

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/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}")
```

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 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
```

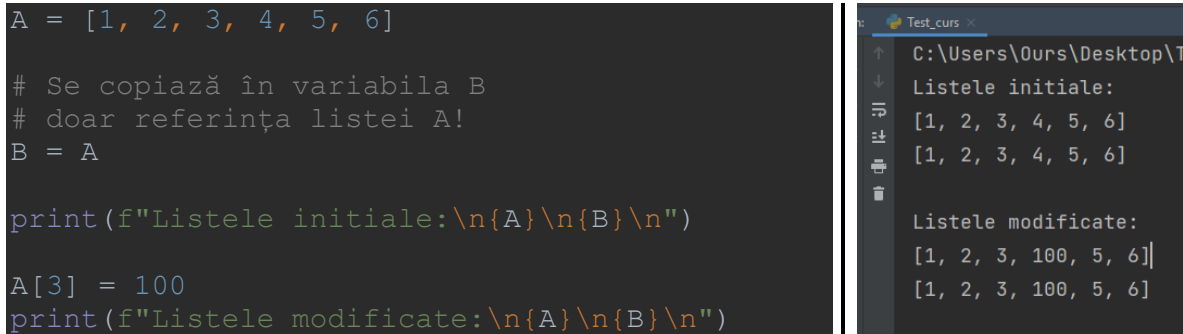
- c) se citesc, pe rând, liniile tabloului dimensional până când se introduce o linie vidă (elementele unei linii vor fi introduse despărțite între ele prin câte un spațiu):

```
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:

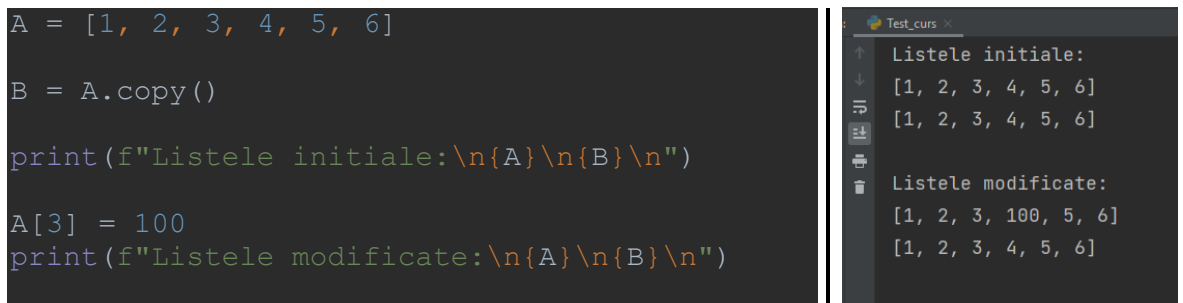
```
[1, 2, 3, 4, 5, 6]
[1, 2, 3, 4, 5, 6]
```

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

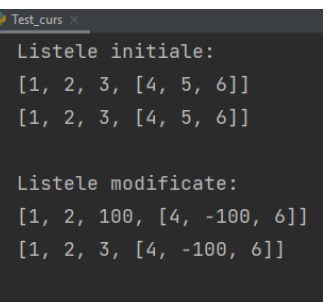
Listele initiale:

```
[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

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]]
```

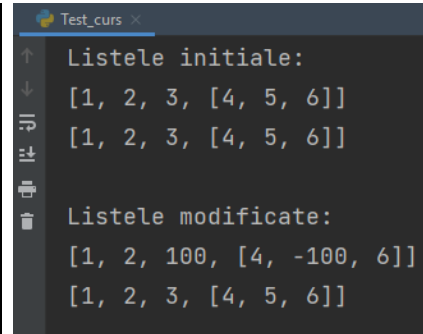
Pentru a rezolva problema anterioară, vom utiliza metoda `deepcopy` din modulul `copy`, care va realiza o copie în adâncime (*deep copy*) a listei A:

```
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!

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/inițializat 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)
```

```
#initalizare 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 șiruri de caractere și liste:

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

	0	1	2	3	4	5	6	7	8	9
T	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 tuple

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 relaționali: <, <=, >, >=, ==, !=*

Observație: În cazul primilor 4 operatori, cele două tuple 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 șirul "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 șir 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 opțional `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.

Exemple:

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 tuplu format 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 executare, 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 = ()
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

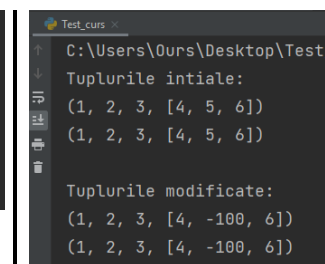
Se observă faptul că primele două variante au timpi de executare foarte buni, în timp ce ultimele două variante au timpi de executare mult mai mari, din cauza faptului că la fiecare operație de concatenare a tuplului (x,) la tuplul curent se creează în memorie o copie a tuplului curent, se adaugă la sfârșitul copiei noua 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 initiale:", a, b, sep="\n")

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



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

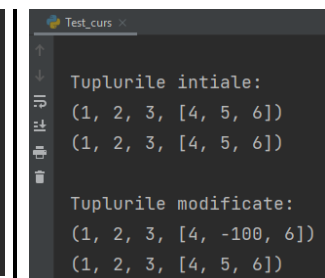
Tuplurile modificate:
(1, 2, 3, [4, -100, 6])
(1, 2, 3, [4, -100, 6])
```

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 initiale:", a, b, sep="\n")

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



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

Tuplurile modificate:
(1, 2, 3, [4, -100, 6])
(1, 2, 3, [4, 5, 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., șir de caractere, listă sau tuplu), așa cum se poate observa din exemplele următoare:

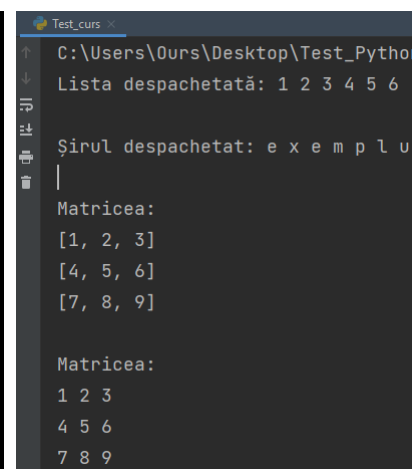
```
lista = [1, 2, 3, 4, 5, 6]
print("Lista despachetată:", *lista)

sir = "exemplu"
print("\nȘirul 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)
```



```
Test_curs
C:\Users\Ours\Desktop\Test_Pytho
Lista despachetată: 1 2 3 4 5 6

Șirul 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
```