Structuri de date. Liste, tupluri, mulțimi, dicționare

LISTE

Despre liste în Python. Crearea unei liste

Un tip de secvențe în Python sunt listele. Pentru lucrul cu există tipul de date (clasa) list.

Elementele unei liste pot avea tipuri diferite (liste neomogene).

Listele sunt secvențe mutabile: elementele unei liste se pot modifica, se pot adăuga și șterge elemente din listă.

Creare

O listă se poate crea în mai multe moduri.

Elementele unei liste se scriu între paranteze drepte, separate prin virgulă:

```
ls = [7, 5, 13] #specificam elementele, intre []
ls = [] #lista vida
ls = [ [1, 3], "doi", [5, 6, 7]] #Liste imbricate + neomogene
ls[0][1] = 11
print(ls)
```

O listă se poate crea și folosind constructorul **`list([iterabil])`** care poate primi ca parametru un obiect iterabil (o secvență) și returnează o listă cu aceleași elemente ca ale obiectului primit ca parametru. Următoarea secvență de cod ilustrează modalități de creare a unei variabile de tip **`list`**:

```
ls = list() #tot lista vida
ls = list("abc") #=> ls= ['a', 'b', 'c']
print(ls)
```

Secvențe de inițializare (comprehensiune)

O listă se poate inițializa folosind și secvențe de inițializare numite și comprehensiune, după denumirea în engleză “list comprehension”, de forma:

```
[expresie for x in iterable]
```

Spre exemplu:

```
ls = [x*x for x in range(1,11)]  #lista cu primele 10 patrate
perfecte
print(ls)
```

```
n = 10
ls = [0 for i in range(n)] # lista cu n elemente egale cu 0,
#utila de exemplu ca vector de frecvente
print(ls)
```

Observație: **`ls = [0]*n`** are ca efect tot crearea unei liste cu n elemente egale cu 0, vom reveni însă la deosebiri între cele două inițializări în secțiunea dedicată operatorului *.

Comprehensiune condiționată

În secvențele de inițializare de tip comprehensiune se pot pune și condiții de filtrare a elementelor pe baza cărora se creează noua listă:

```
[expresie for x in iterable if conditie]
```

Exemple:

1. Crearea unei liste noi cu elementele pozitive ale unei liste date:
```
ls = [1,-2,3,4,-5,6]
ls_poz = [x for x in ls if x>0]
print(ls_poz)
```
2. Determinarea elementelor comune a două mulțimi memorate ca liste:
```
l1 = [2,4,6,8]
l2 = [1,2,3,4,5]
intersectie = [x for x in l1 if x in l2]
print(intersectie)
```

Dacă în secvența de inițializare nu dorim filtrarea elementelor pe baza cărora se creează lista nouă, ci înlocuirea lor după anumite criterii putem folosi operatorul condițional(ternar) `... if... else...`

```
[expresie1 if conditie else expresie2 for x in iterable]
```

Exemplu. Crearea unei liste noi pornind de la o listă dată înlocuind elementele negative cu 0:

```
ls = [1,-2, 3, 4,-5,6]
ls_0 = [x if x>0 else 0 for x in ls]
print(ls_0)
```

În secvența de inițializare pot fi mai multe for-uri (for în for). De exemplu, putem crea o listă de perechi (o pereche având elemente distincte) cu numerele de la 1 la 6 astfel:

```
ls = [(x, y) for x in range(1,7) for y in range(1,7) if x != y]
```

Pentru a înțelege mai bine sintaxa, o scriere echivalentă (!dar mai lentă) fără a folosi comprehensiune ar fi**:**

```
ls = []
for x in range(1,7):
   for y in range(1,7):
       if x != y:
           ls.append((x,y)) #metoda adauga un element la lista
```

Alte exemple de comprehensiune:

1. Citirea elementelor unei liste de numere întregi de la tastatura pe o linie, separate prin spațiu:
   ```
   ls = [int(x) for x in input().split()]
   ```
2. Citirea elementelor unei liste de 5 numere întregi de la tastatura, date fiecare pe câte o linie
   ```
   ls = [int(input()) for i in range(5)]
   ```
3. Determinarea numărului de vocale ale unui cuvânt citit de la tastatură
   ```
   c = len([x for x in input() if x.lower() in "aeiou"] )
   ```
4. Crearea unei matrice cu m linii și n coloane cu toate elementele 0
   ```
   m = int(input())
   n = int(input())
   a = [[0 for j in range(n)] for i in range(m)]
   ```
5. Se citește o propoziție cu cuvinte separate prin spațiu. Să se creeze o listă cu cuvintele care încep cu o consoană, cu toate literele transformate în litere mici.
   ```
   l_vocale = [c.lower() for c in input().split() if
   c[0].lower() not in "aeiou" ]
   print(l_vocale)
   ```
6. Se citește o propoziție cu cuvinte separate prin spațiu. Să se creeze o listă cu cuvintele care sunt palindroame
   ```
   l_palindrom = [c for c in input().split() if c==c[::-1]]
   print(l_palindrom)
   ```

Accesarea elementelor. Parcurgere

Accesarea elementelor unei se face cu metodele deja discutate la secvențe (și la șiruri de caractere).

Amintim că se pot folosi indici negativi (-1 fiind ultimul element din listă), iar prin feliere putem obține subliste din lista inițială. Dacă indicele i nu este valid, apelul `ls[i]` va da eroarea `IndexError` (`ls[i:]` însă nu va da eroare, la feliere indicele care "iese" din limită este înlocuit cu limita). Spre exemplu,

`ls[-k : ]` == lista formată din ultimele k elemente

Pentru a parcurge elementele o listă, ca și în cazul unui șir (de fapt al oricărei secvențe) putem folosi instrucțiunea `for`:

```
ls = [i for i in range(0,10,2)]
for x in ls:
    print(x)
for i in range(len(ls)):
    print(ls[i])
```

Operatori

Pentru liste pot folosi toți operatorii specifici secvențelor:

- **`in, not in`**
- **Concatenare `+, *`**

  Operatorii de concatenare fac o copiere superficială a elementelor listelor concatenate (sunt copiate referințele). Dacă elementele listelor implicate sunt mutabile, modificarea valorii unui element din rezultat (! a valorii), poate duce la modificarea listei inițiale (vezi și secțiunea de copiere și inițializare matrice). Spre exemplu:

```
ls = [1]
ls_c = ls*3
ls_c[0] = 5 # se modifica doar ls_c
print(ls_c, ls)
ls = [[1]]
ls_c = ls * 3
ls_c[0].append(5)
print(ls_c,ls)
ls_c[0]=[3]
print(ls_c,ls)
```

- **Comparare**: **`<, <=, >=, >, >=, ==, !=`** (!elementele testate trebuie să fie comparabile)

  Exemple:

  `[1, 2] == [2, 1]` este **`False`**, lista este o secvență, contează ordinea elementelor

  `[1, 2] == [1, 2]` este **`True`**

  `[1, 2] is [1, 2]` în general **`False`**

  `ls = [1, 3]; ls1 = [1, 2, 3]`

  `ls > ls1` este **`True`**, compararea se face lexicografic (începând cu primul element): avem **`ls[0] = ls1[0]`** și **`ls[1] > ls1[1]`**

  `[1, "a"] > [1, 2]` => eroare (TypeError: '>' not supported between instances of 'str' and 'int')

  Exerciții: Ce valori au expresiile:

```
[1, 40, 2] >  [3, 7]
[1, "b"] <  [1, "a", 2]
```

- **Operatorii** +=, *=

  `ls += ls1` este echivalent cu **`ls.extend(ls1)`**, iar **`ls *= n`** cu concatenare de n ori

Funcții uzuale

Amintim funcțiile uzuale comune pentru subsecvențe (care se pot folosi deci și pentru funcții:

- **`len(ls)`**

  Exemplu: len(["unu", 2, [3,3,3]]) este 2

- **`min(ls), max(ls)`**

  Exemple:
  `min(["un","alt"])` este **`"alt"`**
  `max(["un","alt",5])` dă eroare deoarece lista are elemente care nu sunt comparabile două câte două (**`TypeError`**: '>' not supported between instances of 'int' and 'str')

Pentru liste cu elemente de tip numeric, există funcția **`sum`**:

- **`sum([2,4,5])`** este **`11`**

Căutare

Pentru căutarea unei valori într-o listă se pot folosi metodele comune pentru secvențe. Amintim:

- **`ls.index(x[,start[,end]])`** => returnează poziția primei apariții a lui **`x`** în **`ls`**; dacă se precizează și parametri opționali, aceștia sunt interpretați ca la feliere, căutarea făcându-se începând cu poziția **`start`** dacă se transmite valoare pentru **`start`**, respectiv de la poziția **`start`** la **`end`** (exclusiv **`end`**) dacă au valori ambii parametri opționali; metoda aruncă eroarea ValueError dacă valoarea lui **`x`** nu este element în **`ls`**
- **`ls.count(sub)`** => returnează numărul de apariții ale elementului primit ca parametru în ls

Modificarea conținutului unei liste

O primă variantă de a modifica o listă este prin atribuiri de valori pentru un element accesat prin indice sau unei felii (slice) din listă, sub forma:

- **`ls[i] = x`**
- **`ls[i:j] = iterabil`** => subsecvența cu elem **`i,...j-1`** este înlocuită cu elementele lui **iterabil**
- **`ls[i:j:k] = iterabil`** (de aceeași lungime)

  Exemplu

```
ls = [1,10,20,30,60]
ls[2:4] = [12,14,16,18]
print(ls)
ls[1:1] = [5,6,7]
print(ls)
```

```
ls[1:6:2] = [0,0,0]
print(ls)
ls[1:4] = []
print(ls)
```

Exercițiu: Ce va afișa următoarea secvență de cod?

```
ls = [1,2,3]
ls[1] = [3,4]
print(ls)
ls = [1,2,3]
ls[1:1]=[3,4]
print(ls)
```

Pentru a **șterge** elemente dintr-o listă putem atribui unei felii lista vidă sau putem folosi instrucțiunea del:

- `del ls[i:j]` este echivalent cu `ls[i:j]=[]`
- `del ls[i:j:k]`

Exemplu

```
ls = [1,10,20,30]
del ls[-1]
print(ls)
del ls[0:1]
print(ls)
del ls[:]
print(ls)
del ls
print(ls)
```

Există și metode prin care se poate modifica valoarea unei liste (adăugarea unui element, a unei secvențe, ștergerea unui element dat sau de pe o poziție dată, inserare pe o poziție dată etc)

- `ls.append(x)` – adaugă un **x** la sfârșitul listei ls (ca și `ls[len(ls):] = [x]`)
- `ls.extend(iterabil)` – adaugă **elementele** unei **colecții (lista, șir...)** la sfârșitul listei ls (ca și `ls[len(s):] = iterabil`)

  Exemplu

  ```
  ls = []
  ls.append(1)
  print(ls)
  ls.extend([2, 3])
  print(ls)
  ls.append([2, 3])
  print(ls)
  ls = ls + [5]  # nerecomandat
  print(ls)
  ls = []
  ls.append("ab"); print(ls)
  ```

```
ls.extend("ab"); print(ls)
ls = ls + ["abc"]
print(ls)
ls += "abc"
print(ls)
```

Exemplu – Citirea unui vector de numere întregi cu elementele date câte unul pe o linie

```
ls = []
n = int(input("n="))
print("introduceti sirul, cate un element pe linie")
for i in range(n):
    ls.append(int(input()))
print(ls)
```

- **`ls.insert(i,x)-`** inserează x pe poziția i (similar cu **ls[i:i] = [x])**
- **`ls.pop([i])`** – elimină elementul de pe **poziția** i sau ultimul dacă parametrul i nu este dat (valoarea default pentru este -1)
- **`ls.remove(x)`** – elimină prima apariție a lui x (**ValueError** daca nu există)
- **`ls.clear()`**

  Exemplu

```
ls = [10,20,30,40,50]
ls.insert(2,60)
print(ls)
```

```
ls.remove(10) #prima aparitie
print(ls)
ls = [10,20,30,40,50]
poz = 2
print(f"se elimina elementul {ls.pop(poz)} de pe pozitia {poz} =>
{ls}")
print(f"se elimina ultimul element {ls.pop()} => {ls}")
ls.clear()
print(ls)
```

- **`ls.reverse()`** – oglindirea (inversarea elementelor) listei (**`ls` se modifică**)

  Exemplu

```
ls = [3,5,7]
ls.reverse()
print(ls)
```

Copiere

- **`ls.copy()`-** copiere superficială a conținutului listei, rezultatul fiind un obiect nou în care sunt **copiate referințele** elementelor listei inițiale. Dacă pentru anumite tipuri de date ale elementelor acest lucru poate fi satisfăcător (de exemplu dacă elementele sunt numere sau imutabile), în cazul în care lista inițială avea elemente mutabile (de

exemplu matrice, care este o listă de liste), o astfel de copiere poate să nu fie ceea ce ne dorim, deoarece modificarea conținutului unui element din copie va modifica și lista inițială. În acest caz se poate folosi funcția **`deepcopy()`** din modulul **`copy`**. Amintim că prin atribuire se atribuie referințe, deci după atribuire de tipul **`ls2 = ls1`** ambele variabile vor referi aceeași listă. Pentru a vedea diferența dintre cele modalitățile de copiere analizăm următorul exemplu:

```
ls1 = [1,2]
ls2 = ls1
ls1[0] = 13
print(ls1,ls2) #si ls2 s-a modificat
```

```
ls1 = [1,2]
ls2 = ls1.copy() #ls2 = ls1[:]
ls1[0] = 13
print(ls1,ls2)
ls1 = [[1,2], [3,4]]
ls2 = ls1.copy()
ls1[0][0] = 13
print(f"{ls1}   {ls2}") #se modifica ls2[0]
```

```
import copy
ls1 = [[1,2], [3,4]]
ls2 = copy.deepcopy(ls1)
ls1[0][0] = 13
print(f"{ls1}   {ls2}")
```

Matrice. Tablouri multidimensionale

O matrice (sau, mai general, un tablou multidimensional) se poate memora folosind liste imbricate. O matrice este de fapt o listă de linii, deci o listă în care elementul i este o listă reprezentând linia i:

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

Exemple

1. Citirea unei matrice de la tastatură: Date m și n și o matrice de numere întregi (elementele unei linii se vor da pe o linie separate prin spațiu), să se construiască în memorie o matrice cu aceste elemente și să se afișeze pe ecran. De exemplu, pentru intrarea:

3 4
1 12 400 50
10 2 80 100
9 155 79 46

se va afișa:

```
  1  12 400  50
 10   2  80 100
  9 155  79  46
```

Soluție

```
# varianta 1
ls = input().split()
n = int(ls[0])
m = int(ls[1])
a = []
for i in range(n):
    ls = [int(x) for x in input().split()]
    a.append(ls)
print(a)
for linie in a:
    for x in linie:
        print(f"{x:4}", end="")
    print()

# varianta 2 - cu list comprehension
n, m = [int(x) for x in input().split()]
a = [[int(x) for x in input().split()] for i in range(n)]
print("\n".join(["".join([f"{x:4}" for x in linie]) for linie in a]))
```

2. Inițializarea unei matrice cu 0

Pentru a crea o matrice mxn cu toate elementele egale cu 0 matrice se poate folosi comprehensiune. Deși se poate folosi pentru creare și operatorul de multiplicare * (concatenare repetată), este necesară o bună înțelegere a modului în care se face copierea listei multiplicate pentru a obține efectul dorit. Spre exemplu, o inițializare de tipul **`a=[[0]*n]*m`** nu este ceea ce ne dorim, pentru că modificarea unui element din matrice va duce și la modificarea altor elemente (de ce? ce elemente se vor actualiza dacă modificăm a[0][0]?). Pentru a înțelege mai bine cum se poate crea corect o matrice cu toate elementele 0 considerăm următoarele încercări de inițializare:

```
m = 2; n = 3
a = [[0 for i in range(n)] for j in range(m)]  # CORECT
# fiecare linie este un obiect diferit creat cu secventa de
initializare [0 for i in range(n)]]
a[0][0] = 3
print(a)

a = [[0] * n for i in range(m)]  # CORECT
a[0][0] = 3
print(a)

a = [[0 for i in range(n)]] * m  # INCORECT
# liniile sunt același obiect, prin multiplicare se copiază #referinte
se va crea o singura lista cu [0 for i in range(n)]
# si se vor multiplica n  referinte catre aceasta
a[0][0] = 3
print(a)

a = [[0] * n] * m  # INCORECT, liniile sunt același obiect
a[0][0] = 3
print(a)
```

Sortare

În secțiunea dedicată metodelor comune secvențelor am prezentat metoda de sortare sorted(), care, evident, poate fi folosită și pentru liste:

```
sorted(ls, key=None, reverse=False)
```

Această metodă primește lista ca parametru și returnează o listă nouă cu elementele sortate (!! nu modifică lista **ls**).

În clasa str există însă și o metodă sort pentru sortarea unei liste, care modifică lista care apelează metoda (nu returnează o nouă listă, returnează None):

```
ls.sort(key = None, reverse = False)
```

Exemplu

```
print(sorted([4,1,7,3,6]))
s = "Programarea"
sorted(s)
print(s) #??
```

```
ls = ["Vom", "sorta", "aceasta","lista","de","siruri"]
ls_sorted = sorted(ls)
print(ls)
print(ls_sorted)
ls.sort()
print(ls)
```

Metodele de sortare sunt stabile (în caz de egalitate se păstrează ordinea în care erau elementele în lista inițială).

Vom discuta în continuare despre parametrul cu nume **key**, prin care se poate specifica o cheie de comparare. Parametrul **key** poate primi ca valoare numele unei funcții. Această funcție trebuie să aibă un singur parametru (un element din listă) și să returneze o valoare, care reprezintă cheia asociată, adică valoarea asociată elementului după care se va face sortarea.

Spre exemplu, dacă vrem să sortăm o listă de cuvinte după lungimea cuvintelor, funcția de cheie primește ca parametru un șir și returnează lungimea acestuia (deci este chiar funcția **len**):

```
ls = ["Vom", "sorta", "aceasta", "lista", "de", "siruri"]
ls_sorted = sorted(ls, key = len, reverse = True)
print(ls_sorted) #sortare stabila
```

```
ls = ["Vom", "sorta", "aceasta", "lista", "de", "siruri"]
ls_sorted = sorted(ls, key = str.lower)
print(ls_sorted)
```

Dacă vrem să sortăm o listă de cuvinte în funcție de ultima literă putem face o funcție proprie care să asocieze unui șir ca și cheie ultima literă (funcția va primi ca parametru un șir și va returna ultima sa literă), ca în exemplul următor:

```
def cheie(s):
    return s[-1]
ls = ["Vom", "sorta", "aceasta", "lista","de","siruri"]
ls_sorted = sorted(ls, key = cheie) #sortare stabila
print(ls_sorted)
```

Putem folosi și funcții lambda funcții fără nume, definite chiar la apelul sortării), cu sintaxa:

```
lambda parametru : valoare_returnata
```

De exemplu, pentru a sorta lista `ls` din exemplul anterior sortarea după ultima literă **descrescător**, vom folosi cheia `lambda s : s[-1]` astfel:

```
ls_sorted = sorted(ls, key = lambda s:s[-1], reverse = True)
#sortare stabila
print(ls_sorted)
```

Dacă vrem să folosim criterii de comparare similare celor din C, adică funcții comparator cu doi parametri x și y (reprezentând 2 elemente) care returnează:

- `1` (pozitiv) dacă x>y
- `0` dacă x==y
- `-1` dacă x < y

avem nevoie de o funcție care să transforme o funcție comparator în cheie. Această funcție este `functools.cmp_to_key()`.

Exemplu. Să se sorteze o listă de perechi, folosind următorul criteriu de comparare pentru două perechi: (x,y) > (u,v) dacă și numai dacă y > v sau (y = v și x > u).

```
import functools
def comp_perechi(p,q):
    if p[1] < q[1]:
        return -1
    if p[1] == q[1]:
        return p[0]-q[0]
    return 1

ls = [(3,4), (5,3), (3,1), (1,6), (2,5)]
ls_sorted = sorted(ls, key =  functools.cmp_to_key(comp_perechi))
print(ls_sorted)
```

Sortări după mai multe criterii

Dacă dorim să sortăm după mai multe crierii, putem folosi tupluri, ținând cont că două tupluri se compară componentă cu componentă: astfel, primul element din tuplu va corespunde primului criteriu de sortare, al doilea următorului criteriu etc.

De exemplu, în exemplul anterior în care am folosit comparator pentru a sorta listă de perechi, folosind următorul criteriu de comparare pentru două perechi: (x,y) > (u,v) dacă și numai dacă y > v sau (y = v și x > u), puteam defini și direct o funcție cheie sau puteam folosi lambda expresii. Astfel, pentru o pereche (x,y) cheia asociată va fi tuplul (y,x), deoarece vrem să comparăm întâi a doua componentă, apoi prima:

```
def cheie_pereche(x):
    return x[1],x[0] # return (x[1],x[0])
ls = [(3,4), (5,3), (3,1), (1,6), (2,4)]
ls_sorted = sorted(ls, key=cheie_pereche)
print(ls_sorted)
```

```
ls = [(3,4), (5,3), (3,1), (1,6), (2,4)]
ls_sorted = sorted(ls, key=lambda x: (x[1],x[0]))
print(ls_sorted)
```

Exemple.

1. Considerăm o listă care conține informații despre mai mulți studenți, respectiv pentru fiecare student se cunoaște numele, grupa și nota obținută la examenul de admitere. Să se sorteze studenții în ordinea crescătoare a grupelor, în fiecare grupă studenții să fie sortați descrescător după nota obținută la examenul de admitere, iar în cazul unor note egale studenții să fie sortați alfabetic. Vom considera faptul că informațiile despre fiecare student sunt memorate ca o listă cu 3 elemente:

   ```
   def cheie_student(x):
       # componentele tuplului returnat corespund criteriilor:
       # grupa=> x[1],
       # nota descrescator => -x[2] (!cu minus),
       # nume => x[0]
       return x[1], -x[2], x[0]
   ls_studenti = [["Ionescu Mircea", 181, 7], ["Vasilesu Mihai", 182,
   8], ["Matei Andrei", 181, 9.5],["Antonescu Ionel", 181, 7]]
   ls_sorted = sorted(ls_studenti, key = cheie_student)
   print(ls_sorted)
   ```

2. Să se sorteze o listă de numere naturale astfel încât numerele pare sortate crescător să fie poziționate înaintea celor impare sortate descrescător.

   ```
   def cheie_par(x):
       #criterii:
       #1. par inainte impar =>
       #un numar par trebuie sa fie considerat mai mic decat unul impar
       #asociem 0, respectiv 1 ca prim criteriu unui nr. par, respectiv impar
       #2. numere pare sortate crescator =>
       # al doilea criteriu pentru numar par va fi chiar valoarea lui
       # 2. numere impare sortate descrescator =>
       # al doilea criteriu pentru numar impar va fi minus valoarea lui
   ```

```
    if x%2 == 0:
        return 0, x
    else:
        return 1, -x

ls = [4, 2, 1, 8, 7, 6, 5, 13]
ls.sort(key=cheie_par)
print(ls)
```

Menționăm faptul că funcția **`sorted`** și metoda **`sort`** implementează algoritmul de sortare **`Timsort`**, care a fost creat special în 2002 de Tim Peters pentru a fi implementat în limbajul Python (https://en.wikipedia.org/wiki/Timsort). Algoritmul Timsort este derivat din algoritmii de sortare prin interclasare (Mergesort) și sortare prin inserție (Insertion sort), având complexitatea $\mathcal{O}(n\log_2 n)$ în cazul cel mai defavorabil.

Exercițiu. Să se sorteze lista de studenți de la exercițiul 1 descrescător după note și, în caz de egalitate, crescător după nume.

Implementare internă. Complexitatea operațiilor

Un obiect de tip list este memorat intern ca un vector, de aceea complexitatea operațiilor este cea pe care o cunoaștem de la tablouri unidimensionale (accesarea unui element O(1), ștergerea și inserarea necesită deplasări ale elementelor, deci au complexitate liniară)

Următorul tabel (sursa: https://wiki.python.org/moin/TimeComplexity) sintetizează complexitatea principalelor operații:

Operation	Average Case	Amortized Worst Case
Copy	**O(n)**	**O(n)**
Append	**O(1)**	**O(1)**
Pop last	**O(1)**	**O(1)**
Pop(i)	**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)**

Despre timpul de execuție al unor operații similare (+= vs extend/append vs comprehension, concatenare de sirurui vs join) *

Pentru a construi o listă cu elemente date putem folosi mai multe variante: comprehensiune, concatenare, folosi metoda append, operatorul +=. Pentru a înțelege mai bine ce alegem considerăm următorul exemplu, în care măsurăm timpul de execuție pentru diferite abordări pentru crearea unei liste cu primele 1000 de numere:

```
import time
nr_elemente = 100000
start = time.time()
lista = [x for x in range(nr_elemente)]
stop = time.time()
print(" Comprehensiune: ", 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 = []
for x in range(nr_elemente):
  lista = lista + [x]
stop = time.time()
print(" Operatorul +: ", stop - start, "secunde")
```

După o rulare a acestui program am obținut rezultatele:

- Comprehensiune: 0.006981611251831055 secunde
- Metoda append(): 0.017951250076293945 secunde
- Operatorul +=: 0.0189516544342041 secunde

- Operatorul +: 22.40821361541748 secunde

Amintim că la concatenarea a două secvențe se creează un obiect nou și se copiază conținutul secvențelor concatenate, ceea ce justifică ineficiența folosirii operatorului + de concatenare în cazul în care dorim modificarea unei liste. Rezultate similar obținem și dacă extindem lista cu mai multe elemente simultan (folosind extend în loc de append):

```
nr_concatenari = 10000
ls=list(range(50))
start = time.time()
lista=[]
for i in range(nr_concatenari):
    lista.extend(ls)
stop = time.time()
print("Extend: ", stop - start, "secunde")
start = time.time()
lista = []
for i in range(nr_concatenari):
    lista+=ls
stop = time.time()
print("Operatorul +=: ", stop - start, "secunde")

start = time.time()
lista = []
for x in range(nr_concatenari):
 lista = lista + ls
stop = time.time()
print(" Operatorul +: ", stop - start, "secunde")
```

După o rulare a acestui program am obținut rezultatele

- Extend: 0.01694941520690918 secunde
- Operatorul +=: 0.01994800567626953 secunde
- Operatorul +: 11.493428945541382 secunde

Dacă vrem să formăm un cuvânt prin lipirea a n cuvinte (cu n mare), putem folosi operatorul de concatenare + pentru șiruri sau putem crea o listă de cuvinte și aplica metoda join. Reamintim că la fiecare concatenare de șiruri se face un obiect nou și copierea conținutului șirurilor concatenate, de aceea este recomandata a doua varianta. Acest lucru se poate vedea și în duratele de execuție afișate la rularea următoarei secvențe de cod:

```
nr_concatenari = 10000
s="a"*500
start = time.time()
lista=[]
for i in range(nr_concatenari):
    lista.append(s)
rez="".join(lista)
stop = time.time()
```

```
print("Join: ", stop - start, "secunde")

start = time.time()
rez = ""
for i in range(nr_concatenari):
    rez=rez + s
stop = time.time()
print(" Operatorul +: ", stop - start, "secunde")
```

După o rulare a acestui program am obținut rezultatele

- Join: 0.0029914379119873047 secunde
- Operatorul +: 0.852783203125 secunde

Exemplu - Crearea unei liste cu elemente date de la tastatură

Citirea unei liste cu elemente date de la tastatură se poate face în mai multe moduri. De exemplu, dacă elementele sunt date câte unul pe linie:

```
ls = [0 for i in range(n)]
for i in range(n):
    ls[i] = int(input())

#citire element cu element
ls = []
for i in range(n):
    ls.append(int(input()))

#mai pythonic - comprehensiune
ls = [int(input()) for  i in range(n)]
```

TUPLURI

Despre tupluri. Creare

Un tip de secvențe similar listelor este furnizat în **Python** de clasa **tuple**. Diferența principală este că tuplurile sunt imutabile: nu se pot modifica elemente (referințele lor), nu se pot adăuga sau șterge elemente dintr-un tuplu. Ca și listele, tuplurile pot fi neomogene (de obicei sunt, atunci când sunt folosite ca și structurile din C)

Ca și în cazul listelor, un tuplu se poate crea în mai multe moduri.

Elementele unui tuplu se scriu între paranteze rotunde, separate prin virgulă. Parantezele rotunde se pot omite.

```
t = ()
t = (1, 2, 3)
t = 1, 2, 3 #putem omite()
#Tupluri imbricate
t = ((1,"A"),(2,"B"),3)
```

Pentru a crea un tuplu cu un singur element nu este suficient să îl punem între paranteze, ci trebuie pusă și virgulă după el, altfel parantezele sunt ignorate (setând doar prioritatea???)

```
t = (2) #NU
print(type(t))
t = (2,) #DA
print(type(t))
```

O listă se poate crea și folosind constructorul **tuple([iterabil])** care poate primi ca parametru un obiect iterabil (o secvență) și returnează un tuplu cu aceleași elemente ca ale obiectului primit ca parametru. Următoarea secvență de cod ilustrează modalități de creare a unei variabile de tip **tuple**:

```
t = tuple("abc")
t = tuple(range(3))
```

După cum am precizat deja, un tuplu este imutabil. Astfel, elementele unui tuplu **nu** se pot modifica, mai exact nu i se poate atribui lui t[i] o nouă valoare**. Dacă însă t[i] arată însă spre un obiect mutabil, spre exemplu o listă, putem modifica valoarea acestui obiect** (pentru că astfel nu modificăm referința memorată de t[i]). Astfel, să considerăm următoarele încercări de modifica un tuplu:

```
t = (2,3)
#t[0] = 7  #TypeError: 'tuple' object does not support item
assignment
t = (1, 2, [5,6])
#t[1] = 11 #NU #TypeError: 'tuple' object does not support item
assignment
t[2][0] = 15 #DA - nu modific t[2] (referința memorată în t[2])
t[2].append(7) #DA
print(t)
```

Tuplurile **nu se pot crea prin comprehensiune** (v. subsecțiunea de mai jos Comprehensiune – generator)

Accesarea elementelor. Parcurgere

Accesarea elementelor unei se face cu metodele deja discutate la secvențe, amintite și la clasa list.

Exemplu.

```
t = (2, 4, 16, 8, 10)
print(t[0], t[-1], t[1:3], t[-2:])
for x in t:
    print(x)
```

Operatori și metode specifice secvențelor

Ca și pentru liste, pentru tupluri se pot folosi toți operatorii specifici secvențelor:

- **in, not in**
- **Concatenare +, ***

  Exemplu:

  ```
  t = (1, 2)
  t = t + (3, 4)
  t = t[:3] + (-1, -2, -3) + t[7:]
  ```

- **Comparare:** `<, <=, >=, >, >=, ==, !=` (!lelementele testate trebuie să fie comparabile)
  Exemplu: `(1, 3) > (1,2,3)` este `True` (v. list)
- **Operatorii** += , *=

  Exemplu:

  ```
  t = (1, 2)
  t *= 3
  t += (4,)
  ```

- `min, max, len`
- `count, index`

  Exemplu:

  ```
  t =(4,8,1,8,6)
  print(len(t), t.count(8), t.index(8))
  ```

- `sorted`

  Exemplu:

  ```
  t = (4, 2, 1, 8, 6)
  ls = sorted(t) #ls este list, t nu se modifica
  ```

- Metoda **deepcopy**() se poate folosi si pentru tupluri:

```
import copy
t = (3, [5, 8])
t1 = t
t1[1][0] = 11
print(t1,t)
t1 = copy.deepcopy(t)
t1[1][0] = 15
print(t1,t)
```

Particularități

Există o serie de particularități în lucru cu tupluri, unele deja amintite.

- Tuplu cu un element

  În declararea unui tuplul cu un element x se pune virgulă după unicul element x:

```
x = 2
t = (x,)
    t = (5,)
    t = (0) * 8 #NU este tuplu
t = (0,) * 8
```

- Comprehensiune - generator

  Deși o sintaxă de tip comprehensiune pentru tupluri este corectă:

```
(expresie for x in iterabil)
```

  printr-o astfel de secvență de inițializare se creează un generator, nu un tuplu.

  Elementele unui generator se generează doar atunci când este nevoie de ele (am întâlnit acest mod de generare și la range), nu la declarare. Spre exemplu:

```
t = (i*i for i in range(10))
print(t,type(t)) #nu se afiseaza numerele de la 0 la 9
print(sum(t)) #se genereaza elementele pe rand, nu se creeaza o
lista suplimentara
print(sum(t)) #dupa o generare a tuturor elementelor nu se reia
generarea, se va afișa 0
t = (i*i for i in range(10))
print(next(t)) #genereaza si returneaza urmatorul element
print(next(t))
print(sum(i*i for i in range(10)) ) #se pot omite parantezele de
la generator

l = [1,0,3,0,0,4]
l[:] = [x for x in l if x!=0]
l[:] = (x for x in l if x!=0) #nu se creează lista suplimentara

x,y,z = (int(a) for a in input().split())
```

- ## Împachetare/despachetare. Atribuire de tupluri

Amintim că există atribuire tuplu (tuple assignment)

```
x, y = 1,2
```

care este similar ca scriere cu

```
(x,y) = (1,2)
```

Chiar daca nu se pun parantezele, variabilele separate cu virgula in atribuire sunt un tuplu:

```
t = 1, 2, 3
print(type(t))
```

De asemenea, atribuirea

```
x, y = y, x
```

realizează **interschimbarea**, scrierea `y, x` fiind de fapt o împachetare în tuplu (y, x).

De asemenea, putem atribui unor variabile individuale conținutul unui tuplu:

```
t = 1,2,3
a, b, c = t
print(a, b, c)
```

Aceste facilități le putem folosi și când o funcție returnează mai multe valori (se returnează ca tuplu și rezultatul se poate atribui mai multor variabile):

```
def f():
    x = 1
    y = [10,20]
    return x,y


r = f()
print(f"rezultat {r} de tip {type(r)}")
x, ls = f()
print(f"rezultate {x} de tip {type(x)} si {ls} de tip {type(ls)}")
```

În atribuirea de tupluri se poate folosi operatorul *, pentru a grupa mai multe valori în aceeași variabilă:

```
a, *b, c = 1, 2, 3, 5
print(a,b,c,type(b))    #b este lista
```

Operatorul * se poate folosi și pentru despachetarea valorilor dintr-o secvență, cum am văzut și la `range`.

Exemplu.

```
print(range(4))
print(*range(4))

def f(x,y,z):
     print(x+y+z)
```

```
ls = [3,6,8]
#f(ls) #incorect
f(*ls)
```

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

Exercițiu: Ce afișează următoarea secvență de cod?

```
(a,*b)=(4,5,6,7)
print(a,b)
a,*b = 4,5,6,7
print(a,b)
(*b,a) =(4,5,6,7)
print(a,b)
*b,a = 4,5,6,7
print(a,b)
(*b,) = (1,2,3,4)
print(b)
*b, = (1,2,3,4); print(b)
```

Utilitate

Reamintim în final principalele avantaje ale utilizării tuplurilor:

- Atribuire multiplă
- Returnare de valori multiple
- Împachetare/despachetare (pack/unpack)
- Chei în dicționare
- Există tupluri cu nume – se pot folosi similar structurilor în C