

CURS 3

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)
```

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!**)

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 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, elementele 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 bidimensional 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

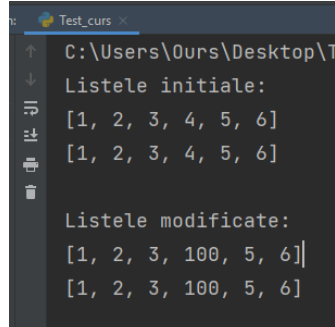
În multe situații dorim să realizăm o copie a unei liste, de exemplu pentru a-i păstra conținutul înainte de o anumită modificare. O variantă greșită de realizare a acestei operații constă în utilizarea unei instrucțiuni de atribuire, așa cum se poate observa în exemplul următor:

```
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")
```



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.

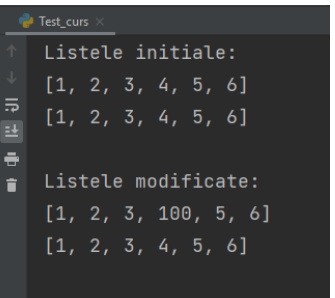
O prima variantă de realizare corectă a unei copii a unei liste o constituie utilizarea metodei `copy` din clasa `list`:

```
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")
```



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!

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)
```

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

#initalizare 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 `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 tuplu 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 relaționali: <, <=, >, >=, ==, !=*

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 șirul "Pop
                        # Ion"

```

Funcții predefinite pentru tuple

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 tuple, o listă sau un șir de caractere. Funcțiile predefinite care se pot utiliza pentru tuple sunt următoarele:

a) **`len(tuple)`**: furnizează numărul de elemente din tuple (lungimea tupleului)

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

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

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

c) **`min(tuple)` / `max(tuple)`**: furnizează elementul minim/maxim în sens lexicografic din tuplele respective (atenție, toate elementele tupleului 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 tuplele T:", min(T))      # -85
print("Maximul din tuplele T:", max(T))      # 101
print()

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

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

T = [20, -30, "101", 17, 100]
print("Minimul din tuplele 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 = 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 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

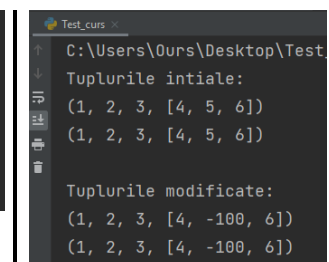
Deoarece tuplurile sunt imutabile, conținutul lor nu poate fi modificat, deci singura problemă care poate să apară în momentul copierii unui tuplu este existența în el a unor referințe spre obiecte mutabile:

```

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*):

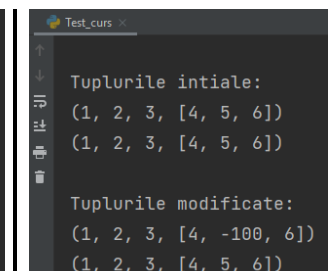
```

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)

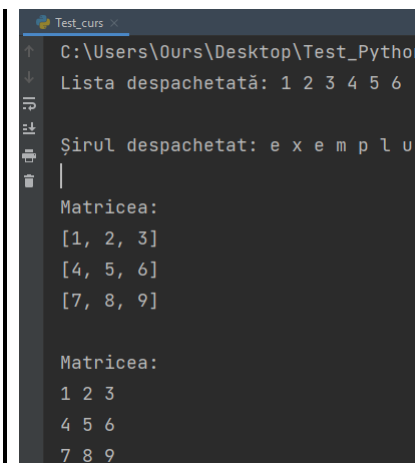
sir = "exemplu"
print("\nȘirul despachetat:", *sir)

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

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

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

```



```

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]

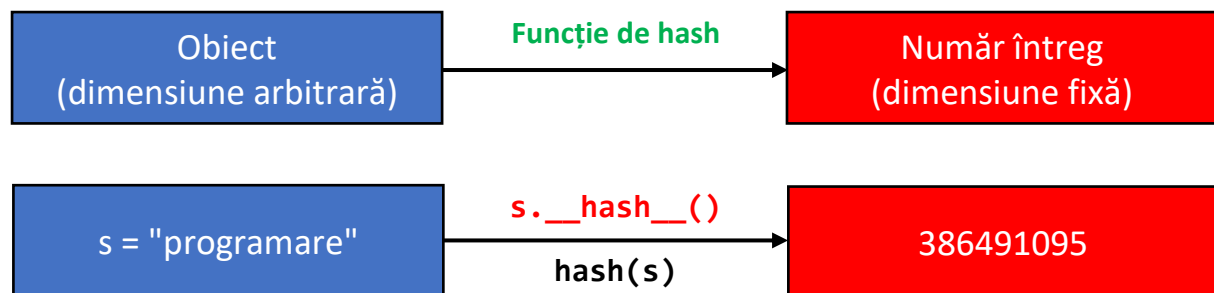
Matricea:
1 2 3
4 5 6
7 8 9

```

Tabele de dispersie (hash table)

În general, o *funcție de dispersie* (*hash function*) este un algoritm care asociază unui șir binar de lungime arbitrară un șir 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)`.

```

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")
  
```

```

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
  
```

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)}")
  
```

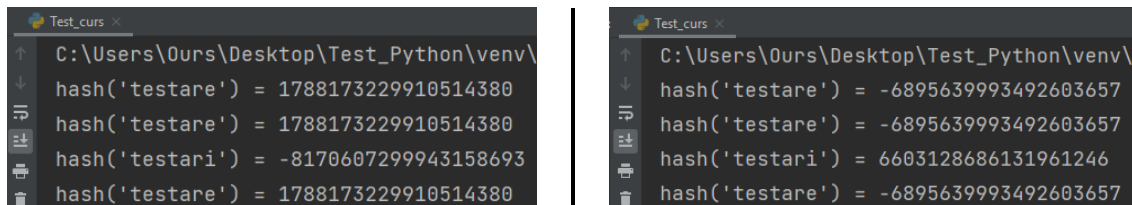
```

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:



The image shows two side-by-side screenshots of a Python terminal window. The left screenshot shows the output of the first code block: hash('testare') = 1788173229910514380, hash('testare') = 1788173229910514380, hash('testari') = -8170607299943158693, and hash('testare') = 1788173229910514380. The right screenshot shows the output of the second code block: hash('testare') = -6895639993492603657, hash('testare') = -6895639993492603657, hash('testari') = 6603128686131961246, and hash('testare') = -6895639993492603657.

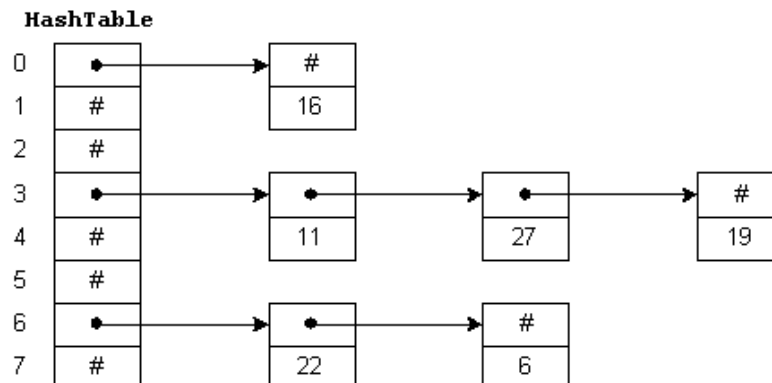
Referitor la prima condiție de mai sus, trebuie să observăm faptul că ea permite existența unor obiecte diferite din punct de vedere al conținutului, dar care au asociate aceeași valoare de dispersie (i.e., dacă `obiect_1 != obiect_2` atunci este posibil ca `hash(obiect_1) == hash(obiect_2)`)! În acest 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 înlănțuite). 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 table de dispersie depind de mai mulți factori (e.g., dimensiunea tablei, 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 $O(1)$. 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/inițializată în mai multe moduri:

- folosind un șir de constante sau o secvență:

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

# șir 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}
```

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

- folosind secvențe de inițializare:

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

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

# grupe distincte
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 incluziune (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. $A \Delta B = (A \setminus B) \cup (B \setminus A)$)

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)       # {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 șir de caractere sau o mulțime.

Funcțiile predefinite care se pot utiliza pentru mulțimi sunt următoarele:

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

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

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

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

- i) **`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!)

Exemple:

```

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'

```

- j) **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`

- k) **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 opțional `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 *mutabile*, 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:

- j) **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}
```

k) **update(secvență)**: adaugă, pe rând, toate elementele din secvența 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)}
```

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

Exemple:

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 dupa 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 dupa stergerea valorii {x}: {S}!")
except KeyError:
    print(f"Valoarea {x} nu apare in multime!")
```

- m) **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"Mulțimea: {S}")

x = 30

S.discard(x)
print(f"Mulțimea după ștergerea valorii {x}: {S}!")
```

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

Exemplu:

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

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

- o) **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 mulțime 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("Mulțimea A: ", A)      # {1, 3, 4, 7}
print("Lista B:", B)          # [1, 2, 3, 4, 6, 8]
```

- p) **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("Mulțimea 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 timpi 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 reunește la copie noua valoare 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 arbitrare 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/inițializat în mai multe moduri**:

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

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

# dicționar 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 inițializare:

```
# dicționar 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}

# dicționar 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 inexistente, 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 tuple:

```
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)]}

```

```
#             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, grupa)]:

```
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 relaționali: ==, !=* (două dicționare sunt egale dacă sunt formate din aceleași perechi **cheie: valoare**, indiferent de ordinea de inserare)

Exemple:

```

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 șir 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!**)

Exemple:

```
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': 3, '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 mutabile**, 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:

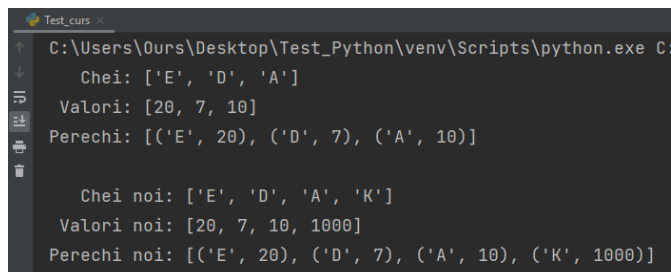
```
d = {'E':20, 'D': 7, 'A': 10}

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()
```



```
Test_curs
C:\Users\Ours\Desktop\Test_Python\venv\Scripts\python.exe C:\Users\Ours\Desktop\Test_Python\venv\Scripts\python.exe
Chei: ['E', 'D', 'A']
Valori: [20, 7, 10]
Perechi: [('E', 20), ('D', 7), ('A', 10)]

Chei noi: ['E', 'D', 'A', 'K']
Valori noi: [20, 7, 10, 1000]
Perechi noi: [('E', 20), ('D', 7), ('A', 10), ('K', 1000)]
```

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

- c) **update(dictionar):** 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 opțional 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}")
```

```
Test_curs
C:\Users\Ours\Desktop\Test_Python\venv\Sc
Dicționarul: {'E': 20, 'D': 7, 'A': 10}

Valoarea asociată cheii D: 7
Dicționarul: {'E': 20, 'A': 10}

Valoarea asociată cheii Z: None
Dicționarul: {'E': 20, 'A': 10}
```

- 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 $O(n)$;
- 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 $O(\log_2 n)$;

- 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 / tuple* (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>) | $\mathcal{O}(n)$ |
| Parcurgere | $\mathcal{O}(n)$ |
| 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> | $\mathcal{O}(\text{len}(\text{secvență}))$ |
| <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 | $\mathcal{O}(\text{len}(\text{secvență}))$ |
| Ștergerea unei secvențe | $\mathcal{O}(n)$ |
| Modificarea unei secvențe | $\mathcal{O}(\text{len}(\text{secvență}) + n)$ |
| 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 | $\mathcal{O}(n)$ | $\mathcal{O}(n^2)$ |
| Parcurgere | | $\mathcal{O}(n)$ |

| | | |
|---|---|--|
| len(mulțime) | | $\mathcal{O}(1)$ |
| update(secvență) | $\mathcal{O}(\text{len}(\text{secvență}))$ | $\mathcal{O}(n * \text{len}(\text{secvență}))$ |
| Reuniunea mulțimilor A și B (operatorul și metoda union) | $\mathcal{O}(\text{len}(A) + \text{len}(B))$ | |
| Intersecția mulțimilor A și B (operatorul &, metoda intersection și metoda intersection_update) | $\mathcal{O}(\min(\text{len}(A), \text{len}(B)))$ | $\mathcal{O}(\text{len}(A) * \text{len}(B))$ |
| Diferența mulțimilor A și B (operatorul - și metoda difference) | $\mathcal{O}(\text{len}(A))$ | |
| Diferența mulțimilor A și B (metoda difference_update) | $\mathcal{O}(\text{len}(B))$ | |
| Diferența simetrică a mulțimilor A și B (operatorul ^ și metoda symmetric_difference) | $\mathcal{O}(\text{len}(A))$ | $\mathcal{O}(\text{len}(A) * \text{len}(B))$ |
| Diferența simetrică a mulțimilor A și B (metoda symmetric_difference_update) | $\mathcal{O}(\text{len}(B))$ | $\mathcal{O}(\text{len}(A) * \text{len}(B))$ |
| Sortare (funcția sorted) | | $\mathcal{O}(n \log_2 n)$ |
| Funcțiile min, max și sum | | $\mathcal{O}(n)$ |

c) dicționare (cu n chei)

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