# §6 СЛОВНИКИ ТА МНОЖИНИ

## Невпорядковані колекції

Розглянути раніше колекції (список, кортеж, рядок) були впорядкованими. Всі елементи у колекції займали свою чітко визначену позицію. До елементів цих типів можна здійснювати доступ за індексами.

Крім впорядкованих типів даних у Python існують невпорядковані колекції, у яких не визначено порядок (послідовність) доступу до елементів. До таких типів відносяться словники та множини.

## Словники

**Словник** у Python – це невпорядкована колекція об'єктів довільних типів з доступом по ключу.

Ключем словника може бути об'єкт незмінюваного типу (число, рядок, кортеж), за допомогою якого можна однозначно звернутися до елемента словника. Словники називають асоціативними масивами або хеш-мапами. Словники у Python належать до **змінюваних** (**mutable**) типів. Значення словника зберігаються в невідсортованому порядку, більш того, ключі можуть зберігатися не в тому порядку, в якому вони додаються до колекції.

Для прикладу розглянемо словник студентів.

Ключ має однозначно ідентифікувати студента, тому у ролі ключа можуть бути лише унікальні для студента характеристики. Наприклад, такими характеристиками можуть бути **номер студентського квитка** або **ідентифікаційний код**. Прізвище та Ім'я не можуть бути ключами, оскільки у одній групі може існувати кілька осіб з однаковим прізвищем та ім'ям.

```
d = {"AA333333" : "Іваненко Іван",
        ... ,
     "BB123123" : "Петренко Петро",
        ... ,
     "AA999999" : "Петренко Петро" }
```

### Створення словників

1. За допомогою літерала. Пари ключ та значення елементів словника записуються через двокрапку. Різні пари розділяються комою, а вся послідовність записується у фігурних дужках:

```
empty_dict = {}          # Порожній словник

d = {'1': 1, 2 : '2'}    # Словник з 2-х пар ключ-значення
                         # ключу '1' відповідає значення 1
                         # ключу 2 відповідає значення '2'

# Словник ключами якого є кортежі (1, 1) та (2, 2),
# а значеннями відповідно два списки
d1 = {(1, 1) : [1, 2, 3], (2, 2) : [4, 5, 6]}
```

2. За допомогою перетворення у словник іншої колекції використовуючи ключове слово `dict`.

```
d = dict(collection)
```

Тут `d` – новий словник, `collection` – колекція у якій записані пари ключ-значення.

```
>>> d = dict(short='dict', long='dictionary')
>>> print(d)
{'short': 'dict', 'long': 'dictionary'}
>>> d = dict([(1, 1), [333, 334], (2, 4)])
>>> print(d)
{1: 1, 2: 4, 333: 334}
```

3. За допомогою методу `fromkeys`.

```
d = dict.fromkeys(collection, initial_value)
```

Тут `collection` – колекція ключів, `initial_value` – не обов'язковий параметр, що задає початкове значення, яке відповідає всім ключам

```
>>> d = dict.fromkeys(['a', 'b'])
>>> print(d)
{'b': None, 'a': None}
>>> d = dict.fromkeys(['a', 'b'], [10, 19])
>>> print(d)
{'b': [10, 19], 'a': [10, 19]}
```

4. За допомогою оператор створення словника (словникоутворення).

Оператор створення словника – це спосіб побудови нового словника на базі колекції, до всіх елементів якої застосовується деякий вираз:

```
D = {key(i) : expr(i) for i in collection if condition}
```

Оператор працює таким чином: змінна `i` послідовно пробігає всі елементи `collection`. Якщо для поточного значення `i` виконується умова `condition`, то в словник `D` додається пара `key(i) : expr(i)`.

```
>>> d = {i // 2: i**2 for i in range(10) if i % 2 == 0}
>>> print(d)
{4: 64, 0: 0, 2: 16, 1: 4, 3: 36}
```

## Доступ до елементів словника

Як було сказано раніше, словник це набір елементів з доступом за ключем. Власне ключ словника, це аналог індексу для списку. Для того, щоб звернутися до відповідного елемента словника, необхідно вказати ключ цього елементу у квадратних дужках:

```
>>> d = {'first' : 1, 'second' : 2}
>>> d['first']
1
>>> d['third']
Traceback (most recent call last):
  File "<input>", line 1, in <module>
KeyError: 'third'
```

Як видно з прикладу, звернення до словника за неіснуючим ключем породжує помилку.

Присвоєння по існуючому ключу перезаписує значення елемента, що відповідає цьому ключу, а присвоєння по новому ключу, розширяє словник новою парою ключ-значення

```
>>> d = {'first': 1, 'second': 2}
>>> d['first'] = 111
>>> d
{'first': 111, 'second': 2}
>>> d['third'] = 3
>>> d
{'first': 111, 'second': 2, 'third': 3}
```

Для видалення пари зі словника, використовують оператор **`del`**

```
>>> d = {'first': 1, 'second': 2}
>>> del d['first']
```

```
>>> d
{'second': 2}
```

## Операції над словниками

Нехай `d` деякий заданий словник. Розглянемо основні операції, що можна здійснювати зі словником.

**Таблиця 6.1. Перевірка приналежності ключа або значення до словника**

| Операція | Опис |
|---|---|
| `key in d` | Повертає **`True`**, якщо ключ `key` входить до `d` |
| `key not in d` | Повертає **`True`**, якщо ключ `key` не входить до `d` |
| `v in d.values()` | Повертає **`True`**, якщо значення `v` входить до `d` |
| `v not in d.values()` | Повертає **`True`**, якщо значення `v` не входить до `d` |

```
>>> d = {'first': 1, 'second': 2}
>>> 'first' in d
True
>>> 'third' in d
False
>>> 222 in d.values()
False
>>> 2 in d.values()
True
```

**Таблиця 6.2. Вибір елементів чи колекцій словника**

| Операція | Опис |
|---|---|
| `d.get(k)` | Повертає значення ключа `k` або **`None`**, якщо ключа `k` немає у словнику. |
| `d.get(k,v)` | Повертає значення ключа `k` або `v`, якщо ключа `k` немає у словнику. |
| `d.items()` | Повертає колекцію всіх пар ключ-значення в словнику `d`. |
| `d.keys()` | Повертає колекцію всіх ключів словника `d`. |
| `d.values()` | Повертає колекцію всіх значень в словнику `d`. |

```
>>> d = {'first': 1, 'second': 2}
>>> d.get('first')
1
>>> d.get("third", 3)
3
```

```
>>> d.items()
dict_items([('first', 1), ('second', 2)])
>>> d.values()
dict_values([1, 2])
```

**Таблиця 6.3. Аналіз словника**

| Операція | Опис |
|---|---|
| `len(d)` | Довжина `d` (кількість пар ключ-значення) |
| `min(d)` | Найменший ключ словника `d` (якщо ключі можна порівнювати) |
| `max(d)` | Найбільший ключ словника `d` (якщо ключі можна порівнювати) |

```
>>> d = {'first': 1, 'second': 2}
>>> len(d)
2
>>> min(d)
'first'
```

Вищенаведені операції можна проводити не лише з ключами, але і з значеннями словника. Для цього потрібно замість словника брати колекцію його значень, наприклад, інструкція

```
min(d.values())
```

знайде найменше значення, що міститься у словнику.

**Таблиця 6.4. Модифікація словника.**

| Операція | Опис |
|---|---|
| `d.clear()` | Видаляє всі елементи зі словника `d` |
| `d.copy()` | Повертає копію словника `d` |
| `d.pop(k)` | Повертає значення ключа `k` і видаляє зі словника елемент з ключем `k` (якщо ключа `k` немає у словнику, то видає помилку) |
| `d.pop(k,v)` | Повертає значення ключа `k` і видаляє зі словника елемент з ключем `k`. Якщо ключа `k` немає у словнику, то повертає значення v |
| `d.popitem()` | Повертає і видаляє довільну пару ключ-значення зі словника `d` (якщо словник `d` порожній, то генерує помилку) |
| `d.update(d1)` | Додає в словник `d` пари ключ-значення з колекції `d1`, які відсутні в словнику `d`, а для кожного ключа, який вже присутній в словнику `d`, виконує заміну відповідним значенням з `d1` (колекція `d1` може бути як словником, так і будь-якою колекцією, що містить пари ключ-значення). |

```
>>> d = {'first': 1, 'second': 2}
>>> d.pop('first')
1
>>> d
{'second': 2}
>>> d = {'first': 1, 'second': 2}
>>> d1  = (('third', 3), ('forth', 4), ('first', 111))
>>> d.update(d1)
>>> d
{'forth': 4, 'third': 3, 'first': 111, 'second': 2}
>>> d.clear()
>>> d
{}
```

### Обхід словників

Нехай `d` заданий словник. Розглянемо основні види обходу словника
1. Обхід словника за ключами

```
for key in d:
    process_iteration
```

Так, наприклад, результат виконання коду

```
>>> d = {'first': 1, 'second': 2, 'third': 3}
>>> for key in d:
        print(key)
```

стане виведення наступного тексту

```
second
third
first
```

2. Обхід словника за значеннями

```
for val in d.values():
    process_iteration
```

```
>>> d = {'first': 1, 'second': 2, 'third': 3}
>>> for val in d.values():
        print(val)

2
3
1
```

3. Обхід словника за парами ключ-значення

```
for key, val in d.items():
    process_iteration
```

```
>>> d = {'first': 1, 'second': 2, 'third': 3}
>>> for key, val in d.items():
        print(key, val)

second 2
third 3
first 1
```

**Приклад 6.1.** Слова у рядку розділяються одним або декількома пропусками. Визначити кількість входжень кожного слова до рядка.

*Розв'язок*. Для розв'язання задачі побудуємо на базі рядка словник, у якому ключами будуть унікальні слова, а значеннями кількість входжень слова до рядка.

```
S = input("введіть рядок: ")
words_list = S.split() # розбиваємо рядок на список слів
d = {}                 # порожній словник слів
for word in words_list: # пробігаємо по всіх словах
    if word not in d:   # якщо слово не входить до списку
                        # додаємо його до словника разом з
                        # його кількістю входжень
        d[word] = words_list.count(word)

print('слова та кількість входжень:', d)
```

**Приклад 6.2.** Визначити слово, яке входить до тексту найбільшу кількість разів.

*Розв'язок*. У цій задачі можемо використати результат отриманий у прикладі 6.1**.** Припустимо, що словник слів прикладу 6.1 вже побудований. Пробіжимо по словнику та знайдемо ключ, якому відповідає найбільше значення.

```
m = 0          # m - максимальна кількість входжень
max_word = '' # max_word - змінна що містить слово,
               # яке входить найбільшу кількість разів
for word, count in d.items(): # пробігаємо словник по
                              # всіх парах ключ-значення
    if m < count:
        m = count
        max_word = word

print('слово, яке входить найчастіше -', max_word)
```

## Множини

**Множина** у Python – це невпорядкована колекція унікальних об'єктів (тобто таких, що не повторюються) довільних типів.

Множини у Python належать до змінюваних (mutable) типів. Елементи зберігаються в невідсортованому порядку та не залежать від того коли вони додаються у множину.

Множини Python використовуються для розв'язання задач у яких має значення лише приналежність елемента до деякої множини або відсутність елемента у множині. Робота з множинами у Python подібна до роботи з дискретними множинами у математиці.

### Створення множин

1. За допомогою літерала. Елементи множини розділяються комою, а вся послідовність записується у фігурних дужках, аналогічно словникам.

```
empty_set = set()    # Порожня множина може створюватися
                     # лише з допомогою інструкції set

M = {1, 2, 3, 3, 3}    # Множина {1, 2, 3}
```

Зверніть увагу на створення порожньої множини. Нагадаємо, що порожні фігурні дужки використовуються для створення словників. Тому, щоб уникнути неоднозначності порожні множини створюються з використанням функції `set`.

2. За допомогою перетворення у множину іншої колекції використовуючи ключове слово `set`.

```
M = set(collection)
```

Тут `M` – нова множна, що будується на базі колекції `collection`.

```
>>> M = set('мама')
>>> print(M)
{'м', 'а'}
```

3. За допомогою оператор створення множини.

Оператор створення множини – це спосіб побудови множини на базі колекції, до всіх елементів якої застосовується деякий вираз:

```
M = {expr(i) for i in collection if condition}
```

Цей оператор працює таким чином: змінна `i` послідовно пробігає всі елементи `collection`. Якщо для поточного значення `i` виконується умова `condition`, то в множину `M` додається елемент `expr(i)`.

```
>>> M = {i for i in "Boing 777" if i.isdigit()}
>>> print(M)
{'7'}
```

## Операції над множинами

Нехай `M` деяка задана множина. Розглянемо основні операції, що можна здійснювати з множиною.

**Таблиця 6.5. Операції відношення.**

| Операція | Опис |
|---|---|
| `x in M` | Повертає **`True`**, якщо елемент `x` входить до `M` |
| `x not in M` | Повертає **`True`**, якщо елемент `x` входить до `M` |
| `M == N` | Повертає **`True`**, якщо множини `M` і `N` складаються з однакових елементів |
| `M != N` | Повертає **`True`**, якщо множини `M` і `N` не рівні |
| `M < N` | Повертає **`True`**, якщо множина `M` є строго підмножиною `N` |
| `M <= N` | Повертає **`True`**, якщо множина `M` є підмножиною `N` або збігається з `N`. |
| `M > N` | Повертає **`True`**, якщо множина `N` є строго підмножиною `M` |
| `M >= N` | Повертає **`True`**, якщо множина `N` є підмножиною `N` або |

| Операція | Опис |
|---|---|
| | збігається з `M`. |

```
>>> M = {1, 2, 3, 444}
>>> 444 in M
True
>>> 222 in M
False
>>> 222 not in M
True
>>> N = {1, 2, 3, 444}
>>> M == N
True
>>> M < N
False
>>> M <= N
True
>>> M = {1, 2}
>>> N = {2, 3, 4}
>>> M <= N
False
```

**Таблиця 6.6. Перетин, об'єднання, різниця.**

| Операція | Альтернативна операція | Опис |
|---|---|---|
| `M \| N` | `M.union(N)` | Об’єднання множин M U N |
| `M & N` | `M.intersection(N)` | Перетин множин M ∩ N |
| `M - N` | `M.difference(N)` | Різниця множин M \ N |
| `M ^ N` | `M.symmetric_difference(N)` | Симетрична різниця множин M та N |

```
>>> M = {1, 2, 3}
>>> N = {2, 3, 4, 5}
>>> M | N
{1, 2, 3, 4, 5}
>>> M & N
{2, 3}
>>> M.intersection({3, 4})
{3}
>>> N - M
{4, 5}
>>> N ^ M
{1, 4, 5}
```

**Таблиця 6.7. Додавання та видалення елементів.**

| Операція | Опис |
|---|---|
| `M.add(x)` | Додає елемент `x` до множини `M` |
| `M.discard(x)` | Видаляє елемент `x` з множини `M`, якщо цей елемент є у множині. |
| `M.remove(x)` | Видаляє елемент `x` з множини `M`. Породжує помилку, якщо елемент `x` відсутній. |
| `M.pop()` | Повертає довільний елемент що входить до множини `M` та видаляє його з множини `M`. Породжує помилку, якщо множина порожня. |
| `M.clear()` | Видаляє всі елементи з множини `M` |
| `M.update(N)` | Оновлює `M` значенням об'єднання `M` та `N` (тобто виконує інструкцію `M = M \| N`) |

```
>>> M = {1, 2, 3}
>>> M.add(444)
>>> M
{1, 2, 3, 444}
>>> M.remove(444)
>>> M
{1, 2, 3}
>>> M.discard(444)
>>> M
{1, 2, 3}
>>> M.pop()
2
>>> M
{1, 3}
```

**Приклад 6.3.** Наведемо розв'язання задачі прикладу 6.1 використовуючи операції із множинами.

*Розв'язок*. Як і в прикладі 6.1 побудуємо словник, що містить слова у якості ключів та число входжень як значення. Щоб не перевіряти, чи міститься вже слово в словнику, побудуємо множину слів, що містяться у тексті.

```
S = input("введіть рядок: ")

words = S.split()           # список всіх слів
words_set = set(words)      # створюємо множину слів
d = {}                      # порожній словник слів

for word in words_set:      # пробігаємо по всіх словах
    d[word] = words.count(word)
```

```
print('слова та кількість входжень', d)
```

## Незмінні множини

Окрім звичайних множин, у Python існують захищені від змін множини (незмінні множини) `frozenset`.

Незмінні множини `frozenset` належать до типу immutable. Фактично, різниця між множиною і незмінною множиною схожа на різницю між списками і кортежами: над незмінними множинами можна проводити всі операції, що і над звичайними множинами, що не змінюють їх. `frozenset` використовують тоді, коли множина після створення не змінюється і потрібна більша швидкодія у порівнянні з використанням звичайних множин.

Створюються незмінні множини перетворенням будь-якої колекції за допомогою інструкції `frozenset`

```
>>> M = frozenset({1, 2, 3})
>>> M
frozenset({1, 2, 3})
```

`frozenset` можуть фігурувати у виразах разом із звичайними множинами. При цьому результат виразу буде того типу, до якого належить перший операнд операції (`set` або `frozenset`).

```
>>> M = frozenset({1, 2, 3})
>>> N = {1, 3, 2}
>>> M == N
True
```

```
>>> M = frozenset({1, 2, 3})
>>> N = {3, 4}
>>> M | N
frozenset({1, 2, 3, 4})
>>> N | M
{1, 2, 3, 4}
```