(s.count("mere", 15))                  #1
```

- **find(subsir, [start], [stop]):** furnizează cel mai mic indice la care începe subșirul dat în sirul curent sau -1 dacă subșirul dat nu apare în sirul curent. Parametrii opționali start și stop pot fi utilizati pentru a specifica poziții din sir între care să fie căutat subșirul respectiv (se va considera inclusiv poziția start și exclusiv poziția stop).

Exemplu:

```
s = "Ana are mere în camerele casei sale."
print(s.find("mere"))                      #8
print(s.find("pere"))                     #-1
```

- **rfind(subsir, [start], [stop]):** furnizează cel mai mare indice la care începe subșirul dat în sirul curent sau -1 dacă subșirul dat nu apare în sirul curent. Parametrii opționali start și stop pot fi utilizati pentru a specifica poziții din sir între care să fie căutat subșirul respectiv (se va considera inclusiv poziția start și exclusiv poziția stop).

Exemplu:

```
s = "Ana are mere în camerele casei sale."
print(s.rfind("mere"))                      #18
print(s.rfind("pere"))                     #-1
```

- **startswith(prefix, [start], [stop]):** verifică dacă sirul curent are prefixul indicat sau nu.

Exemple:

```
s = "programare"
print(s.startswith("pro"))                  #True
print(s.startswith("gram", 2))              #False
print(s.startswith("gram", 3))              #True
print(s.startswith("gram", 3, 6))           #False
```

- **endswith(sufix, [start], [stop]):** verifică dacă sirul curent are sufixul indicat sau nu.

Exemple:

```
s = "programare"
print(s.endswith("are"))                   #True
print(s.endswith("mare", 7))               #False
print(s.endswith("mare", 4))               #True
print(s.endswith("mare", 4, 8))            #False
```

- **replace(subsir, subsir_nou, [max]):** furnizează sirul obținut din sirul curent prin înlocuirea tuturor aparițiilor subșirului indicat cu noul subșir. Parametrul optional max poate fi utilizat pentru a înlocui doar primele max apariții ale subșirului dat cu noul subșir.

Exemple:

```
s = "Ana are ore de floretă cu Dorel!"
s = s.replace("ore", "in")
print(s)                      #Ana are in de flintă cu Dinel!

s = "Ana are ore de floretă cu Dorel!"
s = s.replace("ore", "in", 2)
print(s)                      #Ana are in de flintă cu Dorel!

s = "Ana are ore de floretă cu Dorel!"
s = s.replace("are", "va avea")
print(s)                      #Ana va avea ore de floretă cu Dorel!

s = "Ana are ore de floretă cu Dorel!"
s = s.replace("ore", "")
print(s)                      #Ana are de fltă cu Dl!
```

e) metode pentru împărțirea/construirea unui sir în/din subșiruri:

- `split(separator)`: furnizează o listă care conține subșirurile obținute din sirul curent considerând separatorul indicat (implicit, spațiile albe sunt considerate separatori).

Exemple:

```
s = "Ana are      ore      de floretă cu Dorel!"
w = s.split()
print(w)    #[ 'Ana', 'are', 'ore', 'de', 'floretă', 'cu', 'Dorel!' ]

s = "Ana are ore de floretă cu Dorel!"
w = s.split("e")
print(w)    #[ 'Ana ar', ' or', ' d', ' flor', 'tă cu Dor', 'l!' ]

w = s.split("ore")
print(w)    #[ 'Ana are ', ' de fl', 'tă cu D', 'l!' ]
```

- `join(listă_subșiruri)`: furnizează sirul obținut prin concatenarea subșirurilor date, folosind ca separator sirul curent.

Exemple:

```
s = " ".join(["Ana", "are", "mere", "!"])
print(s)    #Ana are mere !

s = "Ana are ore de floretă cu Dorel!"
w = "...".join(s.split())
print(w)    #Ana...are...ore...de...floretă...cu...Dorel!
```

În afara metodelor prezentate mai sus, clasa `str` conține și alte metode utile pentru prelucrarea sirurilor de caractere. O prezentare detaliată a lor găsiți în pagina <https://docs.python.org/3/library/stdtypes.html?#string-methods>.

CURS 4

Colecții de date

Liste

O *listă* este o secvență mutabilă de valori indexate de la 0. Valorile memorate într-o listă pot fi neomogene (i.e., pot fi de tipuri diferite de date) și, datorită mutabilității, pot fi modificate. Listele au un caracter dinamic, respectiv își modifică automat lungimea în momentul inserării sau ștergerii unui element. Listele sunt instanțe ale clasei `list`.

O listă poate fi creată/initializată în mai multe moduri:

- folosind o listă de constante:

```
# listă vidă
L = []
print(L)

# listă de constante omogene
L = [1, 2, 5, 7, 10]
print(L)

# listă de constante neomogene
L = [1, "Popescu Ion", 151, [9, 9, 10]]
print(L)
```

- folosind secvențe de initializare (*list comprehensions*):

```
# secvență de initializare
L = [x + 1 for x in range(10)]
print(L)                                # [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# secvență de initializare cu placeholders (_)
L = [_ + 1 for _ in range(10)]
print(L)                                # [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# secvențe de initializare condiționale
L = [x**2 for x in range(10) if x % 2 == 0]
print(L)                                # [0, 4, 16, 36, 64]

L = [x**2 if x % 2 == 0 else -x**2 for x in range(10)]
print(L)                                # [0, -1, 4, -9, 16, -25, 36, -49, 64, -81]

L1 = [1, 3, 5, 6, 8, 3, 13, 21]
L2 = [18, 3, 7, 5, 16]
L3 = [x for x in L1 if x in L2]
print(L3)                                # [3, 5, 3]
```

```
# citirea de la tastatură a elementelor unei liste de numere întregi
L = [int(x) for x in input("Valori: ").split()]
print(L)

# calculul sumei cifrelor unui număr natural
print("Suma cifrelor:", sum([int(c) for c in input("x = ")]))
```

Accesarea elementelor unei liste

Elementele unei liste pot fi accesate în mai multe moduri, asemănătoare celor prezentate pentru şirurile de caractere:

- a) *prin indici pozitivi sau negativi*

În limbajul Python, oricărei secvențe (*mulțime iterabilă*) de lungime n îi sunt asociati atât indici pozitivi, cuprinși între 0 și $n - 1$ de la stânga spre dreapta, cât și indici negativi, cuprinși între $-n$ și -1 de la stânga la dreapta.

Exemplu: pentru lista $L = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]$ avem asociati următorii indici:

L	0	1	2	3	4	5	6	7	8	9
	10	20	30	40	50	60	70	80	90	100
	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Astfel, al patrulea element din listă (numărul 40), poate fi accesat atât prin $L[3]$, cât și prin $L[-7]$. Atenție, listele sunt mutabile, deci, spre deosebire de șirurile de caractere, un element poate fi modificat direct (e.g., $L[3] = 400$)!

- b) *prin secvențe de indici pozitivi sau negativi (slice)*

Expresia $lista[st:dr]$ extrage din lista dată sublista cuprinsă între pozițiile st și $dr-1$, dacă $st \leq dr$, sau lista vidă în caz contrar.

Exemple:

```
L[1: 4] == [20, 30, 40] == L[-9 : -6]
L[2] = -10 => L == [10, 20, -10, 40, 50, 60, 70, 80, 90, 100]
L[: 4] == L[0: 4] == [10, 20, 30, 40]
L[4: ] == [50, 60, 70, 80, 90, 100]
L[:] == L
L[5: 2] == [] #pentru că 5 > 2
L[5: 2: -1] == [60, 50, 40]
L[: : -1] == [100, 90, 80, 70, 60, 50, 40, 30, 20, 10] #lista inversată
L[-9: 4] == [20, 30, 40]
L[1: 6] = [-2, -3, -4] => L == [10, -2, -3, -4, 70, 80, 90, 100]
L[1: 1] = [2, 3] => L == [10, 2, 3, 20, 30, 40, 50, 60, 70, 80, 90, 100]
L[1: 3] = [] => L == [10, 40, 50, 60, 70, 80, 90, 100] #ștergere
```

- c) *ștergerea unui element sau a unei secvențe dintr-o listă*

Ștergerea unui element sau a unei secvențe se realizează fie folosind cuvântul cheie `del`, fie atribuind elementului sau secvenței o listă vidă.

Exemple:

```
del L[2] => L == [10, 20, 40, 50, 60, 70, 80, 90, 100]
L[2: 3] = [] => L == [10, 20, 40, 50, 60, 70, 80, 90, 100]
del L[1: 5] => L == [10, 60, 70, 80, 90, 100]
L[1: 5] = [] => L == [10, 60, 70, 80, 90, 100]
```

Operatori pentru liste

În limbajul Python sunt definiți următorii operatori pentru manipularea listelor:

- a) *operatorul de concatenare: +*

Exemplu: `[1, 2, 3] + [4, 5] == [1, 2, 3, 4, 5]`

- b) *operatorul de concatenare și atribuire: +=*

Exemplu:

```
L = [1, 2, 3]
L += [4, 5]
print(L)          # [1, 2, 3, 4, 5]
```

- c) *operatorul de multiplicare (concatenare repetată): **

Exemplu: `[1, 2, 3] * 3 == [1, 2, 3, 1, 2, 3, 1, 2, 3]`

- d) *operatorii pentru testarea apartenenței: in, not in*

Exemplu: expresia `3 in [2, 1, 4, 3, 5]` va avea valoarea True

- e) *operatorii relationali: <, <=, >, >=, ==, !=*

Observație: În cazul primilor 4 operatori, cele două liste vor fi comparate lexicografic, deci elementele efectiv analizate trebuie să fie comparabile, altfel se va genera o eroare!

Exemple:

```
L1 = [1, 2, 3, 100]
L2 = [1, 2, 4]
print(L1 <= L2)          # True

L2 = [1, 2, 4, "Pop Ion"]
print(L1 >= L2)          # False

L2 = [1, 2, "Pop Ion"]
print(L1 == L2)           # False
print(L1 <= L2)           # Eroare, deoarece nu se pot compara
                           # lexicografic numărul 3 și sirul "Pop Ion"
```

Funcții predefinite pentru liste

În limbajul Python sunt predefinite mai multe funcții (*built-in functions* – <https://docs.python.org/3/library/functions.html>), dintre care unele pot fi utilizate pentru mai multe tipuri de date. De exemplu, funcția `len(secvență)` va furniza numărul de elemente dintr-o secvență (iterabil), indiferent dacă aceasta este o listă sau un sir de caractere. Funcțiile predefinite care se pot utiliza pentru liste sunt următoarele:

- a) **`len(listă)`**: furnizează numărul de elemente din listă (lungimea listei)

Exemplu: `len([10, 20, 30, "abc", [1, 2, 3]]) = 5`

- b) **`list(secvență)`**: furnizează o listă formată din elementele secvenței respective

Exemplu: `list("test") = ['t', 'e', 's', 't']`

- c) **`min(listă) / max(listă)`**: furnizează elementul minim/maxim în sens lexicografic din lista respectivă (atenție, toate elementele listei trebuie să fie comparabile între ele, altfel va fi generată o eroare!)

Exemple:

```
L = [100, -70, 16, 101, -85, 100, -70, 28]
print("Minimul din lista:", min(L))      # -85
print("Maximul din lista:", max(L))      # 101
print()
L = [[2, 10], [2, 1, 2], [60, 2, 1], [3, 140, 5]]
print("Minimul din lista:", min(L))      # [2, 1, 2]
print("Maximul din lista:", max(L))      # [60, 2, 1]

L = [20, -30, "101", 17, 100]
print("Minimul din lista:", min(L))
# TypeError: '<' not supported between
# instances of 'str' and 'int'
```

- d) **`sum(listă)`**: furnizează suma elementelor unei liste (evident, toate elementele listei trebuie să fie de tip numeric)

Exemplu: `sum([10, -70, 100, -80, 100, -70]) = -10`

- e) **`sorted(listă, [reverse=False])`**: furnizează o listă formată din elementele listei respective sortate crescător (lista inițială nu va fi modificată!).

Exemplu: `sorted([1, -7, 1, -8, 1, -7]) = [-8, -7, -7, 1, 1, 1]`

Elementele listei pot fi sortate și descrescător, setând parametrul optional `reverse` al funcției `sorted` la valoarea `True`.

Exemplu: `sorted([1, -7, 1, -8], reverse=True) = [1, 1, -7, -8]`

Metode pentru prelucrarea listelor

Metodele pentru prelucrarea listelor sunt, de fapt, metodele încapsulate în clasa `list`. Aşa cum am precizat anterior, listele sunt *mutable*, deci metodele respective pot modifica lista curentă, dacă acest lucru este necesar!

În continuare, vom prezenta mai multe metode pentru prelucrarea listelor, cu observaţia că parametrii scrişi între paranteze drepte sunt optionali:

- a) **`count(valoare)`**: furnizează numărul de apariţii ale valorii respective în listă.

Exemplu:

```
L = [x % 4 for x in range(12)]
print(L)      # [0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3]
n = L.count(2)
print(n)      # 3
```

- b) **`append(valoare)`**: adaugă valoarea respectivă în listă.

Exemplu:

```
L = [x + 1 for x in range(5)]
print(L)      # [1, 2, 3, 4, 5]

L.append("abc")
print(L)      # [1, 2, 3, 4, 5, 'abc']

L.append([10, 20, 30])
print(L)      # [1, 2, 3, 4, 5, 'abc', [10, 20, 30]]
```

- c) **`extend(secvență)`**: adaugă, pe rând, toate elementele din secvența dată în listă.

Exemplu:

```
L = [1, 2, 3]
L.append("test")
print(L)      # [1, 2, 3, 'test']

L = [1, 2, 3]
L.extend("test")
print(L)      # [1, 2, 3, 't', 'e', 's', 't']

L = [1, 2, 3]
L.append([10, 20, 30])
print(L)      # [1, 2, 3, [10, 20, 30]]

L = [1, 2, 3]
L.extend([10, 20, 30, [40, 50]])
print(L)      # [1, 2, 3, 10, 20, 30, [40, 50]]
```

- d) **insert(poziție, valoare)**: inserează în listă valoarea dată înaintea poziției respective.

Exemplu:

```
L = [x + 1 for x in range(5)]
print(L)      # [1, 2, 3, 4, 5]

L.insert(3, "test")
print(L)      # [1, 2, 3, 'test', 4, 5]

L.insert(30, "abc")
print(L)      # [1, 2, 3, 'test', 4, 5, 'abc']
```

- e) **index(valoare)**: furnizează poziția primei apariții, de la stânga la dreapta, a valorii date sau lansează o eroare (`ValueError`) dacă valoarea nu apare în listă.

Exemplu:

Pentru a evita apariția erorii `ValueError`, mai întâi am verificat faptul că valoarea `x` căutată se găsește în listă:

```
L = [x + 1 for x in range(5)]
print(f"Lista: {L}")

x = 30
if x in L:
    p = L.index(x)
    print(f"Valoarea {x} apare in lista pe pozitia {p}!")
else:
    print(f"Valoarea {x} nu apare in lista!")
```

O altă modalitate de utilizare a metodei `index`, mai eficientă, constă în tratarea erorii care poate să apară când valoarea `x` căutată nu se găsește în listă:

```
L = [x + 1 for x in range(5)]
print(L)

x = 30
try:
    p = L.index(x)
    print(f"Valoarea {x} apare in lista pe pozitia {p}!")
except ValueError:
    print(f"Valoarea {x} nu apare in lista!")
```

- f) **remove(valoare)**: șterge din lista curentă prima apariție, de la stânga la dreapta, a valorii date sau lansează o eroare (`ValueError`) dacă valoarea nu apare în listă.

Exemplu:

Pentru a evita apariția erorii `ValueError`, mai întâi am verificat faptul că valoarea `x` pe care dorim să o ștergem se găsește în listă:

```
L = [x + 1 for x in range(5)]
print(f"Lista: {L}")

x = 3
if x in L:
    L.remove(x)
    print(f"Lista dupa stergerea valorii {x}: {L}!")
else:
    print(f"Valoarea {x} nu apare in lista!")
```

O altă modalitate de utilizare a metodei `remove`, mai eficientă, constă în tratarea erorii care poate să apară când valoarea `x` pe care dorim să o ștergem nu se găsește în listă:

```
L = [x + 1 for x in range(5)]
print(L)

x = 30
try:
    L.remove(x)
    print(f"Lista dupa stergerea valorii {x}: {L}!")
except ValueError:
    print(f"Valoarea {x} nu apare in lista!")
```

- g) **`pop([poziție])`**: furnizează elementul aflat pe poziția respectivă și apoi îl șterge. Dacă nu se precizează nicio poziție, atunci funcția va considera implicit ultima poziție din listă.

```
L = [x + 10 for x in range(5)]
print(f"Lista initiala: {L}")
# Lista initiala: [10, 11, 12, 13, 14]

poz = 3
val = L.pop(poz)
print(f"\nValoarea de pe pozitia {poz} era {val}")
# Valoarea de pe pozitia 3 era 13
print(f"Noua lista: {L}")
# Noua lista: [10, 11, 12, 14]

val = L.pop()
print(f"\nValoarea de pe ultima pozitie era {val}")
# Valoarea de pe ultima pozitie era 14
print(f"Noua lista: {L}")
# Noua lista: [10, 11, 12]
```

Dacă poziția precizată ca parametru nu există în listă, atunci va apărea eroarea `IndexError`. Pentru a evita acest lucru, fie mai întâi se verifică faptul că poziția este cuprinsă între 0 și `len(lista)-1`, fie se tratează eroarea respectivă.

- h) **`clear()`**: șterge toate elementele din listă, fiind echivalentă cu `listă = []`.

- i) **reverse()**: oglindește lista, respectiv primul element devine ultimul, al doilea devine penultimul și.a.m.d

Exemplu:

```
L = [x + 1 for x in range(5)]
print(L)      # [1, 2, 3, 4, 5]

L.reverse()
print(L)      # [5, 4, 3, 2, 1]
```

- f) **sort([reverse=False])**: sortează crescător lista, modificând ordinea inițială a elementelor sale. Elementele listei inițiale pot fi sortate și descrescător, setând parametrul optional `reverse` al metodei la valoarea `True`.

Exemplu:

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

L.sort()
print(L)      # [1, 1, 2, 2, 3, 3]

L = [3, 1, 2, 3, 2, 1]
print(L)      # [3, 1, 2, 3, 2, 1]

L.sort(reverse=True)
print(L)      # [3, 3, 2, 2, 1, 1]
```

Crearea unei liste

O listă poate fi creată folosind valori constante, secvențe de valori, valori citite de la tastatură sau valori citite dintr-un fișier. Indiferent de sursa elementelor utilizate pentru crearea listei, există mai multe variante de implementare pe care le putem utiliza: secvențe de inițializare, adăugarea unui element folosind metoda `append` sau operatorul `+=` (ambele variante sunt echivalente!), adăugarea unui element pe o anumită poziție (i.e., accesarea elementelor prin indecsă) sau concatenarea la lista curentă a unei liste formată doar din elementul pe care dorim să-l adăugăm. În continuare, vom testa toate aceste variante din punct de vedere al timpului de executare, creând, de fiecare dată, o listă formată cu 500000 de elemente, respectiv numerele 0, 1, 2, ..., 499999:

```
import time

nr_elemente = 500000

start = time.time()
lista = [x for x in range(nr_elemente)]
stop = time.time()
print("  Initializare: ", stop - start, "secunde")
```

```

start = time.time()
lista = []
for x in range(nr_elemente):
    lista.append(x)
stop = time.time()
print("Metoda append(): ", stop - start, "secunde")

start = time.time()
lista = []
for x in range(nr_elemente):
    lista += [x]
stop = time.time()
print(" Operatorul +=: ", stop - start, "secunde")

start = time.time()
lista = [0] * nr_elemente
for x in range(nr_elemente):
    lista[x] = x
stop = time.time()
print("          Index: ", stop - start, "secunde")

start = time.time()
lista = []
for x in range(nr_elemente):
    lista = lista + [x]
stop = time.time()
print("  Operatorul +: ", stop - start, "secunde")

```

Rezultatele obținute sunt următoarele:

Initializare:	0.031244277954101562 secunde
Metoda append():	0.0468595027923584 secunde
Operatorul +=:	0.04686307907104492 secunde
Index:	0.03124260902404785 secunde
Operatorul +:	859.0856750011444 secunde

Se observă faptul că primele 4 variante au tempi de executare aproximativi egali, iar ultima variantă are un timp de executare mult mai mare din cauza faptului că la fiecare operație de concatenare a listei `[x]` la lista curentă se creează în memorie o copie a listei curente, se adaugă la sfârșitul copiei nouă valoare `x` și apoi referința listei curente se înlocuiește cu referința copiei.

Pentru a crea o listă formată din numere întregi citite de la tastatură, se pot utiliza următoarele variante (derivate din cele prezentate anterior):

- se citește numărul n de elemente din listă și apoi se citesc, pe rând, cele n elemente ale sale;

```
n = int(input("Numarul de elemente din lista: "))
L = []
for i in range(n):
    x = int(input("Element: "))
    L.append(x)
print(f"Lista: {L}")
```

- b) se citesc, pe rând, elementele listei până se întâlnește o anumită valoare (de exemplu, numărul 0):

```
L = []
while True:
    x = int(input("Element: "))
    if x != 0:
        L.append(x)
    else:
        break
print(f"Lista: {L}")
```

- c) se citește numărul n de elemente din listă, se creează o listă formată din n valori nule și apoi se citesc, pe rând, cele n elemente folosind accesarea prin index:

```
n = int(input("Numarul de elemente din lista: "))
L = [0] * n
for i in range(n):
    L[i] = int(input("Element: "))
print(f"Lista: {L}")
```

- d) se citesc toate elementele listei, despărțite între ele printr-un spațiu, într-un sir de caractere și apoi se extrag numerele din sirul respectiv, împărțindu-l în subșirurile delimitate de spații:

```
sir = input("Elementele listei: ")
L = []
for x in sir.split():
    L.append(int(x))
print(f"Lista: {L}")
```

Această variantă poate fi scrisă foarte compact, folosind secvențele de inițializare:

```
L = [int(x) for x in input("Elementele listei: ").split()]
print(f"Lista: {L}")
```

În toate exemplele anterioare, se poate utiliza în locul metodei `append` operatorul `+=`, dar, evident, nu se recomandă utilizarea operatorului `+`.

Elementele unei liste pot fi, de asemenea, liste, ceea ce permite utilizarea lor pentru implementarea unor structuri de date bidimensionale (i.e., de tip matrice). De exemplu, un tablou bidimensional cu m linii și n coloane format din numere întregi poate fi creat în mai multe moduri:

- a) se citesc numerele m și n , apoi se citesc, pe rând, elementele de pe fiecare linie a tabloului bidimensional:

```
m = int(input("Numarul de liniile: "))
n = int(input("Numarul de coloane: "))
T = []
for i in range(m):
    linie = []
    for j in range(n):
        x = int(input(f"T[{i}][{j}] = "))
        linie.append(x)
    T.append(linie)
print(f"Tabloul bidimensional: {T}")
```

Se observă faptul că tabloul bidimensional va fi afișat sub forma unor liste imbricate (e.g., sub forma $[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]$). Pentru a afișa tabloul bidimensional sub forma unei matrice, vom afișa, pe rând, elementele de pe fiecare linie:

```
print("Tabloul bidimensional:")
for linie in T:
    for elem in linie:
        print(elem, end=" ")
    print()
```

În acest caz, tabloul din exemplul de mai sus va fi afișat astfel:

Tabloul bidimensional:

```
1 2 3 4
5 6 7 8
9 10 11 12
```

- b) se citesc numerele m și n , se creează o listă formată din m liste formate, fiecare, din câte n valori nule și apoi se citesc, pe rând, cele elemente tabloului bidimensional folosind accesarea prin index:

```
m = int(input("Numarul de liniile: "))
n = int(input("Numarul de coloane: "))

T = [[0 for x in range(n)] for y in range(m)]

for i in range(m):
    for j in range(n):
        T[i][j] = int(input(f"T[{i}][{j}] = "))

print("Tabloul bidimensional:")
for linie in T:
    for elem in linie:
        print(elem, end=" ")
    print()
```

Atenție, tabloul bidimensional T nu poate fi inițializat prin $T = [[0] * n] * m$, deoarece se va crea o singură listă formată din n valori nule, iar referința sa va fi copiată de m ori în lista T :

```
m = 3    #numarul de liniile
n = 5    #numarul de coloane

# variantă incorrectă: toate liniile vor conține aceeași
# referință!
T = [[0] * n] * m

print("Tabloul inițial:")
for linie in T:
    for elem in linie:
        print(elem, end=" ")
    print()

T[1][3] = 7

print("\nTabloul modificat:")
for linie in T:
    for elem in linie:
        print(elem, end=" ")
    print()
```

După rularea secvenței de cod anterioare, se va obține pe monitor următorul rezultat:

Tabloul inițial:

```
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
```

Tabloul modificat:

```
0 0 0 7 0
0 0 0 7 0
0 0 0 7 0
```

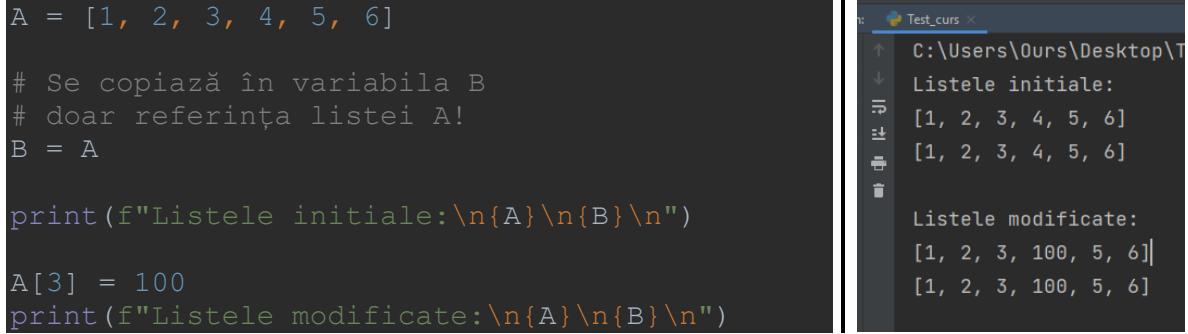
- c) se citesc, pe rând, liniile tabloului dimensional până când se introduce o linie vidă (elementele unei linii vor fi introduse despărțite între ele prin câte un spațiu):

```
T = []
while True:
    linie = input(f"Elementele de pe linia {len(T)}: ")
    if len(linie) != 0:
        T.append([int(x) for x in linie.split()])
    else:
        break
```

Se observă faptul că, în acest caz, liniile nu trebuie să mai aibă toate același număr de elemente!

Realizarea unei copii a unei liste

În multe situații dorim să realizăm o copie a unei liste, de exemplu pentru a-i păstra conținutul înainte de o anumită modificare. O variantă greșită de realizare a acestei operații constă în utilizarea unei instrucțiuni de atribuire, aşa cum se poate observa în exemplul următor:



```
A = [1, 2, 3, 4, 5, 6]

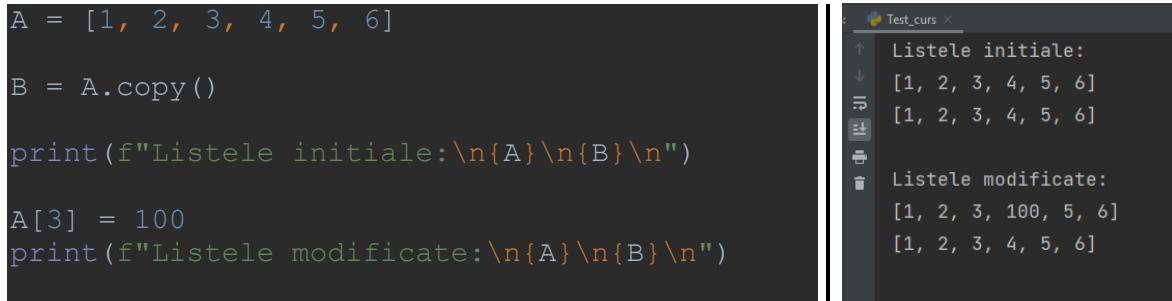
# Se copiază în variabila B
# doar referința listei A!
B = A

print(f"Listele initiale:\n{A}\n{B}\n")

A[3] = 100
print(f"Listele modificate:\n{A}\n{B}\n")
```

În exemplul de mai sus, se va copia în variabila B doar referința listei A, ci nu conținutul său! Din acest motiv, orice modificare efectuată asupra uneia dintre cele două liste (de fapt, două referințe spre un singur obiect de tip `list`!) se va reflecta asupra amânduroră.

O prima variantă de realizare corectă a unei copii a unei liste o constituie utilizarea metodei `copy` din clasa `list`:



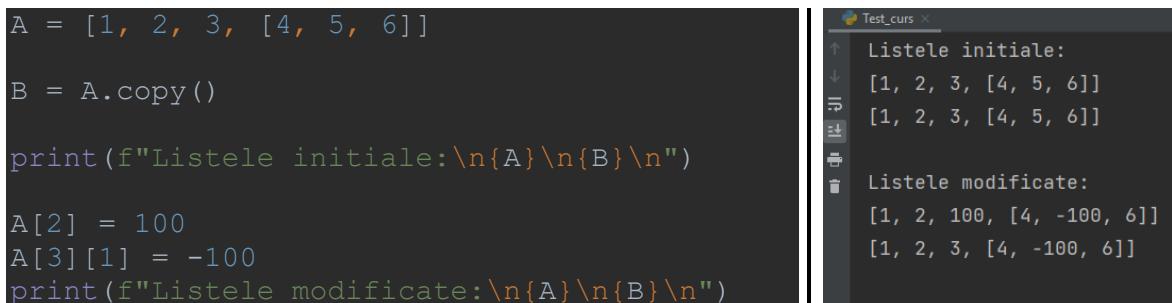
```
A = [1, 2, 3, 4, 5, 6]

B = A.copy()

print(f"Listele initiale:\n{A}\n{B}\n")

A[3] = 100
print(f"Listele modificate:\n{A}\n{B}\n")
```

Totuși, metoda `copy` va realiza doar o copie superficială (*shallow copy*) a listei A, respectiv va copia conținutul listei A, element cu element, în lista B. Din acest motiv, această metodă nu poate fi utilizată dacă lista A conține referințe, aşa cum se poate observa din exemplul următor:



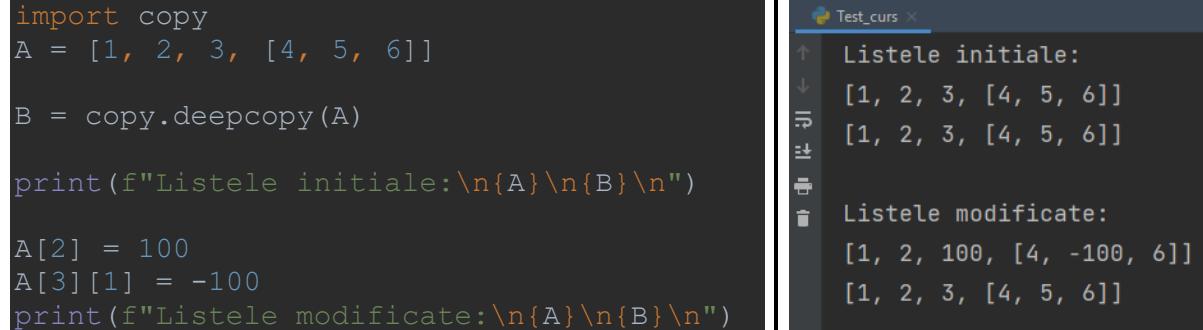
```
A = [1, 2, 3, [4, 5, 6]]

B = A.copy()

print(f"Listele initiale:\n{A}\n{B}\n")

A[2] = 100
A[3][1] = -100
print(f"Listele modificate:\n{A}\n{B}\n")
```

Pentru a rezolva problema anterioară, vom utiliza metoda `deepcopy` din modulul `copy`, care va realiza o copie în adâncime (*deep copy*) a listei A:



```

import copy
A = [1, 2, 3, [4, 5, 6]]
B = copy.deepcopy(A)

print(f"Listele initiale:\n{A}\n{B}\n")

A[2] = 100
A[3][1] = -100
print(f"Listele modificate:\n{A}\n{B}\n")

```

Deși utilizarea acestei metode rezolvă problema copierii unei liste în orice caz, se recomandă utilizarea sa cu precauție, deoarece timpul său de executare poate fi foarte mare în unele cazuri!

Tupluri

Un *tuplu* este o secvență imutabilă de valori indexate de la 0. Valorile memorate într-un tuplu pot fi neomogene (i.e., pot fi de tipuri diferite de date) și, datorită imutabilității, nu pot fi modificate. Tot datorită imutabilității lor, tuplurile sunt mai rapide și ocupă mai puțină memorie decât listele. Tuplurile sunt instanțe ale clasei `tuple`.

Un tuplu poate fi creat/initializat în mai multe moduri:

- folosind constante:

```

# tuplu vid
t = ()
print(t)

# tuplu cu un singur element (atentie la virgula!)
t = (1,)
print(t)

# initializare cu valori constante
t = (123, "Popescu Ion", 9.50)
print(t)

# initializare cu valori constante (varianta fara paranteze)
t = 123, "Popescu Ion", 9.50
print(t)

# initializare cu valori preluate dintr-o lista
t = tuple([123, "Popescu Ion", 9.50])
print(t)

```

```
#initializare cu valori preluate dintr-un sir de caractere
t = tuple("test")           # t = ('t', 'e', 's', 't')
print(t)
```

- folosind secvențe de inițializare (*list comprehensions*):

```
# secvență de inițializare
t = tuple(x + 1 for x in range(10))
print(t)                  # [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# secvențe de inițializare condiționale
L = tuple(x**2 for x in range(10) if x % 2 == 0)
print(t)                  # [0, 4, 16, 36, 64]

# citirea de la tastatură a unui tuplu de numere întregi
t = tuple(int(x) for x in input("Valori: ").split())
print(t)
```

Observați faptul că tuplurile pot fi create folosind secvențe de inițializare doar prin intermediul funcției **tuple**!

Accesarea elementelor unui tuplu

Elementele unui tuplu pot fi accesate în mai multe moduri, asemănătoare celor prezentate pentru siruri de caractere și liste:

- a) *prin indici pozitivi sau negativi*

În limbajul Python, oricărei secvențe (*mulțime iterabilă*) de lungime n îi sunt asociati atât indici pozitivi, cuprinși între 0 și $n - 1$ de la stânga spre dreapta, cât și indici negativi, cuprinși între $-n$ și -1 de la stânga la dreapta.

Exemplu: pentru tuplul $T = (10, 20, 30, 40, 50, 60, 70, 80, 90, 100)$ avem asociatii următorii indici:

T	0	1	2	3	4	5	6	7	8	9
	10	20	30	40	50	60	70	80	90	100
	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Astfel, al patrulea element din tuplu (numărul 40), poate fi accesat atât prin $T[3]$, cât și prin $T[-7]$. Atenție, tuplurile sunt imutabile, deci, spre deosebire de liste, un element nu poate fi modificat direct (e.g., atribuirea $T[3] = 400$ va genera o eroare de tipul **TypeError: 'tuple' object does not support item assignment**!). Totuși, elementul aflat într-un tuplu T pe o poziție p validă (i.e., cuprinsă între 0 și $\text{len}(T)-1$) poate fi modificat sau șters construind un nou tuplu a cărui referință va înlocui referința inițială:

$T = T[:p] + (element_nou,) + T[p+1:]$ (modificare)
 $T = T[:p] + T[p+1:]$ (ștergere)

b) prin secvențe de indici pozitivi sau negativi (slice)

Expresia `tuplu[st:dr]` extrage din tuplul dat un tuplu format din elementele aflate între pozițiile `st` și `dr-1`, dacă `st ≤ dr`, sau un tuplu vid în caz contrar.

Exemple:

```
T = (10, 20, 30, 40, 50, 60, 70, 80, 90, 100)
T[1: 4] == (20, 30, 40)
T[:] == T
T[5: 2] == () #pentru că 5 > 2
T[5: 2: -1] == (60, 50, 40)
T[:: -1] == (100, 90, 80, 70, 60, 50, 40, 30, 20, 10) #tuplul inversat
T[-9: 4] == (20, 30, 40)
```

Operatori pentru tupluri

În limbajul Python sunt definiți următorii operatori pentru manipularea tuplurilor:

a) operatorul de concatenare: `+`

Exemplu: `(1, 2, 3) + (4, 5) == (1, 2, 3, 4, 5)`

b) operatorul de concatenare și atribuire: `+=`

Exemplu:

```
T = (1, 2, 3)
T += (4, 5)
print(T)           # (1, 2, 3, 4, 5)
```

c) operatorul de multiplicare (concatenare repetată): `*`

Exemplu: `(1, 2, 3) * 3 = (1, 2, 3, 1, 2, 3, 1, 2, 3)`

d) operatorii pentru testarea apartenenței: `in`, `not in`

Exemplu: expresia `3 in (2, 1, 4, 3, 5)` va avea valoarea `True`

e) operatorii relationali: `<`, `<=`, `>`, `>=`, `==`, `!=`

Observație: În cazul primilor 4 operatori, cele două tupluri vor fi comparate lexicografic, deci elementele efectiv analizate trebuie să fie comparabile, altfel se va genera o eroare!

Exemple:

```
T1 = (1, 2, 3, 100)
T2 = (1, 2, 4)
print(T1 <= T2)           # True

T2 = (1, 2, 4, "Pop Ion")
print(T1 >= T2)           # False

T2 = (1, 2, "Pop Ion")
print(T1 == T2)            # False
print(T1 <= T2)           # Eroare, deoarece nu se pot compara
                           # lexicografic numărul 3 și sirul "Pop
                           # Ion"
```

Funcții predefinite pentru tupluri

În limbajul Python sunt predefinite mai multe funcții (*built-in functions* – <https://docs.python.org/3/library/functions.html>), dintre care unele pot fi utilizate pentru mai multe tipuri de date. De exemplu, funcția `len(secvență)` va furniza numărul de elemente dintr-o secvență (iterabil), indiferent dacă aceasta este un tuplu, o listă sau un sir de caractere. Funcțiile predefinite care se pot utiliza pentru tupluri sunt următoarele:

- a) **`len(tuplu)`**: furnizează numărul de elemente din tuplu (lungimea tuplului)

Exemplu: `len((10, 20, 30, "abc", [1, 2, 3])) = 5`

- b) **`tuple(secvență)`**: furnizează un tuplu format din elementele secvenței respective

Exemplu: `tuple("test") = ('t', 'e', 's', 't')`

- c) **`min(tuplu) / max(tuplu)`**: furnizează elementul minim/maxim în sens lexicografic din tuplul respectiv (atenție, toate elementele tuplului trebuie să fie comparabile între ele, altfel va fi generată o eroare!)

Exemple:

```

T = (100, -70, 16, 101, -85, 100, -70, 28)
print("Minimul din tuplul T:", min(T))          # -85
print("Maximul din tuplul T:", max(T))          # 101
print()

T = ([2, 10], [2, 1, 2], [60, 2, 1], [3, 140, 5])
print("Minimul din tuplul T:", min(T))          # [2, 1, 2]
print("Maximul din tuplul T:", max(T))          # [60, 2, 1]

T = ("exemplu", "test", "constanta", "rest")
print("Minimul din tuplul T:", min(T))          # constanta
print("Maximul din tuplul T:", max(T))          # test

T = [20, -30, "101", 17, 100]
print("Minimul din tuplul T:", min(T))
# TypeError: '<' not supported between
# instances of 'str' and 'int'

```

- d) **`sum(tuplu)`**: furnizează suma elementelor unui tuplu (evident, toate elementele tuplului trebuie să fie de tip numeric)

Exemplu: `sum((10, -70, 100, -80, 100, -70)) = -10`

- e) **`sorted(tuplu, [reverse=False])`**: furnizează o listă formată din elementele tuplului sortate implicit crescător (tuplul nu va fi modificat!).

Exemplu: `sorted((1, -7, 1, -8, 1, -7)) = [-8, -7, -7, 1, 1, 1]`

Pentru a obține tot un tuplu în urma utilizării funcției `sorted` pentru sortarea unui tuplu, trebuie să folosim funcția `tuple`:

```
T = (1, -7, 1, -8, 1, -7)
T = tuple(sorted(T))
print(T)           # (-8, -7, -7, 1, 1, 1)
```

Elementele tuplului pot fi sortate și descrescător, setând parametrul optional `reverse` al funcției `sorted` la valoarea `True`.

Exemplu: `sorted((1, -7, 1, -8), reverse=True) = [1, 1, -7, -8]`

Metode pentru prelucrarea tuplurilor

Deoarece tuplurile sunt imutabile, metodele pentru prelucrarea tuplurilor sunt, de fapt, metodele specifice listelor, dar care nu modifică lista curentă:

- a) **count(valoare)**: furnizează numărul de apariții ale valorii respective în tuplu.

Exemplu:

```
T = tuple(x % 4 for x in range(12))
print(T)      # (0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3)
n = T.count(2)
print(n)      # 3
```

- b) **index(valoare)**: furnizează poziția primei apariții, de la stânga la dreapta, a valorii date în tuplul curent sau lansează o eroare (`ValueError`) dacă valoarea respectivă nu apare în tuplu.

Exemplu:

Pentru a evita apariția erorii `ValueError`, mai întâi am verificat faptul că valoarea `x` căutată se găsește în tuplu:

```
T = tuple(x + 1 for x in range(5))
print(f"Tuplul: {T}")

x = 30
if x in T:
    p = T.index(x)
    print(f"Valoarea {x} apare in tuplu pe pozitia {p}!")
else:
    print(f"Valoarea {x} nu apare in tuplu!")
```

O altă modalitate de utilizare a metodei `index`, mai eficientă, constă în tratarea erorii care poate să apară când valoarea `x` căutată nu se găsește în tuplul curent:

```

T = tuple(x + 1 for x in range(5))
print(f"Tuplul: {T}")

x = 30
try:
    p = T.index(x)
    print(f"Valoarea {x} apare in tuplu pe pozitia {p}!")
except ValueError:
    print(f"Valoarea {x} nu apare in tuplu!")

```

Crearea unui tuplu

Deoarece tuplurile sunt imutabile, există mai puține variante de a crea un tuplu decât o listă: secvențe de inițializare, adăugarea unui element folosind operatorul `+=`, concatenarea la tuplul curent a unei liste formate doar din elementul curent sau conversia unei liste formată din elementele dorite. În continuare, vom testa toate aceste variante din punct de vedere al timpului de execuțare, creând, de fiecare dată, un tuplu format din 500000 de elemente, respectiv numerele 0, 1, 2, ..., 499999:

```

import time

nr_elemente = 500_000

start = time.time()
tuplu = tuple(x for x in range(nr_elemente))
stop = time.time()
print("  Initializare: ", stop - start, "secunde")

start = time.time()
lista = []
for x in range(nr_elemente):
    lista.append(x)
tuplu = tuple(lista)
stop = time.time()
print("  Dintr-o lista: ", stop - start, "secunde")

start = time.time()
tuplu = tuple()
for x in range(nr_elemente):
    tuplu += (x,)
stop = time.time()
print("  Operatorul +=: ", stop - start, "secunde")

start = time.time()
tuplu = ()
for x in range(nr_elemente):
    tuplu = tuplu + (x,)
stop = time.time()
print("  Operatorul +: ", stop - start, "secunde")

```

Rezultatele obținute sunt următoarele:

```
Initializare: 0.02988576889038086 secunde
Dintr-o lista: 0.06030845642089844 secunde
Operatorul +=: 904.4887342453003 secunde
Operatorul +: 848.3564832210541 secunde
```

Se observă faptul că primele două variante au tempi de executare foarte buni, în timp ce ultimele două variante au tempi de executare mult mai mari, din cauza faptului că la fiecare operație de concatenare a tuplului (`x,`) la tuplul curent se creează în memorie o copie a tuplului curent, se adaugă la sfârșitul copiei nouă valoare `x` și apoi referința tuplului curent se înlocuiește cu referința copiei.

Realizarea unei copii a unui tuplu

Deoarece tuplurile sunt imutabile, conținutul lor nu poate fi modificat, deci singura problemă care poate să apară în momentul copierii unui tuplu este existența în el a unor referințe spre obiecte mutabile:

```
a = (1, 2, 3, [4, 5, 6])
b = a
print(f"Tuplurile intiale:", a, b, sep="\n")

a[3][1] = -100
print("\nTuplurile modificate:", a, b, sep="\n")
```

```
Test_curs >
C:\Users\Ours\Desktop\Test
Tuplurile intiale:
(1, 2, 3, [4, 5, 6])
(1, 2, 3, [4, 5, 6])

Tuplurile modificate:
(1, 2, 3, [4, -100, 6])
(1, 2, 3, [4, -100, 6])
```

Pentru a rezolva problema anterioară, la fel ca în cazul listelor, vom utiliza metoda `deepcopy` din modulul `copy`, care va realiza o copie în adâncime (*deep copy*):

```
import copy

a = (1, 2, 3, [4, 5, 6])
b = copy.deepcopy(a)
print("\nTuplurile intiale:", a, b, sep="\n")

a[3][1] = -100
print("\nTuplurile modificate:", a, b, sep="\n")
```

```
Test_curs >
Tuplurile intiale:
(1, 2, 3, [4, 5, 6])
(1, 2, 3, [4, 5, 6])

Tuplurile modificate:
(1, 2, 3, [4, -100, 6])
(1, 2, 3, [4, -100, 6])
```

Deși utilizarea acestei metode rezolvă problema copierii unui tuplu în orice caz, se recomandă utilizarea sa cu precauție, deoarece timpul său de executare poate fi foarte mare în unele cazuri!

Împachetarea și despachetarea unui tuplu

Limbajul Python pune la dispoziția programatorilor un mecanism complex de atribuire, prin care se pot atribui mai multe valori la un moment dat. Astfel, *împachetarea unui tuplu* (*tuple packing*) permite atribuirea simultană a mai multor valori unui singur tuplu, în timp ce *despachetarea unui tuplu* (*tuple unpacking*) permite atribuirea valorilor dintr-un tuplu mai multor variabile.

Exemplu:

```
t = (1, 2, 3)      # împachetarea celor 3 numere într-un tuplu
print("t = ", t)    # t = (1, 2, 3)

x, y, z = t        # despachetarea tuplului în 3 variabile
print("x = ", x)    # x = 1
print("y = ", y)    # y = 2
print("z = ", z)    # z = 3

t = 4, 5, 6        # împachetarea celor 3 numere într-un tuplu,
                   # fără a utiliza paranteze
print("t = ", t)    # t = (4, 5, 6)
```

Evident, în cazul operației de despachetare, numărul variabilelor din partea stângă a instrucțiunii de atribuire trebuie să coincidă cu numărul elementelor tuplului din partea dreaptă, în caz contrar apărând erori.

Dacă în momentul despachetării unui tuplu nu știm exact numărul elementelor sale, atunci putem să utilizăm operatorul * în fața numelui unei variabile pentru a indica faptul că în ea se vor memora mai multe valori aflate pe poziții consecutive, sub forma unei liste.

Exemplu:

```
t = (1, 2, 3, 4, 5, 6)
x, *y, z = t
print("x = ", x)      # x = 1
print("y = ", y)      # y = [2, 3, 4, 5]
print("z = ", z)      # z = 6

t = (1, 2)
x, y, *z = t
print("x = ", x)      # x = 1
print("y = ", y)      # y = 2
print("z = ", z)      # z = []

t = (131, "Popescu", "Ion", 9.70)
grupa, *nume, medie = t
print("t = ", t)
print("Grupa = ", grupa)    # Grupa = 131
print("Nume = ", nume)       # Nume = ['Popescu', 'Ion']
print("Medie = ", medie)     # Medie = 9.7
```

```
t = (132, "Popa", "Anca", "Maria", 10)
grupa, *nume, medie = t
print("t = ", t)
print("Grupa = ", grupa)      # Grupa = 131
print("Nume = ", nume)        # Nume = ['Popa', 'Anca', 'Maria']
print("Medie = ", medie)      # Medie = 9.7
```

Evident, și în cazul utilizării operatorului *, numărul variabilelor din partea stângă a instrucțiunii de atribuire trebuie să fie în concordanță cu numărul elementelor tuplului din partea dreaptă, în caz contrar apărând erori.

Operația de despachetare poate fi aplicată pentru orice tip de date secvențial (i.e., sir de caractere, listă sau tuplu), aşa cum se poate observa din exemplele următoare:

The screenshot shows a Python code editor with a dark theme. On the left, the code is written in Python, and on the right, the output of the code execution is displayed in the terminal window.

```
lista = [1, 2, 3, 4, 5, 6]
print("Lista despachetată:", *lista)

sir = "exemplu"
print("\nSirul despachetat:", *sir)

matrice = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

print("\nMatricea:")
print(*matrice, sep="\n")

print("\nMatricea:")
for linie in matrice:
    print(*linie)
```

Output:

```
Test_curs x
C:\Users\Ours\Desktop\Test_Pytho
Lista despachetată: 1 2 3 4 5 6
Sirul despachetat: e x e m p l u
|
Matricea:
[1, 2, 3]
[4, 5, 6]
[7, 8, 9]

Matricea:
1 2 3
4 5 6
7 8 9
```

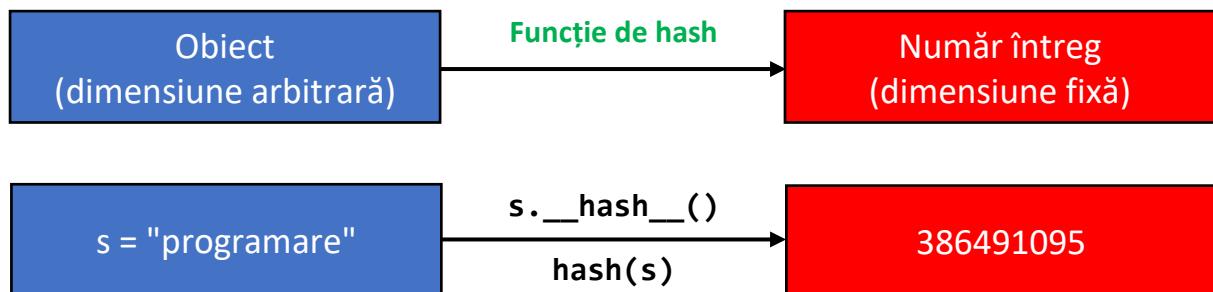
CURS 5

Colecții de date

Tabele de dispersie (hash table)

În general, o *funcție de dispersie (hash function)* este un algoritm care asociază unui sir binar de lungime arbitrară un sir binar de lungime fixă numit *valoare de dispersie* sau *cod de dispersie (hash value, hash code sau digest)*.

În limbajul Python, clasele corespunzătoare tipurilor de date imutabile (i.e., clasele `int`, `float`, `complex`, `bool`, `str`, `tuple` și `frozenset`) conțin metoda `__hash__()` care furnizează valoarea de dispersie asociată unui anumit obiect sub forma unui număr întreg pe 32 sau 64 de biți.



Alternativ, valoarea de dispersie a unui obiect poate fi aflată utilizând funcția predefinită `hash(obiect)`.

<pre> s = "Ana are mere!" print(f"hash('{s}') = {s.__hash__()}") print(f"hash('{s}') = {hash(s)}\n") n = 12345 print(f"hash({n}) = {hash(n)}\n") n = 2**100 print(f"hash({n}) = {hash(n)}\n") n = 3.14 print(f"hash({n}) = {hash(n)}\n") t = (1, 2, 3, 4, 5) print(f"hash({t}) = {hash(t)}\n") </pre>	<pre> Test_curs C:\Users\Ours\Desktop\Test_Python\venv\Scripts\python hash('Ana are mere!') = -2312829570296290796 hash('Ana are mere!') = -2312829570296290796 hash(12345) = 12345 hash(1267650600228229401496703205376) = 549755813888 hash(3.14) = 322818021289917443 hash((1, 2, 3, 4, 5)) = -5659871693760987716 </pre>
---	--

În limbajul Python, orice funcție de dispersie trebuie să satisfacă următoarele două condiții:

1. două obiecte care sunt egale din punct de vedere al conținutului trebuie să fie egale și din punct de vedere al valorilor de dispersie (i.e., dacă `obiect_1 == obiect_2` atunci obligatoriu și `hash(obiect_1) == hash(obiect_2)`);

2. valoarea de dispersie a unui obiect trebuie să rămână constantă pe parcursul executării programului în care este utilizat obiectul respectiv, dar nu trebuie să rămână constantă în cazul unor rulări diferite ale programului.

Putem observa faptul că ambele condiții de mai sus sunt respectate de funcția predefinită `hash`, rulând de mai multe ori următoarea secvență de instrucțiuni:

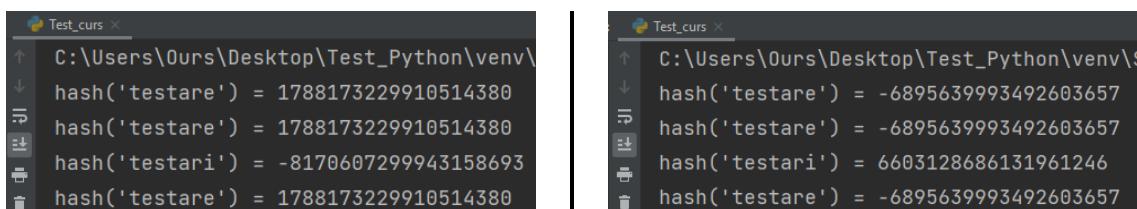
```
s = "testare"
print(f"hash('{s}') = {hash(s)}")

t = "test"
t += "are"
print(f"hash('{t}') = {hash(t)}")

s = s[:len(s)-1] + "i"           # s = "testari"
print(f"hash('{s}') = {hash(s)}")

s = s[:len(s)-1] + "e"           # s = "testare"
print(f"hash('{s}') = {hash(s)}")
```

Astfel, vom observa faptul că la fiecare rulare a secvenței primele două valori afișate și ultima sunt întotdeauna egale, fără a rămâne constante în cazul mai multor rulări:



Run 1 (Left)	Run 2 (Right)
<code>hash('testare')</code> = 1788173229910514380	<code>hash('testare')</code> = -6895639993492603657
<code>hash('testare')</code> = 1788173229910514380	<code>hash('testare')</code> = -6895639993492603657
<code>hash('testari')</code> = -8170607299943158693	<code>hash('testari')</code> = 6603128686131961246
<code>hash('testare')</code> = 1788173229910514380	<code>hash('testare')</code> = -6895639993492603657

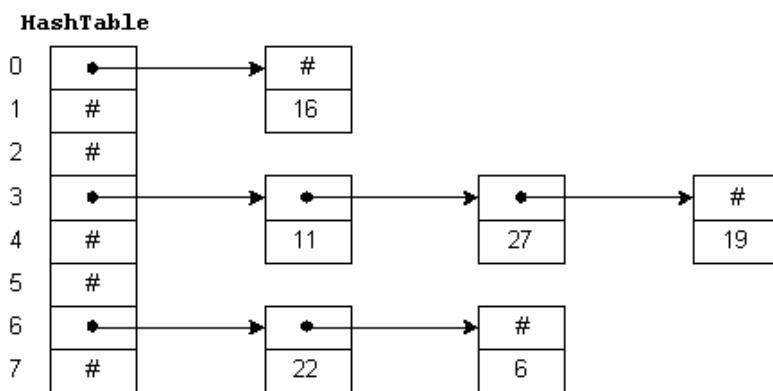
Referitor la prima condiție de mai sus, trebuie să observăm faptul că ea permite existența unor obiecte diferite din punct de vedere al conținutului, dar care au asociate aceeași valoare de dispersie (i.e., dacă `obiect_1 != obiect_2` atunci este posibil ca `hash(obiect_1) == hash(obiect_2)`!). În acest caz, spunem că funcția de dispersie are *coliziuni*. O funcție de dispersie ideală ar trebui să nu aibă coliziuni, dar practic acest lucru este imposibil din cauza *principiului lui Dirichlet*: "*Indiferent de modul în care vom plasa $n + 1$ obiecte în n cutii, va exista cel puțin o cutie care va conține două obiecte*". Astfel, considerând faptul că o funcție de dispersie furnizează valori de dispersie pe 32 biți, rezultă că acestea vor fi numere întregi cuprinse între -2.147.483.648 și 2.147.483.647, deci vor exista 4.294.967.294 valori distincte posibile pentru valorile de dispersie generate de funcția respectivă, ceea ce înseamnă că după aplicarea funcției de dispersie asupra a 4.294.967.294 + 1 obiecte distincte se va obține sigur cel puțin o coliziune!

Referitor la a doua condiție de mai sus, trebuie să observăm faptul că ea restricționează implementarea unei funcții de dispersie doar pentru obiecte imutabile (i.e., al căror conținut nu mai poate fi modificat după ce au fost create)!

Folosind funcții de dispersie se pot crea *tabele de dispersie (hash tables)*, care sunt structuri de date foarte eficiente din punct de vedere al operațiilor de inserare, căutare și ștergere, acestea având, în general, o complexitate computațională medie constantă.

Practic, o tabelă de dispersie este o structură de date asociativă în care unui obiect i se asociază un index (numit și *cheia obiectului*) într-un tablou unidimensional, calculat pe baza valorii de dispersie asociată obiectului respectiv prin intermediul unei funcții de dispersie.

În cazul unei funcții de dispersie ideale (i.e., care nu are coliziuni), fiecărui index din tablou îi va corespunde un singur obiect, dar, în realitate, din cauza, coliziunilor, vor exista mai multe obiecte asociate aceluiași index, care vor fi memorate folosind diverse structuri de date (e.g., liste simplu sau dublu înălanțuite). De exemplu, să considerăm numerele 16, 11, 27, 22, 19 și 6, precum și funcția de dispersie $h(x) = x \% 8$ corespunzătoare unei tabele de dispersie cu 8 elemente. Pe baza valorilor de dispersie, cele 6 numere vor fi distribuite în tabelă de dispersie astfel (sursa imaginii: <https://howtodoinjava.com/java/collections/hashtable-class/>):



Observați faptul că valorile dintr-o listă sunt, de fapt, coliziuni ale funcției de dispersie: $h(11) = h(27) = h(19) = 3!$

Încheiem prin a preciza faptul că performanțele unei tabele de dispersie depind de mai mulți factori (e.g., dimensiunea tabelei, funcția de dispersie utilizată, modalitatea de rezolvare a coliziunilor etc.), dar, în general, operațiile de inserare, căutare și ștergere se vor efectua cu complexitatea medie $\mathcal{O}(1)$. Mai multe detalii referitoare la tabelele de dispersie puteți să găsiți aici: https://itlectures.ro/wp-content/uploads/2020/04/SDA_curs6_hashTables_new.pdf.

Mulțimi

O *mulțime* este o colecție mutabilă de valori fără duplicate. Valorile memorate într-o mulțime pot fi neomogene (i.e., pot fi de tipuri diferite de date), dar trebuie să fie imutabile, deoarece mulțimile sunt implementate intern folosind tabele de dispersie. Mulțimile nu sunt indexate și nu păstrează ordinea de inserare a elementelor. Mulțimile sunt instanțe ale clasei `set`.

O mulțime poate fi creată/initializată în mai multe moduri:

- folosind un sir de constante sau o secvență:

```
# multime vidă
s = set()
print(s)          # set()

# sir de constante numerice
s = {3, 2, 1, 1, 2, 3, 3, 4, 1}
print(s)          # {1, 2, 3, 4}

# listă de numere
s = set([4, 3, 2, 1, 4, 4, 3, 7, 1])
print(s)          # {1, 2, 3, 4, 7}

# sir de caractere
s = set("testare")
print(s)          # {'s', 'a', 't', 'r', 'e'}
```

- folosind secvențe de initializare:

```
# resturi distințe
s = {x % 2 for x in range(100)}
print(s)          # {0, 1}

# prenume distințe
lista = ["Ana", "ION", "ANA", "ana", "George", "Ion", "ION"]
s = set(nume.upper() for nume in lista)
print(s)          # {'GEORGE', 'ANA', 'ION'}

# grupe distințe
lista = [("Popa Anca", 134), ("Popescu Ion", 131),
         ("Ion Mihai", 134), ("Georgescu Ana", 133),
         ("Radu Ileana", 131), ("Pop Ion", 131)]
s = set(t[1] for t in lista)
print(s)          # {131, 133, 134}
```

Accesarea elementelor unei mulțimi

Deoarece elementele unei mulțimi nu sunt indexate, acestea pot fi accesate doar printr-o parcurgere secvențială:

```
s = {1, 2, 3, 2, 2, 4, 1, 1, 7}
for x in s:
    print(x, end=" ")    # 1 2 3 4 7
```

Operatori pentru mulțimi

În limbajul Python sunt definiți următorii operatori pentru manipularea mulțimilor:

- a) *operatorii pentru testarea apartenenței: in, not in*

Exemplu: expresia 3 in {2, 3, 4, 3, 5, 2} va avea valoarea True

- b) *operatorii relaționali: <, <=, >, >=, ==, !=*

Observații:

- În cazul primilor 4 operatori, cele două mulțimi vor fi comparate din punct de vedere al relației matematice de inclusiune (submulțime / supermulțime)!
- În cazul ultimilor 2 operatori, nu se va ține cont de ordinea elementelor din cele două mulțimi comparate, ci doar de valorile lor!

Exemple:

```
S1 = {1, 2, 3, 4, 5, 100}
S2 = {1, 2, 4}
print(S2 < S1)                  # True
print(S1 >= S2)                 # True

S3 = {4, 1, 2, 1, 4, 4}
print(S3 < S2)                  # False
print(S3 <= S2)                 # True
print(S3 == S2)                  # True
```

- c) *Operatorii pentru operații specifice mulțimilor: | (reuniune), & (intersecție), - (diferență), ^ (diferență simetrică, i.e. A Δ B = (A\B) ∪ (B\A))*

Exemple:

```
A = {1, 3, 4, 7}
B = {1, 2, 3, 4, 6, 8}

print("Reuniunea:", A | B)          # {1, 2, 3, 4, 6, 7, 8}
print("Intersecția:", A & B)        # {1, 3, 4}
print("Diferența A\B:", A - B)      # {7}
print("Diferența B\A:", B - A)      # {8, 2, 6}
print("Diferența simetrică:", A ^ B) # {2, 6, 7, 8}
```

Funcții predefinite pentru mulțimi

În limbajul Python sunt predefinite mai multe funcții (*built-in functions* – <https://docs.python.org/3/library/functions.html>), dintre care unele pot fi utilizate pentru mai multe tipuri de date. De exemplu, funcția `len(secvență)` va furniza numărul de elemente dintr-o secvență (iterabil), indiferent dacă aceasta este o listă, un tuplu, un sir de caractere sau o mulțime.

Funcțiile predefinite care se pot utiliza pentru mulțimi sunt următoarele:

- a) **len(mulțime)**: furnizează numărul de elemente din mulțime (cardinalul mulțimii)

Exemplu: `len({2, 1, 3, 3, 2, 1, 7, 3}) = 4`

- b) **set(secvență)**: furnizează o mulțime formată din elementele secvenței respective

Exemplu: `set("teste") = {'e', 't', 's'}`

- c) **min(mulțime) / max(mulțime)**: furnizează elementul minim/maxim în sens lexicografic din mulțimea respectivă (atenție, toate elementele mulțimii trebuie să fie comparabile între ele, altfel va fi generată o eroare!)

Exemplu:

```
S = {100, -70, 16, 100, -85, 100, -70, 28}
print("Minimul din mulțime:", min(S))      # -85
print("Maximul din mulțime:", max(S))      # 101
print()

S = {(2, 10), (2, 1, 2), (60, 2, 1), (3, 140, 5)}
print("Minimul din mulțime:", min(S))      # (2, 1, 2)
print("Maximul din mulțime:", max(S))      # (60, 2, 1)

S = {20, -30, "101", 17, 100}
print("Minimul din mulțime:", min(S))
# TypeError: '<' not supported between
# instances of 'str' and 'int'
```

- d) **sum(mulțime)**: furnizează suma elementelor unei mulțimi (evident, toate elementele mulțimii trebuie să fie de tip numeric)

Exemplu: `sum({1, -7, 10, -8, 10, -7}) = -4`

- e) **sorted(mulțime, [reverse=False])**: furnizează o listă formată din elementele mulțimii respective sortate crescător (mulțimea nu va fi modificată!).

Exemplu: `sorted({1, -7, 1, -8, 1, -7}) = [-8, -7, 1]`

Elementele mulțimii pot fi sortate și descrescător, setând parametrul optional `reverse` al funcției `sorted` la valoarea `True`.

Exemplu: `sorted({1, -7, 1, -8}, reverse=True) = [1, -7, -8]`

Metode pentru prelucrarea mulțimilor

Metodele pentru prelucrarea mulțimilor sunt, de fapt, metodele încapsulate în clasa `set`. Așa cum am precizat anterior, mulțimile sunt *mutable*, deci metodele respective pot modifica mulțimea curentă, dacă acest lucru este necesar!

În continuare, vom prezenta mai multe metode pentru prelucrarea mulțimilor, cu observația că parametrii scriși între paranteze drepte sunt opționali:

- a) **`add(valoare)`**: dacă valoarea nu există deja în mulțime, atunci o adaugă.

Exemplu:

```
S = {1, 3, 5, 7, 9}
print(S)      # {1, 3, 5, 7, 9}

S.add(4)
print(S)      # {1, 3, 4, 5, 7, 9}

S.add(4)
print(S)      # {1, 3, 4, 5, 7, 9}
```

- b) **`update(secvență)`**: adaugă, pe rând, toate elementele din secvență dată în mulțime.

Exemplu:

```
S = {1, 2, 3}
S.add((2, 3, 4, 5, 4, 5))
print(S)      # {1, 2, 3, (2, 3, 4, 5, 4, 5)}

S = {1, 2, 3}
S.update((2, 3, 4, 5, 4, 5))
print(S)      # {1, 2, 3, 4, 5}

S = {1, 2, 3}
S.update([4, 2, 3, 4, (4, 5)])
print(S)      # {1, 2, 3, 4, (4, 5)}
```

- c) **`remove(valoare)`**: șterge din mulțimea curentă valoarea indicată sau lansează o eroare (`KeyError`) dacă valoarea nu apare în mulțime.

Exemplu:

Pentru a evita apariția erorii `KeyError`, mai întâi am verificat faptul că valoarea `x` pe care dorim să o ștergem se găsește în mulțime:

```
S = {3, 2, 1, 1, 2, 3, 3}
print(f"Multimea: {S}")

x = 30
if x in S:
    S.remove(x)
    print(f"Multimea după stergerea valorii {x}: {S}!")
```

```

else:
    print(f"Valoarea {x} nu apare in multime!")

```

O altă modalitate de utilizare a metodei `remove`, mai eficientă, constă în tratarea erorii care poate să apară când valoarea `x` pe care dorim să o ștergem nu se găsește în mulțime:

```

S = {3, 2, 1, 1, 2, 3, 3}
print(f"Multimea: {S}")

x = 30
try:
    S.remove(x)
    print(f"Multimea după stergerea valorii {x}: {S}!")
except KeyError:
    print(f"Valoarea {x} nu apare în multime!")

```

- d) **`discard(valoare)`**: șterge din mulțimea curentă valoarea indicată, dacă aceasta se găsește în mulțime, altfel mulțimea nu va fi modificată.

Exemplu:

```

S = {3, 2, 1, 1, 2, 3, 3}
print(f"Multimea: {S}")

x = 30

S.discard(x)
print(f"Multimea după stergerea valorii {x}: {S}!")

```

- e) **`clear()`**: șterge toate elementele din mulțime.

Exemplu:

```

S = {3, 2, 1, 1, 2, 3, 3}
print(f"Multimea: {S}")      # Multimea: {1, 2, 3}

S.clear()
print(f"Multimea: {S}")      # Multimea: set()

```

- f) **`union(secvență)`, `intersection(secvență)`, `difference(secvență)`, `symmetric_difference(secvență)`**: furnizează mulțimea obținută prin aplicarea operației respective asupra mulțimii curente și secvenței respective (convertită automat în mulțime), fără a modifica mulțimea curentă!

Exemplu:

```

A = {1, 3, 4, 7}
B = [1, 2, 3, 4, 6, 8]

print("Reuniunea:", A.union(B))      # {1, 2, 3, 4, 6, 7, 8}
print("Intersecția:", A.intersection(B))  # {1, 3, 4}

```

```

print("Diferența A\B:", A.difference(B))      # {7}
print("Diferența simetrică:", A.symmetric_difference(B))    # {2, 6, 7, 8}
print("Multimea A: ", A)      # {1, 3, 4, 7}
print("Lista B:", B)          # [1, 2, 3, 4, 6, 8]

```

- g) **intersection_update(secvență), difference_update(secvență), symmetric_difference_update(secvență)**: înlocuiește mulțimea curentă cu mulțimea obținută prin aplicarea operației respective asupra mulțimii curente și secvenței respective (convertită automat în mulțime)!

Exemplu:

```

A = {1, 3, 4, 7}
B = [1, 2, 3, 4, 6, 8]

A.intersection_update(B)
print("Multimea A:", A)      # {1, 3, 4}

A.update([7, 5, 7, 4, 3])
print("Multimea A:", A)      # {1, 3, 4, 5, 7}

A.difference_update(B)
print("Multimea A:", A)      # {5, 7}

A.symmetric_difference_update(B)
print("Multimea A:", A)      # {1, 5, 2, 7, 3, 4, 6, 8}

```

Crearea unei mulțimi

O mulțime poate fi creată folosind valori constante, secvențe de valori, valori citite de la tastatură sau valori citite dintr-un fișier. Indiferent de sursa elementelor utilizate pentru crearea mulțimii, există mai multe variante de implementare pe care le putem utiliza: secvențe de inițializare, adăugarea unui element folosind metoda `add` sau operatorul `+=` sau reunirea la mulțimea curentă a unei mulțimi formate doar din elementul pe care dorim să-l adăugăm. În continuare, vom testa toate aceste variante din punct de vedere al timpului de executare, creând, de fiecare dată, o mulțime formată cu 200000 de elemente, respectiv numerele 0, 1, 2, ..., 199999:

```

import time

nr_elemente = 200000

start = time.time()
multime = set(x for x in range(nr_elemente))
stop = time.time()
print("  Initializare: ", stop - start, "secunde")

```

```

start = time.time()
multime = set()
for x in range(nr_elemente):
    multime.add(x)
stop = time.time()
print("  Metoda add(): ", stop - start, "secunde")

start = time.time()
multime = set()
for x in range(nr_elemente):
    multime |= {x}
stop = time.time()
print("  Operatorul |=: ", stop - start, "secunde")

start = time.time()
multime = set()
for x in range(nr_elemente):
    multime = multime | {x}
stop = time.time()
print("  Operatorul |: ", stop - start, "secunde")

```

Rezultatele obținute sunt următoarele:

```

Initializare: 0.015614748001098633 secunde
Metoda add(): 0.015626907348632812 secunde
Operatorul |=: 0.03124070167541504 secunde
Operatorul |: 408.1307489871979 secunde

```

Se observă faptul că primele 3 variante au tempi de executare mici, aproximativi egali, iar ultima variantă are un timp de executare mult mai mare din cauza faptului că la fiecare operație de reunire a mulțimii $\{x\}$ la mulțimea curentă se creează în memorie o copie a mulțimii curente, se reuneste la copie nouă valoarea x și apoi referința mulțimii curente se înlocuiește cu referința copiei.

Dicționare

Un *dicționar* este o colecție de date asociativă (*tablou asociativ*), deci permite asocierea unei valori arbitrară cu o cheie unică. Astfel, accesarea unei valori dintr-un dicționar nu se realizează prin poziția sa în tablou (index), ci prin cheia sa. Practic, putem privi un dicționar ca fiind o colecție de perechi de forma *cheie: valoare*. Cheile unui dicționar trebuie să fie unice și imutabile, dar pentru valori nu există nicio restricție. Începând cu versiunea Python 3.7, dicționarele păstrează ordinea de inserare a cheilor. Dicționarele sunt instanțe ale clasei `dict`.

Un dicționar poate fi creat/initializat în mai multe moduri:

- folosind constante, operații de inserare sau funcția `dict`:

```
# dictionar vid - {}
d = {}
print(d)

# dictionar cu chei neomogene, dar imutabile
d = {"A": 4, "B": "Popa Ion", 6: -100,
      (1, 2, 3): [100, 200, 300]}
print(d)

# inserarea unor perechi cheie: valoare într-un dicționar
d = {}
d["A"] = 4
d["B"] = "Popa Ion"
d[6] = -100
d[(1, 2, 3)] = [100, 200, 300]
print(d)

# preluarea perechilor cheie: valoare dintr-o listă
# de tupluri de forma (cheie, valoare)
d = dict([('A', 4), ('B', "Popa Ion"), (6, -100),
          ((1, 2, 3), [100, 200, 300])])
print(d)
```

- folosind secvențe de initializare:

```
# dictionar cu perechi literă: cod ASCII
d = {chr(65 + x): 65+x for x in range(5)}
print(d)      # {'A': 65, 'B': 66, 'C': 67, 'D': 68, 'E': 69}

# dicționar cu frecvențele literelor unui cuvânt,
# respectiv perechi literă: frecvență
cuv = "testare"
d = {lit: cuv.count(lit) for lit in set(cuv)}
print(d)      # {'a': 1, 't': 2, 's': 1, 'e': 2, 'r': 1}

# dictionar cu perechi număr: suma cifrelor
lista = [2134, 456, 777, 8144, 9]
d = {x: sum([int(c) for c in str(x)]) for x in lista}
print(d)      # {2134: 10, 456: 15, 777: 21, 8144: 17, 9: 9}
```

Accesarea elementelor unui dicționar

Elementele unui dicționar sunt indexate prin cheile asociate, deci pot fi accesate doar prin intermediul cheilor respective, ci nu prin pozițiile lor:

```
# dicționar cu perechi literă: cod ASCII
d = {chr(65 + x): x+65 for x in range(5)}
print(d)          # {'A': 65, 'B': 66, 'C': 67, 'D': 68, 'E': 69}

# modificare valorii asociate unei chei (i.e., cheia "B")
d["B"] = 10
print(d)          # {'A': 65, 'B': 10, 'C': 67, 'D': 68, 'E': 69}

# ștergere unei chei (i.e., cheia "B")
del d["B"]
print(d)          # {'A': 65, 'C': 67, 'D': 68, 'E': 69}

# inserarea unei chei noi și a unei valori asociate
# (i.e., noii chei "K" i se asociază valoarea 7)
d["K"] = 7
print(d)          # {'A': 65, 'C': 67, 'D': 68, 'E': 69, 'K': 7}
```

Dacă într-un dicționar se încearcă accesarea unei chei nonexistente, atunci va fi lansată o eroare de tipul `KeyError`:

```
# dicționar cu perechi literă: cod ASCII
d = {chr(65 + x): x+65 for x in range(5)}
print(d)          # {'A': 65, 'B': 66, 'C': 67, 'D': 68, 'E': 69}

k = "Z"
print(d[k])      # KeyError: 'Z'
```

Pentru a evita apariția erorii precizate anterior, fie putem să testăm mai întâi existența cheii respective în dicționar, fie să tratăm excepția `KeyError`:

```
# dicționar cu perechi literă: cod ASCII
# d = {'A': 65, 'B': 66, 'C': 67, 'D': 68, 'E': 69}
d = {chr(65 + x): x+65 for x in range(5)}

k = "A"
if k in d:
    print(f"d['{k}'] = {d[k]}")
else:
    print(f"Cheia {k} nu apare în dicționar!")

k = "Z"
try:
    print(f"d['{k}'] = {d[k]}")
except KeyError:
    print(f"Cheia {k} nu apare în dicționar!")
```

Dicționarele pot fi utilizate pentru a organiza eficient anumite informații din punct de vedere al complexității computaționale a operațiilor de inserare, accesare sau ștergere (i.e., o complexitate medie egală cu $\mathcal{O}(1)$). Astfel, printr-o alegere judicioasă a cheilor și valorilor asociate lor, se pot optimiza procesele de actualizare sau interogare a unor date.

De exemplu, să considerăm următoarele informații despre $n = 4$ studenți (numele, grupa și media), memorate într-o listă de tupluri:

```
L = [ ("Marinescu Ioana", 152, 9.85),
      ("Constantinescu Ion", 151, 7.70),
      ("Popescu Ion", 151, 9.70),
      ("Filip Anca", 152, 9.70)]
```

Evident, aproape orice operație de actualizare sau interogare a acestor informații (e.g., afișarea sau modificarea informațiilor referitoare la un student, afișarea studenților dintr-o anumită grupă, afișarea studenților având o anumită medie etc.) se va realiza cu o complexitate maximă egală cu $\mathcal{O}(n)$, deoarece ea va necesita parcurgerea întregii liste.

Dacă într-un program vom efectua multe operații de actualizare sau interogare pe baza numelor studenților, atunci putem să organizăm informațiile într-un dicționar cu perechi de forma **nume**: (**grupă**, **medie**):

```
L = [ ("Marinescu Ioana", 152, 9.85),
      ("Constantinescu Ion", 151, 7.70),
      ("Popescu Ion", 151, 9.70),
      ("Filip Anca", 152, 9.70)]

dict_nume = {}
for t in L:
    dict_nume[t[0]] = (t[1], t[2])

print(f"Dictionar: {dict_nume}")

# dict_nume = {"Marinescu Ioana": (152, 9.85),
#               "Constantinescu Ion": (151, 7.70),
#               "Popescu Ion": (151, 9.70),
#               "Filip Anca": (152, 9.70)}

ns = "Popescu Ion"
print(f"{ns}: {dict_nume[ns]}")

ns = "Popescu Ion"
ms = 9.20
dict_nume[ns] = (dict_nume[ns][0], ms)
print(f"{ns}: {dict_nume[ns]}")

del dict_nume["Popescu Ion"]
print(f"Dictionar: {dict_nume}")
```

În acest caz, operațiile de actualizare și interogare se vor executa cu o complexitate medie mult mai bună, respectiv $\mathcal{O}(1)$. Atenție, această variantă de modelare este corectă doar în cazul în care nu există 2 studenți cu același nume! De ce?

Aceleași informații le putem organiza sub forma unui dicționar cu perechi de forma **grupa**: [lista studenților din grupă], dacă știm că vom efectua multe operații de actualizare sau interogare la nivel de grupă:

```
L = [ ("Marinescu Ioana", 152, 9.85),
      ("Constantinescu Ion", 151, 7.70),
      ("Popescu Ion", 151, 9.70),
      ("Filip Anca", 152, 9.70)]  
  
dict_grupe = {}  
for t in L:  
    if t[1] not in dict_grupe:  
        dict_grupe[t[1]] = [(t[0], t[2])]  
    else:  
        dict_grupe[t[1]].append((t[0], t[2]))  
  
print(f"Dictionar: {dict_grupe}")  
  
# dict_grupe = {152: [("Marinescu Ioana", 9.85),  
#                  ("Filip Anca", 9.70)],  
#               151: [("Constantinescu Ion", 7.70),  
#                      ("Popescu Ion", 9.70)]}  
  
print(dict_grupe[152])  
  
dict_grupe[152].append(("Mihai Carmen", 8.85))  
print(dict_grupe[152])
```

Cu toate acestea, operațiile de actualizare sau interogare a mediei unui student vor avea o complexitate maximă egală cu $\mathcal{O}(n)$, deoarece implică parcurgerea tuturor studenților din grupa studentului respectiv. În acest caz, putem să modificăm structura dicționarului, păstrând informațiile despre studenții dintr-o grupă nu într-o listă, ci într-un dicționar cu perechi de forma **nume: medie**:

```
L = [ ("Marinescu Ioana", 152, 9.85),
      ("Constantinescu Ion", 151, 7.70),
      ("Popescu Ion", 151, 9.70),
      ("Filip Anca", 152, 9.70)]  
  
dict_grupe = {}  
for t in L:  
    if t[1] not in dict_grupe:  
        dict_grupe[t[1]] = {t[0]: t[2]}  
    else:  
        dict_grupe[t[1]][t[0]] = t[2]
```

```
# dict_grupe = {152: {"Marinescu Ioana": 9.85,
#                      "Filip Anca": 9.70},
#               151: {"Constantinescu Ion": 7.70,
#                      "Popescu Ion": 9.70} }

print(dict_grupe[152]["Filip Anca"])

print(dict_grupe[152])

dict_grupe[152]["Mihai Carmen"] = 8.85
print(dict_grupe[152])
```

Astfel, operațiile de actualizare și interogare a notei unui student se vor executa cu o complexitate medie mult mai bună, respectiv $\mathcal{O}(1)$. Atenție, această variantă de modelare este corectă doar în cazul în care nu există 2 studenți cu același nume în aceeași grupă!

Dacă în programul nostru vom efectua multe operații de actualizare sau interogare în funcție de mediile studenților (de exemplu, pentru cazare sau acordarea burselor), atunci vom organiza informațiile sub forma unui dicționar cu perechi de forma **medie: [lista cu tupluri (nume, grupă)]**:

```
L = [ ("Marinescu Ioana", 152, 9.85),
      ("Constantinescu Ion", 151, 7.70),
      ("Popescu Ion", 151, 9.70),
      ("Filip Anca", 152, 9.70)]

dict_medii = {}
for t in L:
    if t[2] not in dict_medii:
        dict_medii[t[2]] = [(t[0], t[1])]
    else:
        dict_medii[t[2]].append((t[0], t[1]))

# dict_medii = {9.85: [ ("Marinescu Ioana", 152)],
#                 9.70: [ ("Filip Anca", 152), ("Popescu Ion",
# 151)],
#                 7.70: [ ("Constantinescu Ion", 152)]}

m = 9.70
print(f"Studentii cu media {m}: {dict_medii[m]}")
```

Operatori pentru dicționare

În limbajul Python sunt definiți următorii operatori pentru manipularea dicționarelor:

a) *operatorii pentru testarea apartenenței la nivel de cheie: in, not in*

Exemplu: expresia 'B' in {'A': 10, 'B': 7, 'C': 4, 'D': 7} va avea valoarea True, iar 7 in {'A': 10, 'B': 7, 'C': 4, 'D': 7} va avea False

- b) operatorii relationali: ==, != (două dicționare sunt egale dacă sunt formate din aceleași perechi cheie: valoare, indiferent de ordinea de inserare)

Exemplu:

```
d1 = {'A': 10, 'B': 7, 'C': 4, 'D': 7}
d2 = {'D': 7, 'A': 10, 'C': 4, 'B': 7}
d3 = {'A': 10, 'B': 17, 'C': 4, 'D': 7}
print(d1 == d2)      # True
print(d1 == d3)      # False
print(d2 != d3)      # True
```

Funcții predefinite pentru dicționare

În limbajul Python sunt predefinite mai multe funcții (*built-in functions* – <https://docs.python.org/3/library/functions.html>), dintre care unele pot fi utilizate pentru mai multe tipuri de date. De exemplu, funcția `len` furnizează numărul de elemente dintr-o secvență (iterabil), indiferent dacă aceasta este o listă, un tuplu, un sir de caractere sau o mulțime, dar și numărul cheilor dintr-un dicționar.

Funcțiile predefinite care se pot utiliza pentru dicționare sunt următoarele:

- a) **`len(dicționar)`**: furnizează numărul cheilor dicționarului

Exemplu: `len({'A': 10, 'B': 7, 'C': 4, 'D': 7}) = 4`

- b) **`dict(secvență)`**: furnizează un dicționar format din elementele secvenței respective, care trebuie să fie toate perechi (primul element al unei perechi este considerat o cheie, iar al doilea reprezintă valoarea asociată cheii respective)

Exemplu: `dict([('A',10), ('B',7), ('C',4), ('D',7)]) = {'A': 10, 'B': 7, 'C': 4, 'D': 7}`

- c) **`min(dicționar) / max(dicționar)`**: furnizează elementul minim/maxim în sens lexicografic din mulțimea respectivă (atenție, toate elementele mulțimii trebuie să fie comparabile între ele, altfel va fi generată o eroare!)

Exemplu:

```
d = {'E':20, 'D': 7, 'A': 10, 'C': 40, 'B': 7}
print(f"Cheia minima: '{min(d)}'")           # Cheia minima: 'A'
print(f"Cheia maxima: '{max(d)}'")           # Cheia maxima: 'E'
```

- d) **`sum(dicționar)`**: furnizează suma cheilor unui dicționar (evident, toate cheile trebuie să fie de tip numeric)

Exemplu: `sum({20: 'E', 7: 'B', 10: 'A', 40: 'C'}) = 77`

- e) **sorted(dicționar, [reverse=False])**: furnizează o listă formată din cheile dicționarului respectiv sortate crescător (dicționarul nu va fi modificat!).

Exemplu: `sorted({20: 'E', 7: 'B', 40: 'C'}) = [7, 20, 40]`

Elementele mulțimii pot fi sortate și descrescător, setând parametrul opțional `reverse` al funcției `sorted` la valoarea `True`.

Exemplu: `sorted({'E': 20, 'B': 7, 'C': 40}, reverse=True) = ['E', 'C', 'B']`

Metode pentru prelucrarea dicționarelor

Metodele pentru prelucrarea dicționarelor sunt, de fapt, metodele încapsulate în clasa `dict`. Așa cum am precizat anterior, dicționarele sunt *mutable*, deci metodele respective pot modifica dicționarul curent, dacă acest lucru este necesar!

În continuare, vom prezenta mai multe metode pentru prelucrarea dicționarelor, cu observația că parametrii scriși între paranteze drepte sunt opționali:

- a) **get(cheie, [valoare eroare])**: furnizează valoarea asociată cheii respective dacă aceasta există în dicționar sau `None` în caz contrar. Dacă se precizează pentru parametrul opțional `valoare eroare` o anumită valoare, aceasta va fi furnizată în cazul în care cheia nu există în dicționar.

Exemplu:

```
d = {'E':20, 'D': 7, 'A': 10, 'C': 40, 'B': 7}

print(d.get('A'))           # 10
print(d.get('Z'))           # None
print(d.get('Z', -1000))    # -1000
```

În general, se recomandă utilizarea metodei `get` în locul accesării directe a unei chei dintr-un dicționar pentru a evita eroarea (de tip `KeyError`) care poate să apară dacă respectiva cheie nu se găsește în dicționar:

```
d = {'E':20, 'D': 7, 'A': 10, 'C': 40, 'B': 7}

if d.get('Z') == 100:        # NU
    print("DA")
else:
    print("NU")

if d['Z'] == 100:           # KeyError: 'Z'
    print("DA")
else:
    print("NU")
```

b) **keys(), values(), items()**: furnizează vizualizări (*dictionary views*) ale cheilor, valorilor sau perechilor (**cheie**, **valoare**) din dicționarul respectiv. Vizualizările sunt iterabile și vor fi actualizate dinamic în momentul modificării dicționarului (<https://docs.python.org/3/library/stdtypes.html#dict-views>). Vizualizările pot fi transformate în liste folosind funcția **list**.

Exemplu:

```
d = {'E':20, 'D': 7, 'A': 10}

k = d.keys()
v = d.values()
p = d.items()

print("    Chei:", list(k))
print(" Valori:", list(v))
print("Perechi:", list(p))

d['K'] = 1000

print()
print("    Chei noi:", list(k))
print(" Valori noi:", list(v))
print("Perechi noi:", list(p))
```

c) **update(dicționar)**: actualizează dicționarul curent folosind perechile **cheie: valoare** din dicționarul transmis ca parametru, astfel: dacă o cheie există deja în dicționarul curent atunci îi actualizează valoarea asociată, altfel adaugă o nouă cheie cu valoarea respectivă în dicționarul curent.

Exemplu:

```
d = {'E':20, 'D': 7, 'A': 10}
t = {'D': -10, 'K': 5}

d.update(t)
print(d)          # {'E': 20, 'D': -10, 'A': 10, 'K': 5}
```

d) **pop(cheie, [valoare implicită])**: șterge din mulțimea curentă cheia indicată, dacă aceasta există în dicționar, și furnizează valoarea asociată cu ea. Dacă cheia indicată nu se găsește în dicționar și parametrul optional **valoare implicită** nu este setat, atunci va apărea o eroare, altfel metoda va furniza valoarea implicită setată.

Exemplu:

```
d = {'E':20, 'D': 7, 'A': 10}
print(f"Dicționarul: {d}\n")
k = 'D'
r = d.pop(k)
print(f"Valoarea asociată cheii {k}: {r}")
print(f"Dicționarul: {d}\n")
k = 'Z'
r = d.pop(k, None)
print(f"Valoarea asociată cheii {k}: {r}")
print(f"Dicționarul: {d}")
```

e) **clear()**: șterge toate elementele din dicționar.

Complexitatea computațională a operațiilor asociate colecțiilor de date

O implementare optimă a unui algoritm presupune, de obicei, și utilizarea unor structuri de date adecvate, astfel încât operațiile necesare să fie implementate cu o complexitate minimă. De exemplu, putem verifica dacă o valoare a fost deja utilizată într-un algoritm în mai multe moduri:

- memorăm toate valorile într-o listă sau un tuplu și testăm, folosind operatorul `in` sau metoda `count`, dacă valoarea respectivă se găsește într-o listă / un tuplu cu n elemente, deci vom avea o complexitate maximă egală cu $\mathcal{O}(n)$;
- dacă valorile sunt deja sortate, atunci putem să utilizăm căutarea binară pentru a testa dacă valoarea respectivă se găsește în lista / tuplul cu n elemente, deci vom avea o complexitate maximă egală cu $\mathcal{O}(\log_2 n)$;
- memorăm valorile într-o mulțime sau un dicționar și atunci putem să testăm existența valorii respective cu complexitate medie egală cu $\mathcal{O}(1)$.

În continuare, vom prezenta complexitatea computațională a operațiilor implementate de colecțiile din limbajul Python (<https://wiki.python.org/moin/TimeComplexity>):

a) *liste / tupluri* (cu n elemente)

Operație / metodă / funcție	Complexitate maximă
Accesarea unui element prin index	$\mathcal{O}(1)$
Ștergerea unui element (instrucțiunea <code>del</code> și metoda <code>remove</code>)	$\mathcal{O}(n)$
Parcursere	$\mathcal{O}(n)$
Căutarea unei valori (operatorii <code>in</code> și <code>not in</code> sau metoda <code>index</code>)	$\mathcal{O}(n)$
<code>len(listă sau tuplu)</code>	$\mathcal{O}(1)$
<code>append(valoare)</code>	$\mathcal{O}(1)$
<code>extend(secvență)</code>	$\mathcal{O}(\text{len(secvență)})$
<code>pop()</code>	$\mathcal{O}(1)$
<code>pop(poziție)</code>	$\mathcal{O}(n)$
<code>count(valoare)</code>	$\mathcal{O}(n)$
<code>insert(poziție, valoare)</code>	$\mathcal{O}(n)$
Extragerea unei secvențe	$\mathcal{O}(\text{len(secvență)})$
Ștergerea unei secvențe	$\mathcal{O}(n)$
Modificarea unei secvențe	$\mathcal{O}(\text{len(secvență)} + n)$
Sortare (funcția <code>sorted</code> și metoda <code>sort</code>)	$\mathcal{O}(n \log_2 n)$
Funcțiile <code>min</code> , <code>max</code> și <code>sum</code>	$\mathcal{O}(n)$
Copiere (metoda <code>copy</code>)	$\mathcal{O}(n)$
Multiplicare de k ori (operatorul <code>*</code>)	$\mathcal{O}(nk)$

b) *mulțimi* (cu n elemente)

Operație / metodă / funcție	Complexitate medie	Complexitate maximă
Testarea apartenenței unei valori (operatorii <code>in</code> și <code>not in</code>)	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Adăugarea unui element (metoda <code>add</code>)	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Ștergerea unui element (metodele <code>remove</code> și <code>discard</code>)	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Creare	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$
Parcursere		$\mathcal{O}(n)$
<code>len(mulțime)</code>		$\mathcal{O}(1)$
<code>update(secvență)</code>	$\mathcal{O}(\text{len(secvență)})$	$\mathcal{O}(n * \text{len(secvență)})$
Reuniunea mulțimilor A și B (operatorul <code> </code> și metoda <code>union</code>)	$\mathcal{O}(\text{len}(A) + \text{len}(B))$	
Intersecția mulțimilor A și B (operatorul <code>&</code> , metoda <code>intersection</code> și metoda <code>intersection_update</code>)	$\mathcal{O}(\min(\text{len}(A), \text{len}(B)))$	$\mathcal{O}(\text{len}(A) * \text{len}(B))$
Diferența mulțimilor A și B (operatorul <code>-</code> și metoda <code>difference</code>)	$\mathcal{O}(\text{len}(A))$	
Diferența mulțimilor A și B (metoda <code>difference_update</code>)	$\mathcal{O}(\text{len}(B))$	
Diferența simetrică a mulțimilor A și B (operatorul <code>^</code> și metoda <code>symmetric_difference</code>)	$\mathcal{O}(\text{len}(A))$	$\mathcal{O}(\text{len}(A) * \text{len}(B))$
Diferența simetrică a mulțimilor A și B (metoda <code>symmetric_difference_update</code>)	$\mathcal{O}(\text{len}(B))$	$\mathcal{O}(\text{len}(A) * \text{len}(B))$
Sortare (funcția <code>sorted</code>)		$\mathcal{O}(n \log_2 n)$
Funcțiile <code>min</code> , <code>max</code> și <code>sum</code>		$\mathcal{O}(n)$

c) *dicționare* (cu n chei)

Operație / metodă / funcție	Complexitate medie	Complexitate maximă
Testarea apartenenței unei chei (operatorii <code>in</code> și <code>not in</code>)	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Accesarea unui element prin cheie	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Ștergerea unui element (instrucțiunea <code>del</code> și metoda <code>pop</code>)	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Creare	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$
Parcursere		$\mathcal{O}(n)$

În continuare, vom determina, folosind diverse colecții de date, frecvențele cuvintelor dintr-o propoziție formată din n cuvinte și citită din fișierul text *text.txt*:

- a) determinăm mulțimea cuvintelor distincte din propoziție și apoi calculăm frecvența fiecărui cuvânt distinct:

```
f = open("text.txt")
prop = f.read()
f.close()

# înlocuim semnele de punctuație cu spații
for sep in ".,:;?!":
    prop = prop.replace(sep, " ")
cuvinte = prop.split()

# calculăm frecvențele cuvintelor distincte
for cuv in set(cuvinte):
    frcuv = cuvinte.count(cuv)
    print(f"Cuvantul {cuv} apare de {frcuv} ori")
```

Această variantă are complexitatea maximă $\mathcal{O}(n^2)$, deoarece pot exista maxim n cuvinte distincte în propoziție, iar metoda `count` are complexitatea $\mathcal{O}(n)$.

- b) calculăm frecvența fiecărui cuvânt distinct din propoziție folosind un dicționar cu perechi de forma *cuvânt:frecvență*:

```
f = open("text.txt")
prop = f.read()
f.close()

# înlocuim semnele de punctuație cu spații
for sep in ".,:;?!":
    prop = prop.replace(sep, " ")
cuvinte = prop.split()

# inițializăm dicționarul cu cuvintele distincte
# din propoziție, fiecare cuvânt cu frecvența 0
frcuv = {cuv: 0 for cuv in set(cuvinte)}

# calculăm frecvența fiecărui cuvânt distinct
for cuv in cuvinte:
    frcuv[cuv] += 1

# afișăm frecvențele cuvintelor distincte
for cuv in frcuv:
    print(f"Cuvantul {cuv} apare de {frcuv[cuv]} ori")
```

Această variantă de rezolvare are complexitatea maximă $\mathcal{O}(n)$, deoarece crearea și actualizarea dicționarului `frcuv` au, fiecare, complexitatea $\mathcal{O}(n)$.

CURS 6

Fișiere text

Un *fișier text* este o secvență de caractere organizată pe linii și stocată în memoria externă (SSD, HDD, DVD, CD etc.). Practic, fișierele text reprezintă o modalitate foarte simplă prin care se poate asigura *persistența datelor*.

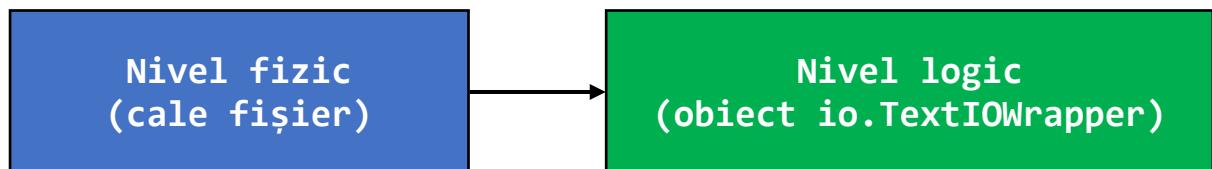
Liniile dintr-un fișier text sunt demarcate folosind caracterele **CR** (*carriage return* sau '\r' sau `chr(13)`) și **LF** (*line feed* sau '\n' sau `chr(10)`), astfel:

- în sistemele de operare *Windows* și *MS-DOS* se utilizează combinația **CR+LF**;
- în sistemele de operare de tip *Unix/Linux* și începând cu versiunea *Mac OS X* se utilizează doar caracterul **LF**;
- în sistemele de operare până la versiunea *Mac OS X* se utilizează doar caracterul **CR**.

Indiferent de modalitatea utilizată pentru demarcarea liniilor într-un anumit sistem de operare, în momentul în care se citesc liniile dintr-un fișier text, caracterele utilizate pentru demarcare vor fi transformate automat într-un singur caracter LF ('\'n')!

Deschiderea unui fișier text

Operația de deschidere a unui fișier (text sau binar!) presupune asocierea fișierului, considerat la nivel fizic (i.e., identificat prin calea sa), cu o variabilă/un obiect dintr-un program, ceea ce va permite manipularea sa la nivel logic (i.e., prin intermediul unor funcții sau metode dedicate):



Practic, după deschiderea unui fișier, obiectul asociat fișierului va conține mai multe informații despre starea fișierul respectiv (dimensiunea în octeți, modalitatea de codificare utilizată, poziția curentă a pointerului de fișier etc.), iar programatorul va putea acționa asupra fișierului într-un mod transparent, folosind metodele puse la dispoziție de obiectul respectiv.

Deschiderea unui fișier se realizează folosind următoarea funcție (reamintim faptul că parantezele drepte indică faptul că parametrul este optional):

```
open("cale fișier", ["mod de deschidere"])
```

Primul parametru al funcției reprezintă calea fișierului, sub forma unui sir de caractere, iar în cazul în care fișierul se află în directorul curent se poate indica doar numele său. Modul de deschidere este utilizat pentru a preciza cum va fi prelucrat fișierul text și se indică prin următoarele litere:

- "r" pentru *citire din fișier* (`read`)
 - fișierul va putea fi utilizat doar pentru a citi date din el;
 - este modul implicit de deschidere a unui fișier text, i.e. se va folosi dacă se omite parametrul "mod de deschidere" în apelul funcției `open`;
 - dacă fișierul indicat prin calea respectivă nu există, atunci se va genera eroarea `FileNotFoundException`;
- "w" pentru *scriere în fișier* (`write`)
 - fișierul va putea fi utilizat doar pentru a scrie date în el;
 - dacă fișierul indicat prin calea respectivă nu există, atunci se va crea unul nou, altfel fișierul existent va fi șters automat și va fi înlocuit cu unul nou;
- "x" pentru *scriere în fișier creat în mod exclusiv* (`exclusive write`)
 - fișierul va putea fi utilizat doar pentru a scrie date în el;
 - dacă fișierul indicat prin calea respectivă nu există, atunci se va crea unul nou, altfel se va genera eroarea `FileExistsError`;
- "a" pentru *adăugare în fișier* (`append`)
 - fișierul va putea fi utilizat doar pentru a scrie informații în el;
 - dacă fișierul există, atunci noile datele se vor scrie imediat după ultimul său caracter, altfel se va crea un fișier nou și datele se vor scrie de la începutul său.

Pentru a trata excepțiile care pot să apară în momentul deschiderii unui fișier, se va folosi o instrucțiune de tipul `try ... except`, aşa cum se poate observa în exemplele următoare:

```
try:
    f = open("exemplu.txt")
    print("Fișier deschis cu succes!")
except FileNotFoundError:
    print("Fișier inexistent!")
```

```
try:
    f = open("C:\Test Python\exemplu.txt", "x")
except FileNotFoundError:
    print("Calea fișierului este greșită!")
except FileExistsError:
    print("Fișierul deja există!!")
```

Din cel de-al doilea exemplu se poate observa faptul că eroarea `FileNotFoundException` va fi generată și în cazul modurilor de deschidere pentru scriere (i.e., modurile "w", "x" și "a") în cazul în care calea fișierului este greșită!

Citirea datelor dintr-un fișier text

În limbajul Python, datele citite dintr-un fișier text vor fi furnizate întotdeauna sub forma unor șiruri de caractere!

Pentru citirea datelor dintr-un fișier text sunt definite următoarele metode:

- **read():** furnizează tot conținutul fișierului text într-un singur șir de caractere

Exemplu:

Program	Fișier de intrare	Ecran
<pre>f = open("exemplu.txt") s = f.read() print(s) f.close()</pre>	Ana are multe pere, mere rosii, si mai multe prune!!!	Ana are multe pere, mere rosii, si mai multe prune!!!

Observați faptul că șirul s conține și caracterele LF ('\n') folosite pentru delimitarea liniilor!

- **readline():** furnizează conținutul liniei curente din fișierul text într-un șir de caractere sau un șir vid când se ajunge la sfârșitul fișierului

Exemplu:

Program	Fișier de intrare	Ecran
<pre>f = open("exemplu.txt") s = f.readline() while s != "": print(s, end="") s = f.readline() f.close()</pre>	Ana are multe pere, mere rosii, si mai multe prune!!!	Ana are multe pere, mere rosii, si mai multe prune!!!

Observați faptul că, de fiecare dată, șirul s conține și caracterul LF ('\n') folosit pentru delimitarea liniilor, mai puțin în cazul ultimei linii!

O modalitate echivalentă de parcursere a unui fișier text linie cu linie constă în iterarea directă a obiectului f asociat:

Exemplu:

Program	Fișier de intrare	Ecran
<pre>f = open("exemplu.txt") for linie in f: print(linie, end="") f.close()</pre>	Ana are multe pere, mere rosii, si mai multe prune!!!	Ana are multe pere, mere rosii, si mai multe prune!!!

- **readlines()**: furnizează toate liniile din fișierul text într-o listă de siruri de caractere

Exemplu:

Program	Fișier de intrare	Ecran
<pre>f = open("exemplu.txt") s = f.readlines() print(s) f.close()</pre>	Ana are multe pere, mere rosii, si mai multe prune!!!	['Ana are multe pere,\n', '\n', 'mere rosii,\n', 'si mai multe prune!!!']

Observați faptul că fiecare sir de caractere din listă, corespunzător unei linii din fișier, conține și caracterul LF ('\n') folosit pentru delimitarea liniilor, mai puțin în cazul ultimei linii!

Cele 3 modalități de citire a datelor dintr-un fișier text sunt echivalente, dar, în cazul în care dimensiunea fișierului este mare, se preferă citirea linie cu linie a conținutului său, deoarece necesită mai puțină memorie.

Deoarece toate metodele folosite pentru citirea datelor dintr-un fișier text furnizează siruri de caractere, dacă vrem să citim datele respective sub forma unor numere trebuie să utilizăm funcții pentru manipularea sirurilor de caractere.

Exemplu: Fișierul text *numere.txt* conține, pe mai multe linii, numere întregi separate între ele prin spații. Scrieți un program care afișează pe ecran suma numerelor din fișier.

Soluția 1 (folosind metoda `read`):

Construim o listă numere care să conțină numerele din fișier, împărțind sirul corespunzător întregului conținut al fișierului în subșiruri cu metoda `split` și transformând fiecare subșir într-un număr cu metoda `str`:

```
f = open("numere.txt")
numere = [int(x) for x in f.read().split()]
print("Suma numerelor din fisier:", sum(numere))
f.close()
```

Soluția 2 (folosind metoda `readline`):

Construim tot o listă care să conțină toate numerele din fișier, dar împărțind în subșiruri doar sirul corespunzător liniei curente:

```
f = open("numere.txt")
numere = []
linie = f.readline()
while linie != "":
    numere.extend([int(x) for x in linie.split()])
    linie = f.readline()
print("Suma numerelor din fisier:", sum(numere))
f.close()
```

Evident, o soluție mai eficientă din punct de vedere al memoriei utilizate constă în calculul sumei numerelor de pe fiecare linie și adunarea sa la suma tuturor numerelor din fișier:

```
f = open("numere.txt")
total = 0
linie = f.readline()
while linie != "":
    numere = [int(x) for x in linie.split()]
    total = total + sum(numere)
    linie = f.readline()
print("Suma numerelor din fisier:", total)
f.close()
```

Soluția 3 (folosind metoda `readlines`):

Vom proceda într-un mod asemănător cu soluția 2, dar parcurgând lista cu liniile fișierului furnizată de metoda `readlines`:

```
f = open("numere.txt")
total = 0
for linie in f.readlines():
    numere = [int(x) for x in linie.split()]
    total = total + sum(numere)
print("Suma numerelor din fisier:", total)
f.close()
```

Atenție, în oricare dintre soluțiile prezentate, metoda `split` trebuie apelată fără parametri, pentru a împărți șirul respectiv în subșiruri considerând toate spațiile albe (care includ caracterele '\n' și '\r')! Altfel, dacă am apela metoda `split` doar cu parametrul " " (un spațiu) ar fi generată eroarea `ValueError` în momentul în care am încerca să transformăm în număr ultimul subșir de pe o linie (mai puțin ultima din fișier), deoarece acesta ar conține și caracterul '\n'!

Scrierea datelor într-un fișier text

În limbajul Python, într-un fișier text se pot scrie doar siruri de caractere. Pentru scrierea datelor într-un fișier text sunt definite următoarele metode:

- **`write(șir)`**: scrie șirul respectiv în fișier, fără a adăuga automat o linie nouă

Exemplu:

Program	Fișier de ieșire
<pre>f = open("exemplu.txt", "w") cuvinte = ["Ana", "are", "mere!"] for cuv in cuvinte: f.write(cuv) f.close()</pre>	Anaaremere!

Dacă dorim să scriem fiecare cuvânt pe o linie nouă, atunci trebuie să modificăm `f.write(cuv)` în `f.write(cuv + "\n")`.

- **writelines(colecție de siruri)**: scrie toate sirurile din colecție în fișier, fără a adăuga automat linii noi între ele

Exemplu:

Program	Fișier de ieșire
<pre>f = open("exemplu.txt", "w") cuvinte = ["Ana", "are", "mere!"] f.writelines(cuvinte) f.close()</pre>	Ana are mere!

Dacă dorim să scriem fiecare cuvânt pe o linie nouă, atunci fie trebuie să adăugăm câte un caracter "\n" la sfârșitul fiecărui cuvânt din listă (de exemplu, prin instrucțiunea `cuvinte = [cuv + "\n" for cuv in cuvinte]`), fie să modificăm `f.writelines(cuvinte)` în `f.writelines("\n".join(cuvinte))`, deci să concatenăm toate sirurile din listă într-unul singur, folosind caracterul "\n" ca separator (evident, în acest caz putem utiliza și metoda `write`).

Închiderea unui fișier text

Indiferent de modalitatea în care a fost deschis un fișier și, implicit, accesat conținutul său, acesta trebuie închis după terminarea prelucrării sale, folosind metoda `close()`. Închiderea unui fișier constă în golirea eventualei zone tampon (buffer) alocate fișierului și ștergerea din memorie a obiectului asociat cu el. Dacă un fișier deschis într-unul dintre modurile de scriere nu este închis, atunci există riscul ca ultimele informații scrise în fișier să nu fie salvate! În mod implicit, un fișier este închis dacă referința obiectului asociat este atribuită unui alt fișier (i.e., se utilizează, din nou, funcția `open`).

Putem elimina necesitatea închiderii explicite a unui fișier putem folosi o instrucțiune `with...as`, astfel:

```
with open("exemplu.txt") as f:
    s = f.readlines()
    print(f.closed)      #False

print(f.closed)          #True
print(s)
```

În acest caz, fișierul va fi închis automat imediat după ce se va termina prelucrarea sa, aşa cum se poate observa din exemplul de mai sus (data membră `closed` permite testarea stării curente a unui fișier, respectiv dacă acesta a fost închis sau nu). Mai mult, închiderea fișierului respectiv se va efectua corect chiar și în cazul apariției unei erori în timpul prelucrării sale. Atenție, chiar dacă se utilizează o instrucțiune

`with...as`, erorile care pot să apară în momentul deschiderii unui fișier trebuie să fie tratate explicit!

În încheiere, vom prezenta un exemplu în care vom utiliza toate cele 3 moduri de deschidere ale unui fișier text.

Mai întâi, vom implementa un program care să genereze un fișier text care va conține pe prima linie o sumă formată din n numere naturale aleatorii (de exemplu, $12 + 300 + 9 + 1234$):

```
import random

numef = input("Numele fisierului: ")
n = int(input("Numarul de valori aleatorii: "))

with open(numef, "w") as f:
    # initializam generatorul de numere aleatorii
    random.seed()
    for k in range(n - 1):
        # generam un numar aleatoriu
        # cuprins intre 1 si 1000
        x = random.randint(1, 1000)
        f.write(str(x) + " + ")
    f.write(str(random.randint(1, 1000)))
```

În continuare, vom implementa un program care să calculeze suma numerelor din fișier și apoi să o scrie la sfârșitul primei linii, după un semn "=" (de exemplu, $12 + 300 + 9 + 1234 = 1555$):

```
numef = input("Numele fisierului: ")

# deschidem fisierul pentru citire și
# calculăm suma numerelor de pe prima linie
f = open(numef, "r")
nr = [int(x) for x in f.readline().split("+")]
s = sum(nr)

# deschidem fisierul pentru adaugare
# și scriem suma la sfârșitul primei linii

# fisierul deschis anterior pentru citire
# va fi inchis implicit
f = open(numef, "a")
f.write(" = " + str(s))
# inchidem explicit fisierul
f.close()
```

Functii

O *funcție* este un set de instrucțiuni, grupate sub un nume unic, care efectuează o prelucrare specifică a unor date în momentul în care este apelată.

Functiile sunt utilizate pentru a modulariza codul dintr-un program, prin împărțirea sa în subprograme (i.e., funcții) care execută fiecare o singură prelucrare bine definită. În plus, funcțiile permit o reutilizare simplă și sigură a codului, prin organizarea lor în module și pachete.

În limbajul Python sunt predefinite mai multe funcții care sunt des utilizate în programe, cum ar fi funcții de conversie (`int`, `str`, `bool` etc.), funcții matematice (`min`, `max`, `sum` etc.), funcții pentru crearea unei colecții (`list`, `dict`, `set` etc.) și.a.m.d. O listă completă a tuturor funcțiilor predefinite din limbajul Python poate fi consultată aici: <https://docs.python.org/3/library/functions.html>.

O funcție poate fi definită de către un utilizator folosind cuvântul cheie `def`, astfel:

```
def nume_funcție(parametrii formali)
    instrucțiuni
```

Parametrii formalii sunt utilizați în antetul unei funcții pentru a preciza datele sale de intrare într-un mod abstract (formal). *Parametrii actuali* ai unei funcții sunt expresii ale căror valori sunt asociate parametrilor formalii în momentul apelării unei funcții.

Parametrii formalii ai unei funcții pot fi de mai multe tipuri în limbajul Python:

- *parametri simpli* au, de obicei, un caracter pozitional, respectiv în momentul apelării unei funcției trebuie să fie înlocuiți cu același număr de parametrii efectivi, compatibili ca tipuri de date:

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

# apelare pozitională
s = suma(3, 7)
print("s =", s)

# apelare prin nume
s = suma(y=7, x=3)
print("s =", s)
```

În acest exemplu, parametrii formalii ai funcției `suma` sunt `x` și `y`, iar constantele 3 și 7 sunt parametrii efectivi. Deoarece în limbajul Python nu se precizează explicit tipurile de date ale parametrilor formalii, compatibilitatea dintre ei și parametrii efectivi poate

fi, în unele cazuri, o noțiune destul de vagă. Astfel, în acest exemplu, parametrii efectivi ai funcției pot fi și două siruri de caractere sau două liste, dar nu pot fi, de exemplu, un număr și un sir de caractere! De asemenea, se observă faptul că parametrii simpli permit și apelarea prin nume.

- *parametri cu valori implicate* sunt utilizați dacă se dorește inițializarea lor cu niște valori implicate, care vor fi utilizate în momentul apelării funcției cu un număr de parametrii efectivi mai mic decât numărul parametrilor formali:

```
def suma(x=0, y=0):
    return x + y

s = suma()                      # x = 0 și y = 0
print("s =", s)                  # s = 0

s = suma(7)                      # x = 7 și y = 0
print("s =", s)                  # s = 7

s = suma(y=19)                   # x = 0 și y = 19
print("s =", s)                  # s = 19

s = suma(7, 10)                   # x = 7 și y = 10
print("s =", s)                  # s = 17
```

Atenție, dacă antetul unei funcții conține și parametri simpli și parametri cu valori implicate, parametrii poziționali trebuie să fie precizați primii, pentru a evita ambiguitățile care pot să apară în momentul apelării funcției! De exemplu, dacă funcția se mai sus ar avea antetul `suma(x=0, y)`, atunci apelul `s = suma(7)` ar putea fi interpretat fie ca `s = suma(7, y)` și ar fi incorect (parametrului `y` nu i-ar fi asociat niciun parametru efectiv), fie ca `s = suma(0, 7)` și ar fi corect. În schimb, antetul `suma(x, y=0)` nu va mai genera nicio ambiguitate în cazul apelului `s = suma(7)`, putând fi interpretat doar ca `s = suma(7, 0)`.

În limbajul Python, se pot defini foarte simplu și funcții cu *număr variabil de parametri*, care sunt utile când nu se poate preciza numărul exact de parametrii ai unei funcții. De exemplu, în același program putem să avem nevoie de o funcție care să calculeze suma a două numere și de o funcție care să calculeze suma a trei numere. Deoarece în limbajul Python nu se pot defini două funcții cu același nume și număr diferit de parametri (de fapt, se pot defini, dar compilatorul o va lua în considerare doar pe ultima definită și orice apelare a primei funcții definite va fi semnalată ca eroare), înseamnă că ar trebui să definim două funcții cu nume diferite, dar care sunt foarte asemănătoare din punct de vedere al prelucrărilor efectuate.

Definirea unei funcții cu număr variabil de parametri se realizează adăugând simbolul * înaintea unui parametru, ceea ce indică faptul că parametrul respectiv va conține, sub forma unui tuplu, un număr oarecare de parametri efectivi. De exemplu, o funcție cu număr variabil de parametri care calculează suma acestora se poate defini și apela astfel:

```

def suma(*args):
    sa = 0
    for x in args:
        sa = sa + x
    return sa
s = suma()
print("s =", s)           #s = 0

s = suma(7)                #s = 7
print("s =", s)

s = suma(1, 2, 3, 4)      #s = 10
print("s =", s)

lista = [x for x in range(1, 11)]
s = suma(*lista)
print("s =", s)           #s = 55

```

Evident, o funcție poate să aibă un singur parametru variabil, iar eventualii parametrii existenți după el trebuie să fie precizați explicit, prin numele lor, în momentul apelării funcției (deoarece ei nu mai pot fi accesati pozitional, fiind precedați de un număr necunoscut de parametrii). De exemplu, următoarea funcție calculează suma parametrilor variabili care sunt cel puțini egali cu valoarea `minim`:

```

def suma(*valori, minim):
    sv = 0
    for x in valori:
        if x >= minim:
            sv = sv + x
    return sv

s = suma(7, minim=5)          #s = 7
print("s =", s)

s = suma(1, 2, 3, minim=10)    #s = 0
print("s =", s)

lista = [x for x in range(1, 11)]
s = suma(*lista, minim=8)
print("s =", s)               #s = 27

```

Dacă în exemplul de mai sus nu vom preciza explicit valoarea parametrului `minim` (de exemplu, scriind `s = suma(1, 2, 3, 10)`), atunci se va semnala o eroare de tipul `TypeError` (pentru exemplul considerat aceasta va fi `TypeError: suma() missing 1 required keyword-only argument: 'minim'`).

Bineînțeles, dacă o funcție cu număr variabil de parametri are și parametri simpli și parametri cu valori implicate, atunci trebuie respectată și regula ca parametrii simpli să

fie scriși înaintea celor cu valori implicite! De exemplu, următoarea funcție `suma` calculează suma parametrilor variabili cuprinși între valorile `minim` și `maxim` care sunt și multiplii ai numărului `t`:

```
def suma(t, *valori, minim=0, maxim=100):
    sv = 0
    for x in valori:
        if minim <= x <= maxim and x % t == 0:
            sv = sv + x
    return sv

s = suma(7, 14, 19, 21, -56)                                     #s = 35
print("s =", s)

s = suma(7, 14, 19, 21, -56, minim=-100)                         #s = -21
print("s =", s)

s = suma(7, 14, 19, 21, -56, minim=-100, maxim=0)      #s = -56
print("s =", s)
```

Datorită operației implicită de împachetare a mai multor valori sub forma unui tuplu, o funcție poate să furnizeze mai multe valori. Astfel, o instrucțiune de forma `return a, b` este echivalentă cu `return (a, b)`, așa cum se poate observa din următorul exemplu:

```
def suma_prod(x, y):
    return x + y, x * y

s, p = suma_prod(3, 7)
print("s =", s, "\tp =", p)

(s, p) = suma_prod(3, 7)
print("s =", s, "\tp =", p)

t = suma_prod(3, 7)
print("s =", t[0], "\tp =", t[1])
print("Suma si produsul:", *t)
```

Observați faptul că la apelarea funcției se poate utiliza operația de despachetare a unui tuplu, complementară celei de împachetare utilizată în instrucțiunea `return`!

Modalități de transmitere a parametrilor

În limbajele C/C++ transmiterea unui parametru efectiv către o funcție se poate realiza în două moduri:

- *transmitere prin valoare*: se transmite o copie a valorii parametrului efectiv, deci modificările efectuate asupra parametrului respectiv în interiorul funcției nu se reflectă și în exteriorul său;
- *transmitere prin adresă/referință*: se transmite adresa parametrului efectiv, deci modificările efectuate asupra parametrului respectiv în interiorul funcției se reflectă și în exteriorul său.

În limbajul Python, orice parametru efectiv al unei funcții este o referință către un obiect (i.e., nu se transmite obiectul în sine, ci o referință spre el) care se transmite implicit prin valoare (i.e., se transmite o copie a referinței respective), deci modificarea referinței respective în interiorul funcției nu se va reflecta în exteriorul său. Din acest motiv, mecanismul de transmitere a parametrilor către o funcție în limbajul Python se numește *transmitere prin referință la obiect (call by object reference)*.

Exemplul 1:

Funcția	Executarea programului	Rezultat
<pre>def functie(t): t = 100 # Pas 3 x = 7 # Pas 1 functie(x) # Pas 2 print("x =", x)</pre>		x = 7

Explicație: După pasul 2, variabilele x și copie_x vor conține aceeași referință (spre obiectul 7), dar după pasul 3 doar copie_x se va modifica (va conține referința obiectului 100), deoarece x este o referință transmisă prin valoare.

Exemplul 2:

Funcția	Executarea programului	Rezultat
<pre>def functie(t): t = t.upper() # Pas 3 x = "test" # Pas 1 functie(x) # Pas 2 print("x =", x)</pre>		x = test

Explicație: vezi exemplul anterior!

Exemplul 3:

Funcția	Executarea programului	Rezultat
<pre>def functie(t): t.append(100) # Pas 3 x = [1, 2, 3] # Pas 1 functie(x) # Pas 2 print("x =", x)</pre>	<p>Pas 1: <code>x</code> points to <code>[1, 2, 3]</code>. Pas 2: <code>copie_x</code> points to the same list. Pas 3: The list is modified to <code>[1, 2, 3, 100]</code>.</p>	<code>x = [1, 2, 3, 100]</code>

Explicație: După pasul 2, variabilele `x` și `copie_x` vor conține aceeași referință, iar după pasul 3 conținutul listei se va modifica prin intermediul referinței din `copie_x` (însă fără a modifica referința listei!), deci variabila `x` va putea accesa lista modificată.

Exemplul 4:

Funcția	Executarea programului	Rezultat
<pre>def functie(t): t = t + [100] # Pas 3 x = [1, 2, 3] # Pas 1 functie(x) # Pas 2 print("x =", x)</pre>	<p>Pas 1: <code>x</code> points to <code>[1, 2, 3]</code>. Pas 2: <code>copie_x</code> points to the same list. Pas 3: The list is modified to <code>[1, 2, 3, 100]</code>.</p>	<code>x = [1, 2, 3]</code>

Explicație: După pasul 2, variabilele `x` și `copie_x` vor conține aceeași referință (spre lista `[1,2,3]`), dar la pasul 3 operatorul de concatenare (`+`) va crea o nouă listă `[1,2,3,100]`, deci după pasul 3 doar `copie_x` va conține referința noii liste. Practic, deși o listă este mutabilă, elementul 100 a fost adăugat într-o manieră specifică obiectelor imutabile!

În concluzie, mecanismul de *transmitere prin referință la obiect* a parametrilor efectivi către o funcție în limbajul Python utilizează transmiterea prin valoare a referințelor parametrilor efectivi (copii ale referințelor lor, deci modificarea acestora nu va fi vizibilă în exteriorul funcției) și acționează astfel:

- dacă obiectul asociat referinței este *imutabil*, atunci modificarea parametrului respectiv în interiorul funcției nu se va reflecta în exteriorul funcției deoarece conținutul obiectului asociat referinței nu poate fi modificat (din cauza imutabilității), iar crearea unei referințe noi nu se va reflecta în exteriorul funcției (din cauza transmiterii prin valoare a referinței);
- dacă obiectul asociat referinței este *mutabil*, atunci modificarea conținutului parametrului respectiv în interiorul funcției se va reflecta în exteriorul funcției deoarece nu se va modifica referința obiectului.

Atenție la exemplul 4 de mai sus, deoarece acolo nu se modifică direct conținutul obiectului mutabil de tip listă, ci se creează un nou obiect (tot o listă mutabilă) care conține noua listă și se atribuie referința sa copiei referinței parametrului!

CURS 7

Funcții

Variabile locale și globale

O variabilă definită în interiorul unei funcții se numește *variabilă locală* și este vizibilă (i.e., poate fi utilizată) doar în interiorul funcției respective.

O variabilă definită în afara oricărei funcții se numește *variabilă globală* și este vizibilă în tot modulul respectiv (i.e., poate fi utilizată în interiorul oricărei funcții).

Exemplu:

```
def afisare():
    print("x = ", x)      # se va utiliza variabila globală x,
                          # deoarece nu există o variabilă locală x
x = 100
afisare()                # x = 100
```

Dacă într-o funcție există definită o variabilă locală având același nume cu o variabilă globală, atunci, implicit, se va utiliza variabila locală în interiorul funcției:

```
def afisare():
    x = 200
    print("x = ", x)      # se va utiliza variabila locală x

x = 100
afisare()                  # x = 200
print("x = ", x)          # x = 100 (variabila globală)
```

Dacă într-o funcție există definită o variabilă locală având același nume cu o variabilă globală, atunci putem utiliza variabila globală precizând în interiorul funcției acest lucru:

```
def afisare():
    global x
    print("x = ", x)      # se va utiliza variabila globală x
    x = 200

x = 100
afisare()                  # x = 100
print("x = ", x)          # x = 200
```

Atenție, dacă în exemplul de mai sus ar lipsi declararea `global x`, atunci ar fi generată eroarea `UnboundLocalError: local variable 'x' referenced before assignment`, deoarece definirea unei variabile locale `x` prin `x=200` va determina interpretatorul să nu

mai caute o variabilă globală cu numele `x`! Același lucru se va întâmpla și în exemplul următor:

```
def f():
    x = x + 100           # eroare!
    print("x = ", x)

x = 200
f()
print("x = ", x)
```

În acest caz, eroarea apare deoarece interpretatorul consideră faptul că o instrucțiune de atribuire de forma `"x ="` reprezintă o declarare prin inițializare a unei variabile locale `x`, dar, evident, expresia `x + 100` nu poate fi evaluată, variabila locală `x` nefiind inițializată!

Functii imbricate

În limbajul Python putem defini o funcție în interiorul altei funcții, aşa cum se poate observa din exemplul următor:

```
def combinari(n, k):
    def factorial(x):
        p = 1
        for i in range(1, x+1):
            p = p * i
        return p

    return factorial(n) // (factorial(k) * factorial(n-k))

print(combinari(5, 3))
```

O funcție `g` definită în interiorul unei funcții `f` este locală funcției `f`, deci funcția `g` poate fi apelată doar în interiorul funcției `f` (i.e., funcția `g` este o funcție auxiliară pentru `f`).

O funcție imbricată are acces implicit la parametrii funcției în care este definită și la variabilele sale locale:

```
def calcul(x, y):
    n = 2
    def medie(k):
        return (x**n + y**n) / k

    return medie(2)

print("m = ", calcul(3, 4))          # m = 12.5
```

Atenție, deși o funcție imbricată are acces implicit la variabilele locale ale funcției în care este definită, ea nu le poate modifica deoarece va fi generată o eroare de tipul `UnboundLocalError`. Pentru a utiliza o variabilă locală într-o funcție imbricată, variabila locală trebuie declarată în interiorul funcției imbricate folosind cuvântul cheie `nonlocal`:

```
def f():
    n = 100
    def aux():
        nonlocal n
        n = n * 2

    aux()
    return n

print(f())           # 200
```

Practic, prin utilizarea cuvântului cheie `nonlocal` pentru declararea unei variabile în interiorul unei funcții imbricate îi cerem interpretatorului să caute definirea variabilei respective în cel mai apropiat spațiu de nume exterior funcției imbricate, mai puțin în cel global (pentru a accesa variabilele globale se folosește cuvântul cheie `global`, așa cum deja am menționat):

```
incr = 100
def f():
    n = 7
    incr = 10

    def g():
        nonlocal incr, n
        n = n + incr
        return n

    def h():
        global incr
        nonlocal n
        n = n + incr
        return n

    print("nonlocal incr:", g())      # nonlocal incr: 17
    print("global incr:", h())       # global incr: 117

f()
```

În limbajul Python, o funcție poate să returneze o funcție imbricată, așa cum se poate observa din exemplul următor:

```
def putere(baza):

    def paux(exponent):
        return baza ** exponent

    return paux
putere10 = putere(10)

print(type(putere10))      # <class 'function'>
print(putere10.__name__)    # paux
print(putere10(3))         # 1000
```

Practic, `putere10` este o referință spre funcția `paux` particularizată pentru `baza = 10`, deci poate fi utilizată la fel ca orice altă funcție.

Transmiterea unei funcții ca parametru al altei funcții (callback)

Există mai multe situații în care este necesar să utilizăm *mecanismul de callback*, respectiv să transmitem o funcție `f` ca parametru al unei funcții `g`, astfel încât funcția `g` să poată apela funcția `f` când acest lucru este necesar. De exemplu, notificările utilizate în aplicațiile mobile utilizează un mecanism asemănător mecanismului de callback, respectiv o aplicație de tip server înregistrează faptul că un anumit utilizator a acceptat să primească notificări și în momentul în care apare un anumit eveniment pe server (de exemplu, când sunt depuși sau retrăși bani dintr-un cont bancar), server-ul îi trimite utilizatorului respectiv o notificare. De asemenea, mecanismul de callback este intens utilizat în programarea interfețelor grafice, respectiv sistemul de operare primește o funcție pe care să o apeleze în momentul apariției unui anumit eveniment (e.g., apăsarea unui buton, închiderea unei ferestre, selectarea unei opțiuni dintr-o listă etc.).

Mecanismul de callback mai este utilizat și în *programarea generică*, respectiv în scrierea unor funcții care realizează prelucrări generice ale unei funcții a cărei expresie nu este cunoscută (e.g., reprezentarea grafică a unei funcții, calculul unei integrale etc.).

În continuare, vom prezenta o funcție generică pentru calculul unor sume. De exemplu, să considerăm următoarele 3 sume:

$$\begin{aligned} S_1 &= 1 + \frac{1}{2} + \cdots + \frac{1}{n} \\ S_2 &= 1^2 + 2^2 + \cdots + n^2 \\ S_3 &= e^1 + e^2 + \cdots + e^n \end{aligned}$$

Evident, putem să definim câte o funcție pentru calculul fiecărei sume, dar această soluție nu este scalabilă. Putem observa cu ușurință faptul că toate cele 3 sume au următoarea formă generală:

$$S_k = \sum_{i=1}^n f_k(i)$$

unde prin $f_k(i)$ am notat termenul de rang i al sumei S_k pentru $k \in \{1, 2, 3\}$, deci, pentru sumele de mai sus, termenii generali sunt $f_1(i) = \frac{1}{i}$, $f_2(i) = i^2$ și $f_3(i) = e^i$. Astfel, putem scrie o funcție generică pentru calculul unei astfel de sume, care va avea parametrii n și fk , unde fk va fi o funcție care implementează termenul general al unei anumite sume:

```
import math

def suma_generica(n, fk):
    s = 0
    for i in range(1, n+1):
        s = s + fk(i)
    return s

def fk_1(i):
    return 1/i

def fk_2(i):
    return i**2

n = 10
s = suma_generica(n, fk_1)
print("Suma 1:", s)                      # Suma 1: 2.9289682539682538

s = suma_generica(n, fk_2)
print("Suma 2:", s)                      # Suma 2: 385

s = suma_generica(n, math.exp)
print("Suma 3:", s)                      # Suma 3: 34843.77384533132
```

Observați faptul că putem apela funcția `suma_generica` utilizând pentru termenul general și funcții predefinite!

Funcții anonime (lambda expresii)

O *funcție anonimă (lambda expresie)* este o funcție foarte simplă, fără nume, definită folosind cuvântul cheie `lambda`, astfel:

`lambda parametrii: expresie`

O funcție anonimă poate să aibă unul sau mai mulți parametrii, dar corpul său trebuie să fie format dintr-o singură expresie (e.g., nu poate conține instrucțiuni sau mai multe expresii). În momentul apelării unei funcții anonime, expresia asociată va fi evaluată, iar rezultatul obținut va fi furnizat direct, fără a fi necesară utilizarea cuvântului cheie `return` în interiorul funcției anonime.

Exemple:

- *suma a două numere:* `lambda x, y: x+y`
- *testarea parității unui număr întreg:* `lambda x: x % 2 == 0`
- *testarea divizibilității:* `lambda x, y: False if y == 0 else x % y == 0`
- *suma cifrelor unui număr:* `lambda x: sum([int(cf) for cf in str(x)])`
- *numărul vocalelor dintr-un sir:* `lambda sir: len([lit for lit in sir if lit in "aeiouAEIOU"])`
- *numărul valorilor dintr-o listă L strict mai mari decât o valoare v:* `lambda L, v: len([x for x in L if x > v])`

O funcție anonimă poate fi apelată direct, în momentul definirii sale, lucru care nu este posibil în cazul funcțiilor obișnuite:

```
print((lambda x, y: x+y)(5, 7))      # 12
print((lambda s: len([c for c in s if c in "aeiou"]))("examene")) # 4
print((lambda x, y: x % y == 0)(24, 6)) # True
```

O referință spre o funcție anonimă poate fi atribuită unei variabile, pentru a fi utilizată ca o funcție obișnuită, dar acest lucru nu se recomandă deoarece contrazice ideea de funcție anonimă (se preferă definirea unei funcții obișnuite, cu nume):

```
f = lambda x, y: x+y
s = f(5, 7)
print(f"s = {s}")    # s = 12
```

O funcție poate returna o funcție anonimă, permitând astfel crearea unor funcții particularizate în raport de anumite criterii:

```
def selectorFunctie(tip):
    if tip == "suma":
        return lambda x, y: x + y
    elif tip == "diferenta":
        return lambda x, y: x - y
    elif tip == "produs":
        return lambda x, y: x * y
    else:
        return None

f = selectorFunctie("suma")
s = f(5, 7)
print(f"s = {s}")    # s = 12
```

```
f = selectorFunctie("produs")
p = f(5, 7)
print(f"p = {p}") # p = 35
```

O funcție anonimă poate fi utilizată în cadrul mecanismului de callback, pentru a elimina definițiile funcțiilor transmise ca parametrii (evidenț, dacă funcțiile respective sunt suficient de simple pentru a fi implementate ca niște funcții anone):

```
import math

def suma_generica(n, fk):
    s = 0
    for i in range(1, n+1):
        s = s + fk(i)
    return s

n = 10
s = suma_generica(n, lambda x: 1 / x)
print("Suma 1:", s) # Suma 1:
2.9289682539682538

s = suma_generica(n, lambda x: x ** 2)
print("Suma 2:", s) # Suma 2: 385

s = suma_generica(n, math.exp)
print("Suma 3:", s) # Suma 3:
34843.77384533132
```

Încheiem prin a preciza faptul că funcțiile anone sunt utilizate, de obicei, în *programarea funcțională*, o altă paradigmă de programare implementată în limbajul Python, alături de programarea procedurală și programarea orientată pe obiecte (<https://realpython.com/python-functional-programming/>).

Sortarea colecțiilor de date

În limbajul Python, colecțiile pot fi sortate crescător folosind funcția predefinită `sorted`, care furnizează o listă cu elementele colecției respective sortate crescător:

- `sorted([2, 1, 5, 2, 1, 4]) = [1, 1, 2, 2, 4, 5]`
- `sorted((2, 1, 5, 2, 1, 4)) = [1, 1, 2, 2, 4, 5]`
- `sorted({2, 1, 5, 2, 1, 4}) = [1, 2, 4, 5]`
- `sorted({"d": 2, "a": 1, "f": 0, "b": 3}) = ['a', 'b', 'd', 'f']`

Observați faptul că, indiferent de tipul colecției sortate (i.e., listă, tuplu sau mulțime), rezultatul este furnizat sub forma unei liste, iar în cazul unui dicționar funcția va furniza doar o listă a cheilor dicționarului sortate crescător. Dacă dorim să obținem perechile

unui dicționar sortate crescător în funcție de chei, trebuie să sortăm o listă cu tuplurile corespunzătoare intrărilor sale:

```
sorted({"d": 2, "a": 1, "b": 3, "f": 0}.items()) = [('a', 1), ('b', 3), ('d', 2), ('f', 0)]
```

În cazul sortării unui sir de caractere, funcția `sorted` va returna o listă formată din caracterele sirului, ordonate crescător:

```
sorted("exemplu") = ['e', 'e', 'l', 'm', 'p', 'u', 'x']
```

Dacă este necesar, putem să transformăm lista de caractere furnizată de metoda `sorted` înapoi într-un sir de caractere, folosind metoda `join`, astfel:

```
"".join(sorted("exemplu")) = "eelmpux"
```

Observație: O listă poate fi sortată direct, prin rearanjarea elementelor sale în ordine crescătoare, folosind metoda `sort` din clasa `list`:

```
L = [2, 1, 5, 2, 1, 4]
L.sort()
print("L =", L) # L = [1, 1, 2, 2, 4, 5]
```

Dacă ulterior nu mai avem nevoie de lista inițială, atunci se recomandă utilizarea metodei `sort` în locul funcției `sorted`, deoarece nu utilizează memorie suplimentară!

Deoarece funcția `sorted` și metoda `sort` din clasa `list` au aceeași parametrii opționali, în continuare vom prezenta doar funcția `sorted`, deoarece ea poate fi utilizată pentru orice structură de date iterabilă.

Pentru a realiza o sortare descrescătoare a elementelor unei structuri de date iterabile, parametrul opțional `reverse` al funcției `sorted` trebuie setat la valoarea `True`:

- `sorted([2, 1, 5, 2, 1, 4], reverse=True) = [5, 4, 2, 2, 1, 1]`
- `"".join(sorted("exemplu", reverse=True)) = "xupmlee"`

Evident, sortarea unei structuri de date iterabile se poate realiza doar în cazul în care elementele sale sunt comparabile, altfel se va genera o eroare de tipul `TypeError`. De exemplu, în cazul apelului `sorted([100, -10, "12345", 70])` se va genera eroarea `TypeError: '<' not supported between instances of 'str' and 'int'!`

Dacă o listă este formată din tupluri comparabile, atunci acestea vor fi sortate în ordine lexicografică. De exemplu, prin apelul `sorted([(‘g’, 1), (‘f’, 7), (‘b’, 3), (‘a’, 2), (‘f’, 0), (‘b’, 1)])` se va obține lista `[(‘a’, 2), (‘b’, 1), (‘b’, 3), (‘f’, 0), (‘f’, 7), (‘g’, 1)]`, în care tuplurile au fost sortate în ordinea crescătoare a primelor componente, iar în cazul în care acestea erau egale în ordinea crescătoare a componentelor secundare. În cazul unei liste de siruri de caractere, sortarea se va realiza folosind tot ordinea lexicografică. De exemplu, prin apelul

`sorted(["prune", "mere", "ananas", "pere", "mango"])` se va obține lista `["ananas", "mango", "mere", "pere", "prune"]`.

Pentru realizarea unor sortări complexe, eventual bazate pe mai multe criterii, putem să asociem fiecărui element al unei colecții o *cheie* pe bază căreia să se realizeze operația de sortare. Acest lucru se realizează folosind parametrul optional `key` al funcției `sorted`, respectiv atribuindu-i acestuia numele unei funcții care asociază unui element al colecției cheia dorită. De exemplu, pentru a sorta crescător numerele dintr-o listă în funcție de sumele cifrelor lor, vom folosi pentru parametrul optional `key` funcția `sumaCifre`, care calculează suma cifrelor unui număr natural `nr`:

```
def sumaCifre(nr):
    return sum([int(c) for c in str(nr)])

L = [30, 27, 111, 71, 101, 107, 12, 81, 202, 18]
print(sorted(L, key=sumaCifre))
```

În urma executării programului, se va afișa următoarea listă (am evidențiat grupurile de numere care au aceeași sumă a cifrelor):

$$[\underbrace{101}_2, \underbrace{30, 111, 12}_3, \underbrace{202}_4, \underbrace{71, 107}_8, \underbrace{27, 81, 18}_9]$$

Practic, funcția `sumaCifre` a fost apelată pentru fiecare element al listei înainte ca acesta să fie comparat cu alte elemente, iar valoarea obținută (cheia elementului) a fost utilizată în comparații în locul numărului respectiv!

Deoarece funcția `sorted` implementează *o metodă de sortare stabilă*, în lista sortată se va păstra ordinea relativă din lista inițială a elementelor cu chei egale. De exemplu, în lista inițială, numărul 27 se află înaintea numărului 81, iar numărul 81 se află înaintea numărului 18, iar această ordine relativă este păstrată și în lista sortată.

Putem rescrie mai concis secvența de cod de mai sus, utilizând o funcție anonimă în locul funcției `sumaCifre`:

```
L = [30, 27, 111, 71, 101, 107, 12, 81, 202, 18]
print(sorted(L, key=lambda n: sum([int(c) for c in str(n)])))
```

Dacă dorim să sortăm numerele din listă în ordinea crescătoare a sumelor cifrelor lor, iar în cazul în care sumele cifrelor sunt egale să le sortăm crescător după valorile lor, atunci vom asocia fiecărui număr un tuplu format din suma cifrelor sale și el însuși:

```
def sumaCifre(nr):
    sc = sum([int(c) for c in str(nr)])
    return sc, nr

L = [30, 27, 111, 71, 101, 107, 12, 81, 202, 18]
print("L =", sorted(L, key=sumaCifre))
```

Astfel, se va afișa lista [101, 12, 30, 111, 202, 71, 107, 18, 27, 81], deoarece cheile asociate elementelor listei inițiale [30, 27, 111, 71, 101, 107, 12, 81, 202, 18] au fost tuplurile (3, 30), (9, 27), (3, 111), (8, 71), (2, 101), (8, 107), (3, 12), (9, 81), (4, 202), (9, 18), iar sortarea elementelor listei a fost realizată comparând lexicografic aceste tupluri!

Putem rescrie mai concis secvența de cod de mai sus, utilizând o funcție anonimă care furnizează un tuplu, în locul funcției `sumaCifre`:

```
L = [30, 27, 111, 71, 101, 107, 12, 81, 202, 18]
print(sorted(L, key=lambda n: (sum([int(c) for c in str(n)]), n)))
```

Observație: Pentru a sorta descrescător o colecție în funcție de o cheie numerică k este suficient să folosim cheia -k. De exemplu, pentru a sorta numerele dintr-o listă în ordinea crescătoare a sumelor cifrelor lor, iar în cazul în care sumele cifrelor sunt egale să le sortăm descrescător după valorile lor, atunci vom utiliza următoarea funcție pentru chei:

```
def sumaCifre(nr):
    sc = sum([int(c) for c in str(nr)])
    return sc, -nr
```

Funcția utilizată pentru a calcula cheia unui element poate fi și o funcție predefinită. De exemplu, putem utiliza funcția predefinită `len` pentru a sorta o listă de siruri de caractere în ordinea crescătoare a lungimilor lor:

```
L = ["prune", "pere", "ananas", "mere", "mango"]
print(sorted(L, key=len)) # ['pere', 'mere', 'prune', 'mango', 'ananas']
```

În continuare, vom mai prezenta câteva exemple de sortări complexe:

- Să se sorteze o listă de numere naturale astfel încât numerele pare sortate crescător să fie poziționate înaintea celor impare sortate descrescător.

Pentru a realiza această sortare, vom asocia fiecărui număr natural nr cheia (0, nr) dacă el este par, respectiv cheia (1, -nr) dacă el este impar. Practic, prima componentă a cheii este chiar restul împărțirii numărului nr la 2 (i.e., paritatea sa), iar cea de-a doua componentă este utilizată pentru a sorta crescător sau descrescător numerele cu aceeași paritate!

```
def paritate(nr):
    return nr % 2, nr if nr % 2 == 0 else -nr

L = [int(x) for x in input("Lista: ").split()]
L = sorted(L, key=paritate)
print("Lista sortata:", L)
```

De exemplu, dacă lista inițială este [52, 27, 111, 71, 101, 17, 107, 12, 18], atunci, după sortare, se va obține lista [12, 18, 52, 111, 107, 101, 71, 27, 17].

Putem utiliza o funcție anonimă în locul funcției `paritate`:

```
L = [int(x) for x in input("Lista: ").split()]
L = sorted(L, key=lambda n: (0, n) if n % 2 == 0 else (1, -n))
print("Lista sortata:", L)
```

- b) Considerăm o listă care conține informații despre mai mulți studenți, respectiv pentru fiecare student se cunoaște numele, grupa și nota obținută la examenul de admitere. Să se sorteze studenții în ordinea crescătoare a grupelor, în fiecare grupă studenții să fie sortați descrescător după nota obținută la examenul de admitere, iar în cazul unor note egale studenții să fie sortați alfabetic.

Vom considera faptul că informațiile despre fiecare student sunt memorate într-un tuplu, astfel:

```
L = [( "Popescu Ion", 131, 9.25),
      ( "Ionescu Ana", 133, 8.75),
      ( "Popa Marian", 131, 9.85),
      ( "David Maria", 132, 8.95),
      ("Gheorghe Ana", 131, 9.85),
      ("Popescu Anca", 132, 9.15),
      ("Corbu Florin", 133, 8.05),
      ("Gheorghe Dan", 132, 9.15)]
```

Cheia asociată unui student va consta din componentele tuplului respectiv, în ordinea specificată în criteriile de sortare:

```
def cheie_student(t):
    return t[1], -t[2], t[0]

S = sorted(L, key=cheie_student)
print(*S, sep="\n")
```

După sortarea listei, studenții vor fi afișați în următoarea ordine:

```
("Gheorghe Ana", 131, 9.85)
( "Popa Marian", 131, 9.85)
( "Popescu Ion", 131, 9.25)
("Gheorghe Dan", 132, 9.15)
("Popescu Anca", 132, 9.15)
( "David Maria", 132, 8.95)
( "Ionescu Ana", 133, 8.75)
("Corbu Florin", 133, 8.05)
```

Putem simplifica secvența de cod utilizând o funcție anonimă în locul funcției `cheie_student`:

```
S = sorted(L, key=lambda t: (t[1], -t[2], t[0]))
print(*S, sep="\n")
```

- c) Să se sorteze sirurile de caractere dintr-o listă L în ordinea descrescătoare a lungimilor prefixelor maximale comune cu un sir dat s. De exemplu, dacă lista este L = ["apasator", "apartament", "exemplu", "ars", "test", "aparat", "amic"] și s = "aparator", atunci lista sortată va fi L = ["aparat", "apartament", "apasator", "ars", "amic", "exemplu", "test"].

Pentru a determina prefixul maximal comun a două siruri, vom considera, pe rând, toate prefixele unuia dintre siruri, în ordinea descrescătoare a lungimilor, și vom verifica dacă el este prefix și pentru cel de-al doilea sir:

```
def prefixMaxim(s, t):
    for i in range(len(s), 0, -1):
        if t.startswith(s[:i]):
            return i
    return 0
```

Evident, funcția se poate optimiza din punct de vedere al complexității computaționale considerând prefixele celui mai scurt sir dintre cele două!

Din păcate, funcția `prefixMaxim` are 2 parametri, deci nu o putem utiliza pentru a furniza cheile asociate sirurilor din listă (funcția ar trebui să aibă un singur parametru, respectiv un element al listei)! Totuși, putem să rezolvăm această problemă folosind funcții imbricate, respectiv definim funcția `prefixMaxim` cu un singur parametru, sirul dat s, și în interiorul său definim o funcție auxiliară `pmax` cu un singur parametru t, care va determina prefixul maximal comun dintre sirurile s și t, iar funcția `prefixMaxim` va returna funcția `pmax`:

```
def prefixMaxim(s):
    def pmax(t):
        for i in range(len(s), 0, -1):
            if t.startswith(s[:i]):
                return i, t
        return 0, t

    return pmax
```

Astfel, prin apelul `prefixMaxim(s)`, vom obține o funcție cu un singur parametru, care va determina lungimea prefixului comun maximal dintre un sir oarecare t (parametrul funcției) și sirul dat s, iar această funcție poate fi utilizată pentru a furniza cheia asociată unui element:

```
L = ["apasator", "apartament", "exemplu", "ars", "test",
      "aparat", "amic"]
s = "aparator"

L.sort(key=prefixMaxim(s), reverse=True)
print(L)
```

Cum ar trebui să modificăm funcția `prefixMaxim` astfel încât sirurile având aceeași lungime a prefixului comun maximal să fie sortate alfabetic? În acest exemplu, nu putem înlocui funcția `prefixMaxim` cu o funcție anonimă. De ce?

În încheiere, menționăm faptul că funcția `sorted` și metoda `sort` implementează algoritmul de sortare *Timsort*, care a fost creat special în 2002 de Tim Peters pentru a fi implementat în limbajul Python (<https://en.wikipedia.org/wiki/Timsort>). Algoritmul *Timsort* este derivat din algoritmii de sortare prin interclasare ([MergeSort](#)) și sortare prin inserție ([Insertion sort](#)), având complexitatea $\mathcal{O}(n \log_2 n)$ în cazul cel mai defavorabil.

Generatori

Un *generator* este un tip de funcție a cărei executare nu se termină în momentul în care returnează o valoare, ceea ce îi permite furnizarea mai multor valori într-o manieră secvențială (i.e., sub forma unui *iterator*). Un exemplu de generator predefinit în limbajul Python este `range`, care furnizează valorile întregi dintr-un anumit interval una câte una. Pentru a returna valori în mod repetat, un generator folosește instrucțiunea `yield` în locul instrucțiunii `return`:

```
def generator_numere_pare(n):
    x = 0
    while x <= n:
        yield x
        x += 2
```

Valorile furnizate de un generator pot fi accesate secvențial, folosind, de exemplu, o instrucțiune `for`:

```
for x in generator_numere_pare(100):
    print(x, end=" ")
```

O altă posibilitate de accesare secvențială a valorilor furnizate de un generator o reprezintă utilizarea funcției `next(iterator, [valoare_implicită])`, care permite, în general, accesarea următoarei valori dintr-un iterator sau furnizarea unei valori implicite, precizată prin parametrul opțional `valoare_implicită`, în momentul în care iteratorul nu mai conține nicio valoare:

```
nr_pare = generator_numere_pare(100)
x = next(nr_pare, -1)
while x != -1:
    print(x, end=" ")
    x = next(nr_pare, -1)
```

Practic, pentru a putea furniza mai multe valori într-o manieră secvențială, contextul de apel al unui generator nu este eliminat de pe stiva programului în momentul executării unei instrucțiuni `yield`. Astfel, după revenirea dintr-un apel, un generator îți poate continua executarea din starea în care se afla înaintea apelului respectiv!

Instrucția `return` vidă poate fi utilizată pentru a întrerupe executarea unui generator:

```
def generator_numere_pare(n):
    x = 0
    while True:
        yield x
        if x == n:
            return
        x += 2
```

Un generator poate rula "la infinit", aşa cum se poate observa din urmatorul exemplu:

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

Evident, în acest caz nu se pot utiliza modalitățile de accesarea a valorilor furnizate menționate anterior, ci accesarea acestora trebuie să fie întreruptă explicit:

<pre>for x in generator_numere_pare(): print(x, end=" ") if x == 100: break</pre>	<pre>nr_pare = generator_numere_pare() x = next(nr_pare, -1) while x <= 100: print(x, end=" ") x = next(nr_pare, -1)</pre>
---	---

Întreruperea accesării valorilor furnizate de un generator infinit nu va conduce la oprirea sa, aşa cum se poate observa din urmatorul exemplu (se vor afişa numerele pare de la 0 la 20):

```
nr_pare = generator_numere_pare()
for x in nr_pare:
    print(x, end=" ")
    if x == 10:
        break

for x in nr_pare:
    print(x, end=" ")
    if x == 20:
        break
```

Pentru a întrerupe forțat executarea unui generator, infinit sau nu, trebuie apelată metoda `close` (se vor afişa doar numerele pare de la 0 la 10):

```
nr_pare = generator_numere_pare()
for x in nr_pare:
    print(x, end=" ")
```

```
if x == 10:  
    break  
  
nr_pare.close()  
  
for x in nr_pare:  
    print(x, end=" ")  
    if x == 20:  
        break
```

În concluzie, un generator reprezintă o modalitate foarte simplă de creare a unui iterator în limbajul Python. În plus, generatorii permit o utilizare eficientă a memoriei, valorile fiind furnizate secvențial.

CURS 8

Complexitatea algoritmilor

Un *algoritm* este o succesiune de prelucrări care se aplică mecanic asupra unor date de intrare în scopul obținerii unor date de ieșire.

Principalele caracteristici ale unui algoritm sunt:

- *corectitudine* – proprietatea unui algoritm de a furniza date de ieșire corecte pentru orice date de intrare;
- *determinism* – pentru un anumit set de date de intrare un algoritm trebuie să furnizeze întotdeauna același date de ieșire;
- *generalitate* – un algoritm nu trebuie să rezolve doar o problemă particulară, ci o întreagă clasă de probleme;
- *claritate* – prelucrările efectuate de un algoritm trebuie să nu fie ambiguie;
- *finitudine* – un algoritm trebuie să se termine într-un timp finit (i.e., după un număr finit de pași).

În practică, finitudinea unui algoritm nu este suficientă pentru a-i garanta utilitatea, deoarece s-ar putea ca timpul după care el se va termina să fie prea mare în raport cu cerințele noastre. Din acest motiv, un algoritm trebuie analizat și din punctul de vedere al *eficienței* sale, respectiv să verificăm dacă algoritmul se termină într-un anumit timp maxim, convenabil în practică din punctul nostru de vedere. Dacă, în plus, numărul de pași după care algoritmul se termină este minim, atunci algoritmul respectiv este *optim*.

De exemplu, dacă am calcula suma numerelor naturale mai mici sau egale decât un n dat adunând, pe rând, fiecare număr de la 1 la n am obține un algoritm care s-ar termina într-un timp finit și, în plus, ar fi eficient pentru valori "rezonabile" ale lui n . Totuși, algoritmul optim constă în calculul sumei $1 + 2 + \dots + n$ folosind formula $\frac{n(n+1)}{2}$.

În practică, eficiența unui algoritm se studiază prin prisma complexității sale computaționale, respectiv a resurselor necesare pentru executarea sa. Cele mai importante resurse luate în considerare în evaluarea complexității computaționale a unui algoritm sunt *timpul de executare* și *spațiul de memorie* necesar. Deoarece exprimarea timpului de executare în unități de timp nu este relevantă, aceasta depinzând foarte mult de configurația hardware și software a sistemului de calcul, se preferă exprimarea acestuia printr-o expresie care estimează numărul maxim de operații elementare efectuate de algoritmul respectiv în raport de dimensiunile datelor de intrare, folosind notația asimptotică $\mathcal{O}(\text{expresie})$. De exemplu, dacă un algoritm are complexitatea computațională $\mathcal{O}(n^2)$ înseamnă că dimensiunea datelor sale de intrare este egală cu n (variabila din expresie) și algoritmul efectuează aproximativ n^2 operații elementare (expresia) pentru a rezolva problema respectivă. Pentru a exprima complexitatea unui algoritm din punct de vedere al spațiului de memorie necesar se utilizează aceeași notație, însă precizând explicit acest lucru (în mod implicit, complexitatea unui algoritm se referă la timpul său de executare).

Utilizarea notației asymptotice permite o comparare simplă a performanțelor a doi sau mai mulți algoritmi care rezolvă o aceeași problema, fiind evident faptul că un algoritm este cu atât mai eficient cu cât numărul de operații elementare efectuate și, eventual, spațiul de memorie necesar este mai mic.

Operațiile elementare pe care le efectuează un algoritm sunt:

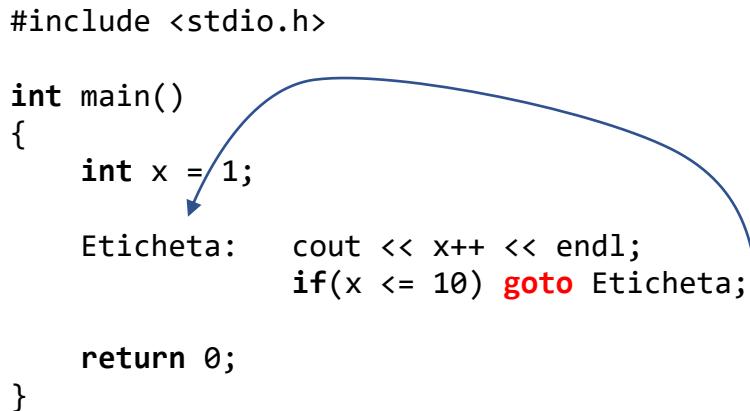
- operația de atribuire și operațiile aritmetice;
- operația de decizie și **operația de salt**;
- operațiile de citire/scriere.

În programarea actuală, de tip structurat, nu se mai permite utilizarea **operației de salt** (instrucțiunea `goto` din limbajele C/C++). Totuși, instrucțiunile repetitive sunt descompuse în astfel de instrucțiuni înapoi de a fi executate de procesor! De exemplu, o instrucțiuni repetitive poate fi simulață în limbajul C++ astfel:

```
#include <stdio.h>

int main()
{
    int x = 1;
    Etiqueta: cout << x++ << endl;
    if(x <= 10) goto Etiqueta;

    return 0;
}
```



Astfel, pentru a estima complexitatea unei instrucțiuni repetitive care execută de n ori un anumit bloc de instrucțiuni, vom considera faptul că instrucțiunea repetitive este atomică (i.e., nu vom lua în considerare numărul de operații elementare efectuate pentru executarea sa), deci complexitatea sa totală va fi $\mathcal{O}(n \cdot \text{complexitate_bloc_instrucțiuni})$:

Instrucțiune repetitive	Complexitatea totală	Exemplu
<code>for i in range(n):</code> instrucțiune cu complexitatea $\mathcal{O}(1)$	$\mathcal{O}(n)$	<code>n = int(input("n = "))</code> <code>L = []</code> <code>for i in range(n):</code> <code> L.append(i+1)</code>
<code>for i in range(n):</code> instrucțiune cu complexitatea $\mathcal{O}(m)$	$\mathcal{O}(n \cdot m)$	<code>#presupunem că len(s) = n</code> <code>#și len(t) = m</code> <code>s = input("s: ")</code> <code>t = input("t: ")</code> <code>for x in s:</code> <code> print(x, t.count(x))</code>
<code>for i in range(n):</code> instrucțiune cu complexitatea $\mathcal{O}(1)$ <code>for i in range(m):</code> instrucțiune cu complexitatea $\mathcal{O}(1)$	$\mathcal{O}(n + m)$	<code>s = input("s: ")</code> <code>t = input("t: ")</code> <code>#presupunem că len(s) = n</code> <code>#și len(t) = m</code> <code>L = []</code> <code>for x in s:</code> <code> L.append(x)</code> <code>for x in t:</code> <code> L.append(x)</code>

Instrucțiune repetitivă	Complexitatea totală	Exemplu
<pre>for i in range(n): for j in range(m): instrucțiune cu complexitatea O(1)</pre>	$O(n \cdot m)$	<pre>n = int(input("n = ")) m = int(input("m = ")) L = [] for i in range(n): for j in range(m): L.append((i, j))</pre>
<pre>for i in range(n): for j in range(m): instrucțiune cu complexitatea O(p)</pre>	$O(n \cdot m \cdot p)$	<pre>#presupunem că A este o #matrice cu n linii și m #coloane, iar L o listă #cu p elemente for x in L: print(f"Valoarea {x}: ") for i in range(n): for j in range(m): if x == A[i][j]: print(i, j)</pre>
<pre>for i in range(n): { instrucțiune cu complexitatea O(m) instrucțiune cu complexitatea O(p) }</pre>	$O(n(m + p))$	<pre>#presupunem că A este o #listă cu n elemente, B o #listă cu m elemente, iar #C o listă cu p elemente for x in A: p = B.count(x) q = C.count(x) if p >= 1 and q >= 1: print(x)</pre>

Evident, exemplele de mai sus rămân valabile pentru orice alt tip de instrucțiune repetitivă (e.g., instrucțiunea `while`), dar în exemplele prezentate am preferat utilizarea instrucțiunii repetitive `for` pentru a evidenția într-un mod simplu faptul că instrucțiunea respectivă se execută de n ori. De asemenea, în locul unei instrucțiuni cu o anumită complexitate putem considera și un bloc de instrucțiuni cu aceeași complexitate. Atenție, în limbajul Python unele dintre exemplele de mai sus pot fi scrise mult mai concis, folosind, de exemplu, secvențe de inițializare. De exemplu, primul exemplu de mai sus poate fi scris condensat astfel: `L = [i+1 for i in range(int(input("n = ")))]`. Evident, complexitatea rămâne aceeași, respectiv $O(n)$.

Practic, complexitatea computațională a unui algoritm se determină estimând numărul maxim de operații elementare efectuate în raport de dimensiunile datelor de intrare. De exemplu, complexitatea maximă a algoritmului clasic pentru calculul valorii maxime dintr-o listă formată din n valori se poate determina astfel:

Instrucțiune	Operații elementare		
<code>n = int(input("Numar elemente: "))</code>	1 afișare + 1 citire		
<code>lista = []</code>	1 atribuire		
<code>for i in range(n):</code>	de n ori:	3n operații elementare	
<code>elem = int(input("Element:"))</code>	1 afișare + 1 citire		
<code>lista.append(elem)</code>	1 atribuire		
<code>maxim = lista[0]</code>	1 atribuire		
<code>for i in range(1, n):</code>	de n-1 ori:	2(n-1) operații elementare	
<code>if lista[i] > maxim:</code>	1 operație de decizie		
<code>maxim = lista[i]</code>	1 atribuire		
<code>print("Maximul:", maxim)</code>	1 afișare		
TOTAL:	$5n + 3$ operații elementare		

În concluzie, complexitatea algoritmului din punct de vedere al timpului de executare este $\mathcal{O}(5n + 3) \approx \mathcal{O}(n)$, iar complexitatea din punct de vedere al memoriei utilizate este tot $\mathcal{O}(n)$, deoarece se memorează cele n valori într-o listă.

Expresiile utilizate în notația asymptotică a complexității computaționale se simplifică folosind următoarele două reguli (se presupune faptul că $n \rightarrow \infty$):

- constantele (multiplicative sau aditive) sunt ignorate: $\mathcal{O}(5n + 3) \approx \mathcal{O}(5n) \approx \mathcal{O}(n)$
- dintr-o expresie se păstrează doar termenul dominant: $\mathcal{O}(3n^2 + 5n + 7) \approx \mathcal{O}(3n^2) \approx \mathcal{O}(n^2)$ sau $\mathcal{O}(2^n + 3n^2) \approx \mathcal{O}(2^n)$

În continuare, vom analiza din punct de vedere al complexității computaționale mai mulți algoritmi:

a) *algoritmul de sortare prin selecția minimului*

Instructiune	Operații elementare		
<code>n = int(input("Numar elemente: "))</code>	1 afişare + 1 citire		
<code>lst = []</code>	1 atribuire		
<code>for i in range(n):</code>	de n ori:	3n operații elementare	
<code>elem = int(input("Element: "))</code>	1 afişare + 1 citire		
<code>lst.append(elem)</code>	1 atribuire		
<code>for i in range(n-1):</code>	de $(n - 1) + \dots + 1 = \frac{n(n-1)}{2}$ ori:	maxim $\frac{3n(n-1)}{2}$ operații elementare	
<code>for j in range(i+1, n):</code>			
<code>if lst[i] > lst[j]:</code>	1 operație de decizie	2 atribuiriri (maxim!)	
<code>lst[i], lst[j] = lst[j], lst[i]</code>			
<code>print("Lista sortata:")</code>	1 afişare		
<code>for i in range(n):</code>	de n ori:	n operații elementare	
<code>print(lst[i], end=" ")</code>	1 afişare		
TOTAL:		$\frac{3n^2+5n+8}{2}$ operații elementare	

În concluzie, complexitatea acestui algoritm din punct de vedere al timpului de executare este $\mathcal{O}\left(\frac{3n^2+5n+8}{2}\right) \approx \mathcal{O}(n^2)$, iar complexitatea din punct de vedere al memoriei utilizate este tot $\mathcal{O}(n)$, deoarece se memorează cele n valori într-o listă.

b) *determinarea valorilor distincte (fără duplicate) dintr-o listă L formată din n numere întregi*

O variantă de rezolvare a acestei probleme constă în utilizarea unei liste auxiliare *distincte* care să memoreze valorile distincte găsite până la un moment dat. Practic, parcurgem lista *L* element cu element și dacă elementul curent nu se găsește în lista *distincte*, atunci îl adăugăm la sfârșitul său:

```

L = [int(x) for x in input("Lista: ").split()]

distincte = []
for x in L:
    if x not in distincte:
        distincte.append(x)

print("Elementele distincte:")
print(*distincte, end=" ")

```

Se observă faptul că instrucțiunile marcate cu roșu induc complexitatea algoritmului, respectiv $\mathcal{O}(nd)$, unde d reprezintă numărul valorilor distincte din lista inițială. Deoarece complexitatea unui algoritm trebuie exprimată doar în funcție de dimensiunile datelor de intrare (iar valoarea d nu reprezintă o dimensiune a datelor de intrare!), vom aproxima complexitatea sa maximă prin $\mathcal{O}(n^2)$, care se obține când numărul valorilor distincte d este aproximativ egal cu numărul n al valorilor din lista L . De asemenea, putem observa faptul că algoritmul are complexitatea $\mathcal{O}(n)$ când numărul valorilor distincte d este mult mai mic decât n (de exemplu, d va fi egal cu 10 dacă vom considera elementele listei L ca fiind doar cifre). Complexitatea din punct de vedere al memoriei utilizate este $\mathcal{O}(n)$. Evident, un algoritm cu o complexitate medie mai bună, respectiv $\mathcal{O}(n)$, se poate obține folosind colecții de date de tip mulțime sau de tip dicționar!

c) suma cifrelor unui număr natural n

Varianta clasică de rezolvare a acestei probleme constă în adăugarea fiecărei cifre a numărului n într-o sumă și apoi eliminarea sa:

```

n = int(input("n = "))

s = 0
while n != 0:
    s = s + n % 10
    n = n // 10

print(f"Suma cifrelor: {s}")

```

Se observă faptul că numărul operațiilor elementare efectuate este proporțional cu numărul c al cifrelor numărului n , deci este $\mathcal{O}(c)$. Complexitatea unui algoritm trebuie exprimată doar în funcție de dimensiunile datelor de intrare, trebuie să calculăm numărul c în funcție de numărul n . Presupunând faptul că numărul n are c cifre, rezultă că $10^{c-1} \leq n < 10^c$. Logaritările inegalitățea în baza 10 și obținem că $\log_{10} 10^{c-1} \leq \log_{10} c < \log_{10} 10^c$, deci $c - 1 \leq \log_{10} n < c$. Din ultima inegalitate rezultă că $\lceil \log_{10} n \rceil = c - 1$, deci $c = \lceil \log_{10} n \rceil + 1$. În concluzie, complexitatea acestui algoritm este $\mathcal{O}(\lceil \log_{10} n \rceil)$. Atenție, aceeași complexitate are și următoarea variantă, mult mai concisă, specifică limbajului Python:

```
print(f"Suma cifrelor: {sum([int(x) for x in input('n = ')])}")
```

Clase uzuale de complexitate computațională

În continuare, vom prezenta clasele de complexitate computațională cele mai des întâlnite, în ordine crescătoare:

a) **clasa $\mathcal{O}(1)$ – complexitate constantă**

Exemplu: suma a două numere sau alte formule simple (e.g., rezolvarea ecuației de gradul I sau II, calculul minimului sau maximului dintre două numere etc.), adăugarea unui element la sfârșitul unei liste, determinarea numărului de elemente dintr-o colecție, accesarea unui element al unei liste prin indexul său

b) **clasa $\mathcal{O}(\log_b n)$ – complexitate logaritmică**

Exemplu: suma cifrelor unui număr natural - $\mathcal{O}(\log_{10} n)$, operația de căutare binară a unei valori într-o listă sortată cu n elemente - $\mathcal{O}(\log_2 n)$

c) **clasa $\mathcal{O}(n)$ – complexitate liniară**

Exemplu: citirea/scrierea/o singură parcurgere a unui liste cu n elemente, testarea apartenenței unei valori la o listă cu n elemente sau un sir format din n caractere, numărarea aparițiilor unei valori într-o listă cu n elemente sau într-un sir format din n caractere

d) **clasa $\mathcal{O}(n \log_2 n)$**

Exemplu: metodele de sortare *Quicksort* (sortarea rapidă), *Mergesort* (sortarea prin interclasare), *Timsort* (implementată în funcția `sort` din Python) și *Heapsort* (sortarea cu ansamblu)

e) **clasa $\mathcal{O}(n^2)$ – complexitate pătratică**

Exemplu: metoda de sortare prin interschimbare, metoda de sortare *Bubblesort*, compararea fiecărui element al unui liste cu n elemente cu toate celelalte elemente din listă, citirea/scrierea/o singură parcurgere a unui matrice pătratice de dimensiune n

f) **clasa $\mathcal{O}(n^k), k \geq 3$ – complexitate polinomială**

Exemplu: sortarea fiecărei linii dintr-o matrice pătratică de dimensiune n folosind sortarea prin interschimbare sau *Bubblesort* - $\mathcal{O}(n^3)$, algoritmul Roy-Floyd-Warshall pentru determinarea drumurilor minime într-un graf orientat ponderat - $\mathcal{O}(n^3)$

g) **clasa $\mathcal{O}(a^n), a \geq 2$ – complexitate exponențială**

Exemplu: generarea tuturor submulțimilor unei mulțimi cu n elemente - $\mathcal{O}(2^n)$, partiționarea unei mulțimi cu n elemente în două submulțimi cu aceeași sumă a elementelor - $\mathcal{O}(2^n)$

Observații:

1. Complexitatea unui algoritm NU poate fi mai mică decât complexitatea citirii datelor de intrare și scrierii datelor de ieșire! De exemplu, complexitatea minimă a oricărui algoritm de generare a tuturor submulțimilor unei mulțimi cu n elemente este $\mathcal{O}(2^n)$, deoarece, indiferent de metoda de generare a submulțimilor, trebuie afișate cele 2^n submulțimi!
2. Complexitatea unei operații poate fi mai mică decât complexitatea unui algoritm care o implementează, deoarece un algoritm presupune citirea unor date de intrare și scrierea unor date de ieșire! De exemplu, operația de căutare binară a unei valori într-o listă sortată cu n elemente are complexitatea $\mathcal{O}(\log_2 n)$, dar un algoritm care utilizează această operație utilizează presupune și citirea listei sortate crescător, deci complexitatea algoritmului va fi $\mathcal{O}(n + \log_2 n) \approx \mathcal{O}(n)$.
3. Evident, există și alte clase de complexitate relativ des întâlnite, în afara celor menționate anterior. De exemplu, există clasa $\mathcal{O}(m + n)$ – interclasarea a două liste sortate crescător având m , respectiv n elemente; clasa $\mathcal{O}(m \cdot n)$ – parcurgerea unui tablou bidimensional cu m linii și n coloane; clasa $\mathcal{O}(m \cdot n \cdot p)$ – înmulțirea unei matrice cu m linii și n coloane cu o matrice cu n linii și p coloane etc.
4. Există complexități mai mari decât cea exponentială (e.g., generarea tuturor permutărilor unei mulțimi cu n elemente are complexitatea $\mathcal{O}(n!)$), dar algoritmii cu o astfel de complexitate sunt, în realitate, inutilizabili sau, mai precis, sunt utilizabili doar pentru valori foarte mici ale lui n . De exemplu, considerând un PC pe care interpreterul limbajului Python execută aproximativ 500000 de operații elementare pe secundă, adică o operație elementară durează aproximativ $2 \cdot 10^{-6}$ secunde, generarea tuturor submulțimilor unei mulțimi cu $n = 30$ de elemente va dura aproximativ $2^{30} \cdot 2 \cdot 10^{-6}$ secunde ≈ 2140 de secunde ≈ 35 de minute, pentru $n = 50$ va dura aproximativ $2^{50} \cdot 2 \cdot 10^{-6}$ secunde ≈ 625500 ore ≈ 26062 de zile ≈ 72 de ani, iar pentru $n = 100$ de elemente va dura aproximativ $2^{100} \cdot 2 \cdot 10^{-6}$ secunde $\approx 8 \cdot 10^{16}$ ani ≈ 80 de milioane de miliarde de ani, adică o valoare de 6 milioane de ori mai mare decât vîrsta estimată a Universului!

În continuare, vom prezenta câțiva algoritmi a căror complexitate se estimează mai greu:

a)

```

n = int(input("n = "))
i = 0
p = 1
while i <= n:
    j = 1
    while j <= p:
        print(j, end= " ")
        j = j + 1
    print()
    i = i + 1
    p = p * 2

```

După o analiză superficială, acest algoritm pare să aibă complexitatea $\mathcal{O}(n \cdot p)$, datorită celor două instrucțiuni `while` imbricate. Analizând cu atenție algoritmul, vom observa faptul că, pentru fiecare valoare i cuprinsă între 0 și n , el afișează numerele de la 1 la 2^i , ceea ce înseamnă că, în total, va afișa $2^0 + 2^1 + \dots + 2^n = 2^{n+1} - 1$ numere, deci complexitatea sa este $\mathcal{O}(2^n)$, adică o complexitate exponențială! Această complexitate exponențială este indusă de faptul că variabila p își dublează valoarea la fiecare pas, deci are o creștere exponențială. Dacă variabila p ar avea o creștere liniară (e.g., $p = p + 1$), atunci, pentru fiecare valoare i cuprinsă între 0 și n , se vor afișa numerele de la 1 la $i + 1$, ceea ce înseamnă că, în total, va afișa $1 + 2 + \dots + (n + 1) = \frac{(n+1)(n+2)}{2}$ numere, deci complexitatea ar fi $\mathcal{O}(n^2)$, adică o complexitate pătratică!

b)

```
a = int(input("a = "))
b = int(input("b = "))

p = 1
while p < a:
    p = p * 2
while p <= b:
    print(p, end=" ")
    p = p * 2
```

După o analiză superficială, acest algoritm pare să aibă complexitatea $\mathcal{O}(b)$, datorită celor două instrucțiuni `while` secvențiale. Analizând cu atenție algoritmul, vom observa faptul că la fiecare pas valoarea variabilei p se dublează, deci, numărul total de iterații k pe care le va efectua algoritmul se obține rezolvând inecuația $2^k \leq b$, de unde obținem $k \leq \log_2 b$. În concluzie, acest algoritm, care afișează puterile lui 2 cuprinse între a și b , are complexitatea $\mathcal{O}(\log_2 b)$, deci o complexitate logaritmică!

c)

```
v = [int(x) for x in input("Valorile: ").split()]
n = len(v)
i = 0
j = n - 1
while i < j:
    while i < n and v[i] < 0:
        i = i + 1
    while j >= 0 and v[j] >= 0:
        j = j - 1
    if i < j:
        v[i], v[j] = v[j], v[i]
print("\nValorile:\n", v, sep="")
```

După o analiză superficială, acest algoritm pare să aibă complexitatea $\mathcal{O}(n^2)$, unde n este numărul de elemente din lista v , datorită celor două instrucțiuni `while` secvențiale imbricate în altă instrucțiune `while`. Analizând cu atenție algoritmul, vom observa faptul că, în realitate, cei 2 indici i și j nu se suprapun, ci parcurg, fiecare, o porțiune din lista v (i.e., indicele i parcurge lista de la stânga spre dreapta, iar indicele j de la dreapta spre

stânga), iar în momentul în care cei 2 indici se "întâlnesc", algoritmul se termină. Astfel, per total, cei 2 indici vor parcurge o singură dată întreaga listă v , deci complexitatea algoritmului este $\mathcal{O}(n)$, respectiv o complexitate liniară. Algoritmul realizează un fel de sortare a elementelor listei, respectiv mută valorile negative înaintea celor pozitive, fără ca valorile negative sau cele pozitive să fie sortate conform unui criteriu. Metoda de parcurgere utilizată se numește *metoda arderii lumânării la două capete (two pointers)* și, într-o formă puțin modificată, este folosită și în metoda de sortare Quicksort.

Funcții recursive

În general, prin *recursivitate* se înțelege proprietatea unor noțiuni de a se defini prin ele însese. De exemplu, numerele naturale poate fi definite recursiv folosind următoarele două axiome ale lui Peano (https://en.wikipedia.org/wiki/Peano_axioms): "Zero este un număr natural." și "Sucesorul oricărui număr natural este tot un număr natural.".

În programare, o *funcție recursivă* este o funcție care se autoapeleză, direct sau indirect.

Deoarece recursivitatea indirectă (i.e., o funcție f apelează o funcție g , iar funcția g apeleză, la rândul său, funcția f) este foarte rar utilizată în programare (de exemplu, pentru a calcula media aritmetică-geometrică: https://en.wikipedia.org/wiki/Arithmetic-geometric_mean), în continuare vom prezenta doar recursivitatea directă.

Un exemplu clasic de funcție recursivă îl reprezintă calculul factorialului unui număr natural n (i.e., $n! = 1 \cdot 2 \cdot \dots \cdot n$), folosind următoarea relație de recurență:

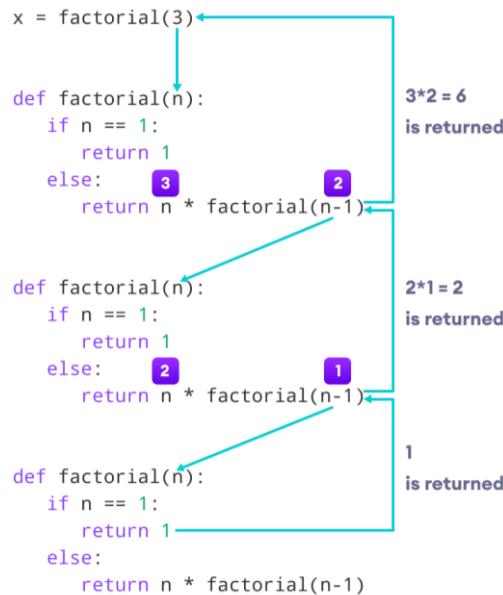
$$n! = \begin{cases} 1, & \text{dacă } n = 1 \\ n \cdot (n - 1)!, & \text{dacă } n \geq 1 \end{cases}$$

O funcție care implementează în limbajul Python relația de recurență de mai sus este următoarea:

```
def factorial(n):
    if n == 1:
        return 1
    return n * factorial(n - 1)
```

În momentul apelării unei funcții, *contextul de apel* asociat (numele, variabilele locale, parametrii, adresa de revenire etc.) se salvează în stiva alocată programului, iar în momentul terminării executării sale, contextul de apel este eliminat din stivă.

De exemplu, pentru apelul `f = factorial(3)`, stiva programului (reprezentată invers) va avea următoarea evoluție (sursa imaginii: <https://www.programiz.com/python-programming/recursion>):



Observație: Orice funcție recursivă trebuie să conțină, pe lângă componenta recurrentă (care conține autoapelurile), și o condiție de oprire (care nu conține niciun autoapel) realizabilă. În caz contrar, în momentul apelării sale, se va produce o recursivitate "infinită", care va conduce la depășirea numărului maxim de apeluri recursive permis (implicit, acesta este egal cu 1000) și la apariția unei erori. De exemplu, în cazul apelului `f = factorial(1000)`, se va afișa următoarea eroare:

The screenshot shows a Python code editor in PyCharm with the file `Test_curs.py` open. The code defines a `factorial` function that returns 1 for `n == 0` and otherwise returns `n * factorial(n - 1)`. A call to `print(factorial(1000))` is made at the end. The terminal window shows a stack trace indicating that the maximum recursion depth was exceeded. The error message is: `RecursionError: maximum recursion depth exceeded in comparison`.

```

def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)

print(factorial(1000))

```

Numărul maxim de apeluri recursive care pot fi salvate pe stivă poate fi aflat folosind metoda `getrecursionlimit` și modificat folosind metoda `setrecursionlimit`, ambele din modulul `sys`:

```

import sys

def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)

rmax = sys.getrecursionlimit()
print("Numarul maxim de apeluri recursive initial:", rmax)

sys.setrecursionlimit(5000)
rmax = sys.getrecursionlimit()
print("Numarul maxim de apeluri recursive modificat:", rmax)

n = 1000
print(str(n) + "! = " + str(factorial(n)))

```

În continuare, vom prezenta câteva exemple clasice de funcții recursive:

a) *Sirul lui Fibonacci* este definit prin următoarea relație de recurență:

$$f_n = \begin{cases} 0, & \text{dacă } n = 0 \\ 1, & \text{dacă } n = 1 \\ f_{n-2} + f_{n-1}, & \text{dacă } n \geq 2 \end{cases}$$

O implementare directă a acestei relații, care va furniza valoarea termenului de rang n a sirului lui Fibonacci, este următoarea funcție recursivă:

```

def fibo(n):
    if n <= 1:
        return n
    return fibo(n-2) + fibo(n-1)

```

Pentru $n \geq 40$ se observă faptul că timpul de executare este destul de mare și crește în raport cu valoarea lui n . În capitolul următor, dedicat complexității computaționale, se va demonstra faptul că numărul de apeluri recursive efectuate este exponențial în raport cu n , aceasta fiind cauza timpului mare de executare. O implementare eficientă se poate obține iterativ:

```

def fibo(n):
    if n <= 1:
        return n

    a, b = 0, 1
    for i in range(n):
        a, b = b, a+b
    return a

```

- b) *Algoritmul lui Euclid* permite determinarea celui mai mare divizor comun a două numere întregi nenule a și b prin împărțiri repetitive, respectiv: cât timp restul împărțirii lui a la b este nenul înlocuim a cu b și b cu restul împărțirii lui a la b , iar ultimul rest nenul obținut va fi $\text{cmmdc}(a, b)$. De exemplu, pentru $a = 120$ și $b = 18$, cel mai mare divizor comun se va calcula astfel:

a	b	$a \% b$
120	18	12
18	12	6
12	6	0
6	0	

Ultimul rest nenul este egal cu 6, deci vom obține $\text{cmmdc}(120, 18) = 6$. Se observă faptul că ultimul rest nenul este egal cu ultimul împărțitor, deci o variantă de implementare iterativă a acestui algoritm este următoarea:

```
def cmmdc(a, b):
    r = a % b
    while r != 0:
        a, b = b, r
        r = a % b
    return b
```

Analizând algoritmul, putem deduce foarte ușor următoarea formulă de recurență pentru calculul celui mai mare divizor comun a două numere întregi:

$$\text{cmmdc}(a, b) = \begin{cases} a, & \text{dacă } b = 0 \\ \text{cmmdc}(b, a \% b), & \text{dacă } b \neq 0 \end{cases}$$

Implementarea acestei relații de recurență printr-o funcție recursivă este foarte simplă:

```
def cmmdc(a, b):
    if b == 0:
        return a
    return cmmdc(b, a%b)
```

Observăm faptul că funcțiile au complexități egale, respectiv numărul de iterații este aproximativ egal cu numărul de autoapeluri.

- c) *Calculul sumei cifrelor unui număr natural* se poate realiza folosind următoarea relație de recurență:

$$sc(n) = \begin{cases} n, & \text{dacă } n < 10 \\ n \% 10 + sc(n // 10), & \text{dacă } n \geq 10 \end{cases}$$

Implementarea acestei relații de recurență printr-o funcție recursivă este banală:

```
def sc(n):
    if n < 10:
        return n
    return n%10 + sc(n//10)
```

Ce relație există între numărul de autoapeluri din varianta recursivă și numărul de iterații din varianta iterativă?

- d) *Suma elementelor dintr-o listă* se poate defini recursiv ca fiind suma dintre primul element al listei și suma elementelor din restul listei dacă lista este nevidă, respectiv 0 dacă lista este vidă:

```
def suma_lista(L):
    if len(L) == 0:
        return 0
    return L[0] + suma_lista(L[1:])
```

Într-un mod similar se poate calcula și suma elementelor strict pozitive dintr-o listă:

```
def sumapoz_lista(L):
    if len(L) == 0:
        return 0
    if L[0] > 0:
        return L[0] + sumapoz_lista(L[1:])
    else:
        return sumapoz_lista(L[1:]),
```

- e) *Frecvența unei litere într-un sir de caractere* se poate calcula recursiv astfel:

```
def frecventa(litera, sir):
    if len(sir) == 0:
        return 0
    return int(sir[0] == litera) + frecventa(litera, sir[1:]),
```

- f) Din numărul 4 se poate obține orice număr natural nenul aplicând în mod repetat următoarele operații:

1. împărțirea numărului la 2;
2. adăugarea cifrei 4 la sfârșitul numărului;
3. adăugarea cifrei 0 la sfârșitul numărului.

De exemplu, numărul 101 se poate obține prin sirul de operații $4 \rightarrow 40 \rightarrow 404 \rightarrow 202 \rightarrow 101$, iar numărul 133 prin sirul de operații $4 \rightarrow 2 \rightarrow 24 \rightarrow 12 \rightarrow 6 \rightarrow 64 \rightarrow 32 \rightarrow 16 \rightarrow 8 \rightarrow 84 \rightarrow 42 \rightarrow 424 \rightarrow 212 \rightarrow 106 \rightarrow 1064 \rightarrow 532 \rightarrow 266 \rightarrow 133$.

Pentru a afișa sirul de operații prin care se poate obține un număr natural nenul din numărul 4, vom aplica asupra sa operațiile inverse operațiilor date:

- 1'. înmulțirea numărului cu 2;
- 2'. eliminarea ultimei cifre, dacă aceasta este 4;
- 3'. eliminarea ultimei cifre, dacă aceasta este 0.

De exemplu, pentru numărul 101 vom obține următorul sir de operații $101 \rightarrow 202 \rightarrow 404 \rightarrow 40 \rightarrow 4$.

```
def numar4(n):
    if n != 4:
        if n % 10 == 0 or n%10 == 4:
            numar4(n//10)
        else:
            numar4(2*n)
            print(" ->", n, end="")
    else:
        print(4, end="")
```

Complexitatea funcțiilor recursive se exprimă în raport de numărul apelurilor efectuate pentru o anumită dimensiune a datelor de intrare. O modalitate pentru determinarea efectivă a complexității unei funcții recursive o vom prezenta în capitolul următor, dedicat tehnicii de programare *Divide et Impera*.

Încheiem prezentarea funcțiilor recursive menționând faptul că, în general, acestea consumă multă memorie (pentru salvarea contextelor de apel), sunt mai greu de depanat decât funcțiile iterative și, de multe ori, necesită un timp de executare mai mare. Din aceste motive, se recomandă utilizarea unei funcții recursive doar în cazul în care complexitatea sa computațională este echivalentă cu cea a variantei iterative, dar implementarea sa este mai simplă.

TEHNICA DE PROGRAMARE "GREEDY"

1. Prezentare generală

Tehnica de programare Greedy este utilizată, de obicei, pentru rezolvarea problemelor de optimizare, adică a celor probleme în care se cere determinarea unei submulțimi a unei mulțimi date pentru care se minimizează sau se maximizează valoarea unei funcții obiectiv. Formal, o problemă de optimizare poate fi enunțată astfel: "*Fie A o mulțime nevidă și $f: \mathcal{P}(A) \rightarrow \mathbb{R}$ o funcție obiectiv asociată mulțimii A , unde prin $\mathcal{P}(A)$ am notat mulțimea tuturor submulțimilor mulțimii A . Să se determine o submulțime $S \subseteq A$ astfel încât valoarea funcției f să fie minimă/maximă pe S (i.e., pentru orice altă submulțime $T \subseteq A, T \neq S$, valoarea funcției obiectiv f va fi cel puțin/cel mult egală cu valoarea funcției obiectiv f pe submulțimea S).*"

O problemă foarte simplă de optimizare este următoarea: "*Fie A o mulțime nevidă de numere întregi. Să se determine o submulțime $S \subseteq A$ cu proprietatea că suma elementelor sale este maximă.*". Se observă faptul că funcția obiectiv nu este dată în formă matematică și nu se precizează explicit faptul că suma elementelor submulțimii S trebuie să fie maximă în raport cu suma oricărei alte submulțimi, acest lucru subînțelegându-se. Formal, problema poate fi enunțată astfel: "*Fie $A \subseteq \mathbb{Z}, A \neq \emptyset$ și $f: \mathcal{P}(A) \rightarrow \mathbb{R}$, $f(S) = \sum_{x \in S} x$. Să se determine o submulțime $S \subseteq A$ astfel încât valoarea funcției f să fie maximă pe S , i.e. $\forall T \subseteq A, T \neq S \Rightarrow f(T) \leq f(S)$ sau, echivalent, $\forall T \subseteq A, T \neq S \Rightarrow \sum_{x \in T} x \leq \sum_{x \in S} x$.*"

Evident, orice problemă de acest tip poate fi rezolvată prin *metoda forței brute*, astfel: se generează, pe rând, toate submulțimile S ale mulțimii A și pentru fiecare dintre ele se calculează $f(S)$, iar dacă valoarea obținută este mai mică/mai mare decât minimul/maximul obținut până în acel moment, atunci se actualizează minimul/maximul și se reține submulțimea S . Deși aceasta rezolvare este corectă, ea are o complexitate exponențială, respectiv $\mathcal{O}(2^{|A|})$!

Tehnica de programare Greedy încearcă să rezolve problemele de optimizare adăugând în submulțimea S , la fiecare pas, cel mai bun element disponibil din mulțimea A din punct de vedere al optimizării funcției obiectiv. Practic, metoda Greedy încearcă să găsească optimul global al funcției obiectiv combinând optimele sale locale. Totuși, prin combinarea unor optime locale nu se obține întotdeauna un optim global! De exemplu, să considerăm cel mai scurt drum posibil dintre București și Arad (un optim local), precum și cel mai scurt drum posibil dintre Arad și Ploiești (alt optim local). Combinând cele două optime locale nu vom obține un optim global, deoarece, evident, cel mai scurt drum de la București la Ploiești nu trece prin Arad! Din acest motiv, aplicare tehnicii de programare Greedy pentru rezolvarea unei probleme trebuie să fie însotită de o demonstrație a corectitudinii (optimalității) criteriului de selecție pe care trebuie să-l îndeplinească un element al mulțimii A pentru a fi adăugat în soluția S .

De exemplu, să considerăm *problema plății unei sume folosind un număr minim de monede*. O rezolvare de tip Greedy a acestei probleme ar putea consta în utilizarea, la fiecare pas, a unui număr maxim de monede cu cea mai mare valoare admisibilă. Astfel, pentru monede cu valorile de 8\$, 7\$ și 5 \$, o sumă de 23\$ va fi plătită în următorul mod: $23\$ = 2*8\$ + 7\$ = 2*8\$ + 1*7\$$, deci se vor utiliza 3 monede, ceea ce reprezintă o soluție optimă. Dacă vom considera monede cu valorile de 8\$, 7\$ și 1 \$, o sumă de 14\$ va fi plătită în următorul mod: $14\$ = 1*8\$ + 6\$ = 1*8\$ + 6*1\$$, deci se vor utiliza 7 monede, ceea ce nu reprezintă o soluție optimă (i.e., $2*7\$$). Mai mult, pentru monede cu 8\$, 7\$ și 5 \$, o sumă de 14\$ nu va putea fi plătită deloc: $14\$ = 1*8\$ + 6\$ = 1*8\$ + 1*5\$ + 1\$$, deoarece restul rămas, de 1\$, nu mai poate fi plătit (evident, soluția optimă este $2*7\$$). În concluzie, acest algoritm de tip Greedy, numit *algoritmul casierului*, nu furnizează întotdeauna o soluție optimă pentru plata unei sume folosind un număr minim de monede. Totuși, pentru anumite valori ale monedelor, el poate furniza o soluție optimă pentru orice sumă dată (de exemplu, pentru monedele din Statele Unite ale Americii: <https://personal.utdallas.edu/~sxb027100/cs6363/coin.pdf>)

Revenind la problema determinării unei submulțimi S cu sumă maximă, observăm faptul că aceasta trebuie să conțină toate elementele pozitive din mulțimea A , deci criteriul de selecție este ca elementul curent din A să fie pozitiv (demonstrația optimalității este banală). Dacă mulțimea A nu conține niciun număr pozitiv, care va fi soluția problemei?

În anumite probleme, criteriul de selecție poate fi aplicat mai eficient dacă se realizează o prelucrare inițială a elementelor mulțimii A – de obicei, o sortare a lor. De exemplu, să considerăm următoarea problemă: "Fie A o mulțime nevidă formată din n numere întregi. Să se determine o submulțime $S \subseteq A$ având exact k elemente ($k \leq n$) cu proprietatea că suma elementelor sale este maximă". Evident, submulțimea S trebuie să conțină cele mai mari k elemente ale mulțimii A , iar acestea pot fi selectate în două moduri:

- de k ori se selectează maximul din mulțimea A și se elimină (sau doar se marchează – important este ca, la fiecare pas, să nu mai luăm în considerare maximul determinat anterior), deci această soluție va avea complexitatea $\mathcal{O}(kn)$, care oscilează între $\mathcal{O}(n)$ pentru valori ale lui k mult mai mici decât n și $\mathcal{O}(n^2)$ pentru valori ale lui k apropiate de n ;
- sortăm crescător elementele mulțimii A și apoi selectăm ultimele k elemente, deci această soluție va avea complexitatea $\mathcal{O}(k + n \log_2 n) \approx \mathcal{O}(n \log_2 n)$, care nu depinde de valoarea k .

În plus, a doua variantă de implementare are avantajul unei implementări mai simple decât prima.

În concluzie, pentru o mulțime A cu n elemente, putem considera următoarea formă generală a unui algoritm de tip Greedy:

```
prelucrarea inițială a elementelor mulțimii A
S = []
for x in A:
    if elementul x verifică criteriul de selecție:
        S.append(x)
afișarea elementelor mulțimii S
```

Se observă faptul că, de obicei, un algoritm de acest tip are o complexitate relativ mică, de tipul $\mathcal{O}(n \log_2 n)$, dacă prin sortarea (prelucrarea) elementelor mulțimii A cu complexitatea $\mathcal{O}(n \log_2 n)$ se poate ulterior testa criteriul de selecție în $\mathcal{O}(1)$. Dacă nu se realizează prelucrarea inițială a elementelor mulțimii A , atunci algoritmul (care trebuie puțin adaptat) va avea complexități de tipul $\mathcal{O}(n)$ sau $\mathcal{O}(n^2)$, induse de complexitatea verificării criteriului de selecție. Evident, acestea nu sunt toate complexitățile posibile pentru un algoritm de tip Greedy, ci doar sunt cele mai des întâlnite!

2. Minimizarea timpului mediu de așteptare

La un ghișeu, stau la coadă n persoane p_1, p_2, \dots, p_n și pentru fiecare persoană p_i se cunoaște timpul său de servire t_i . Să se determine o modalitate de reașezare a celor n persoane la coadă, astfel încât timpul mediu de așteptare să fie minim.

De exemplu, să considerăm faptul că la ghișeu stau la coadă $n = 6$ persoane, având timpii de servire $t_1 = 7, t_2 = 6, t_3 = 5, t_4 = 10, t_5 = 6$ și $t_6 = 4$. Evident, pentru ca o persoană să fie servită, aceasta trebuie să aștepte ca toate persoanele aflate înaintea sa la coadă să fie servite, deci timpii de așteptare ai celor 6 persoane vor fi următorii:

Persoana	Timpul de servire (t_i)	Timp de așteptare (a_i)
p_1	7	7
p_2	6	$7 + 6 = 13$
p_3	3	$13 + 3 = 16$
p_4	10	$16 + 10 = 26$
p_5	6	$26 + 6 = 32$
p_6	3	$32 + 3 = 35$
Timpul mediu de așteptare (M):		$\frac{7 + 13 + 16 + 26 + 32 + 35}{6} = \frac{129}{6} = 21.5$

Deoarece timpul de servire al unei persoane influențează timpii de așteptare ai tuturor persoanelor aflate după ea la coadă, se poate intui foarte ușor faptul că minimizarea

timpului mediu de așteptare se obține rearanjând persoanele la coadă în ordinea crescătoare a timpilor de servire:

Persoana	Timpul de servire (t_i)	Timp de așteptare (a_i)
p_3	3	3
p_6	3	$3 + 3 = 6$
p_2	6	$6 + 6 = 12$
p_5	6	$12 + 6 = 18$
p_1	7	$18 + 7 = 25$
p_4	10	$25 + 10 = 35$
Timpul mediu de așteptare (M):		$\frac{3 + 6 + 12 + 18 + 25 + 35}{6} = \frac{99}{6} = 16.5$

Practic, minimizarea timpului mediu de așteptare este echivalentă cu minimizarea timpului de așteptare al fiecarei persoane, iar minimizarea timpului de așteptare al unei persoane se obține minimizând timpii de servire ai persoanelor aflate înaintea sa!

Pentru a demonstra mai simplu corectitudinea algoritmului, mai întâi vom renumera persoanele $p_1, p_2, \dots, p_i, \dots, p_j, \dots, p_n$ în ordinea crescătoare a timpilor de servire, astfel încât vom avea $t_1 \leq t_2 \leq \dots \leq t_i \leq \dots \leq t_j \leq \dots \leq t_n$. De asemenea, vom presupune faptul că timpii individuali de servire t_1, t_2, \dots, t_n nu sunt toți egali între ei (în acest caz, problema ar fi trivială), deci există $i < j$ astfel încât $t_i < t_j$. În continuare, presupunem faptul că această modalitate P_1 de aranjare a persoanelor la coadă (o permutare, de fapt) nu este optimă, deci există o altă modalitate optimă P_2 de aranjare $p_1, p_2, \dots, p_j, \dots, p_i, \dots, p_n$ diferită de cea inițială, în care $t_j > t_i$ (practic, am interschimbat persoanele p_i și p_j din varianta inițială, adică persoana p_j se află acum pe poziția i în coadă, iar persoana p_i se află acum pe poziția j , unde $i < j$).

În cazul primei modalități de aranjare P_1 , timpul mediu de servire M_1 este egal cu:

$$\begin{aligned} M_1 &= \frac{t_1 + (t_1 + t_2) + \dots + (t_1 + \dots + t_i) + \dots + (t_1 + \dots + t_j) + \dots + (t_1 + \dots + t_n)}{n} = \\ &= \frac{nt_1 + (n-1)t_2 + \dots + (n-i+1)t_i + \dots + (n-j+1)t_j + \dots + 2t_{n-1} + t_n}{n} \end{aligned}$$

În cazul celei de-a doua modalități de aranjare P_2 , timpul mediu de servire M_2 este egal cu:

$$\begin{aligned} M_2 &= \frac{t_1 + (t_1 + t_2) + \dots + (t_1 + \dots + t_j) + \dots + (t_1 + \dots + t_i) + \dots + (t_1 + \dots + t_n)}{n} = \\ &= \frac{nt_1 + (n-1)t_2 + \dots + (n-i+1)t_j + \dots + (n-j+1)t_i + \dots + 2t_{n-1} + t_n}{n} \end{aligned}$$

Comparăm acum M_1 cu M_2 , calculând diferența dintre ele:

$$\begin{aligned} M_1 - M_2 &= \frac{(n-i+1)t_i + (n-j+1)t_j - (n-i+1)t_j - (n-j+1)t_i}{n} = \\ &= \frac{t_i(n-i+1-n+j-1) + t_j(n-j+1-n+i-1)}{n} = \\ &= \frac{t_i(-i+j) + t_j(-j+i)}{n} = \frac{-t_i(i-j) + t_j(i-j)}{n} = \frac{(t_j - t_i)(i-j)}{n} \end{aligned}$$

Deoarece $i < j$ și $t_j > t_i$, obținem faptul că $M_1 - M_2 = \frac{(t_j - t_i)(i-j)}{n} < 0$ (evident, $n \geq 1$), ceea ce implică $M_1 < M_2$. Acest fapt contrazice optimalitatea modalității de aranjare P_2 , deci presupunerea că modalitatea de aranjare P_1 (în ordinea crescătoare a timpilor de servire) nu ar fi optimă este falsă!

Atenție, soluția acestei probleme constă într-o rearanjare a persoanelor p_1, p_2, \dots, p_n , deci în implementarea acestui algoritm nu este suficient să sortăm crescător timpii de servire, ci trebuie să memorăm perechi de forma (p_i, t_i) , folosind, de exemplu, un tuplu, iar apoi să le sortăm crescător după componenta t_i .

```
# functie folosita pentru sortarea crescătoare a persoanelor
# în raport de timpii de servire (cheia)
def cheieTimpServire(t):
    return t[1]

# funcția afișează, într-un format tabelar, timpii de servire
# și timpii de așteptare ai persoanelor
# ts = o listă cu timpii individuali de servire
def afisareTimp(ts):
    print("Persoana\tTimp de servire\tTimp de așteptare")
    # timpul de așteptare al persoanei curente
    tcrt = 0
    # timpul total de așteptare
    ttotal = 0
    for t in ts:
        tcrt = tcrt + t[1]
        ttotal = ttotal + tcrt
        print(str(t[0]).center(len("Persoana")),
              str(t[1]).center(len("Timp de servire")),
              str(tcrt).center(len("Timp de așteptare")), sep="\t")
    print("Timpul mediu de așteptare:", round(ttotal/len(ts), 2))

# timpii de servire ai persoanelor se citesc de la tastatură
aux = [int(x) for x in input("Timpii de servire: ").split()]
# asociem fiecărui timp de servire numărul de ordine al persoanei
tis = [(i+1, aux[i]) for i in range(len(aux))]
```

```

print("Varianta inițială:")
afisareTimpI(tis)

# sortăm persoanele în ordinea crescătoare a timpilor de servire
tis.sort(key=cheieTimpServire)

print("\nVarianta optimă:")
afisareTimpI(tis)

```

Evident, complexitatea algoritmului este dată de complexitatea operației de sortare utilizate, deci complexitatea sa, optimă, este $\mathcal{O}(n \log_2 n)$.

Încheiem prezentarea acestei probleme precizând faptul că este o problemă de planificare, forma sa generală fiind următoarea: "Se consideră n activități cu duratele t_1, t_2, \dots, t_n care partajează o resursă comună. Știind faptul că activitățile trebuie efectuate sub excludere reciprocă (respectiv, la un moment dat, resursa comună poate fi alocată unei singure activități), să se determine o modalitate de planificare a activităților astfel încât timpul mediu de așteptare să fie minim".

3. Planificarea optimă a unor spectacole într-o singură sală

Considerăm n spectacole S_1, S_2, \dots, S_n pentru care cunoaștem intervalele lor de desfășurare $[s_1, f_1], [s_2, f_2], \dots, [s_n, f_n]$, toate dintr-o singură zi. Având la dispoziție o singură sală, în care putem să planificăm un singur spectacol la un moment dat, să se determine numărul maxim de spectacole care pot fi planificate fără suprapuneri. Un spectacol S_j poate fi programat după spectacolul S_i dacă $s_j \geq f_i$.

De exemplu, să considerăm $n = 7$ spectacole având următoarele intervale de desfășurare:

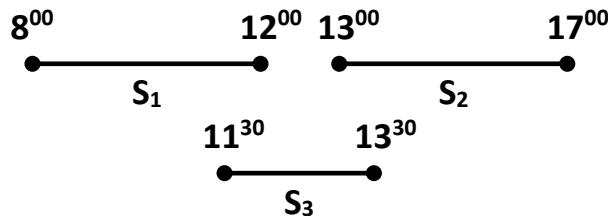
$S_1: [10^{00}, 11^{20})$
 $S_2: [09^{30}, 12^{10})$
 $S_3: [08^{20}, 09^{50})$
 $S_4: [11^{30}, 14^{00})$
 $S_5: [12^{10}, 13^{10})$
 $S_6: [14^{00}, 16^{00})$
 $S_7: [15^{00}, 15^{30})$

Se observă faptul că numărul maxim de spectacole care pot fi planificate este 4, iar o posibilă soluție este S_3, S_1, S_5 și S_7 . Atenție, soluția nu este unică (de exemplu, o altă soluție optimă este S_3, S_1, S_5 și S_6)!

Deoarece dorim să găsim o rezolvare de tip Greedy a acestei probleme, vom încerca planificarea spectacolelor folosind unul dintre următoarele criterii:

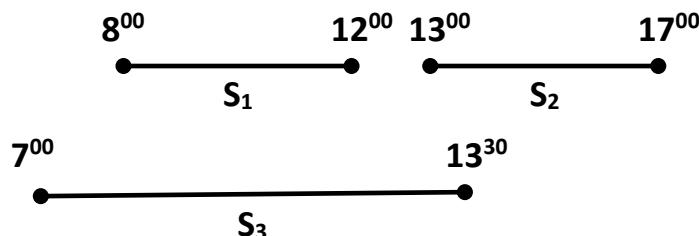
- în ordinea crescătoare a duratelor;
- în ordinea crescătoare a orelor de început;
- în ordinea crescătoare a orelor de terminare.

În cazul utilizării criteriului a), se observă ușor faptul că nu vom obține întotdeauna o soluție optimă. De exemplu, să considerăm următoarele 3 spectacole:



Aplicând criteriul a), vom planifica prima dată spectacolul S_3 (deoarece durează cel mai puțin), iar apoi nu vom mai putea planifica nici spectacolul S_1 și nici spectacolul S_2 , deoarece ambele se suprapun cu spectacolul S_3 , deci vom obține o planificare formată doar din S_3 . Evident, planificarea optimă, cu număr maxim de spectacole, este S_1 și S_2 .

De asemenea, în cazul utilizării criteriului b), se observă ușor faptul că nu vom obține întotdeauna o soluție optimă. De exemplu, să considerăm următoarele 3 spectacole:



Aplicând criteriul b), vom planifica prima dată spectacolul S_3 (deoarece începe primul), iar apoi nu vom mai putea planifica nici spectacolul S_1 și nici spectacolul S_2 , deoarece ambele se suprapun cu el, deci vom obține o planificare formată doar din S_3 . Evident, planificarea optimă este S_1 și S_2 .

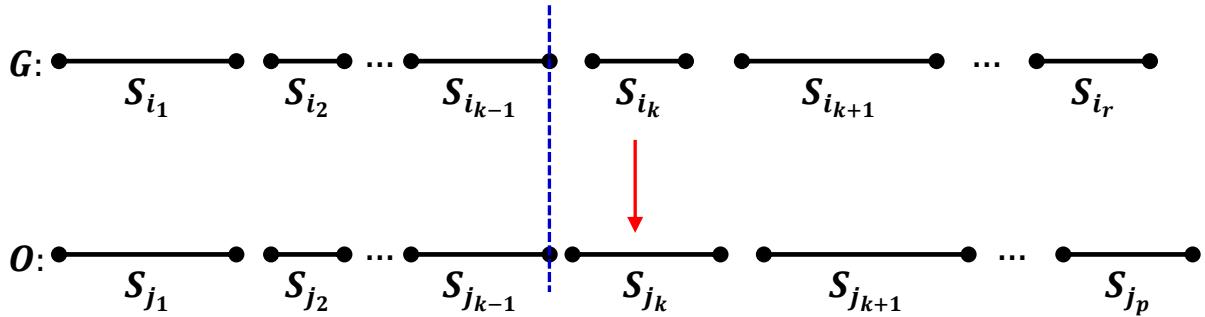
În cazul utilizării criteriului c), se observă faptul că vom obține soluțiile optime în ambele exemple prezentate mai sus:

- în primul exemplu, vom planifica mai întâi spectacolul S_1 (deoarece se termină primul), apoi nu vom putea planifica spectacolul S_3 (deoarece se suprapune cu S_1), dar vom putea planifica spectacolul S_2 , deci vom obține planificarea optimă formată din S_1 și S_2 ;
- în al doilea exemplu, vom proceda la fel și vom obține planificarea optimă formată din S_1 și S_2 .

Practic, criteriul c) este o combinație a criteriilor a) și b), deoarece un spectacol care durează puțin și începe devreme se va termina devreme!

Pentru a demonstra optimalitatea criteriului c) de selecție, vom utiliza o demonstrație de tipul *exchange argument*: vom considera o soluție optimă furnizată de un algoritm oarecare (nu contează metoda utilizată!), diferită de soluția furnizată de algoritmul de tip Greedy, și vom demonstra faptul că aceasta poate fi transformată, element cu element, în soluția furnizată de algoritmul de tip Greedy. Astfel, vom demonstra faptul că și soluția furnizată de algoritmul de tip Greedy este tot optimă!

Fie G soluția furnizată de algoritmul de tip Greedy și o soluție optimă O , diferită de G , obținută folosind orice alt algoritm:



Deoarece soluția optimă O este diferită de soluția Greedy G , rezultă că există un cel mai mic indice k pentru care $S_{i_k} \neq S_{j_k}$. Practic, este posibil ca ambii algoritmi pot să selecteze, până la pasul $k - 1$, aceleasi spectacole în aceeași ordine, adică $S_{i_1} = S_{j_1}, \dots, S_{i_{k-1}} = S_{j_{k-1}}$. Spectacolul S_{j_k} din soluția optimă O poate fi înlocuit cu spectacolul S_{i_k} din soluția Greedy G fără a produce o suprapunere, deoarece:

- spectacolul S_{i_k} începe după spectacolul $S_{j_{k-1}}$, deoarece spectacolul S_{i_k} a fost programat după spectacolul $S_{i_{k-1}}$ care este identic cu spectacolul $S_{j_{k-1}}$, deci $s_{i_k} \geq f_{i_{k-1}} = f_{j_{k-1}}$;
- spectacolul S_{j_k} se termină după spectacolul S_{i_k} , adică $f_{j_k} \geq f_{i_k}$, deoarece, în caz contrar ($f_{j_k} < f_{i_k}$) algoritmul Greedy ar fi selectat spectacolul S_{j_k} în locul spectacolului S_{i_k} ;
- spectacolul S_{i_k} se termină înaintea spectacolului $S_{j_{k+1}}$, adică $f_{i_k} \leq s_{j_{k+1}}$, deoarece am demonstrat anterior faptul că $f_{i_k} \leq f_{j_k}$ și $f_{j_k} \leq s_{j_{k+1}}$ (deoarece spectacolul $S_{j_{k+1}}$ a fost programat după spectacolul S_{j_k}).

Astfel, am demonstrat faptul că $f_{j_{k-1}} \leq s_{i_k} < f_{j_k} \leq s_{j_{k+1}}$, ceea ce ne permite să înlocuim spectacolul S_{j_k} din soluția optimă O cu spectacolul S_{i_k} din soluția Greedy G fără a produce o suprapunere. Repetând raționamentul anterior, putem transforma primele r elemente din soluția optimă O în soluția G furnizată de algoritmul Greedy.

Pentru a încheia demonstrația, trebuie să mai demonstrăm faptul că ambele soluții conțin același număr de spectacole, respectiv $r = p$. Presupunem prin absurd faptul că $r \neq p$. Deoarece soluția O este optimă, rezultă faptul că $p > r$ (altfel, dacă $p < r$, ar însemna că soluția optimă O conține mai puține spectacole decât soluția Greedy G , ceea ce i-ar contrazice optimalitatea), deci există cel puțin un spectacol $S_{j_{r+1}}$ în soluția optimă

O care nu a fost selectat în soluția Greedy G . Acest lucru este imposibil, deoarece am demonstrat anterior faptul că orice spectacol S_{j_k} din soluția optimă se termină după spectacolul S_{i_k} aflat pe aceeași poziție în soluția Greedy (adică $f_{j_k} \geq f_{i_k}$), deci am obținut relația $f_{i_r} \leq f_{j_r} \leq s_{j_{r+1}}$, ceea ce ar însemna că spectacolul $S_{j_{r+1}}$ ar fi trebuit să fie selectat și în soluția Greedy G ! În concluzie, presupunerea că $r \neq p$ este falsă, deci $r = p$.

Astfel, am demonstrat faptul că putem transforma soluția optimă O în soluția G furnizată de algoritm Greedy, deci și soluția furnizată de algoritm Greedy este optimă!

În concluzie, algoritmul Greedy pentru rezolvarea problemei programării spectacolelor este următorul:

- sortăm spectacolele în ordinea crescătoare a orelor de terminare;
- planificăm primul spectacol (problema are întotdeauna soluție!);
- pentru fiecare spectacol rămas, verificăm dacă începe după ultimul spectacol programat și, în caz afirmativ, îl planificăm și pe el.

Citirea datelor de intrare are complexitatea $\mathcal{O}(n)$, sortarea are complexitatea $\mathcal{O}(n \log_2 n)$, programarea primului spectacol are complexitatea $\mathcal{O}(1)$, testarea spectacolelor rămase are complexitatea $\mathcal{O}(n - 1)$, iar afișarea planificării optime are cel mult complexitatea $\mathcal{O}(n)$, deci complexitatea algoritmului este $\mathcal{O}(n \log_2 n)$.

În continuare, vom prezenta implementarea algoritmului în limbajul Python:

```
# functie folosita pentru sortarea crescătoare a spectacolelor
# în raport de ora de sfârșit (cheia)
def cheieOraSfârșit(sp):
    return sp[2]

# citim datele de intrare din fișierul text "spectacole.txt"
fin = open("spectacole.txt")
# lsp = lista spectacolelor, fiecare spectacol fiind memorat
# sub forma unui tuplu (ID, ora de început, ora de sfârșit)
lsp = []
crt = 1
for linie in fin:
    aux = linie.split("-")
    # aux[0] = ora de început a spectacolului curent
    # aux[1] = ora de sfârșit a spectacolului curent
    lsp.append((crt, aux[0].strip(), aux[1].strip()))
    crt = crt + 1
fin.close()

# sortăm spectacolele în ordinea crescătoare a timpilor de sfârșit
lsp.sort(key=cheieOraSfârșit)
```

```

# posp = o listă care conține o programare optima a spectacolelor,
# inițializată cu primul spectacol
posp = [lsp[0]]
# parcurgem restul spectacolelor
for sp in lsp[1:]:
    # dacă spectacolul curent începe după ultimul spectacol
    # programat, atunci îl programăm și pe el
    if sp[1] >= posp[len(posp)-1][2]:
        posp.append(sp)

# scriem datele de ieșire în fișierul text "programare.txt"
fout = open("programare.txt", "w")
fout.write("Numarul maxim de spectacole: "+str(len(posp))+"\n")
fout.write("\nSpectacolele programate:\n")
for sp in posp:
    fout.write(sp[1]+" - "+sp[2]+ " Spectacol "+str(sp[0])+"\n")
fout.close()

```

Pentru exemplul de mai sus, fișierele text de intrare și de ieșire sunt următoarele:

spectacole.txt	programare.txt
10:00-11:20	Numarul maxim de spectacole: 4
09:30-12:10	
08:20-09:50	Spectacolele programate:
11:30-14:00	08:20-09:50 Spectacol 3
12:10-13:10	10:00-11:20 Spectacol 1
14:00-16:00	12:10-13:10 Spectacol 5
15:00-15:30	15:00-15:30 Spectacol 7

Încheiem prezentarea acestei probleme precizând faptul că este tot o problemă de planificare, forma sa generală fiind următoarea: "*Se consideră n activități pentru care se cunosc intervalele orare de desfășurare și care partajează o resursă comună. Știind faptul că activitățile trebuie efectuate sub excludere reciprocă (respectiv, la un moment dat resursa comună poate fi alocată unei singure activități), să se determine o modalitate de planificare a unui număr maxim de activități care nu se suprapun.*".