CURS 5
Colecții de date (continuare)

2. TUPLURI / TUPLE

Tuplu = o secvență IMUTABILĂ de valori indexată de la 0

- **Valorile pot fi de tipuri de date diferite (neomogene).**
- **Tuplurile sunt mai rapide și folosesc mai puțină memorie decât listele.**
- **Tuplurile sunt instanțe ale clasei `tuple`.**

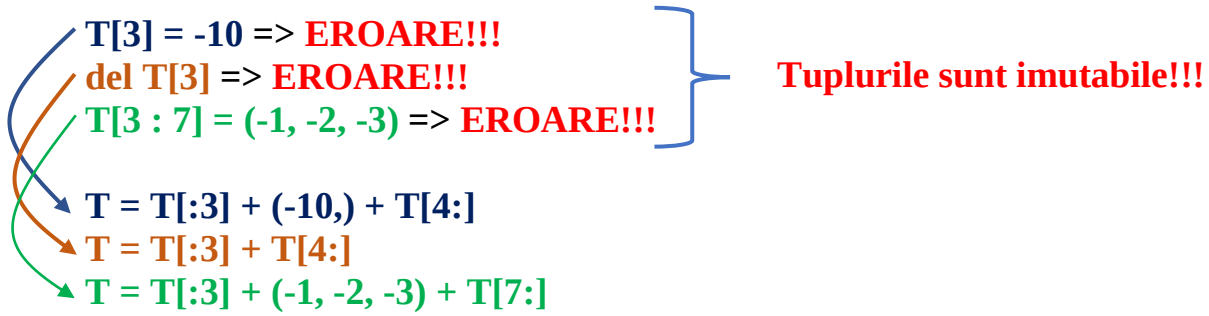
Crearea unui tuplu:

```
# tuplu vid
t = ()
print(t)

# prin initalizare
t = (1, 2, 3)
print(t)

# prin initalizare (varianta)
t = 1, 2, 3
print(t)

# prin initalizare (un element)
t = (1,)
print(t)
```

Accesarea elementelor unui tuplu:

a) prin indici pozitivi sau negativi

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



Operatori pentru tupluri:

a) concatenare: +
b) concatenare și atribuire: +=
c) multiplicare (concatenare repetată): *
d) testarea apartenenței: in, not in
e) operatori relaționali: <, <=, >, >=, ==, !=

Elementele testate trebuie să fie comparabile!

Funcții predefinite pentru tupluri:

a) `len(tuplu)` — **len((10, 20, 30, "abc", [1, 2, 3])) = 5**
b) `tuple(secvență)` — **tuple("test") = ('t', 'e', 's', 't')**
c) `min(tuplu)` și `max(tuplu)` — **Toate elementele tuplului trebuie să fie comparabile între ele!**

Metode pentru tupluri:

a) `count(valoare)` = numărul de apariții ale valorii respective în tuplu

```
T = tuple(x % 4 for x in range(22))
print(T)

n = T.count(2)
print(n)
```

b) `index(valoare)` = furnizează poziția primei apariții, de la stânga la dreapta, a valorii date sau lansează o eroare (ValueError) dacă valoarea nu apare în tuplu!

```
T = tuple(x + 1 for x in range(5))
print(T)

x = 3
if x in T:
    p = T.index(x)
    print(x, "apare in tuplu pe pozitia", p)
else:
    print(x, "nu apare in tuplu!")
```

```
T = tuple(x + 1 for x in range(5))
print(T)

x = 30
try:
    p = T.index(x)
    print(x, "apare in tuplu pe pozitia", p)
except:
    print(x, "nu apare in tuplu!")
```

Crearea unui tuplu din elemente citite:

```
import time

nr_elemente = 100_000

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

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

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

Timpi de executare:

```
   Initializare:  0.003989219665527344 secunde
  Operatorul +=:  0.012964963912963867 secunde
   Operatorul +:  8.101027488708496 secunde
```

Copierea unui tuplu: atenție la cazul în care elementele tuplului sunt obiecte mutabile!

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

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

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

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

Metoda deepcopy() realizează o copie în profunzime (deep copy) care rezolvă problema anterioară (dar este lentă!):

```
import copy

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

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

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

Tuplurile noi:
(1, 2, 3, [4, -100, 6])
(1, 2, 3, [4, 5, 6])
```

Împachetarea/despachetarea unui tuplu

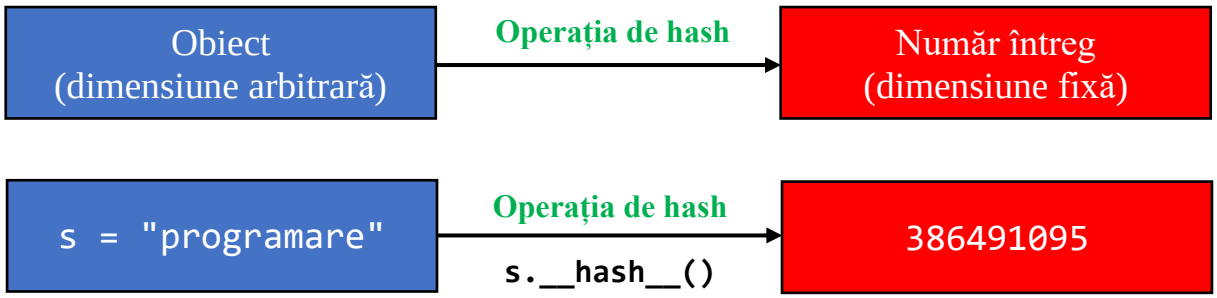
```
t = (1, 2, 3)          #împachetarea celor 3 numere într-un tuplu
print("t = ", t)
x, y, z = t            #despachetarea tuplului în 3 numere
print("x = ", x)
print("y = ", y)
print("z = ", z)

t = (131, "Popescu", "Ion", 9.70)
grupa, *nume, medie = t
print("t = ", t)
print("Grupa = ", grupa)
print("Nume = ", nume)
print("Medie = ", medie)

matrice = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
for linie in matrice:
    print(*linie)
```

Tabele de dispersie (hash table)

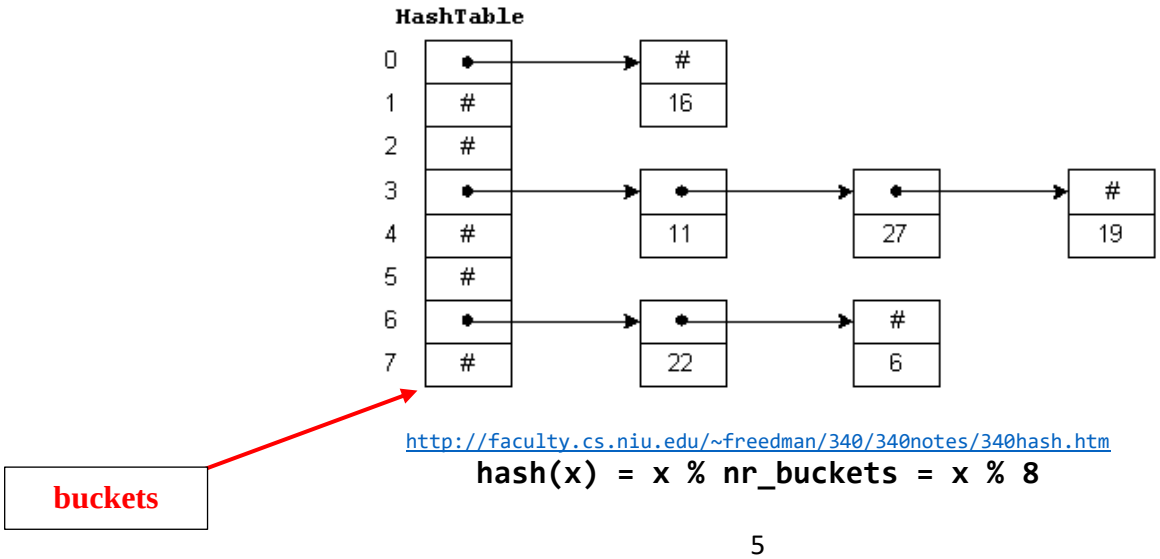
O funcție de dispersie este un algoritm care se aplică asupra un șir binar de lungime arbitrară (de exemplu, asupra reprezentării binare a unui obiect) și furnizează un șir binar de lungime fixă (de obicei, lungimea este multiplu de 128 de biți).

Proprietăți:

- **sensibilă la modificarea datelor membre ale obiectului**
- **rapidă**
- **să aibă cât mai puține coliziuni**
- **coliziune: pentru două obiecte diferite se obține aceeași valoare de hash**



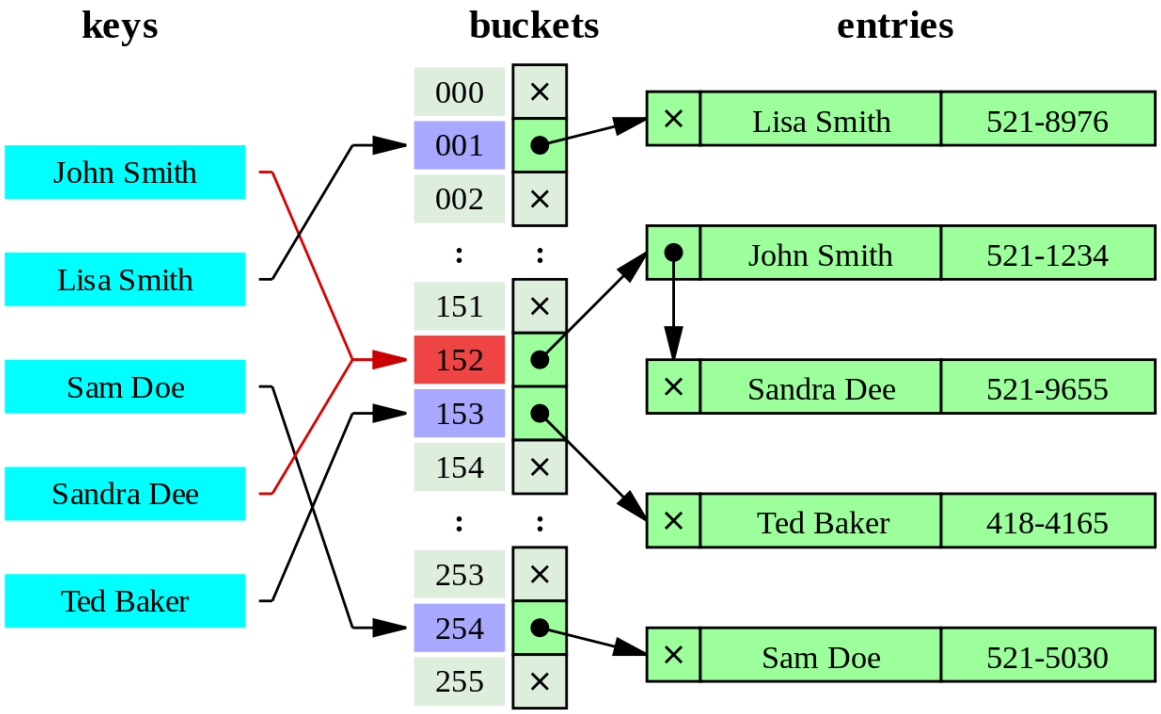

```
s = "testare"
print(s, s.__hash__())

s = s[:len(s)-1]
print(s, s.__hash__())

s = s + "e"
print(s, s.__hash__())
```



http://faculty.cs.niu.edu/~freedman/340/340notes/340hash.htm

hash(x) = x % nr_buckets = x % 8

index-ul bucket-ului -> __hash__()
compararea ob căutat cu obiectele din lista asociată bucket-ului -> __eq__()



https://cp-algorithms.com/string/string-hashing.html#toc-tgt-0

$$p = 31,\ m = 10^9+9$$
$$hash("BAC") = 2 + 1*31 + 3*31^2 = 2961$$
$$hash("BAD") = 2 + 1*31 + 4*31^2 = 3877$$
$$hash("BCA") = 2 + 3*31 + 1*31^2 = 2978$$

3. MULȚIMI

Mulțime = o colecție MUTABILĂ de valori fără duplicate

- **Mulțimile sunt instanțe ale clasei `set`.**
- **Valorile pot fi de tipuri de date diferite (neomogene), dar trebuie să fie "hash-uibile" (de exemplu, să fie imutabile).**
- **Mulțimile NU sunt indexate!**
- **Mulțimile NU păstrează ordinea de inserare a valorilor!**

Crearea unei mulțimi:

```
s = {1, 2, 3, 2, 2, 4, 1, 1, 7}
print(s)

s = set()
print(s)

s = set([1, 2, 3, 2, 2, 4, 1, 1, 7])
print(s)

s = set("testare")
print(s)

s = {x % 2 for x in range(100)}
print(s)
```

Accesarea elementelor unei mulțimi:

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

Operatori pentru mulțimi:

a) testarea apartenenței: in, not in

b) operatori relaționali: <, <=, >, >=, ==, !=

```
{1, 2, 3} == {2, 1, 3}
```

Testează relația de incluziune (submulțime/supermulțime)!!!

c) operații specifice mulțimilor: | (reuniune), & (intersecție), - (diferență), ^ (diferență simetrică = (A\B) U (B\A))

```
A = {1, 3, 5, 7, 10, 14}
B = {1, 2, 3, 4, 6, 8, 10, 13, 17}

print("Reuniunea:", A | B)
print("Intersecția:", A & B)
print("Diferența:", A - B)
print("Diferența simetrică:", A ^ B)
```

Funcții predefinite pentru mulțimi:

a) len(mulțime)
b) set(secvență)
c) min(mulțime) și max(mulțime)

Metode pentru mulțimi:

a) add(valoare) = ...

b) update(colecție iterabilă) = ...

c) remove(element) = ... (eroare)

d) discard(element) = ... (fără eroare)

e) clear() = ...

f) union(colecție iterabilă)
   intersection(colecție iterabilă)
   difference(colecție iterabilă)
   symmetric_difference(colecție iterabilă)

```
s = "examen la programare"

for litera in set(s):
    print("Litera", litera, "are frecventa", s.count(litera))
```

4. MULȚIMI IMUTABILE (clasa frozenset)

- Creare: s = frozenset([1,2,3,1,2,5,2])
- NU conține metodele add, update, remove și discard!

4. DICȚIONARE

Dicționar = o colecție MUTABILĂ de perechi de forma cheie:valoare

- **Dicționarele sunt instanțe ale clasei `dict`.**
- **Cheile trebuie să fie unice și imutabile, iar pentru valori nu există restricții.**
- **Mulțimile NU sunt indexate prin poziție, ci prin cheie!**
- **Dicționarele păstrează ordinea de inserare a perechilor!**

Crearea unui dicționar:

```
d = {}
print(d)

d = {"A": 4, "B": 7, 6: -100, (1, 2, 3): [100, 200, 300]}
print(d)

d = {}
d["B"] = "Popescu Ion"
d[(1, 2, 3)] = [100, 200, 300]
d[17] = "A"
print(d)

d = dict([("B", "Popescu Ion"), ((1, 2, 3), [100, 200, 300]), (17,
"A")])
print(d)

d = {chr(65 + x): x for x in range(10)}
print(d)
```

Accesarea unui element: se realizează prin cheie!

```
d = {chr(65 + x): x for x in range(5)}
print(d)

d["B"] = 100
print(d)

d["K"] = 7
print(d)

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

# print(d.get("Z", -1000))

if d.get("Z") == 100:
```

```
    print("DA")
else:
    print("NU")
```

Operatori pentru dicționare:

a) testarea apartenenței: in, not in (cheie!)
b) operatori relaționali: ==, !=

```
{"A":1, "B":2} == {"B":2, "A":1}
```

Funcții predefinite pentru dicționare:

a) len(dicționar)
b) dict(secvență)
c) min(dicționar) și max(dicționar) -> după cheie!

Metode pentru dicționare:

a) keys() = ...
b) values() = ...
c) items() = ...

```
d = {chr(65 + x): x for x in range(5)}

print("Chei:", d.keys())
print("Valori:", d.values())
print("Intrari:", d.items())

for (k, v) in d.items():
    print(k, v)
```

d) update(dicționar) = ...

```
d = {chr(65 + x): x for x in range(10)}
t = {chr(65 + x): x*100 for x in range(5, 8)}
t["Z"] = 1000

print(d)
d.update(t)
print(d)
```

e) pop(cheie) = ...(Eroare)
pop(cheie, valoare implicită) = ...

```
d = {chr(65 + x): x for x in range(10)}
print(d)
r = d.pop("Z", -1000)
print(r)
print(d)
```

```
s = "examen la programare"
d = {c: s.count(c) for c in set(s)}
print(d)
```

COMPLEXITĂȚILE COMPUTAȚIONALE ALE OPERAȚIILOR ASOCIATE COLECȚIILOR DE DATE

1. LISTE

Operation	Average Case	Amortized Worst Case
Copy	O(n)	O(n)
Append	O(1)	O(1)
Pop last	O(1)	O(1)
Pop intermediate	O(n)	O(n)
Insert	O(n)	O(n)
Get Item	O(1)	O(1)
Set Item	O(1)	O(1)
Delete Item	O(n)	O(n)
Iteration	O(n)	O(n)
Get Slice	O(k)	O(k)
Del Slice	O(n)	O(n)
Set Slice	O(k+n)	O(k+n)
Extend	O(k)	O(k)
Sort	O(n log n)	O(n log n)
Multiply	O(nk)	O(nk)
x in s	O(n)	
min(s), max(s)	O(n)	
Get Length	O(1)	O(1)

2. MULȚIMI

Operation	Average case	Worst Case
x in s	O(1)	O(n)
Union s|t	O(len(s)+len(t))	
Intersection s&t	O(min(len(s), len(t))	O(len(s) * len(t))
Difference s-t	O(len(s))	
Symmetric Difference s^t	O(len(s))	O(len(s) * len(t))

3. DICȚIONARE

Operation	Average Case	Amortized Worst Case
k in d	O(1)	O(n)
Copy	O(n)	O(n)
Get Item	O(1)	O(n)
Set Item	O(1)	O(n)
Delete Item	O(1)	O(n)
Iteration	O(n)	O(n)

https://wiki.python.org/moin/TimeComplexity