

CURS 4

Colecții de date

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)

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

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

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

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

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

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

Accesarea elementelor unui tuplu

Elementele unui tuplu pot fi accesate în mai multe moduri, asemănătoare celor prezentate pentru 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 $st \leq dr$, 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 tuplelor:

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 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(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 = 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

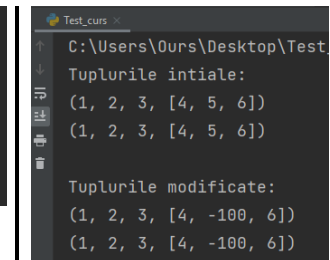
copie a tuplului curent, se adaugă la sfârșitul copiei noua valoare x și apoi referința tuplului curent se înlocuiește cu referința copiei.

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])

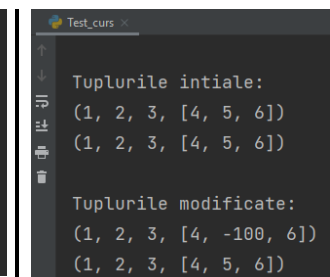
Tuplurile modificate:
(1, 2, 3, [4, -100, 6])
(1, 2, 3, [4, -100, 6])
```

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)

```

```

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]

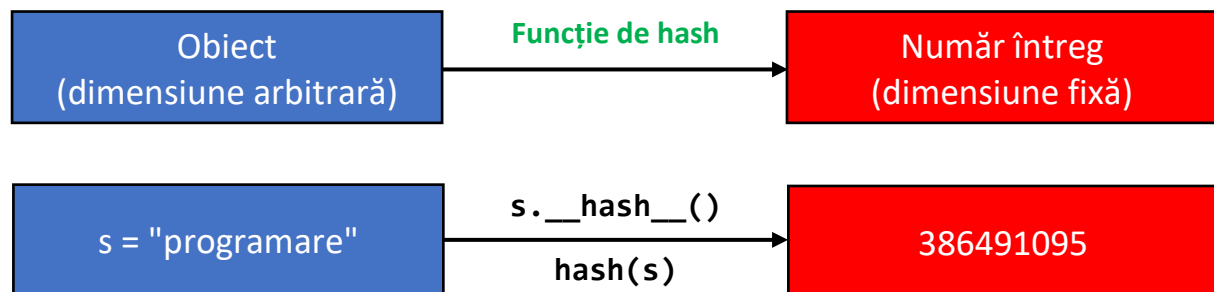
Matricea:
1 2 3
4 5 6
7 8 9

```

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")

```

```

Test_curs
C:\Users\Ours\Desktop\Test_Python\venv\Scripts\python
hash('Ana are mere!') = -2312829570296290796
hash('Ana are mere!') = -2312829570296290796

hash(12345) = 12345

hash(1267650600228229401496703205376) = 549755813888

hash(3.14) = 322818021289917443

hash((1, 2, 3, 4, 5)) = -5659871693760987716

```

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:

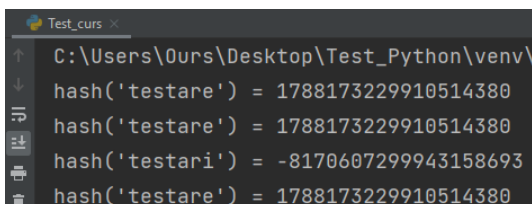
```
s = "testare"
print(f"hash('{s}') = {hash(s)}")

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

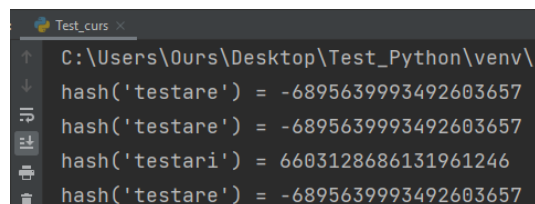
s = s[:len(s)-1] + "i"           #s = "testari"
print(f"hash('{s}') = {hash(s)}")

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:



```
Test_curs
C:\Users\Ours\Desktop\Test_Python\venv\
hash('testare') = 1788173229910514380
hash('testare') = 1788173229910514380
hash('testari') = -8170607299943158693
hash('testare') = 1788173229910514380
```



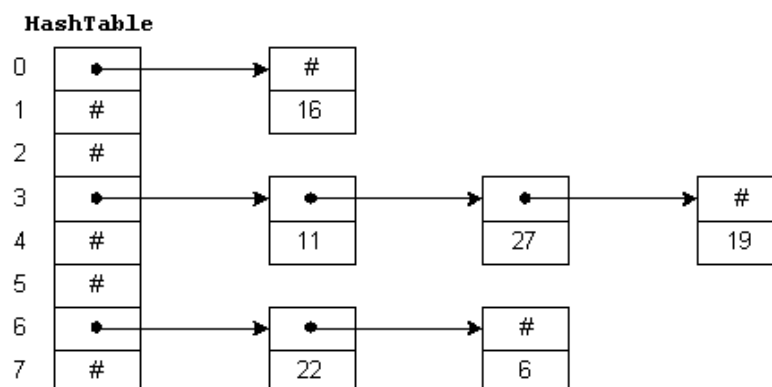
```
Test_curs
C:\Users\Ours\Desktop\Test_Python\venv\
hash('testare') = -6895639993492603657
hash('testare') = -6895639993492603657
hash('testari') = 6603128686131961246
hash('testare') = -6895639993492603657
```

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 înlanțuite). De exemplu, să considerăm numerele 16, 11, 27, 22, 19 și 6, precum și funcția de dispersie $h(x) = x \% 8$ corespunzătoare unei table 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 tablele 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 table 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:

```
# multe 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. $A \triangle B = (A \setminus B) \cup (B \setminus A)$)

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:

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

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

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

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

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

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'
```

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

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

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

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

Elementele mulțimii pot fi sortate și descrescător, setând parametrul 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:

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

Exemplu:

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

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

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

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

c) **`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!")
```

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

Exemplu:

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

x = 30

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

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

Exemplu:

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

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

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

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

Exemplu:

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

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

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

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

A.symmetric_difference_update(B)
print("Mulțimea 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, aproximativ 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.