Capitolul 4. Siruri avansat, tuplu, lista, set, dictionar

Pag.

4.1 **Siruri de caractere avansat** 02

4.1.1 Indexarea sirurilor de caractere 02

4.1.2 Slice intr-un sir de caractere 03

4.1.3 Type 04

4.1.4 Mutabilitate 06

4.2 **Tuplu** 07

4.2.1 Definitie, generalitati, ***for, len, type*** intr-un tuplu 07

4.2.2 Metode de manipulare a tuplurilor 09

4.3 **Lista** 09

4.3.1 Definitie, generalitati, ***del, len, type*** intr-o lista 09

4.3.2 Metode de manipulare a listelor 11

4.3.3 Indexare si slicing intr-o lista 14

4.3.4 Aplicatii ***for*** intr-o lista 15

4.4 **Set** 17

4.4.1 Definitie, generalitati, ***del, len, type*** intr-un set 17

4.4.2 Metode de manipulare a seturilor 19

4.5 **Dictionar** 22

4.5.1 Definitie, generalitati, ***del, len, type*** intr-un dictionar 22

4.5.2 Metode de manipulare a dictionarelor 24

4.5.3 Aplicatii ***for*** intr-un dictionar 26

4.6 **Recapitulare** 27

4.6.1 Recapitulare 27

4.1 Siruri de caractere avansat

4.1.1 Indexarea sirurilor de caractere

Indexarea sirurilor de caractere, este un procedeu prin care identificam in mod unic fiecare caracter al unui sir, prin intermediul unui numar asociat. Acest lucru este foarte util atunci cand cautam in interiorul sirului diferite caractere sau subsiruri.

0	1	2	3	4	5
c	u	v	a	n	t
-6	-5	-4	-3	-2	-1

Dupa cum se observa in imagine, indexarea se poate face de la stanga la dreapta, incepand cu zero si pana la numarul de caractere al sirului minus unu sau de la dreapta catre stanga, incepand cu minus unu si pana la minus numarul de caractere al sirului.

Putem identifica in mod unic un character prin expresia: **sir[index]**.

```
>>>x = 'cuvant'
>>> print ( x[ 2 ] )
v
>>> print ( x[ -2 ] )
n
```

Atentie, daca indexul este inexistent (depaseste plaja de valori determinata de lunginea sirului) va fi generata o eroare:

```
>>> print ( x[ 6 ] )
Traceback (most recent call last):
  File "<input>", line 1, in <module>
IndexError: string index out of range
```

In functie subsirul cautat si pozitia ocupata putem utiliza indexul pozitiv sau pe cel negativ. Daca nu stim care este lungimea sirului putem aplica functia len(). Indexul maxim va fi **len(sir) – 1**.

Putem sa ne folosim de indecsi in combinatie cu una din buclele invatate, while sau for.

In exemplul urmator utilizam o bucla pentru a returna ***fiecare litera a sirului si indexul*** corespunzator:

```
>>>index = 0                    # initializam o variabila index
>>>cuv = 'academia'
>>>for lit in cuv:
        print ( index, '-', lit )    # printeaza combinatia index - litera
        index = index + 1            # incrementeaza variabila pentru continuarea buclei
0 - a

1 - c

2 - a

3 - d

4 - e

5 - m

6 - i

7 - a
```

Putem ***numara aparitiile*** unei litere intr-un sir:

```
>>>count = 0                  # initializam o variabila count, care numara aparitiile
>>>for lit in cuv:
        if lit == 'a':
        count = count + 1     # incrementeaza valoarea variabilei (numarul aparitiilor)
>>>print(count)
3
```

4.1.2 Slice intr-un sir de caractere

Slicing este o metoda de cautare intr-un sir de caractere, bazata pe indexul acestuia si care returneaza o portiune din sir determinata de compozitia slicingului folosind urmatoarele tipare:

```
>>>cuv = 'Telecinemateca'
          0 1 2 3 4 5 6 7  8  9 10 11 12 13

>>>print(cuv[ 0:4 ])    # incepe de la primul index, se  termina la al doilea fara sa-l includa
Tele
>>>print(cuv[ 4:8 ])    # incepe cu primul index, se termina la al doilea fara sa-l includa
cine
>>>print(cuv[ 4:20 ])   # daca al doilea index depaseste len(string) se opreste la sfarsitul
                        stringului
cinemateca
>>>print(cuv[ :4 ])     # returneaza primele caractere fara sa includa argumentul final
Tele
>>>print(cuv[ 12: ])    # returneaza toate caracterele incepand cu argumentul dinaintea ':'
ca
>>>print(cuv[ : ])      # returneaza tot stringul
Telecinemateca
>>>copy_cuv = cuv[ : ]          # facem o copie a stringului intr-o alta variabila
>>>print(cuv[ 0 : 13 : 2 ])
Tlcnmtca                # slicing intre primul si al doilea index cu pas dat de al treilea numar
```

4.1.3 Type

Type este o functie care determina natura obiectului in Python, tipul de date, clasa din care face parte.

```
>>> print( type( x )  )                    # indica tipul de date al variabilei
<class 'str'>
>>> print( type( 'string' ) )
<class 'str'>
>>> print( type( 25 ) )
<class 'int'>
>>> print( type( 42.5 ) )
<class 'float'>
>>> print( type( True ) )
<class 'bool'>
```

```
>>> print( type( None ) )
<class 'NoneType'>
```

In python exista tipuri de date, obiecte predefinite, cum ar fi numerele, stringurile, clasele, instantele claselor sau liste, dictionare, etc. Pentru unele dintre ele ati vazut mai sus rezultatul aplicarii functiei type.

Putem utiliza rezultatul type (str, int, float,etc) in expresii si conditii:

```
>>> x = 3
>>> if type ( x ) == int:
        print ( x ** x )
27
```

Pornind de la tipul unui obiect putem sa facem conversii de la float la intreg si invers, de la sir de caractere la numar intreg sau float, dupa caz, (daca este compus doar din cifre) si invers. Pentru aceasta utilizam functiile str(), int() si float(), care vor primi ca argument valoarea de convertit. Un exemplu simplu este convertirea unui sir de caractere captat de la tastatura, cu input, intr-un numar.

```
>>> print( 'Adresa de broadcast este: ' + str( 255 ) )
                                        # poate fi util cand concatenam doua siruri
'Adresa de broadcast este: 255'
>>> print( int( '255' ) + 45 )          # expresii numerice
300
>>> print( float( '22.5' ) / 3 )
7.5
>>> an_nastere = input("Introduceti anul nasterii: ")
>>> varsta = 2017 - int( an_nastere )
```

4.1.4 Mutabilitate

Un obiect, in Python, este mutabil daca poate fi modificat dupa ce l-am creat deja. Imutabil inseamna ca odata creat nu poate fi modificat. Exista diferente de la un tip de date la altul in ceea ce priveste mutabilitatea.

Ce trebuie retinut este ca numerele, sirurile de caractere si tuplurile sunt ***imutabile***. Pe de alta parte, listele si dictionarele sunt ***mutabile***. O situatie aparte avem pentru set. Exista set, care este mutabil si frozenset, care este imutabil. De asemenea, mutabilitatea functioneaza diferit pentru fiecare tip de obiect in parte.

Un sir de caractere este ***imutabil***. Daca vom incerca sa inlocuim un caracter vom obtine o eroare.

```
>>> x = 'sapa'
>>> x[0] = 'm'
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Atentie, daca folosim metoda replace, cu ajutorul careia putem face modificari intr-un sir de caractere, modificarea nu va fi posibila in sirul initial, ci doar in cel creat cu metoda replace, dar care nu este considerata mutabilitate. Practic, cu ajutorul metodei replace, cream un nou sir.

```
>>> y = x.replace( x[0], 'm' )
>>> print ( y )
mapa
```

Acelasi lucru altfel:

```
>>> z = x.replace('sa', 'ram')
>>> print ( z )
rampa
```

4.2 Tuplu

4.2.1 Definitie, generalitati, *for*, *len*, *type* intr-un tuplu

Tuplu este o colectie de date, de regula eterogene. Un tuplu poate contine urmatoarele tipuri de date:

- Numere;
- Siruri de caractere;
- Liste;
- Dictionare;
- Variabile care contin aceste tipuri de date;
- Boolean;
- Alte tupluri;
- Alte tipuri de date.

Un tuplu este reprezentat prin paranteze rotunde. Eelementele unui tuplu sunt despartite de virgula.

Sintaxa: un tuplu poate fi creat in mai multe moduri:

```
>>> x = tuple()          # initializarea unui tuplu fara date (empty tuple)
>>> print ( x )
()
>>>y = 'x',              # utilizam un singur element urmat de virgula
>>> print ( y )
('x',)
>>> z = tuple( 'acum' )  # un tuplu iterabil, caracterele devin elemente ale tuplului
>>> print ( z )
('a', 'c', 'u', 'm')
>>> w = 1,2,3,4          # pornind de la elemente despartite de virgula
>>> print ( w )
```

(1, 2, 3, 4)

Elementele unui tuplu sunt indexabile si numarabile. Un tuplu este o variabila imutabila. Elementele tuplului nu pot fi modificate, nu putem adauga sau sterge elemente, ceea ce face ca un tuplu sa fie ***imutabil***. In schimb, putem schimba cu totul componenta unui tuplu, atribuindu-i alte elemente.

Un tuplu permite elemente duplicate.

In bazele de date, un rand (o inregistrare) este asimilata unui tuplu.

Exemplu, un tuplu care contine variabila var (variabila trebuie sa fie definita anterior crearii tuplului), numarul 1, stringul "aici", un alt tuplu ('x', 'y') si Boolean, True:

```
>>> var = 10
>>> tuplu_1 = ( var, 1,  'aici', ( 'x', 'y' ), True )
```

Putem aplica for prin elementeele unui tuplu.

Unui tuplu i se pot aplica functiile ***len*** si ***type***, astfel:

```
>>> print ( len ( tuplu_1 ) )          # numara cate elemente are tuplul
5
>>> print ( type ( tuplu_1 ) )         # returneaza clasa din care face parte
<class 'tuple'>
>>> for element in tuplu_1:            # returneaza elementele unui tuplu
        print ( element )
10
1
'aici'
( 'x', 'y' )
True
```

Tuplurilor li se pot aplica si operatorii de comparare. Compararea se va face element cu element in functie de pozitia pozitia(indexul) acestuia. Pentru a fi comparabile, elementele cu acelasi index trebuie sa aiba acelasi tip de date.

4.2.2 Metode de manipulare a tuplurilor

```
>>> print ( dir( tuplu_1 ) )          # returneaza metodele aplicabile pentru tuplu
'index', 'count'
```

Dupa cum se observa, avem doua metode de manipulare a tuplurilor.

index primeste ca parametru un element al tuplului si returneaza indexul acestuia. Indexarea intr-un tuplu se aplica elementelor, incepand de la zero, pentru primul element.

count primeste ca parametru un element al tuplului si numara aparitiile acestuia.

```
>>> print ( tuplu_1.index( 'aici' ) )
2
>>> print ( tuplu_1.count( 'aici' ) )
1
```

4.3 Lista

4.3.1 Definitie, generalitati, *del*, *len*, *type* intr-o lista

Lista este o colectie de date, de regula omogene. O lista poate contine urmatoarele tipuri de date:

- Numere;
- Siruri de caractere;
- Tupluri;
- Alte liste;
- Dictionare;
- Variabile care contin aceste tipuri de date;
- Boolean
- Alte tipuri de date.

Sintaxa: o lista poate fi creata in mai multe moduri:

```
>>> list()                  # initializarea unei liste fara date (empty list). Acelasi rezultat si cu [].
[]
>>> a,b = [ 'x' ], [1, 2, 3]          # utilizam elementele listei despartite de virgula
>>> print ( a )
[ 'x' ]
>>> print ( b )
[1, 2, 3]


>>> [ i for i in  'acum' ]          # iterand cu for printr-un sir. Echivalent cu list('acum' )
[ 'a', 'c', 'u', 'm' ]
```

Elementele unei liste sunt cuprinse in paranteze drepte. De regula, elementele sunt de acelasi tip. Totusi, ca si tuplul, lista permite si elemente eterogene si duplicate.

Lista este mutabila. Ca efect, putem inlocui, adauga si sterge elemente din lista, ceea ce o face un obiect mult mai puternic si util in programare decat tuplul.

Unei liste i se pot aplica functiile ***del, len*** si ***type***, astfel:

```
>>> lista2 = [ 1, 2, 3 , 5, 8, 13 ]
>>> lista2 [ 5 ] = 19               # mutabilitate, inlocuim elementul cu indexul 5
>>> print ( lista2 )
[ 1, 2, 3 , 5, 8, 19 ]
>>> lista2[ -6 ] = 10               # mutabilitate, inlocuim elementul cu indexul -6
>>> print ( lista2 )
[ 10, 2, 3 , 5, 8, 19 ]
>>> print ( len ( lista2 ) )        # numara cate elemente are lista
6
>>> print ( type ( lista2 ) )       # returneaza clasa din care face parte
<class 'list'>
```

>>> **del (lista2[4])** sau

>>> **del lista2[4]** # stergem elementul cu indexul 4 din lista.

>>> **del (lista2)** sau

>>> **del lista2** # stergem toata lista, echivalent cu **lista2 = [ ]**.

4.3.2 Metode de manipulare a listelor

>>> **dir(lista2)** # returneaza metodele aplicabile pentru o lista

'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort'

Metodele de manipulare se pot schimba, pe masura ce noi versiuni de Python se dezvolta. Pentru a fi la curent verificati periodic ce metode sunt disponibile prin comanda dir() aplicata tipului corespunzator (lista, tuplu, dictionar, etc.).

Metoda append(valoare) adauga elementul "valoare" in lista. Elementul va fi adaugat la sfarsitul listei. Daca dorim sa asiguram unicitatea elementelor dintr-o lista va trebui sa conditionam adaugarea de inexistenta elementului.

>>> **lista2 = [1, 2, 3, 4, 5, 6]**

>>> **lista2.append(20)**

>>> **print (lista2)**

[1, 2, 3, 4, 5, 6, 20]

Metoda copy( ) creeaza o copie a listei. Cele doua liste sunt egale, dar nu ocupa acelasi spatiu de memorie. Am facut aceasta precizare, deoarece, in cazul crearii unei liste prin atribuire, cu expresia lista_x = lista_y acestea sunt egale si ocupa acelasi spatiu de memorie. In acest din urma caz o modificare in prima lista se propaga si in cea de-a doua, ceea ce in cazul copy nu se intampla.

>>> **lista3 = lista2.copy( )**

>>> **print (lista3)**

[1, 2, 3, 4, 5, 6, 20]

Metoda clear( ) sterge toate elementele listei.

```
>>> lista3.clear( )
>>> print ( lista3 )
[ ]
```

Metoda count(valoare) numara aparitiile elementului "valoare" in lista.

```
>>> print ( lista2.count( 5 ) )
1
```

Metoda extend( ) poate primi ca argument un string sau o lista. In cazul in care argumentul este string va face split si va introduce ca elemente ale listei fiecare caracter. In cazul in care argumentul este o lista va adauga elementele listei primite ca argument la lista careia ii aplicam metoda.

```
>>> lista2.extend( 'acum' )      # adauga elementele [ 'a', 'c', 'u', 'm' ] echivalent cu
                                 # lista2.extend( [ 'a', 'c', 'u', 'm' ] )
>>> print ( lista2 )
[ 1, 2, 3, 4, 5, 6, 20, 'a', 'c', 'u', 'm' ]
>>> lista2.extend( [ 20, 30 ] )          # adauga elementele [ 20, 30 ]
>>> print ( lista2 )
[ 1, 2, 3, 4, 5, 6, 20, 'a', 'c', 'u', 'm' , 20, 30 ]
```

Metoda index(valoare[, pozitie_inceput]) returneaza indexul primei aparitii a elementului "valoare". Cautarea se va face implicit de la inceputul sirului (pozitia de inceput poate sa lipseasca, implicit zero).

Daca dorim sa caute incepand cu alta pozitie vom mentiona al doilea argument.

```
>>> print ( lista2.index( 20 ) )          # cauta de la inceputul sirului
6
```

```
>>> print ( lista2.index( 20, 7 ) )      # cauta de la indexul cu nr 7
11
```

Metoda insert(index, valoare) atribuie o noua valoare elementului cu indexul primit ca argument. Este echivalenta cu **lista[index] = valoare**.

```
>>> lista2.insert( 6, 25 )
>>> print ( lista2 )
[ 1, 2, 3, 4, 5, 6, 25, 'a', 'c', 'u', 'm' , 20, 30 ]
>>> lista2 [ 1 ] = 50
>>> print ( lista2 )
[ 50, 2, 3, 4, 5, 6, 25, 'a', 'c', 'u', 'm' , 20, 30 ]
```

Metoda pop([index]) sterge elementul cu indexul primit ca argument si printeaza elementul sters. Daca nu-l mentionam va sterge **ultimul** element din lista.

```
>>> lista2.pop( 7 )
'a'
>>> print ( lista2 )
[ 50, 2, 3, 4, 5, 6, 25, 'c', 'u', 'm' , 20, 30 ]
>>> lista2.pop( )
30
>>> print ( lista2 )
[ 50, 2, 3, 4, 5, 6, 25, 'c', 'u', 'm' , 20 ]
```

Metoda remove(valoare) sterge elementul "valoare" din lista.

```
>>> lista2.remove( 'c' )
>>> lista2.remove( 'u' )
>>> lista2.remove( 'm' )
```

>>> **print (lista2)**

[50, 2, 3, 4, 5, 6, 25, 20]

Metoda reverse( ) inverseaza ordinea elementelor unei liste.

>>> **lista2.reverse( )**

>>> **print (lista2)**

[20, 25, 6, 5, 4, 3, 2, 50]

Metoda sort( ) sorteaza elementele listei. Atentie, elementele trebuie sa fie de acelasi tip, in caz contrar va fi generata o eroare.

>>> **lista2.sort( )**

>>> **print (lista2)**

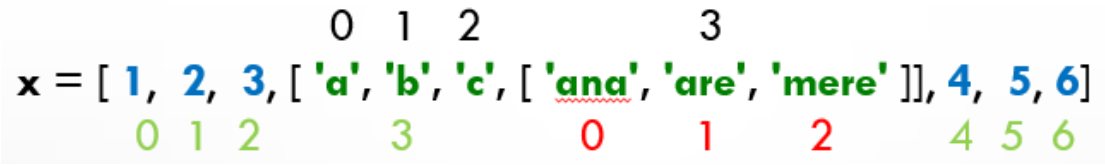
[2, 3, 4, 5, 6, 20, 25, 50]

Metodele copy si clear nu exista in versiunea 2.x. Metoda sort, in versiunea 2.x accepta elemente de tipuri diferite.

4.3.3 Indexare si slicing intr-o lista

Inainte de a discuta despre inxexare si slicing intr-o lista, trebuie sa stiti ca o lista poate cuprinde o alta lista, care la randul ei poate sa cuprinda o alta lista, etc. Acestea se numesc liste imbricate (nested lists).

Pentru a exemplifica cum se aplica indexarea si slicing-ul intr-o lista am apelat la un exemplu care contine 7 elemente, indexate de la 0 la 6 (culoarea verde). Aceasta contine o alta lista cu 4 elemente, indexate de la 0 la3 (culoarea negru), dintre care, al treilea element este tot o lista, cu 3 elemente, indexata de la 0 la 2 (culoarea rosu)



Indexarea este similara cu cea a tuplurilor, fiecarui eleement al unei liste fiindu-i atribuit un numar de la zero la n-1, in care n este numarul de elemente.

```
>>> print (x[0:2])          # similar cu print (x[:2]), elementul cu indexul 2 nu este inclus
[1, 2]

>>> print (x[4:])           # de la indexul 4 pana la sfarsit
[4, 5, 6]

>>> print (x[2:5])          # elementul cu inddexul 5 nu este inclus
[3, ['a', 'b', 'c', ['ana', 'are', 'mere']], 4]

>>> print (x[3][3][1:])     # slice in elementul cu indexul 3 al listei de baza, elementul cu
                            # indexul 3 din lista imbricata si respectiv de la indexul 1 pana la
                            # sfarsit in ultima lista imbricata
['are', 'mere']]


>>> print (x[:])            # toata lista
[ 1,  2,  3, ['a', 'b', 'c', ['ana', 'are', 'mere']], 4,  5, 6]

>>> print (x[0:6:2])        # slice cu pas de 2 (din 2 in 2), echivalent cu print (x[::2])

[1, 3, 5]
```

4.3.4 Aplicatii *for* intr-o lista

Cea mai simpla aplicatie este redarea elementelor unei liste:

```
>>> lista_4 = ['pisica', 'caine', 'cal', 'oaie', 'vaca', 'capra']
>>> for i in lista_4:
        print ( i )              # va printa toate elementele, unul sub altul
pisica
caine
cal
oaie
vaca
capra
```

O alta posibilitate este sa prelucram elementele listei. Spre exemplu le putem adauga un numar de identificare:

```
>>> nr = 1
>>> lista_4.sort()
>>> for i in lista_4:
        print ( str(nr) + ' ' + i )     # concatenarea se face doar pentru siruri de caractere
        nr += 1                         # incrementam nr pentru urmatorul element
1 caine
2 cal
3 capra
4 oaie
5 pisica
6 vaca
```

sau folosind functia **enumerate()**:

```
>>> for x, y in enumerate(lista_4):
        print ( x + 1, y )              # enumerate incrementeaza valori cronologice de la 0

1 caine
2 cal
3 capra
4 oaie
5 pisica
6 vaca
```

Sau aceeasi prelucrare pentru crearea unei noi liste cu elementele modificate:

```
>>> nr = 1
>>> lista_4.sort()
>>> lista_5 = [ ]
>>> for i in lista_4:
        lista_5.append( str(nr) + ' ' + i ) )  # concatenarea este doar pentru siruri de caractere
        nr += 1

>>> print ( lista_5 )

['1 caine', '2 cal', '3 capra', '4 oaie', '5 pisica', '6 vaca']
```

Putem sa determinam pentru fiecare element literele componente (sau cuvintele) in cazul in care elementele sunt texte:

```
>>> for index in range(len(lista_4)):
        print( lista_4[ index ],', contine urmatoarele caractere:' )
        for animal in lista_4[ index ]:
            print (animal.split())
caine  contine urmatoarele caractere:
['c']
['a']
['i']
['n']
['e']
cal, contine urmatoarele caractere:
.........
```

Posibilitatile sunt multiple si vor putea fi folosite pe masura ce invatam despre alte tipuri de date si pe masura ce ne dezvoltam abilitatile de programatori.

4.4 Set

4.4.1 Definitie, generalitati, *del*, *len*, *type* intr-un set

Setul este o colectie de date, asemanatoare cu lista, dar cu elemente unice. Un set poate contine urmatoarele tipuri de date:

- Numere;
- Siruri de caractere;
- Tupluri;
- Variabile care contin aceste tipuri de date;
- Boolean;

Sintaxa: un set va contine elementele intre acolade { }.

Putem transforma un set intr-o lista si invers. In cazul transformarii unei liste in set, eventualele duplicate din lista vor fi eliminate.

Un set nu este indexabil. Putem folosi bucle si operatori decizionali cu un set, putem aplica functia len() pentru determinarea numarului de elemente ale unui set si putem utiliza operatorul ***in*** pentru a verifica existenta unui element in set.

```
>>> set_1 = { }                    # initializarea unui set fara date (empty set)
>>> set_2 = { 1, 2, 3 }            # crearea unui set cu elementele despartite de virgula
>>> print( set_2 )
{ 1, 2, 3 }
>>> lista_6 = list( set_2 )        # crearea unei liste cu elementele unui set
>>> print( lista_6 )
[ 1, 2, 3 ]
>>> set_3 = set( lista_4 )         # crearea unui set cu elementele unei liste
>>> print( set_3 )
{'oaie', 'cal', 'pisica', 'caine', 'capra', 'vaca'}

>>> for i in set_2:
        print ( i )                 # va printa toate elementele, unul sub altul
1
2
3


>>> print( type( set_3 ) )
<class 'set'>
>>> print( len( set_4 ) )          # lungimea setului (numarul de elemente)
6
```

4.4.2 Metode de manipulare a seturilor

Vizualizam metodele aplicabile unui set si explicam cum sunt utilizate:

```
>>> dir( set_4 )
```

'add', 'clear', 'copy', 'difference', 'difference_update', 'discard', 'intersection', 'intersection_update', 'isdisjoint', 'issubset', 'issuperset', 'pop', 'remove', 'symmetric_difference', 'symmetric_difference_update', 'union', 'update'

Metoda add(valoare), adauga elementul valoare daca nu exista:

```
>>>set1 = {1, 2, 3, 4, 5}
>>>set2 = {3, 4, 5, 6, 7, 8}

>>>set1.add ( 6 )                    # Adauga elementul 6
>>> print (  set1 )
{1, 2, 3, 4, 5, 6}
```

Metoda clear( ), sterge toate elementele setului:

```
>>>set2.clear ( )
>>>set2 = {3, 4, 5, 6, 7, 8}        # Adaugam din nou elementele setului
```

Metoda copy( ), creaza o copie a setului. Desi setul va cuprinde aceleasi elemente (este egal), nu va ocupa acelasi spatiu de memorie.

```
>>>set3 = set1.copy ( )             # Creaza o copie a setului
>>>id ( set3 )
>>>id ( set1 )                      # Testam spatiiul de memorie ocupat

>>>print ( set3 is set1 )

False
>>>print ( set3 == set1 )
True
```

Metoda difference(set) returneaza elementele setului 1, mai putin elementele comune cu setul 2 (sterge din setul 1 elementele care se gasesc si in setul 2 si le returneaza pe cele ramase):

```
>>>print ( set1 - set2 )
{1, 2}

>>>print ( set1.difference ( set2 ) )
{1, 2}
```

Metoda discard(valoare), elimina elementul valoare daca exista:

```
>>>set1.discard ( 6 )
>>>print ( set1 )
{1, 2, 3, 4, 5}
```

Metoda intersection(set), returneaza doar elementele comune din cele doua seturi:

```
>>>print ( set1 & set2 )

{3, 4, 5}
>>>print ( set2.intersection ( set1 ) )
{3, 4, 5}
```

Metoda pop( ), sterge **primul** element al setului:

```
>>>set1.pop ( )
>>>print ( set1 )
{2, 3, 4, 5}
```

Metoda remove(valoare), sterge elementul valoare daca exista. In caz contrar returneaza eroare:

```
>>>set2.remove ( 6 )
>>>print ( set2 )
{3, 4, 5, 7, 8}
```

Metoda* symmetric_difference *(set), reuneste toate elementele celor doua seturi, mai putin elementele comune, care vor fi eliminate:

```
>>>print ( set1 ^ set2 )

{2, 7, 8}
>>>print ( set2.symmetric_difference ( set1 ) )
{2, 7, 8}
```

Metoda* union *(set), returneaza valorile unice din cele 2 (reuneste toate elementele celor doua seturi, inclusiv elementele comune, care vor incluse o singura data (asigurand unicitatea elementelor din set):

```
>>> print ( set2 | set3  )

{2, 3, 4, 5, 7, 8}
>>> print ( set2.union ( set3 ) )
{2, 3, 4, 5, 7, 8}
```

Metoda* update *(set), adauga la set elementele setului primit ca argument (elimina duplicatele):

```
>>>set2.update ( set3 )
>>>print ( set2 )
{1, 2, 3, 4, 5, 6, 7, 8}

>>>len ( set2 )
8
>>>if 8 in set2:                          # cautare intr-un set dupa numele elementului
        print ( 'Element gasit!' )

Element gasit
```

4.5 Dictionar

4.5.1 Definitie, generalitati, *del*, *len*, *type* intr-un dictionar

Dictionar (dict) este un tip de date compus din cheie si valoare. Cheia poate fi numerica, sir de caractere sau chiar un tuplu care nu are in componenta elemente mutabile (este compus din string si numere). Poate contine ca valori: numere, stringuri, boolean, tuplu, lista, set, dictionar, variabile care contin tipurile permise.

Dictionarele ne permit sa lucram cu baze de date. Ele sunt asimilate cu alte elemente identice din alte limbaje (Associative Arrays - Perl / PHP, HashMap – Java, Property Bag - C# / .Net).

Sintaxa este de forma: **{cheie_1: valoare_1, cheie_2: valoare_2, ...}**

Dictionarele sunt indexabile dupa cheie. Cheile trebuie sa fie unice. Mutabilitatea este asigurata la dictionare prin intermediul cheilor. Practic, putem adauga un element cu cheie noua si valoarea corespunzatoare, putem sterge un element sau putem modifica valoarea unui element pastrand cheia.

Un dictionar poate fi creat prin mai multe medode. Principalele metode sunt descrise in continuare:

Direct prin scrierea elementelor de tip cheie: valoare:

```
>>>d1 = {1: 'rosu', 2: 'oranj', 3: 'galben', 4: 'verde', 5: 'albastru', 6: 'ingigo', 7: 'violet'}
```

Prin elemente de tip cheie = valoare, utilizand dict():

```
>>>d2 = dict(rosu=1, oranj=2, galben=3, verde=4, albastru=5, ingigo=6, violet=7)
```

Utilizand functiile zip() si dict(), din doua liste cuprinzand cheile si valorile:

```
>>>d3 = dict(zip( [1, 2, 3, 4, 5, 6, 7], ['rosu', 'oranj', 'galben', 'verde', 'albastru',
'ingigo', 'violet'] ))
```

Utilizand tupluri cu cheile si valorile:

```
>>>d4 = dict([(2, 'oranj'), (1, 'rosu'), (3, 'galben'), (4, 'verde'), (5, 'albastru'), (6, 'ingigo'), (7, 'violet')])
```

Utilizand dict() si elemente de tip cheie: valoare:

```
>>>d5 = dict( {1: 'rosu', 2: 'oranj', 3: 'galben', 4: 'verde', 5: 'albastru', 6: 'ingigo', 7: 'violet'} )
```

Prin oricare dintre metodele prin care le-am creat putem observa ca obtinem acelasi rezultat, dictionarele sunt egale (cu exceptia d2):

```
>>>if d1 == d3 == d4 == d5:
        print ('\nAdevarat')
>>>else:
        print ('\nFals')
Adevarat
```

Putem utiliza si asa numita ***comprehensions method***:

```
>>>{x: x**3 for x in (1, 2, 3, 4)}
{1: 3, 2: 8, 3: 27, 4: 64}
```

Un dictionar fara elemente se poate initializa cu **{ }** sau cu **dictionar = dict()**.

Dictionarele nu sunt ordonabile (nu au o metoda similara sort, cum au listele). Totusi, daca dorim sa sortam cheile, valorile sau elementele unui dictionar o putem face cu ajutorul functiei **sorted**().

```
>>> sorted(d2.keys())
['albastru', 'galben', 'ingigo', 'oranj', 'rosu', 'verde', 'violet']
>>> sorted(d2.values())
[1, 2, 3, 4, 5, 6, 7]
>>> sorted(d2.items())
```

[('albastru', 5), ('galben', 3), ('ingigo', 6), ('oranj', 2), ('rosu', 1), ('verde', 4), ('violet', 7)]

Putem adauga noi elemente intr-un dictionar utilizand urmatoarea sintaxa: **dict[cheie] = valoare**.

```
>>>dict1 =  {1: 'Cal', 2: 'Caine', 3: 'Vaca', 4: 'Pisica'}

>>>dict1[5] = 'Oaie'
>>>dict1[7] = 'Iepure'
>>>print ( dict1 )
```

{1: 'Cal', 2: 'Caine', 3: 'Vaca', 4: 'Pisica', 5: 'Oaie', 7: 'Iepure'}

Unui dictionar i se poate aplica functia **len**(). Aceasta va returna numarul de elemente ale dictionarului.

Functia **type**(), va returna tipul de date al dictionarului.

Functia **del**() are doua forme. **del**(**dictionar[cheie**]) va sterge elementul cu cheia data in timp ce **del**(**dictionar**) va sterge intregul dictionar.

```
>>> print ( type( dict1 ) )
```

<class 'dict'>

```
>>> print ( len( dict1 ) )
```

6

```
>>> del( d4[ 3 ] )          # sterge elementul cu cheia 3 si valoarea 'Vaca'
>>> del( d5 )                # sterge intreg dictionarul
```

4.5.2 Metode de manipulare a dictionarelor

Metodele aplicabile dictionarelor pot fi vizualizate cu ajutorul functiei **dir**():

```
dir(dict())
'clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem', 'setdefault', 'update', 'values'
```

O sa discutam in continuare cele mai uzuale metode utilizate:

Metoda copy( ), creaza o copie a dictionarului:

```
>>>dict2 = dict1.copy ( )
>>>print ( dict2 )
{1: 'Cal', 2: 'Caine', 3: 'Vaca', 4: 'Pisica', 5: 'Oaie', 7: 'Iepure'}
```

Metoda clear( ), sterge toate elementele dictionarului dictionarului:

```
>>>dict2 = dict1.clear ( )
>>>print ( dict2 )
{ }
```

Metoda get(cheie [,valoare_default]), returneaza valoarea corespunzatoare cheii sau valoarea default, in cazul in care cheia nu exista:

```
>>> print ( dict1.get (6,  'Oaie') )      # Cheia 6 nu exista, va returna valoarea default
Oaie

>>> print ( dict1.get (1,  'Oaie') )      # Cheia 1 exista, va returna valoarea aferenta
Cal
```

Metoda update({ cheie : valoare }), adauga in dictionar elementele primite ca argument :

```
>>>dict2 = dict1.update ( {11: 'Magar', 12: 'Capra'} )
>>>print ( dict2 )
{1: 'Cal', 2: 'Caine', 3: 'Vaca', 4: 'Pisica', 5: 'Oaie', 7: 'Iepure', 11: 'Magar', 12: 'Capra'}
```

Metoda keys( ), returneaza o lista cu cheile dictionarului :

```
>>> print ( dict1.keys( ) )
dict_keys([1, 2, 3, 4, 5, 7, 11, 12])
```

Metoda values( ), returneaza o lista cu valorile din dictionar :

```
>>> print ( dict1.values( ) )
dict_values(['Cal', 'Caine', 'Vaca', 'Pisica', 'Oaie', 'Iepure', 'Magar', 'Capra'])
```

Metoda items( ), returneaza o lista cu elementele dictionarului sub forma unei liste cuprinzand tupluri (cheie, valoare):

```
>>> print ( dict1.items( ) )
dict_items ([(1, 'Cal'), (2, 'Caine'), (3, 'Vaca'), (4, 'Pisica'), (5, 'Oaie'), (7, 'Iepure'), (11, 'Magar'), (12, 'Capra')])
```

Metoda pop(cheie), sterge elementul cu cheia data si printeaza valoarea stearsa. Default sterge primul element:

```
>>>dict1.pop ( 11 )
'Magar'
```

4.5.3 Aplicatii *for* intr-un dictionar

Putem lista diferite combinatii (exemplu: valoare – cheie) sau alte expresii calculate pe baza acestora:

```
>>>dict3 = {'alfa': 10, 'beta': 20, 'gama': 30}
>>>for k in dict3:
        print ( dict3[k], k )
10 alfa
20 beta
30 gama
```

Acelasi lucru si cu:

```
>>>for k, v in dict3.items():
        print ( v, k )
10 alfa
20 beta
30 gama
```

Putem utiliza functia enumerate(), cu ajutorul careia obtinem un index numeric (incepand cu zero), corespunzator pozitiilor primite ca argument, iar valorile astfel obtinute le introducem intr-un dictionar:

```
>>>Unit_adm = dict()
>>>for k, v in enumerate(['Municipiu', 'Oras', 'Comuna', 'Sat']):
        print(k+1, v)
        Unit_adm[k + 1] = v
>>>print (Unit_adm)
1 Municipiu
2 Oras
3 Comuna
4 Sat
{1: 'Municipiu', 2: 'Oras', 3: 'Comuna', 4: 'Sat'}
```

4.6 Recapitulare

4.6.1 Recapitulare

Indexarea stringurilor si a altor tipuri de date;

Slicing in siruri, tuplu, lista, slicing cu pas;

Type;

Conversii de tipuri de date;

Tuplu – ce variabile accepta, cum se definesc, mutabilitate? Metode aplicabile: 'count', 'index';

Lista – ce variabile accepta, cum se definesc, mutabilitate? Metode aplicabile: 'append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort';

Set – ce variabile accepta, cum se definesc, mutabilitate? Metode aplicabile: 'add', 'clear', 'copy', 'difference', 'discard', 'intersection', 'pop', 'remove', 'symmetric_difference', 'union', 'update';

Dictionar – ce variabile accepta, cum se definesc, mutabilitate? Metode aplicabile: 'clear', 'copy', 'get', 'items', 'keys', 'pop', 'update','values';

Utilizarea buclelor cu noile tipuri de date.