МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ЛЬВІВСЬКА ПОЛІТЕХНІКА»

Кафедра “Захист інформації”

Програмування з використанням кортежів, словників та множин

МЕТОДИЧНІ ВКАЗІВКИ
до лабораторної роботи № 6
з курсу «Програмування скриптовими мовами»
для студентів спеціальності
«Кібербезпека»

Затверджено
на засіданні кафедри
"Захист інформації"
протокол № 01 від 29.08.2024 р.

Львів – 2024

Програмування з використанням кортежів, словників та множин: Методичні вказівки до лабораторної роботи № 6 з курсу «Програмування скриптовими мовами» для студентів спеціальності «Кібербезпека» / Укл. *Я. Р. Совин* – Львів: Національний університет "Львівська політехніка", 2024. – 25 с.

Укладач: Я. Р. Совин, канд. техн. наук, доцент

Відповідальний за випуск: В. Б. Дудикевич, д.т.н., професор

Рецензенти: А. Я. Горпенюк, канд. техн. наук, доцент
Ю. Я. Наконечний, канд. техн. наук, доцент

Мета роботи – ознайомитись з кортежами, словниками і множинами та їх можливостями у мові Python**.**

1. ТЕОРЕТИЧНІ ВІДОМОСТІ

1.1. Кортежі (tuple)

Кортежі (tuple) це послідовності, що відносяться до *незмінних* (immutable) типів даних. Іншими словами, можна отримати елемент за індексом, але змінити його не можна:

```
>>> t = (1, 2, 3) # Створюємо кортеж
>>> t[0]          # Отримуємо елемент за індексом
1
>>> t[0] = 50     # Змінити елемент за індексом не можна!
Traceback (most recent call last):
  File "<pyshell#106>", line 1, in <module>
    t[0] = 50 # Змінити елемент за індексом не можна!
TypeError: 'tuple' object does not support item assignment
```

Кортежі, як і списки, є послідовностями елементів, для зберігання гетерогенних даних. Використовуються у випадках, коли дані постійні впродовж всього часу виконання програми, до них можна віднести, наприклад, константи. Також кортежі служать для повернення декількох значень з функції.

Кортежі можна вважати списками-константами в яких заборонено модифікувати, додавати або вилучати елементи. Перевагою кортежів є вища швидкодія ніж у списків. Тому якщо не передбачається зміна елементів послідовності та її складу, то ефективніше застосовувати кортежі, а не списки.

Створити кортеж можна за допомогою таких дій:

♦ за допомогою функції *tuple(<Послідовність>])*. Функція дозволяє перетворити будь-яку послідовність в кортеж. Якщо параметр не вказано, створюється порожній кортеж:

```
>>> tuple()                     # Створюємо порожній кортеж
()
>>> tuple("String")             # Перетворимо рядок в кортеж
('S', 't', 'r', 'i', 'n', 'g')
>>> tuple([1, 2, 3, 4, 5])      # Перетворимо список в кортеж
(1, 2, 3, 4, 5)
>>> t = tuple(range(10))
>>> t
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

♦ вказавши всі елементи через кому всередині круглих дужок (або без дужок) за шаблоном:

```
   <tupleVariable> = (<element1>, <element2>, ..., <elementN>) або
      <tupleVariable> = <element1>, <element2>, ..., <elementN>
>>> t1 = ()                 # Створюємо порожній кортеж
>>> t2 = (5,)               # Створюємо кортеж з одного елемента
>>> t3 = (1, "str", (3, 4)) # Кортеж з трьох елементів
>>> t4 = 1, "str", (3, 4)   # Кортеж з трьох елементів
>>> t1, t2, t3, t4
((), (5,), (1, 'str', (3, 4)), (1, 'str', (3, 4)))
```

Зверніть особливу увагу на другий рядок прикладу. Щоб створити кортеж з одного елемента, необхідно в кінці вказати кому, - саме коми формують кортеж, а не круглі дужки. Якщо всередині круглих дужок немає ком, буде створено об'єкт іншого типу:

```
>>> t = (5); type(t)      # Отримали число, а не кортеж!
<class 'int'>
>>> t = ("str"); type(t) # Отримали рядок, а не кортеж!
<class 'str'>
```

Четвертий рядок у вихідному прикладі також доводить, що не дужки формують кортеж, а коми. Пам'ятайте, що будь-який вираз в мові Python можна помістити в круглі дужки, а щоб отримати кортеж, необхідно вказати коми.

Позиція елемента в кортежі задається індексом. Зверніть увагу на те, що нумерація елементів кортежу (як і списку) починається з 0, а не з 1. Як і всі послідовності, кортежі підтримують звернення до елементу за індексом, отримання зрізу, конкатенацію (оператор +), повторення (оператор *), перевірку на входження (оператор *in*) і невходження (оператор *not in*):

```
>>> t = (1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> t[0] # Отримуємо значення першого елемента кортежу
1
>>> t[::-1] # Змінюємо порядок слідування елементів
(9, 8, 7, 6, 5, 4, 3, 2, 1)
>>> t[2:5] # Отримуємо зріз
(3, 4, 5)
>>> 8 in t, 0 in t # Перевірка на входження
(True, False)
>>> (1, 2, 3) * 3 # Повторення
(1, 2, 3, 1, 2, 3, 1, 2, 3)
>>> (1, 2, 3) + (4, 5, 6) # Конкатенація
(1, 2, 3, 4, 5, 6)
```

З допомогою кортежів можна одночасно присвоювати значення декільком змінним з використанням душок або без:

```
>>> (x, y) = (5, 10)
>>> x
5
>>> y
10
>>> x, y = 5, 10
>>> x
5
>>> y
10
```

При позиційному присвоєнні перед однією з змінних зліва від оператора = можна вказати зірочку. У цій змінній буде зберігатися список, що складається з «зайвих» елементів. Якщо таких елементів немає, список буде порожнім:

```
>>> x = (1, 2, 3, 4)
>>> a, b, *c = x
>>> a, b, c
(1, 2, [3, 4])
>>> a, *b, c = x
>>> a, b, c
(1, [2, 3], 4)
```

```
>>> *a, b, c = x
>>> a, b, c
([1, 2], 3, 4)
```

Кортежі можуть бути вкладені:

```
>>> t = (12345, 54321, 'hello!')
>>> u = t, (1, 2, 3, 4, 5); u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> u[0]
(12345, 54321, 'hello!')
>>> u[0][1]
54321
```

Хоча кортежі незмінні вони можуть містити змінні типи даних, наприклад, списки:

```
>>> v = ([1, 2, 3], [3, 2, 1]); v
([1, 2, 3], [3, 2, 1])
```

Кортеж змінити неможна, але можна змінити, наприклад, список, що входить в кортеж (one-level-deep immutability):

```
>>> t = (1, [1, 3], "3")
>>> t[1]
[1, 3]
>>> t[1][0] = "1"
>>> t
(1, ['1', 3], '3')
```

Кортежі підтримують функції *len()*, *min()*, *max()*, методи *index()* і *count()*:

```
>>> t = (1, 2, 3) # Створюємо кортеж
>>> len(t) # Отримуємо кількість елементів
3
>>> t = (1, 2, 1, 2, 1)
>>> max(t)
2
>>> min(t)
1
>>> t.index(1), t.index(2) # Шукаємо елементи в кортежі
(0, 1)
```

Якщо потрібно відсортувати кортеж то його можна спочатку перетворити у список, виконати сортування методом *sort()* і знову перетворити у кортеж:

```
>>> t = ("cc", "aa", "dd", "bb")
>>> tmp = list(t)
>>> tmp.sort()
>>> tmp
['aa', 'bb', 'cc', 'dd']
>>> t = tuple(tmp)
>>> t
('aa', 'bb', 'cc', 'dd')
```

У таблиці 1 представлено порівняння основних операцій доступних для списків та кортежів.

Табл. 1

Порівняння властивостей списку та кортежу

Властивість	Список	Кортеж
Змінність (Mutability)	змінний (mutable)	незмінний (immutable)
Створення	*lst = [i, j]*	*tpl = (i, j)*

Доступ до елементу	*A = lst[i]*	*a = tpl[i]*
Модифікація елементу	*lst[i] = a*	Неможлива
Додавання елементів	*lst += [a]*	Неможливе
Видалення елементів	*del lst[i]*	Неможливе
Зріз (Slicing)	*lst[i:j:k]*	*tpl[i:j:k]*
Присвоєння зрізу	*lst[i:j] = []*	Неможливе
Ітерації	*for elem in lst: ...*	*for elem in tpl: ...*

1.2. Множини (set і frozenset)

Множина (set) – це невпорядкована послідовність унікальних елементів. Її основна відмінність від щойно розглянутих типів даних – зберігання лише унікальних значень. Неунікальні значення автоматично відкидаються:

```
>>> set([0, 1, 1, 2, 3, 3, 4])
{0, 1, 2, 3, 4}
```

Елементи множини не зберігаються в жодному заданому порядку та не можуть бути доступні за індексами:

```
>>> s = {1, 2, 3, 4, 5, 6}
>>> s[0]
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in <module>
    s[0]
TypeError: 'set' object does not support indexing
```

Елементи множини повинні бути незмінні та хешовані. Це означає, що цілі числа, числа з плаваючою комою, рядки та кортежі можуть бути елементами множини, а списки, словники і самі множини – ні.

Оголосити множину можна за допомогою функції *set()*:

```
>>> s = set()
>>> s
set()
```

Функція *set()* також дозволяє перетворити елементи послідовності у множину:

```
>>> set("string") # Перетворимо рядок
{'s', 'i', 'n', 'g', 't', 'r'}
>>> set([1, 2, 3, 4, 5]) # Перетворимо список
{1, 2, 3, 4, 5}
>>> set((1, 2, 3, 4, 5)) # Перетворимо кортеж
{1, 2, 3, 4, 5}
>>> set([1, 2, 3, 1, 2, 3]) # Залишаються тільки унікальні елементи
{1, 2, 3}
```

Перебрати елементи множини дозволяє цикл *for*:

```
>>> for i in set([1, 2, 3]): print(i)
1
2
3
```

Отримати кількість елементів множини дозволяє функція *len()*:

```
>>> len(set([1, 2, 3]))
3
```

Для роботи з множинами призначені наступні оператори і відповідні їм методи:

♦ | та *union()* – об'єднують дві множини:

```
>> s = set([1, 2, 3])
>>> s.union(set([4, 5, 6 ])), s | set([4, 5, 6])
({1, 2, 3, 4, 5, 6}, {1, 2, 3, 4, 5, 6})
```

Якщо елемент вже міститься у множині, то він повторно доданий не буде:

```
>>> set([1, 2, 3]) | set([1, 2, 3])
{1, 2, 3}
```

♦ *a* |= *b* та *a.update (b)* – додають елементи множини *b* в множину *a*:

```
>>> s = set([1 , 2, 3])
>>> s.update(set([4, 5, 6]))
>>> s
{1, 2, 3, 4, 5, 6}
>>> s |= set([7, 8, 9])
>>> s
{1, 2, 3, 4, 5, 6, 7, 8, 9}
```

♦ - та *difference()* – обчислюють різницю множин:

```
>>> set([1, 2, 3]) - set([1, 2, 4])
{3}
>>> s = set([1, 2, 3])
>>> s.difference(set([1, 2, 4]))
{3}
```

♦ *a* -= *b* та *a.difference_update(b)* – видаляють елементи з множини *a*, які існують і в множині *a*, і в множині *b*:

```
>>> s = set([1, 2, 3])
>>> s.difference_update (set([1, 2, 4]))
>>> s
{3}
>>> s -= set([3, 4, 5])
>>> s
set()
```

♦ & та *intersection()* – перетин множин. Дозволяє отримати елементи, які існують в обох множинах:

```
>>> set([1, 2, 3]) & set([1, 2, 4])
{1, 2}
>>> s = set([1, 2, 3])
>>> s.intersection(set([1, 2, 4]))
{1, 2}
```

♦ *a* &= *b* та *a.intersection_update(b)* – в множині *a* залишаться елементи, які існують і в множині *a*, і в множині *b*:

```
>>> s = set([1, 2, 3])
>>> s.intersection_update(set([1, 2, 4]))
>>> s
{1, 2}
>>> s &= set([1, 6, 7])
>>> s
{1}
```

♦ ^ та *symmetric_difference()* – повертають всі елементи обох множин, за вийнятком елементів, які присутні в обох цих множинах:

```
>>> s = set([1, 2, 3])
```

```
>>> s ^ set([1, 2, 4]), s.symmetric_difference(set([1, 2, 4]))
({3, 4}, {3, 4})
>>> s ^ set([1, 2, 3]), s.symmetric_difference(set([1, 2, 3]))
(set(), set())
>>> s ^ set([4, 5, 6]), s.symmetric_difference(set([4, 5, 6]))
({1, 2, 3, 4, 5, 6}, {1, 2, 3, 4, 5, 6})
```

♦ *a* ^= *b* та *a.symmetric_difference_update(b)* – в множині *a* будуть всі елементи обох множин, крім тих, що присутні в обох цих множинах:

```
>>> s = set([1, 2, 3])
>>> s.symmetric_difference_update(set([1, 2, 4]))
>>> s
{3, 4}
>>> s ^= set([3, 5, 6])
>>> s
{4, 5, 6}
```

На рис. 1 показано основні операції з множинами. Сірим кольором на діаграмах відзначено елементи, які будуть входити в множину, що є результатом застосування відповідного оператора.

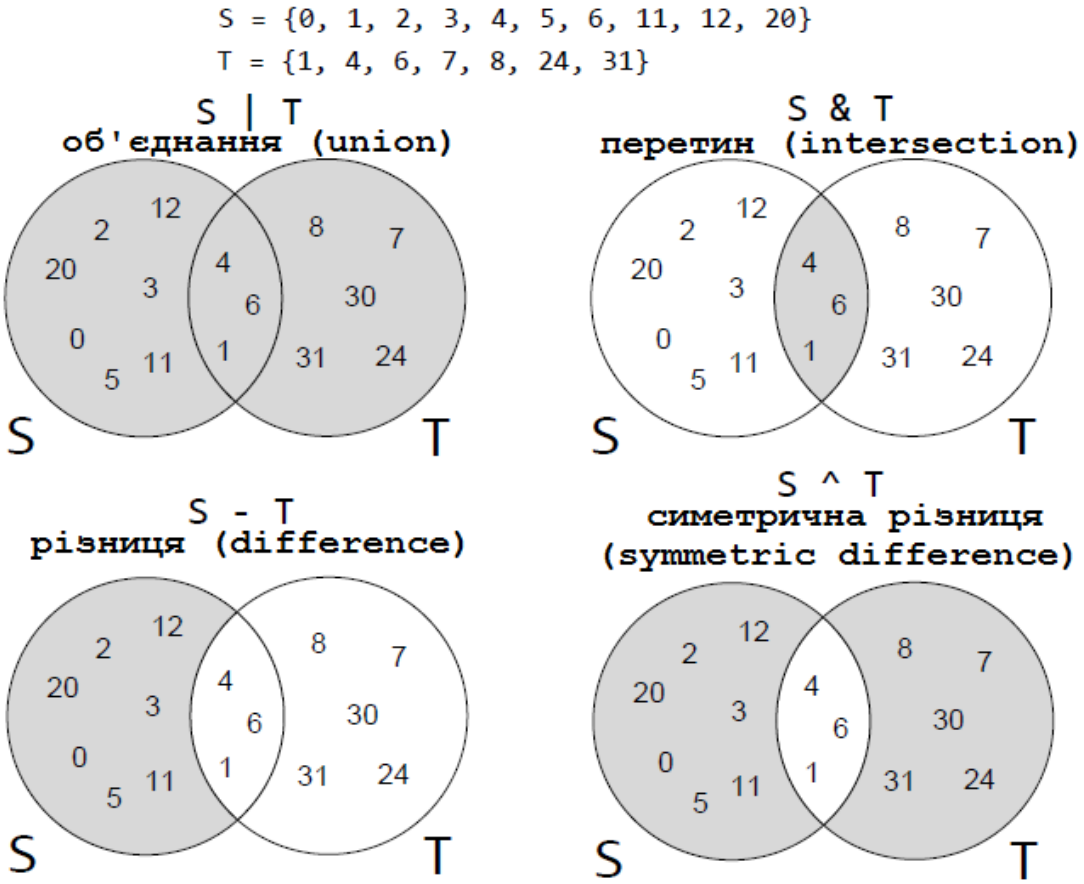

Рис. 1. Операції з множинами в Python

Оператори порівняння множин:

♦ *in* – перевірка наявності елемента в множині:

```
>>> s = set([1, 2, 3, 4, 5])
>>> 1 in s, 12 in s
(True, False)
```

♦ *not in* – перевірка відсутності елементу в множині:

```
>>> s = set([1, 2, 3, 4, 5])
>>> 1 not in s, 12 not in s
```

```
(False, True)
```

♦ == – перевірка на рівність:

```
>>> set([1, 2, 3]) == set([1, 2, 3])
True
>>> set([1, 2, 3]) == set([3, 1, 2])
True
>>> set([1, 2, 3]) == set([1, 2, 3, 4])
False
```

♦ *a* <= *b* та *a.issubset(b)* – перевіряють, чи входять всі елементи множини *a* в множину *b*:

```
>>> s <= set([1, 2]), s <= set([1, 2, 3, 4])
(False, True)
>>> s.issubset(set([1, 2])), s.issubset(set([1, 2, 3, 4]))
(False, True)
```

♦ *a* < *b* - перевіряє, чи входять всі елементи множини *a* в множину *b*, причому множина *a* не має дорівнювати множині *b*:

```
>>> s < set([1, 2, 3]), s < set([1, 2, 3, 4])
(False, True)
```

♦ *a* > = *b* та *a.issuperset(b)* - перевіряють, чи входять всі елементи множини *b* в множину *a*:

```
>>> s = set([1, 2, 3])
>>> s >= set([1, 2]), s >= set([1, 2, 3, 4])
(True, False)
>>> s.issuperset(set([1, 2])), s.issuperset(set([1, 2, 3, 4]))
(True, False)
```

♦ *a* > *b* - перевіряє, чи входять всі елементи множини *b* в множину *a*, причому множина *a* не має дорівнювати множині *b*:

```
>>> s = set([1, 2, 3])
>>> s > set([1, 2]), s >set([1, 2, 3])
(True, False)
```

♦ *a.isdisjoint(b)* - перевіряє, чи є множина *a* й *b* повністю різними, тобто не містять жодного однакового елемента:

```
>>> s = set([1, 2, 3])
>>> s.isdisjoint(set([4, 5, 6]))
True
>>> s.isdisjoint(set([1, 3, 5]))
False
```

Табл. 2

Операції над множинами в Python. *A* та *B* множини

Операція	Математична нотація	Синтаксис Python	Тип результату	Опис
Union	$A \cup B$	*A* | *B*	*set*	Елемент в *A* або *B* чи обох
Intersection	$A \cap B$	*A* & *B*	*set*	Спільні елементи для *A* і *B*
Set Difference	$A - B$	*A* - *B*	*set*	Елементи в *A*, але не в *B*
Symmetric Difference	$A \oplus B$	*A* ^ *B*	*set*	Елементи в *A* або *B*, але не в обох

Set Membership	$x \in A$	*x in A*	*bool*	*x* є елементом *A*
Set Membership	$x \notin A$	*x not in A*	*bool*	*x* не є елементом *A*
Set Equality	$A = B$	*A == B*	*bool*	*A* і *B* містять однакові елементи
Subset	$A \subseteq B$	*A <= B*	*bool*	Кожен елемент в *A* також є елементом *B*
Proper Subset	$A \subset B$	*A < B*	*bool*	*A* є підмножиною *B*, проте *B* містить щонайменше один елемент не з *A*

Для роботи з множинами призначені наступні методи:

♦ *copy()* – створює копію множини. Зверніть увагу на те, що оператор = привласнює лише посилання на той же об'єкт, а не копіює його:

```
>>> s = set([1, 2, 3])
>>> c = s; s is c # За допомогою = копію створити не можна!
True
>>> c = s.copy() # Створюємо копію об'єкта
>>> c
{1, 2, 3}
>>> s is c # Тепер це різні об'єкти
False
```

♦ *add(<Елемент>)* – додає *<Елемент>* в множину:

```
>>> s = set([1, 2, 3])
>>> s.add(4); s
{1, 2, 3, 4}
```

♦ *remove(<Елемент>)* – видаляє *<Елемент>* з множини. Якщо елемент не знайдений, то генерується виняток *KeyError*:

```
>>> s = set([1, 2, 3])
>>> s.remove(3); s # Елемент існує
{1, 2}
>>> s.remove(5) # Елемент не існує
Traceback (most recent call last):
  File "<pyshell#37>", line 1, in <module>
    s.remove(5) # Елемент не існує
KeyError: 5
```

♦ *discard(<Елемент>)* – видаляє *<Елемент>* з множини, якщо він присутній. Якщо зазначений елемент не існує, ніякого виключення не генерується:

```
>>> s = set([1, 2, 3])
>>> s.discard(3); s # Елемент існує
{1, 2}
>>> s.discard(5); s # Елемент не існує
{1, 2}
```

♦ *pop()* – видаляє довільний елемент з множини і повертає його. Якщо елементу немає, генерується виключення *KeyError*:

```
>>> s = set([1, 2])
>>> s.pop(), s
(1, {2})
>>> s.pop(), s
```

```
(2, set())
>>> s.pop() # Якщо немає елементів, то буде помилка
Traceback (most recent call last):
  File "<pyshell#45>", line 1, in <module>
    s.pop() # Якщо немає елементів, то буде помилка
KeyError: 'pop from an empty set'
```

♦ *clear()* – видаляє всі елементи з множини:

```
>>> s = set([1, 2, 3])
>>> s.clear(); s
set()
```

Крім генераторів списків і генераторів словників, мова Python 3 підтримує *генератори множин*. Їх синтаксис схожий на синтаксис генераторів списків, але вираз розташовується у фігурних дужках, а не в квадратних. Так як результатом є множина, всі повторювані елементи будуть видалені:

```
>>> {x for x in [1, 2, 1, 2, 1, 2, 3]}
{1, 2, 3}
```

Генератори множин можуть мати складну структуру. Наприклад, складатися з декількох вкладених циклів *for* і (або) містити оператор розгалуження *if* після циклу. Створимо з елементів вихідного списку множину, що містить тільки унікальні елементи з парними значеннями:

```
>>> {x for x in [1, 2, 1, 2, 1, 2, 3] if x % 2 == 0}
{2}
```

Мова Python підтримує ще один тип множин – *frozenset*. На відміну від типу *set*, множини типу *frozenset* не можна змінити. Оголосити таку множину можна за допомогою функції *frozenset()*:

```
>>> f = frozenset()
>>> f
frozenset()
>>> type(f)
<class 'frozenset'>
```

Оскільки множини не є незмінними і хешованими, вони не можуть бути елементами інших множин. Для вирішення цієї проблеми в Python існує ще один тип множини: фіксовані множини – *frozenset*, який веде себе як множина, але не може змінюватися після створення. Фіксовані множини можуть бути елементами інших множин:

```
>>> s1 = set({1, 2, 3})
>>> s2 = {s1, 4, 5, 6}
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    s2 = {s1, 4, 5, 6}
TypeError: unhashable type: 'set'
>>> s3 = frozenset(s1)
>>> s2 = {s3, 4, 5, 6}
>>> s2
{frozenset({1, 2, 3}), 4, 5, 6}
```

Функція *frozenset()* дозволяє також перетворити елементи послідовності у множину:

```
>>> frozenset("string") # Перетворимо рядок
frozenset({'r', 't', 's', 'n', 'g', 'i'})
>>> frozenset([1, 2, 3, 4, 4]) # Перетворимо список
```

```
frozenset({1, 2, 3, 4})
>>> frozenset((1, 2, 3, 4, 4)) # Перетворимо кортеж
frozenset({1, 2, 3, 4})
```

Множини *frozenset* підтримують оператори, які не змінюють саму множину, а також такі методи: *copy()*, *difference()*, *intersection()*, *issubset()*, *issuperset()*, *symmetric_difference()* та *union()*.

У табл. 3 підсумовано основні розглянуті вище операції з множинами.

Табл. 3

Найпоширеніші оператори та функції роботи з множинами

Операція	Опис	Приклад
s = set()	Створює пусту множину	*x = set()*
s = set(seq)	Створює множину з елементів послідовності *seq*	*x = set([1, 2, 3])*
s = {elem1, elem2, ..., elemN}	Створює множину з заданих елементів	*x = {1, 2, 5}*
len(s)	Повертає кількість елементів в множині *s*	*len(x)*
elem in s	Повідомляє, чи присутній елемент *elem* в множині *s*	*3 in {1, 3, 5}*
elem not in s	Повідомляє, чи не присутній елемент *elem* в множині *s*	*3 not in {1, 3, 5}*
s.add(elem)	Додає елемент *elem* в множину *s*	*x.add(7)*
s.discard(elem)	Видаляє елемент *elem* з множини *s*	*x.discard(3)*
s.remove(elem)	Видаляє елемент *elem* з множини *s*. Якщо не знайдено генерує вийняток	*x.remove(9)*
s.clear()	Видаляє всі елементи з множини *s*	*x.clear()*
s.issubset(t)	Перевіряє чи є множина *s* підмножиною *t*	*x.issubset({1,4})*
s == t	Перевіряє чи множини *s* і *t* еквівалентні	*x == {1, 7, 9}*
s != t	Перевіряє чи множини *s* і *t* не еквівалентні	*x != {1, 7, 9}*
s.union(t)	Об’єднання множин *s* і *t*	*x.union({0, 8})*
s.intersection(t)	Перетин множин *s* і *t*	*x.intersection({0, 8})*
s.difference(t)	Різниця множин *s* і *t*	*x.difference ({0, 8})*
s.symmetric_difference(t)	Симетрична різниця множин *s* і *t*	*x.symmetric_difference(y)*

1.3. Словники (dictionary)

Словники - це набори об'єктів, доступ до яких здійснюється не за індексом, а за ключем. Як ключ можна вказати незмінний об'єкт, наприклад: число, рядок або кортеж. Елементи словника можуть містити об'єкти довільного типу даних і мати необмежену ступінь вкладеності. Фактично словники це відображення між унікальними ключами (keys) та довільними значеннями (values), де кожному ключу відповідає тільки одне значення. Слід також зауважити, що словники – *невпорядкована* змінна колекція елементів, відповідно елементи в словниках розташовуються в довільному порядку. Щоб отримати елемент, необхідно вказати ключ, який використовувався при збереженні значення.

Словники відносяться до *відображень*, а не до *послідовностей*. З цієї причини функції, призначені для роботи з послідовностями, а також операції вилучення зрізу, конкатенації, повторення і т. д., до словників не застосовні. Так само як і списки, словники відносяться до змінюваних типів даних. Іншими словами, ми можемо не тільки отримати значення за ключем, а й змінити його.

1.3.1. Створення словника

Створити словник можна за допомогою таких дій:

♦ за допомогою функції *dict()*. Формати функції:

```
dict(<Ключ>=<Значення1>[...,<Ключ> = <ЗначенняN>])
dict(<Словник>)
dict(<Список кортежів з двома елементами (Ключ, Значення)>)
dict(<Список списків з двома елементами [Ключ, Значення]>)
```

Якщо параметри не зазначені, то створюється порожній словник. Приклади:

```
>>> d = dict(); d # Створюємо порожній словник
{}
>>> d = dict(a = 1, b = 2); d
{'a': 1, 'b': 2}
>>> d = {
     1: "Facebook",
     2: "Twitter",
     3: "Instagram",
     4: "LinkedIn",
     5: "Google+"
     }; d
{1: 'Facebook', 2: 'Twitter', 3: 'Instagram', 4: 'LinkedIn', 5:
'Google+'}
>>> d = dict({"a": 1, "b": 2}); d     # Словник
{'a': 1, 'b': 2}
>>> d = dict([("a", 1), ("b", 2)]); d # Список кортежів
{'a': 1, 'b': 2}
>>> d = dict([["a", 1], ["b", 2]]); d # Список списків
{'a': 1, 'b': 2}
```

Об'єднати два списки в список кортежів дозволяє функція *zip()*:

```
>>> k = ["a","b"]          # Список з ключами
>>> v = [1, 2]             # Список зі значеннями
>>> list(zip(k, v))        # Створення списку кортежів
```

```
[('a', 1), ('b', 2)]
>>> d = dict(zip(k, v)); d # Створення словника
{'a': 1, 'b': 2}
```

♦ вказавши всі елементи словника всередині фігурних дужок. Це найбільш часто використовуваний спосіб створення словника. Між ключем і значенням вказується двокрапка, а пари «Ключ/Значення» записуються через кому. Приклад:

```
>>> d = {}; d # Створення порожнього словника
{}
>>> d ={"a": 1, "b": 2}; d
{'a': 1, 'b': 2}
```

♦ заповнивши словник поелементно. У цьому випадку ключ вказується всередині квадратних дужок:

```
>>> d = {} # Створюємо порожній словник
>>> d["a"] = 1 # Додаємо елемент1 (ключ "a")
>>> d["b"] = 2 # Додаємо елемент2 (ключ "b")
>>> d
{'a': 1, 'b': 2}
```

♦ за допомогою методу *dict.fromkeys(<Послідовність>[,<Значення>])*. Метод створює новий словник, ключами якого будуть елементи послідовності, переданої першим параметром, а їх значеннями - величина, передана другим параметром. Якщо другий параметр не вказано, то значенням елементів словника буде значення *None*. Приклад:

```
>>> d = dict.fromkeys(["a", "b", "c"])
>>> d
{'a': None, 'b': None, 'c': None}
>>> d = dict.fromkeys(["a", "b", "c"], 0) # Вказано список
>>> d
{'a': 0, 'b': 0, 'c': 0}
>>> d = dict.fromkeys(("a", "b", "c"), 0) # Вказано кортеж
>>> d
{'a': 0, 'b': 0, 'c': 0}
```

При створенні словника в змінній зберігається посилання на об'єкт, а не сам об'єкт. Це обов'язково слід враховувати при груповому привласненні. Групове присвоювання можна використовувати для чисел і рядків, але для списків і словників цього робити не можна. Розглянемо приклад:

```
>>> d1 = d2 = {"a": 1, "b": 2} # Нібито створили два об'єкти
>>> d2["b"] = 10
>>> d1, d2 # Змінилося значення в двох змінних!!!
({'a': 1, 'b': 10}, {'a': 1, 'b': 10})
```

Як видно з прикладу, зміна значення в змінній *d2* призвела також до зміни значення у змінній *d1*. Тобто, обидві змінні посилаються на один і той же об'єкт, а не на два різних об'єкта. Щоб отримати два об'єкти, необхідно проводити роздільне присвоювання:

```
>>> d1, d2 = {"a": 1, "b": 2}, {"a": 1, "b": 2}
>>> d2["b"] = 10
>>> d1, d2
({'a': 1, 'b': 2}, {'a': 1, 'b': 10})
```

Створити поверхневу копію словника дозволяє функція *dict()*:

```
>>> d1 = {"a": 1, "b": 2} # Створюємо словник
>>> d2 = dict(d1) # Створюємо поверхневу копію
```

```
>>> d1 is d2 # Оператор показує, що це різні об'єкти
False
>>> d2["b"] = 10
>>> d1, d2 # Змінилося тільки значення у змінній d2
({'a': 1, 'b': 2}, {'a': 1, 'b': 10})
```

Крім того, для створення поверхневої копії можна скористатися методом *copy()*:

```
>>> d1 = {"a": 1, "b": 2} # Створюємо словник
>>> d2 = d1.copy() # Створюємо поверхневу копію
>>> d1 is d2 # Оператор показує, що це різні об'єкти
False
>>> d2["b"] = 10
>>> d1, d2 # Змінилося тільки значення у змінній d2
({'a': 1, 'b': 2}, {'a': 1, 'b': 10})
```

Щоб створити повну копію словника, слід скористатися функцією *deepcopy()* з модуля *copy*:

```
>>> d1 = {"a": 1, "b": [20, 30, 40]}
>>> d2 = dict(d1) # Створюємо поверхневу копію
>>> d2["b"][0] = "test"
>>> d1, d2 # Змінилися значення в двох змінних!!!
({'a': 1, 'b': ['test', 30, 40]}, {'a': 1, 'b': ['test', 30, 40]})
>>> import copy
>>> d3 = copy.deepcopy(d1) # Створюємо повну копію
>>> d3["b"][1] = 800
>>> d1, d3 # Змінилося значення тільки в змінній d3
({'a': 1, 'b': ['test', 30, 40]}, {'a': 1, 'b': ['test', 800, 40]})
```

В якості ключів можуть використовуватися незмінні об’єкти: числа, рядки кортежі і т. п. У таблиці 4 показано, які з вбудованих типів Python є незмінними, хешованими і придатними для використання в якості ключів.

Табл. 4

Типи Python придатні для використання як ключі словника

Тип Python	Незмінний	Хешований	Ключ словника
int	Так	Так	Так
float	Так	Так	Так
boolean	Так	Так	Так
complex	Так	Так	Так
str	Так	Так	Так
bytes	Так	Так	Так
bytearray	Ні	Ні	Ні
list	Ні	Ні	Ні
tuple	Так	Іноді	Іноді
set	Ні	Ні	Ні
frozenset	Так	Так	Так
dictionary	Ні	Ні	Ні

1.3.2. Операції над словниками

Звернення до елементів словника здійснюється за допомогою квадратних дужок, в яких вказується ключ. Як ключ можна вказати незмінний об'єкт-наприклад: число, рядок або кортеж.

Виведемо всі елементи словника:

```
>>> d = {1: "int", "a": "str", (1, 2): "tuple"}
>>> d[1], d["a"], d[(1, 2)]
('int', 'str', 'tuple')
```

Якщо елемент, відповідний вказаною ключу, відсутня в словнику, то генерується виняток *KeyError*:

```
>>> d = {"a": 1, "b": 2}
>>> d["c"] # Звернення до неіснуючого елементу
Traceback (most recent call last):
  File "<pyshell#114>", line 1, in <module>
    d["c"] # Звернення до неіснуючого елементу
KeyError: 'c'
```

Перевірити існування ключа можна за допомогою оператора *in*. Якщо ключ знайдений, то повертається значення *True*, в іншому випадку - *False*. Приклади:

```
>>> d = {"a": 1, "b": 2}
>>> "a" in d # Ключ існує
True
>>> "c" in d # Ключ не існує
False
```

Перевірити, чи відсутній який-небудь ключ в словнику, дозволить оператор *not in*. Якщо ключ відсутній, повертається *True*, інакше - *False*. Приклади:

```
>>> d = {"a": 1, "b": 2}
>>> "c" not in d # Ключ не існує
True
>>> "a" not in d # Ключ існує
False
```

Метод *get(<Ключ>[,<Значення за замовчуванням>])* дозволяє уникнути генерації винятку *KeyError* при відсутності в словнику зазначеного ключа. Якщо ключ присутній в словнику, то метод повертає значення, відповідне цьому ключу. Якщо ключ відсутній, то повертається *None* або значення, вказане у другому параметрі. Приклад:

```
>>> d = {"a": 1, "b": 2}
>>> d.get("a"), d.get("c"), d.get("c", 800)
(1, None, 800)
```

Крім того, можна скористатися методом *setdefault(<Ключ>[,<Значення за замовчуванням>])*. Якщо ключ присутній в словнику, то метод повертає значення, що відповідає цьому ключу. Якщо ключ відсутній, то в словнику створюється новий елемент зі значенням, зазначеним у другому параметрі. Якщо другий параметр не вказано, значенням нового елемента буде *None*.

Приклад:

```
>>> d = {"a": 1, "b": 2}
>>> d.setdefault("a"), d.setdefault("c"), d.setdefault("d", 0)
(1, None, 0)
>>> d
{'a': 1, 'b': 2, 'c': None, 'd': 0}
```

Так як словники відносяться до змінюваних типів даних, ми можемо змінити елемент за ключем. Якщо елемент із зазначеним ключем відсутній в словнику, то він буде доданий в словник:

```
>>> d = {"a": 1, "b": 2}
>>> d["a"] = 800      # Зміна елемента по ключу
>>> d["c"] = "string" # Буде додано новий елемент
>>> d
{'a': 800, 'b': 2, 'c': 'string'}
```

Отримати кількість ключів в словнику дозволяє функція *len()*:

```
>>> d = {"a": 1, "b": 2}
>>> len(d) # Отримуємо кількість ключів в словнику
2
```

Видалити елемент зі словника можна за допомогою оператора *del*:

```
>>> d = {"a": 1, "b": 2}
>>> del d["b"]; d # Видаляємо елемент з ключем "b"
{'a': 1}
```

1.3.3. Перебір елементів словника

Перебрати всі елементи словника можна за допомогою циклу *for*, хоча словники і не є послідовностями. Для прикладу виведемо елементи словника двома способами. Перший спосіб використовує метод *keys()*, який повертає об'єкт з ключами словника. У другому випадку ми просто вказуємо словник як параметр. На кожній ітерації циклу буде повертатися ключ, за допомогою якого всередині циклу можна отримати значення, що відповідає цьому ключу:

```
>>> d = {"x": 1, "y": 2, "z": 3}
>>> for key in d.keys(): # Використання методу keys()
        print("({0} => {1})".format(key, d[key]), end = " ")
# Виведе: (y => 2) (x => 1) (z => 3)
>>> for key in d: # Словники також підтримують ітерації
        print("({0} => {1})".format(key, d[key]), end = " ")
# Виведе: (y => 2) (x => 1) (z => 3)
```

Оскільки словники є невпорядкованими структурами, елементи словника виводяться в довільному порядку. Щоб вивести елементи з сортуванням за ключами, слід отримати список ключів, а потім скористатися методом *sort()*:

Приклад:

```
>>> d = {"x": 1, "y": 2, "z": 3}
>>> k = list(d.keys()) # Отримуємо список ключів
>>> k.sort() # Сортуємо список ключів
>>> for key in k:
     print("({0} => {1})".format(key, d[key]), end = " ")
# Виведе: (x => 1) (y => 2) (z => 3)
```

Для сортування ключів замість методу *sort()* можна скористатися функцією *sorted()*.

Приклад:

```
>>> d = {"x": 1, "y": 2, "z": 3}
>>> for key in sorted(d.keys()):
    print("({0} => {1})". format(key, d[key]), end = " ")
# Виведе: (x => 1) (y => 2) (z => 3)
```

Так як на кожній ітерації повертається ключ словника, функції *sorted()* можна відразу передати об'єкт словника, а не результат виконання методу *keys()*:

```
>>> d = {"x": 1, "y": 2, "z": 3}
>>> for key in sorted(d):
        print("({0} => {1})". format(key, d[key]), end = " ")
# Виведе: (x => 1) (y => 2) (z => 3)
```

Розглянемо ще приклад:

```
contacts = {"Fred": 7235591, "Mary": 3841212, "Bob": 3841212,
"Sarah": 2213278}
print("Fred's number is", contacts["Fred"])
```

Результат:

```
Fred's number is 7235591
```

Виведемо імена контактів:

```
print("My Contacts:")
for key in contacts:
    print(key)
```

Результат:

```
My Contacts:
Fred
Mary
Bob
Sarah
```

Виведемо імена контактів та номера телефонів:

```
print("My Contacts:")
for key in contacts:
    print("%-10s %d" % (key, contacts[key]))
```

Результат:

```
Fred       7235591
Mary       3841212
Bob        3841212
Sarah      2213278
```

Ще приклад підрахунку слів у тексті:

```
>>> sample_string = "To be or not to be"
>>> occurrences = {}
>>> for word in sample_string.split():
     occurrences[word] = occurrences.get(word, 0) + 1
>>> for word in occurrences:
     print("The word", word, "occurs", occurrences[word], \
     "times in the string")
```

Результат:

```
The word To occurs 1 times in the string
The word be occurs 2 times in the string
The word or occurs 1 times in the string
The word not occurs 1 times in the string
The word to occurs 1 times in the string
```

1.3.4. Методи для роботи зі словниками

Для роботи зі словниками призначені наступні методи:

♦ *keys()* - повертає об'єкт *dict_keys*, що містить всі ключі словника. Цей об'єкт підтримує ітерації, а також операції над множинами. Приклад:

```
>>> d1, d2 = {"a": 1, "b": 2}, {"a": 3, "c": 4, "d": 5}
```

```
>>> d1.keys(), d2.keys() # Отримуємо об'єкт dict_keys
(dict_keys(['a', 'b']), dict_keys(['a', 'c', 'd']))
>>> list(d1.keys()), list(d2.keys()) # Отримуємо список ключів
(['a', 'b'], ['a', 'c', 'd'])
>>> for k in d1.keys(): print(k, end = " ")
a b
>>> d1.keys() | d2.keys() # Об'єднання
{'c', 'd', 'b', 'a'}
>>> d1.keys() - d2.keys() # Різниця
{'b'}
>>> d2.keys() - d1.keys() # Різниця
{'c', 'd'}
>>> d1.keys() & d2.keys() # Однакові ключі
{'a'}
>>> d1.keys() ^ d2.keys() # Унікальні ключі
{'c', 'd', 'b'}
```

♦ *values()* - повертає об'єкт *dict_values*, що містить всі значення словника. Цей об'єкт підтримує ітерації. Приклади:

```
>>> d = {"a": 1, "b": 2}
>>> d.values() # Отримуємо об'єкт dict_values
dict_values([1, 2])
>>> list(d.values()) # Отримуємо список значень
[1, 2]
>>> [v for v in d.values()]
[1, 2]
```

♦ *items()* - повертає об'єкт *dict_items*, що містить всі ключі і значення в вигляді кортежів. Цей об'єкт підтримує ітерації. Приклади:

```
>>> d = {"a": 1, "b": 2}
>>> d.items() # Отримуємо об'єкт dict_items
dict_items([('a', 1), ('b', 2)])
>>> list(d.items()) # Отримуємо список кортежів
[('a', 1), ('b', 2)]
```

♦ *<Ключ> in <Словник>* - перевіряє існування зазначеного ключа в словнику. Якщо ключ знайдений, то повертається значення *True*, в іншому випадку - *False*. Приклади:

```
>>> d = {"a": 1, "b": 2}
>>> "a" in d # Ключ існує
True
>>> "c" in d # Ключ не існує
False
```

♦ *<Ключ> not in <Словник>* - перевіряє відсутність зазначеного ключа в словнику. Якщо такого ключа немає, то повертається значення *True*, в іншому випадку - *False*. приклади:

```
>>> d = {"a": 1, "b": 2}
>>> "c" not in d # Ключ не існує
True
>>> "a" not in d # Ключ існує
False
```

♦ *get(<Ключ>[,<Значення за замовчуванням>])* - якщо ключ присутній в словнику, то метод повертає значення, відповідне цьому ключу. Якщо ключ

відсутній, то повертається *None* або значення, вказане у другому параметрі. Приклад:
```
>>> d = {"a": 1, "b": 2}
>>> d.get("a"), d.get("c"), d.get("c", 800)
(1, None, 800)
```
♦ *setdefault(<Ключ>[,<Значення за замовчуванням>])* - якщо ключ присутній в словнику, то метод повертає значення, відповідне цьому ключу. Якщо ключ відсутній, то створює в словнику новий елемент зі значенням, зазначеним у другому параметрі. Якщо другий параметр не вказано, значенням нового елемента буде *None*. Приклад:
```
>>> d = {"a": 1, "b": 2}
>>> d.setdefault("a"), d.setdefault("c"), d.setdefault("d", 0)
(1, None, 0)
>>> d
{'a': 1, 'b': 2, 'c': None, 'd': 0}
```
♦ *pop(<Ключ>[,<Значення за замовчуванням>])* - видаляє елемент із зазначеним ключем і повертає його значення. Якщо ключ відсутній, то повертається значення з другого параметра. Якщо ключ відсутній, і другий параметр не вказано, генерується виняток *KeyError*. Приклади:
```
>> d = {"a": 1,"b": 2, "c": 3}
>>> d.pop("a"), d.pop("n", 0)
(1, 0)
>>> d.pop("n") # Ключ відсутній і немає другого параметра
Traceback (most recent call last):
  File "<pyshell#194>", line 1, in <module>
    d.pop("n") # Ключ відсутній і немає другого параметра
KeyError: 'n'
>>> d
{'b': 2, 'c': 3}
```
♦ *popitem()* - видаляє довільний елемент і повертає кортеж з ключа і значення. Якщо словник порожній, генерується виняток *KeyError*. Приклади:
```
>>> d = {"a": 1, "b": 2}
>>> d.popitem() # Видаляємо довільний елемент
('b', 2)
>>> d.popitem() # Видаляємо довільний елемент
('a', 1)
>>> d.popitem() # Словник порожній. Генерується виключення
Traceback (most recent call last):
  File "<pyshell#199>", line 1, in <module>
    d.popitem() # Словник порожній. Генерується виключення
KeyError: 'popitem(): dictionary is empty'
```
♦ *clear()* - видаляє всі елементи словника. Метод нічого не повертає в якості значення. Приклад:
```
>>> d = {"a": 1, "b": 2}
>>> d.clear() # Видаляємо всі елементи
>>> d # Словник тепер порожній
{}
```
♦ *update()* - додає елементи в словник. Метод змінює поточний словник і нічого не повертає. Формати методу:
```
update(<Ключ1>=<Значення1>[, ..., <КлючN>=<ЗначенняN>])
update(<Словник>)
```

```
update(<Список кортежів з двома елементами>)
update(<Список списків з двома елементами>)
```

Якщо елемент із зазначеним ключем вже присутній в словнику, то його значення буде перезаписано. Приклади:

```
>>> d = {"a": 1, "b": 2}
>>> d.update (c = 3, d = 4)
>>> d
{'a': 1, 'b': 2, 'c': 3, 'd': 4}
>>> d.update({"c": 10, "d": 20}) # Словник
>>> d # Значення елементів перезаписані
{'a': 1, 'b': 2, 'c': 10, 'd': 20}
>>> d.update([("d", 80), ("e", 6)]) # Список кортежів
>>> d
{'a': 1, 'b': 2, 'c': 10, 'd': 80, 'e': 6}
>>> d.update([["a", "str"], ["i", "t"]]) # Список списків
>>> d
{'a': 'str', 'b': 2, 'c': 10, 'd': 80, 'e': 6, 'i': 't'}
```

♦ *copy()* - створює поверхневу копію словника:

```
>>> d1 = {"a": 1, "b": [10, 20]}
>>> d2 = d1.copy() # Створюємо поверхневу копію
>>> d1 is d2 # Це різні об'єкти
False
>>> d2["a"] = 800 # Змінюємо значення
>>> d1, d2 # Змінилося значення тільки в d2
({'a': 1, 'b': [10, 20]}, {'a': 800, 'b': [10, 20]})
>>> d2["b"][0] = "new" # Змінюємо значення вкладеного списку
>>> d1, d2 # Змінилися значення і в d1, і в d2!!!
({'a': 1, 'b': ['new', 20]}, {'a': 800, 'b': ['new', 20]})
```

Щоб створити повну копію словника, слід скористатися функцією *deepcopy()* з модуля *copy()*.

1.3.5. Генератори словників

Крім генераторів списків, мова Python 3 підтримує *генератори словників*. Синтаксис генераторів словників схожий на синтаксис генераторів списків, але має дві відмінності:

♦ вираз заноситься в фігурні дужки, а не квадратні;

♦ всередині виразу перед циклом *for* вказуються два значення через двокрапку, а не одне. Значення, розташоване ліворуч від двокрапки, стає ключем, а значення, розташоване праворуч від двокрапки, - значенням елементу.

Приклад:

```
>>> keys = ["a", "b"] # Список з ключами
>>> values = [1, 2] # Список зі значеннями
>>> {k: v for (k, v) in zip(keys, values)}
{'a': 1, 'b': 2}
>>> {k: 0 for k in keys}
{'a': 0, 'b': 0}
```

Генератори словників можуть мати складну структуру. Наприклад, складатися з декількох вкладених циклів *for* і (або) містити оператор розгалуження *if* після циклу. Створимо новий словник, який містить лише елементи з парними значеннями, з вихідного словника:

```
>>> d = {"a": 1, "b": 2, "c": 3, "d": 4}
>>> {k: v for (k, v) in d.items() if v % 2 == 0}
{'b': 2, 'd': 4}
```

У табл. 5 підсумовано основні розглянуті вище операції з множинами.

Таблиця 5

Найпоширеніші оператори та функції роботи з словниками

Операція	Опис	Приклад
d = {}	Створює пустий словник	*x = {}*
d = dict()	Створює пустий словник	*x = dict()*
d = {k_1:v_1, k_2:v_2, ..., k_n:v_n}	Створює словник з заданих пар ключ (*k*) – значення (*v*)	*x = {"a": 1, "b": 2}*
d = dict(c)	Створює копію словника *c*	*x = dict(y)*
len(d)	Повертає кількість елементів у словнику *d*	*len(x)*
key in d	Повідомляє, чи присутній елемент *elem* в множині *s*	*"a" in {"a": 1, "b": 2}*
key not in d	Повідомляє, чи не присутній елемент *elem* в множині *s*	*"c" not in {"a": 1, "b": 2}*
d[key] = value	Додає нову пару *key/value* до словника d якщо ключ не існує. Якщо існує – обновляє пов'язане значення *value*	*x["c"] = 3*
value = d[key]	Повертає значення *value* асоційоване з заданим ключем *key*	*v = x["a"]*
d.update(k1:v1, k2:v2, ..., kn:vn)	Додає пари ключ (*k*) – значення (*v*) в словник	*x.update("q": 0, "s": 1)*
d.get(key, default)	Повертає значення *value* асоційоване з заданим ключем *key* або значення *default*, якщо ключ *key* не існує	*x.get("b", 0)*
d.setdefault(key, default)	Якщо ключ *key* є в словнику *d*, то повертає асоційоване значення. Якщо ключ відсутній, то створюється нова пара *key/default*	*x.setdefault("q", 0)*
d.pop(key)	Видаляє ключ *key* та асоційоване з ним значення *value* з словника *d*	*x.pop("b")*
d.keys()	Повертає послідовність, що містить всі ключі *key* в словнику *d*	*x.keys()*
d.values()	Повертає послідовність, що містить всі значення *values* в словнику *d*	*x.values()*

d.items()	Повертає всі пари ключ-значення словника *d* у вигляді кортежів	*x.items()*
del d[key]	Видаляє пару ключ-значення з словника *d* за заданим ключем *key*	*del x["b"]*

2. ЗАВДАННЯ

2.1. Домашня підготовка до роботи

1. Вивчити теоретичний матеріал.

2.2. Виконати в лабораторії

1. Написати програму яка створює і виводить кортеж, що містить послідовність цілих чисел з *n* елементів задану формулою згідно таблиці 6 Для створеного кортежу:
    a. Виведіть елементи з індексами від 3 до 5.
    b. Замініть перший елемент останнім.
    c. Об’єднайте початковий кортеж і отриманий на кроці b.
    d. Додайте до кортежу ще три елементи зі значеннями перших трьох.
    e. Виведіть максимальне і мінімальне значення в кортежі.
    f. Видаліть всі елементи менші за середньоарифметичне значення.

Табл. 6

Варіант	**Формула**	**Кількість елементів списку**	**Початкове значення n**	**Крок**
1	7n + 3	10	2	1
2	n^2	15	-2	2
3	n^3+ n	20	0	1
4	5n + 7	12	4	3
5	3n + 1	8	2	6
6	n^2 + 3n + 5	17	1	4
7	4n + 5	11	-5	1
8	n^2 + 7n	16	4	4
9	n^2+ 3n + 1	22	1	3
10	9n + 4	13	0	1
11	7n - 2	10	4	1
12	3n^2 + 6n + 1	18	7	8
13	9n + 1	11	-5	1
14	5n^2 + 6n	16	2	4
15	2n^2+ 9n - 1	22	5	1
16	2n + 11	13	4	5
17	5n + 6	10	4	3
18	4n^2 + 8n + 2	18	7	4
19	4n + 11	11	-5	1

20	3n^2 + 2n + 1	15	2	3
21	5n^2+ 7n - 2	22	1	2
22	3n + 11	14	0	2
23	6n + 7	12	4	7
24	2n^2 + 8n + 5	20	3	5
25	3n^2 + 4n + 1	14	2	4

2. Написати програму яка створює три множини з ПІБ: *set1* - містить літери вашого прізвища, *set2* - літери вашого імені, *set3* – літери по батькові (використовуйте літери українського алфавіту у нижньому регістрі). Визначте і виведіть:
    a. Всі літери, які є в set2 і set3.
    b. Літери, які є і в set1 або set2.
    c. Літери set2, яких немає в set1.
    d. Чи є set1 підмножиною set3.
    e. Літери, які є в set1 або set2, але не в обидвох.
    f. Всі літери, які є в двох множинах з трьох.
    g. Всі літери, які є лише в одній множині з трьох.
    h. Всі літери алфавіту, яких немає в жодній з множин.
    i. Всі літери алфавіту, яких немає у кожній з трьох множин.
3. Написати програму яка створює і виводить список (базу даних, БД), який складається з словників, в яких атрибути об’єкта виступають як ключі (табл. 7). Програма не повинна використовувати функції чи класи. Організуйте діалоговий режим із вводом з клавіатури, який дозволяє робити такі операції:
    a. Вивести всю БД.
    b. Додавати елементи до БД.
    c. Відсортувати БД за заданим атрибутом.
    d. Видаляти елементи за заданим індексом.
    e. Видаляти елемент за заданим значенням.
    f. Виводити всі елементи за заданим атрибутом.

Табл. 7

Варіант	**Об’єкт**	**Ключ: Значення**
1	Автомобіль	Виробник, модель, рік випуску, пробіг, макс. швидкість
2	Книга	Автор, назва, видавництво, рік видання, кількість сторінок,
3	Пасажирський літак	Виробник, модель, рік випуску, кількість пасажирів, макс. швидкість
4	Студент	Прізвище, група, рік вступу, середній бал
5	Користувач	Логін, пароль, телефон, е-мейл, кількість авторизацій
6	Фільм	Назва, режисер, рік випуску, бюджет, тривалість, розмір файлу
7	Пиво	Назва, виробник, міцність, ціна, термін зберігання в

		днях
8	Працівник	Назва фірми, посада, телефон, е-мейл, оклад
9	Пісня	Виконавець, назва, альбом, тривалість, розмір файлу
10	Смартфон	Виробник, модель, ціна, ємність батареї, обсяг пам’яті
11	Процесор	Виробник, модель, тактова частота, ціна, розсіювана потужність
12	Квартира	Власник, адреса, поверх, площа, кількість кімнат
13	Собака	Порода, кличка, вік, вага
14	Ноутбук	Виробник, процесор, ціна, діагональ, час роботи від акумулятора
15	Країна	Назва, столиця, населення, площа, ВВП
16	Нобелівський лауреат	Прізвище, рік народження, країна, рік вручення, галузь
17	Купюра	Валюта, номінал, рік випуску, курс до долара
18	Дисципліна	Назва, викладач, семестр, балів за поточний контроль, балів за екзамен
19	Річка	Назва, континент, довжина, басейн, середньорічний стік
20	Місто	Назва, країна, населення, рік заснування, площа

3. ЗМІСТ ЗВІТУ

1. Мета роботи.
2. Повний текст завдання згідно варіанту.
3. Лістинг програми.
4. Результати роботи програм (у текстовій формі та скріншот).
5. Висновок.

4. КОНТРОЛЬНІ ЗАПИТАННЯ

1. В чому відмінність кортежу від списку?
2. Які особливості та операції підтримують множини?
3. Які є методи для роботи з словниками ?

5. СПИСОК ЛІТЕРАТУРИ

1. Learn to Program with Python 3. A Step-by-Step Guide to Programming, Second Edition / Irv Kalb. – Mountain View: Apress, 2018. – 361 p.
2. The Python Workbook. A Brief Introduction with Exercises and Solutions, Second Edition / Ben Stephenson. – Cham: Springer, 2014. – 218 p.
3. Python Pocket Reference, Fifth Edition / Mark Lutz. – Sebastopol: O’Reilly Media, Inc., 2014. – 264 p.
4. Learn Python 3 the Hard Way / Zed A. Shaw. – Boston: Addison-Wesley, 2017. – 321 p.
5. A Python Book: Beginning Python, Advanced Python, and Python Exercises / Dave Kuhlman. – Boston: MIT, 2013. – 278 p.

НАВЧАЛЬНЕ ВИДАННЯ

Програмування з використанням кортежів, словників та множин

МЕТОДИЧНІ ВКАЗІВКИ
до лабораторної роботи № 6
з курсу «Програмування скриптовими мовами»
для студентів спеціальності
«Кібербезпека»

Укладач: Я. Р. Совин, канд. техн. наук, доцент