

CURS 4

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*.

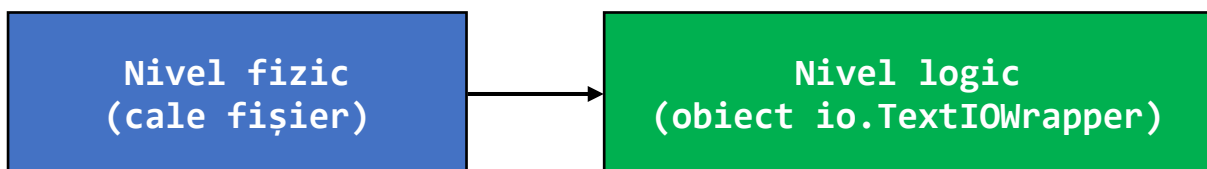
Liniile dintr-un fișier text sunt demarcate folosind caracterele **CR** (*carriage return* sau `'\r'` sau `chr(13)`) și **LF** (*line feed* sau `'\n'` sau `chr(10)`), astfel:

- 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 linii 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 fișierului 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 opțional):

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

Primul parametru al funcției reprezintă calea fișierului, sub forma unui șir 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 `FileNotFoundError`;
- **"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`, așa 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 `FileNotFoundError` 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>	<p>Ana are multe pere,</p> <p>mere rosii,</p> <p>si mai multe prune!!!</p>	<p>Ana are multe pere,</p> <p>mere rosii,</p> <p>si mai multe prune!!!</p>

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>	<p>Ana are multe pere,</p> <p>mere rosii,</p> <p>si mai multe prune!!!</p>	<p>Ana are multe pere,</p> <p>mere rosii,</p> <p>si mai multe prune!!!</p>

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 parcurgere 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>	<p>Ana are multe pere,</p> <p>mere rosii,</p> <p>si mai multe prune!!!</p>	<p>Ana are multe pere,</p> <p>mere rosii,</p> <p>si mai multe prune!!!</p>

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

Exemplu:

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

Observați faptul că fiecare șir 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 șiruri de caractere, dacă vrem să citim datele respective sub forma unor numere trebuie să utilizăm funcții pentru manipularea șirurilor 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 șirul 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 șirul 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 șiruri 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 șiruri)**: scrie toate șirurile 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>	Anaaremere!

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, fie să modificăm `f.writelines(cuvinte)` în `f.writelines("\n".join(cuvinte))`, deci să concatenăm toate șirurile din listă într-unul singur, folosind caracterul `"\n"` ca separator.

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, așa 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 si
# calculam 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
# si scriem suma la sfarsitul primei linii

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

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/inițializată 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 inițializare (*list comprehensions*):

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

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

# secvențe de inițializare 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 șirurile de caractere:

a) *prin indici pozitivi sau negativi*

În limbajul Python, oricărei secvențe (*mulțime iterabilă*) de lungime n îi sunt asociați 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 asociați următorii indici:

	0	1	2	3	4	5	6	7	8	9
L	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 sublistă 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 relaționali: <, <=, >, >=, ==, !=*

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 șirul "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 șir 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!)

Example:

```
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 opțional `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`. Așa cum am precizat anterior, listele sunt *mutabile*, deci metodele respective pot modifica lista curentă, dacă acest lucru este necesar!

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

a) **`count(valoare)`**: furnizează numărul de apariții 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.

Exemple:

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.

Exemple:

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 ș.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 opțional `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 indecși) 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 timpi 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 noua 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):

- a) 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 șir de caractere și apoi se extrag numerele din șirul 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}")
```

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 linii: "))
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 linii: "))
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 linii
n = 5    #numarul de coloane

# variantă incorectă: 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

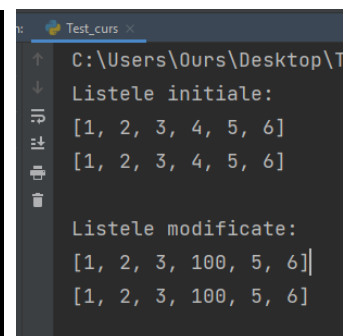
În multe situații dorim să realizăm o copie a unei liste, de exemplu pentru a-i păstra conținutul înainte de o anumită modificare. O variantă greșită de realizare a acestei operații constă în utilizarea unei instrucțiuni de atribuire, așa cum se poate observa în exemplul următor:

```
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")
```



```
Test_curs
C:\Users\Ours\Desktop\T
Listele initiale:
[1, 2, 3, 4, 5, 6]
[1, 2, 3, 4, 5, 6]

Listele modificate:
[1, 2, 3, 100, 5, 6]
[1, 2, 3, 100, 5, 6]
```

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 amândurora.

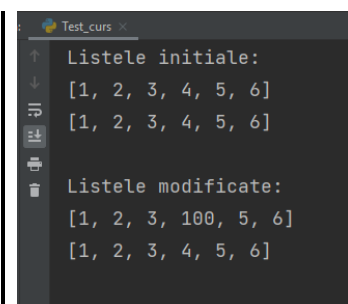
O prima variantă de realizare corectă a unei copii a unei liste o constituie utilizarea metodei `copy` din clasa `list`:

```
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")
```



```
Test_curs
Listele initiale:
[1, 2, 3, 4, 5, 6]
[1, 2, 3, 4, 5, 6]

Listele modificate:
[1, 2, 3, 100, 5, 6]
[1, 2, 3, 4, 5, 6]
```

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, așa cum se poate observa din exemplul următor:

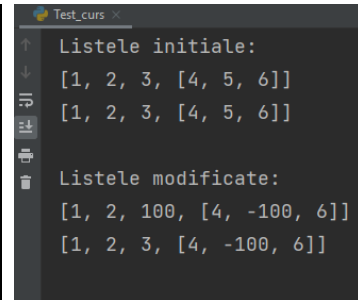
```

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")

```



```

Test_curs x
Listele initiale:
[1, 2, 3, [4, 5, 6]]
[1, 2, 3, [4, 5, 6]]

Listele modificate:
[1, 2, 100, [4, -100, 6]]
[1, 2, 3, [4, -100, 6]]

```

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

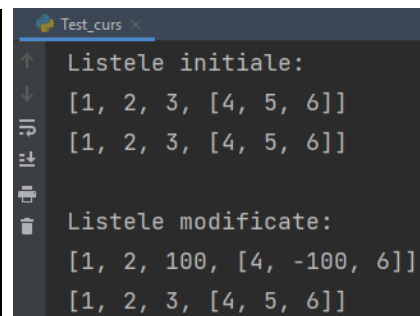
```

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")

```



```

Test_curs x
Listele initiale:
[1, 2, 3, [4, 5, 6]]
[1, 2, 3, [4, 5, 6]]

Listele modificate:
[1, 2, 100, [4, -100, 6]]
[1, 2, 3, [4, 5, 6]]

```

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!