Secvențe

Despre secvențe

O colecție este un obiect care grupează mai multe obiecte într-o singură unitate, aceste obiecte fiind organizate în anumite forme.

Prin intermediul colecțiilor avem acces la diferite **tipuri de date** cum ar fi liste, mulțimi matematice, tabele de dispersie, etc. Colecțiile sunt utile atât pentru memorarea şi manipularea datelor, cât şi pentru transmiterea unor informații de la o metodă la alta.

O secvență este o colecție de elemente indexate (**de la 0**).

Secvențele pot fi neomogene (pot avea **elemente cu tipuri diferite**), spre exemplu:

```
ls = [1, "ab", 2.5] # o lista
print(type(ls))
for i in range(len(ls)):
  print(ls[i], type(ls[i]))
```

Cele mai cunoscute tipuri de secvențe în Python sunt:

- **Liste** - clasa `list`: `ls = [5, 7, 3]`
- **Tupluri** - clasa `tuple`: `t = (5, 7, 3)`
- **Șiruri de caractere** – clasa `str`: `s = "sirul"`
- `range`

După cum elementele secvenței pot fi modificate sau nu, secvențele se clasifică în două categorii:

- **mutabile**: lista `list`

```
ls = [3,5];   ls[0] = 1;  print(ls)
```

- **imutabile**: tupluri, șiruri de caractere

```
s = "ab"
s[0]= 'a' #TypeError: 'str' object does not support item assignment
```

Vom prezenta în cele ce urmează o serie de operații uzuale comune pentru secvențele din Python (se pot folosi și pentru liste, tupluri, și pentru șiruri de caractere). În cursurile următoare vom prezenta pe rând cele mai importante tipuri de secvențe din Python evidențiind caracteristicile lor particulare, fără a prezenta din nou de fiecare data operațiile comune.

CONVENȚIE: O parte dintre funcțiile sau metodele pe care le vom descrie în acest curs au și parametri opționali (pentru care nu este obligatoriu de trimis o valoare la apel). Când vom scrie antetul unei funcții/metode vom pune între paranteze drepte parametri opționali. Spre exemplu:

`functie(x [, y])` – `x` este parametru obligatoriu, `y` este opțional (funcția se poate apela cu un parametru sau cu doi)

`functie(x [, y [, z]])` - `x` este parametru obligatoriu, `y` și `z` sunt opționali, dar `z` poate primi valoare doar dacă și pentru `y` se transmite o valoare (apelul în acest caz poate fi `f(x)`, `f(x,y)` sau `f(x,y,z)`)

Operații uzuale

- ## Lungime

  Lungimea unei secvențe se poate determina folosind funcția len:

  ```
  s = "sir"; print(len(s))
  ls = [3,1,5]; print(len(ls))
  ```

- ## Minim și maxim

  Minimul și maximul unei secvențe se poate determina folosind funcțiile **`min`** și **`max`**:

  ```
  s = "sir"; print(min(s))
  ls = [3,1,5]; print(min(ls))
  ls = [3, "a"]; print(min(ls)) #TypeError, atentie la colectii neomogene
  ```

  După cum vom vedea detalia la sortări, funcțiile de minim și maxim pot primi ca parametru și o cheie după care să se compare elementele din colecție:

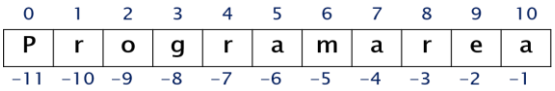
  ```
  ls = [3, -7, 2]
  print(min(ls, key=lambda x:abs(x))) #min din modulele elementelor
  ```

- ## Accesarea elementelor și subsecvențelor. Feliere (Slice)

  O secvență este indexată de la 0; elementul de pe poziția i este s[i]:

  `s[i]` = elementul de pe poziția/indexul i (numerotare de la 0)

  Indexul i **poate fi negativ**, și atunci se consideră **relativ la sfârșitul secvenței** (**len(s)+i**). De exemplu, pentru **s = "Programarea",** pentru a accesa elementul de pe poziția 2 (caracterul "o") se pot folosi atât apelul s[2], cât și s[-9], indicii elementelor pentru cuvântul Programarea fiind indicați în imaginea următoare:

  Dacă indicele i nu este valid, apelul s[i] va da eroarea **`IndexError`**.

  Cu ajutorul indicilor în Pyhon se pot accesa și subsecvențe din secvența inițială, astfel:

  `s[i:j]` = subsecvența formată cu elementele dintre pozițiile i și j, **exclusiv j** (cu indicii i, i+1, i+2,..., j-1), cu precizările:

  - indicii pot fi și negativi și pot lipsi
  - dacă indicele i lipsește, este considerat 0
  - dacă indicele j lipsește, este considerat len(s)
  - dacă indicii depășesc len(s), atunci sunt considerați len(s) (indicele care "iese" din limită este înlocuit cu limita)

  Apelul **`s[:]`** întoarce toată secvența; mai exact, în **cazul unei liste întoarce o copie a acesteia** (pentru ca este mutabilă) , în cazul unui șir de caractere întoarce chiar șirul.

Spre exemplu, pentru un șir **`s = "Programarea Algoritmilor"`** avem

- **`s[-p:]`** = sufixul de lungime p a lui s
- **`s[:p]`** = prefixul de lungime p a lui s
- **`s[::-1]`** = șirul s inversat
- **`s[:]`** este egal cu șirul s
- **`s[:len(s)+2]`** are aceeași valoare ca și s (deși s[**`len(s)+1`**] va da eroare, la feliere indicele care "iese" din limită este înlocuit cu limita)

Observație: În cazul secvențelor mutabile (de exemplu liste) **`s[:]`** este de fapt o copie a lui s, deci **`s[:]==s`** este **`True`**, dar **`s[:] is s`** este **`False`**.

Exemplu: Următorul tabel ilustrează modul în care putem accesa subsecvențe. În coloana din dreapta este trecut rezultatul afișat pe ecran pentru șirul din stânga.

`s = "Programarea"`

`print(s[3:7])`	`gram`
`print(s[:-4])`	`Program`
`print(s[:3])`	`Pro`
`print(s[5:])`	`amarea`
`print(s[-8:])`	`gramarea`
`print(s[5:2])`	
`print(s[-5:-2])`	`mar`
`print(s[12:13])`	

Se pot accesa, de asemenea, și subșiruri din secvența inițială, cu o sintaxă mai generală, în care precizăm însă și pasul cu care crește indicele i:

`s[i:j:k]` = secvența formată cu elementele dintre pozițiile **i** și **j**, exclusiv **j**, cu pasul **k** (cu indicii **i**, **i+k**, **i+2k**,**...**), cu precizările:

- dacă i sau j lipsesc, se consideră a fi valori "de final", după caz
- dacă pasul k este pozitiv și i și j depășesc len(s), sunt considerați len(s)
- dacă pasul k este negativ și i și j depășesc len(s)-1, sunt considerați len(s)-1

Spre exemplu, pentru un șir s = "Programarea Algoritmilor" avem

`s[::-1]` = șirul s inversat

Exemplu: Următorul tabel ilustrează modul în care putem accesa subșiruri cu pasul k din secvența inițială. În coloana din dreapta este trecut rezultatul afișat pe ecran pentru șirul din stânga.

```
s = "Programarea"
```

`print(s[1:6:2])`	`rga`
`print(s[6:1:-2])`	`mro`
`print(s[1::2]) #index impar`	`rgaae`
`print(s[6::-2])`	`mroP`
`print(s[12:7:-1])`	`aer`
`print(s[:3:-1])`	`aeramar`
`print(s[::-1])`	`aeramargorP`
`print(s[-2:-5:1])`	
`print(s[-2:-5:-1])`	`era`

Exercițiu – Ce va afișa următoarea secvență de cod?

```
ls = [3, 1, 11, 4, 7, 9, 5]
print(ls[1:6:2])
print(ls[6:1:-2])
print(ls[1::2])
print(ls[6::-2])
print(ls[12:5:-1])
print(ls[:3:-1])
print(ls[::-1])
print(ls[-2:-5:1])
print(ls[-2:-5:-1])
```

- ## Parcurgerea

  O secvență poate fi parcursă element cu element folosind for.

  ```
  s = "un sir" #s = [12, 13, 14]
  for c in s:
      print(c)
  ```

  Ca și în C/C++, se pot parcurge indicii i de la 0 la lungime (exclusiv) și accesa fiecare element s[i], dar caracteristic limbajului este „for-each”, parcurgerea element cu element (utilizând iteratorul):

  ```
  ls = [3, 1, 6, 10]
  for x in ls:
      print(x)
  for i in range(len(ls)):
      print(ls[i])
  ```

  Dacă dorim să accesăm și indicele și valoarea fiecărui element din colecție putem folosi funcția enumerate, care returnează un iterator al colecției cu elementele perechi de forma (indice, element din colecție):

```
for i,c in enumerate(s):
    print("pozitia ",i, " elementul ", c)
```

- ## Compararea – operatori relaționali

Pentru secvențe se pot folosi operatorii relaționali, compararea făcându-se element cu element. De exemplu:

```
s1 = "un sir"
s2 = "alt sir"
print(s1 < s2) #fals, prima litera din s1 „u” este mai mare decat „a”
t1 = (1, 4, 5)
t2 = (1, 1, 2, 3)
print(t1 > t2) #adevarat, al doilea element din t1 > cel din t2
l1 = [1, 2, 3]
l2 = [2, 3, 1]
print(l1 == l2)#fals, ordinea conteaza, secventa este ordonata
```

- ## Teste de apartenență – operatorii in, not in

Apartenența unui element la o secvență se poate verifica folosind operatorii **in**, **not in** de care am amintit în cursul privind elementele fundamentale de limbaj, ca în exemplele:

```
s = "un sir"
if "a" not in s:
    print("sirul nu contine litera a ")
if "si" in s: #pentru siruri putem verifica si subsecvente
    print("sirul contine secventa si")
ls = [3,1,4,0,8]
if 0 in ls:
    print("lista are elemente nule")
```

- ## Căutări. Interogări

Când căutăm o valoare într-o colecție, ne putem dori să aflăm:

- dacă valoarea există - atunci putem folosi **operatorul `in`**
- de câte ori apare în colecție - atunci putem folosi metoda **`count`**
- poziția (prima, toate) pe care apare, și atunci putem folosi metoda **`index`**

Când apelăm funcții de căutare trebuie să citim în documentație ce se întâmplă dacă valoarea căutată nu se găsește în colecție. Spre exemplu, dacă o valoare nu există în secvență, funcția index dă eroarea **`ValueError`**, nu returnează -1, False sau None ca în alte limbaje. Pentru ca programul să își continue execuția în acest caz (altfel se va opri cu eroarea **`ValueError`**), vom prezenta și metode de tratare a excepțiilor. Pentru șiruri de caractere există o metodă similară cu index, numită find, care în caz că valoarea nu este în listă returnează -1 (vom detalia la șiruri).

Astfel :

- **`s.index(x[,start[,end]])`** => returnează poziția primei apariții a lui x în s; 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 s
- **`s.count(x[, start[, end]]))`** => returnează numărul de apariții în s ale valorii x primite ca parametru; dacă se precizează și parametri opționali, aceștia sunt interpretați ca la feliere, numărarea făcându-se în subsecvența s[start:] dacă se transmite valoare pentru start, respectiv s[start:end] dacă au valori ambii parametri opționali

Exemple:

```
s = 'acesta este un sir'
print("Sirul 'es' apare de ", s.count('es'), " ori")
print("Litera 'e' apare de ", s.count('e',0,10), " ori printre
primele 10 caractere ale sirului")
```

```
s = "programarea"
x = s.index('a')
print("Prima pozitie pe care apare litera 'a' este ", x)
x = s.index('a', x+1)
print("A doua pozitie pe care apare litera 'a' este ", x)
x = s.index('ar')
print("Prima pozitie pe care apare sirul 'ar' este ", x)
x = s.index('b')
print("Prima pozitie pe care apare litera 'b' este ", x)
```

La execuția apelului **`s.index('b')`** este dată eroarea **`ValueError`** și execuția programului se întrerupe, nu se mai ajunge la ultima linie. Dacă vrem însă ca utilizatorului să i se afișeze un mesaj corespunzător și execuția să continue, putem trata excepția **`ValueError`** folosin blocuri **`try...except`**:

```
s = 'Programarea'
try:
  x = s.index('b')
  print("Prima pozitie pe care apare litera b este ", x)
except ValueError:
  print("Litera b nu apare in sir")
```

În clauza **`except`** putem să nu specificăm numele erorii, și atunci se vor executa instrucțiunile din **`except`** pentru orice eroare aruncată de instrucțiunile din **`try`** (în practică este de preferat să tratăm erorile diferențiat), spre exemplu:

```
ls = [6,8,10,2,8,5]
x = -1
c = int(input())
try:
    x = ls.index(c)
except:
    pass
print(x)
```

În cazul în care valoare **c** nu este în listă, variabila **x** va rămâne cu valoarea **-1**. O secvență de cod similară, fără a folosi instrucțiunea **pass** ar fi:

```
ls = [6,8,10,2,8,5]
c = int(input())
try:
  x = ls.index(c)
except:
  x = -1
print(x)
```

(atenție, variabila **x** se inițializează atât în **try**, cât și în **except**; dacă înlocuim atribuirea **x=-1** cu **pass**, în cazul în care **c** nu este în **ls** variabila **x** nu se va creea și afișarea va da eroarea **NameError: name 'x' is not defined)**

- ## Concatenari

Folosind operatorul + sau înmulțirea cu un număr se pot concatena secvențe.

1. **s + t =** o secvență nouă, obținută prin concatenarea secvențelor s și t

Exemplu:

```
s = "Programarea"
t = "Algoritmilor"
print("s=", s, "t=", t, id(s),id(t))
s + t
print("s=", s, "t=", t, id(s),id(t)) # s și t nu se modifică
s = s + " " + t
print("s=", s, "t=", t, id(s), id(t))
#obiect nou, nu adauga la vechiul s
```

La concatenarea de obiecte **imutabile** se creează mereu **un nou obiect**, elementele secvențelor care se concatenează fiind copiate (atenție, se copiază referințele, după cum vom ilustra în exemplul următor). Astfel, operațiile de concatenări succesive de secvențe pot avea **complexitate mare. Este recomandat să folosim extinderi de liste și join** pentru a evita concatenări repetate

Pentru a ilustra cum se face copierea (!superficială, de referințe) elementelor la concatenarea celor două secvențe pentru a obține secvența nouă, considerăm următoarele exemple:

```
s = [1,4,6]
t = [8,10]
print("s=", s, "t=", t, id(s),id(t))
s + t
print("s=", s, t, id(s),id(t))
s1 = s +  t
print("s1=", s1,"s=",s, "t=", t, id(s1), id(s),id(t))
s1[0] = 12
print(s1,s) #s nu se modifica
s[0] = 14
print(s1,s) #s1 nu se modifica
```

În exemplul de mai sus avem **`s1 = s + t;`** dacă modificăm **`s1[0]`** nu se modifică **`s[0]`** și nici invers (dacă modificăm **`s[0]`** nu se modifică **`s1[0]),`** atribuirea **`s1[0] = 12`** având ca efect mutarea referinței **`s1[0]`** către alt obiect.

Să considerăm însă și exemplul următor:

```
s = [[1,3],4,6]
t = [8,10]
s1 = s +  t
print("s1=", s1,"s=",s, "t=", t, id(s[0]),id(s1[0]))
s1[0][0] = 14
print(s1,s)  #se modifica si s, s[0] si s1[0] refera acelasi
obiect, s-a copiat referinta
s[0][0] = 15
print(s1,s) #se modifica si s1
```

În acest exemplu avem tot **`s1 = s + t`**, dar **`s[0]`** este o listă; atunci **`s1[0]`** va fi referință către aceeași listă. Astfel, dacă modificăm primul element din **`s1[0]`** se va modifica și **`s[0]`** (și invers), ambele referind același obiect; dacă însă ii vom atribui lui s1[0] o nouă valoare (!un obiect nou), atunci **`s1[0]`** și **`s[0]`** vor referi obiecte diferite:

```
s1[0]=[4,55,19]
print(s1,s)
s1[0][0] = 18
print(s1,s) #s nu se mai modifica
```

Dacă vrem să adăugăm elemente la finalul unei liste este indicat să folosim metoda **`extend`** astfel: **`ls.extend(ls1)`** (sau operatorul **`+=`**), nu concatenare de tipul **`ls = ls + ls1`**, aceasta fiind mai lentă (se creează un obiect nou copiind elementele din ls și ls1, apoi ls va arăta către acesta, deci va avea alt id). Vom reveni asupra metodelor din clasa list în cursul dedicat acestui tip de secvență.

```
import time
ls = list(range(1, 500))
lsc = []
start_time = time.time()
for i in range(10000):
    ls.extend(lsc)
print(time.time() - start_time)
ls = list(range(1, 500))
lsc = []
start_time = time.time()
for i in range(10000):
    ls = ls + lsc
print(time.time() - start_time)
```

2. Multiplicarea unei secvențe de n ori

```
s * n
```

```
n * s
```

Rezultatul se obține concatenând de n ori secvența s (dacă n<0 se obține secvența vidă). De exemplu:

```
n = 4
s = "ab"
print(s*n)
```

Amintim că se copiază referințe (copiere "superficială"), de aceea trebuie atenție la copierea secvențelor care au ca elemente la rândul lor alte secvențe mutabile, de exemplu liste de liste.

Exercițiu: Ce afișează următoarea secvență de cod? Justificați

```
ls = [[1]]*3 #lista cu 3 elemente, fiecare element este lista [1]
print(ls)
ls[1].append(2)
print(ls) #se modifica toate cele 3 elemente, s-au multiplicat referintele
```

- ## Sortarea

Sortarea unei secvențe se poate face folosind funcția **`sorted`**.

Funcția **returnează** o **listă** (!indiferent de tipul secvenței pe care o sortăm) cu elementele secvenței ordonate. Funcția **nu** modifică secvența primită ca parametru

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

Exemple:

```
s = "alfabet"
ls = sorted(s)
print(ls) #rezultatul este lista, desi s e sir
```

```
ls = [7, 2, 4, 3]
ls1 = sorted(ls, reverse = True) #descrescator
print("lista sortata descrescator: ", ls1)
print("lista initiala nu s-a modificat: ",ls)
```

Vom reveni cu detalii legate de cum putem specifica un criteriu propriu de ordonare când vom discuta despre liste și funcții.