# Python Core

**ЗМІСТ**

[Python Core 1](#_Toc145426441)

[Встановлення та налаштування оточення 4](#_Toc145426442)

[**Модуль 1: Типи даних у Python** 6](#_Toc145426443)

[**Вступ**​ 6](#_Toc145426444)

[**Системний підхід у програмуванні**​ 7](#_Toc145426445)

[**Python**​ 7](#_Toc145426446)

[**Змінні**​ 7](#_Toc145426447)

[**Інструкція**​ 8](#_Toc145426448)

[**Вирази Python**​ 9](#_Toc145426449)

[**Оператори та операнди**​ 9](#_Toc145426450)

[**Порядок виконання операцій у виразі**​ 10](#_Toc145426451)

[**Коментарі**​ 10](#_Toc145426452)

[**Типи даних**​ 10](#_Toc145426453)

[**Тип None**​ 11](#_Toc145426454)

[**Числа**​ 11](#_Toc145426455)

[**Рядки**​ 11](#_Toc145426456)

[**Операції над рядками**​ 12](#_Toc145426457)

[**Створення змінних типу bool**​ 12](#_Toc145426458)

[**Вирази порівняння**​ 13](#_Toc145426459)

[**Вбудовані функції**​ 13](#_Toc145426460)

[**Введення даних**​ 14](#_Toc145426461)

[**Приведення типів**​ 14](#_Toc145426462)

[**Запуск програми​** 15](#_Toc145426463)

[**Модуль 2: Керуючі конструкції. Винятки** 15](#_Toc145426464)

[**Керуючі конструкції. Винятки** 15](#_Toc145426465)

[**Умовне виконання**​ 15](#_Toc145426466)

[**Умови у Python**​ 17](#_Toc145426467)

[**Логічні вирази**​ 17](#_Toc145426468)

[**Булева алгебра**​ 18](#_Toc145426469)

[**Тернарні операції**​ 19](#_Toc145426470)

[**Блоки інструкцій**​ 19](#_Toc145426471)

[**Цикли**​ 21](#_Toc145426472)

[**Цикл for**​ 21](#_Toc145426473)

[**Цикл while**​ 22](#_Toc145426474)

[**«Нескінченні цикли» та break**​ 22](#_Toc145426475)

[**Завершення ітерації за допомогою continue**​ 23](#_Toc145426476)

[**Винятки**​ 24](#_Toc145426477)

[**Механізм обробки винятків**​ 25](#_Toc145426478)

[**Основні типи виключень у Python**​ 26](#_Toc145426479)

[**Модуль 3: Функції** 26](#_Toc145426480)

[**Функції** 27](#_Toc145426481)

[**Створення та виклик функцій**​ 27](#_Toc145426482)

[**Аргументи функції**​ 27](#_Toc145426483)

[**Повернення результату**​ 28](#_Toc145426484)

[**Локальні змінні​** 29](#_Toc145426485)

[**Global**​ 29](#_Toc145426486)

[**Ключові аргументи**​ 30](#_Toc145426487)

[**Змінна кількість параметрів**​ 31](#_Toc145426488)

[**Контейнери для зберігання аргументів функцій**​ 32](#_Toc145426489)

[**Кортежі**​ 32](#_Toc145426490)

[**Словники**​ 33](#_Toc145426491)

[**Рекурсія**​ 33](#_Toc145426492)

[**Імпорт пакетів та модулів**​ 34](#_Toc145426493)

[**Точка входу**​ 36](#_Toc145426494)

[**Модуль 4: Структури даних** 37](#_Toc145426495)

[**Колекції**​ 37](#_Toc145426496)

[**Списки**​ 37](#_Toc145426497)

[**Впорядковані контейнери**​ 38](#_Toc145426498)

[**Доступ за індексом**​ 38](#_Toc145426499)

[**Зрізи у Python (Slice)**​ 39](#_Toc145426500)

[**Використання методів об'єктів**​ 40](#_Toc145426501)

[**Методи списків**​ 40](#_Toc145426502)

[**Словники**​ 42](#_Toc145426503)

[**Методи словників**​ 42](#_Toc145426504)

[**Цикли та словники**​ 43](#_Toc145426505)

[**Множини**​ 44](#_Toc145426506)

[**Методи множин**​ 45](#_Toc145426507)

[**Математичні операції над множинами**​ 46](#_Toc145426508)

[**Використання кортежів як ключів**​ 46](#_Toc145426509)

[**Рядки**​ 47](#_Toc145426510)

[**Загальні для усіх колекцій операції**​ 48](#_Toc145426511)

[**Перевірка на входження**​ 49](#_Toc145426512)

[**Кількість елементів**​ 49](#_Toc145426513)

[**Перебір усіх елементів колекції в циклі for**​ 50](#_Toc145426514)

[**Робота з файловою системою, пакет pathlib**​ 50](#_Toc145426515)

[**Обробка аргументів командного рядка**​ 52](#_Toc145426516)

[Модуль 5: Просунута робота з рядками 53](#_Toc145426517)

[**Ще варіанти створення рядків**​ 53](#_Toc145426518)

[**Спеціальні символи**​ 54](#_Toc145426519)

[**Методи рядків**​ 55](#_Toc145426520)

[**Пошук у рядку**​ 55](#_Toc145426521)

[**Розбиття рядка на декілька підрядків**​ 56](#_Toc145426522)

[**Нові рядки на основі рядків**​ 56](#_Toc145426523)

[**Translate**​ 57](#_Toc145426524)

[**Форматування рядків (метамова форматування)**​ 57](#_Toc145426525)

[**Модифікатори**​ 58](#_Toc145426526)

[**Специфікація**​ 59](#_Toc145426527)

[**Регулярні вирази**​ 60](#_Toc145426528)

[**Функція search​** 60](#_Toc145426529)

[**Findall**​ 61](#_Toc145426530)

[**Підстановка, sub**​ 61](#_Toc145426531)

[**Складання регулярних виразів**​ 61](#_Toc145426532)

[**Модуль 6: Робота з файлами** 62](#_Toc145426533)

[**Файлові дескриптори** 63](#_Toc145426534)

[**Відкриття та закриття файлів** 63](#_Toc145426535)

[**Можливі режими відкриття файлів:** 64](#_Toc145426536)

[**Читання та запис у файл** 64](#_Toc145426537)

[**Навігація по файлу** 66](#_Toc145426538)

[**Менеджер контексту** 67](#_Toc145426539)

[**Байт-рядки, масив байтів** 68](#_Toc145426540)

[**bytes** 69](#_Toc145426541)

[**Створення байт-рядків** 69](#_Toc145426542)

[**Виведення байт-рядків** 70](#_Toc145426543)

[**Кодування рядків (ASCII, UTF-8, CP1251)** 70](#_Toc145426544)

[**Масив байтів** 71](#_Toc145426545)

[**Порівняння рядків** 72](#_Toc145426546)

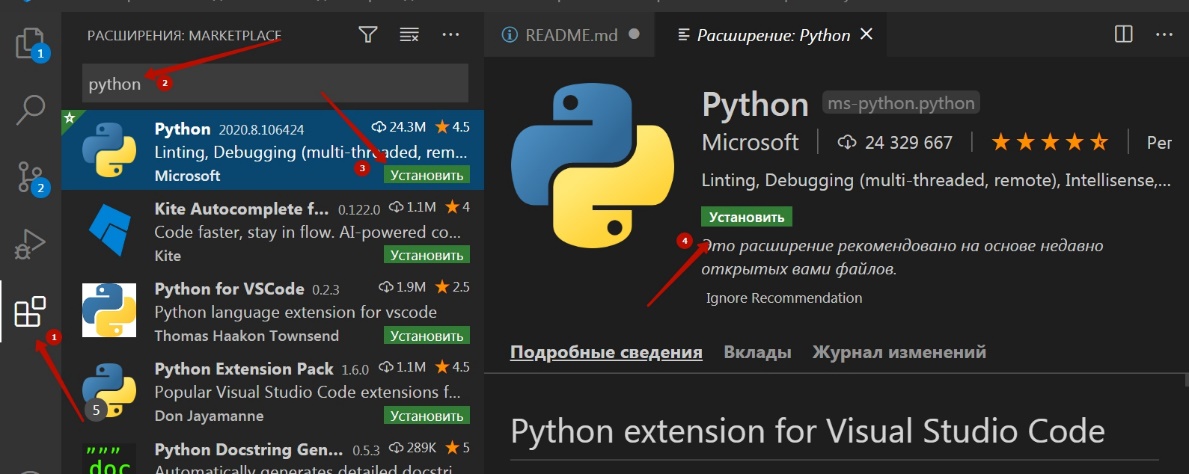
[**Робота з нетекстовими файлами у Python** 72](#_Toc145426547)

[**Робота з архівами** 73](#_Toc145426548)

## Встановлення та налаштування оточення

**Запуск програми**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/#%D0%B7%D0%B0%D0%BF%D1%83%D1%81%D0%BA-%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%B8)

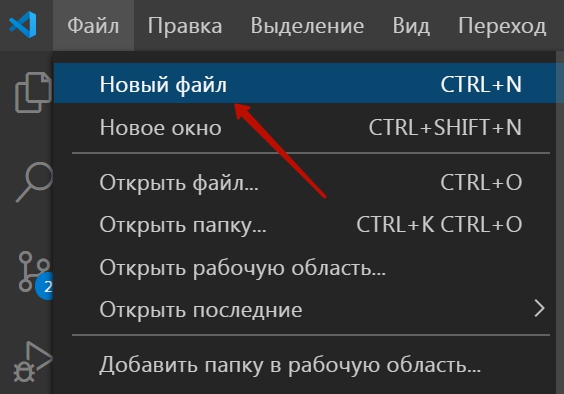
Для того щоб запустити програму мовою програмування Python на локальному комп'ютері, вам необхідно:

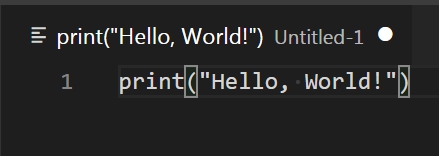
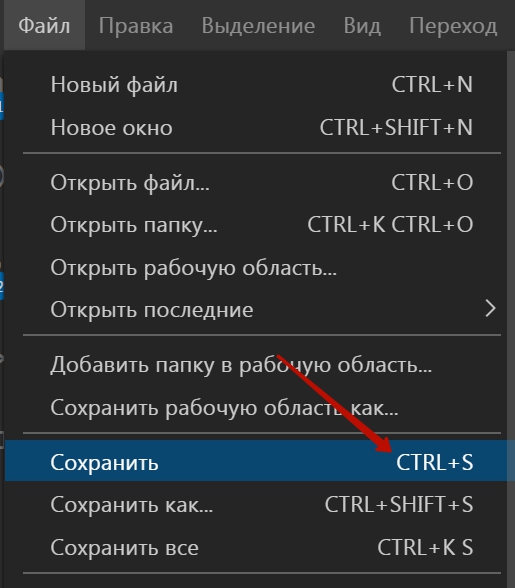
* Завантажити та встановити інтерпретатор Python <https://www.python.org/downloads/>
* Завантажити та налаштувати IDE
  + Завантажити та встановити програму [Microsoft VS Code](https://code.visualstudio.com/)
  + Встановити розширення Python у Microsoft VS Code
  + 

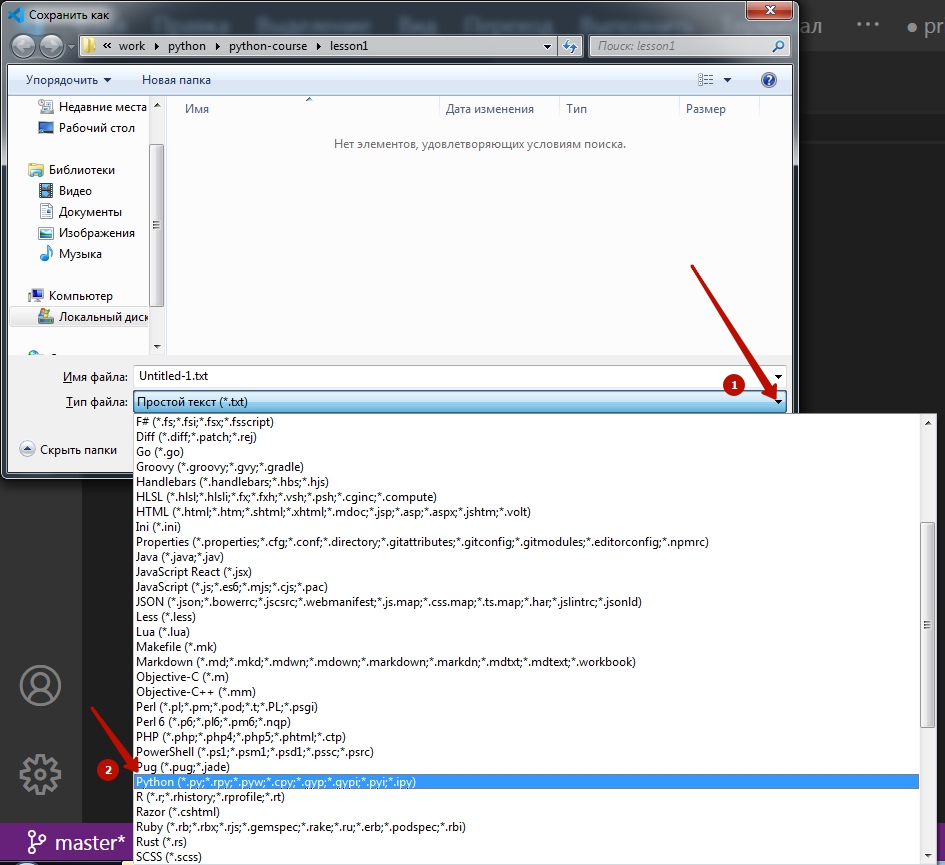
**Перша програма**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/#%D0%BF%D0%B5%D1%80%D1%88%D0%B0-%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%B0)

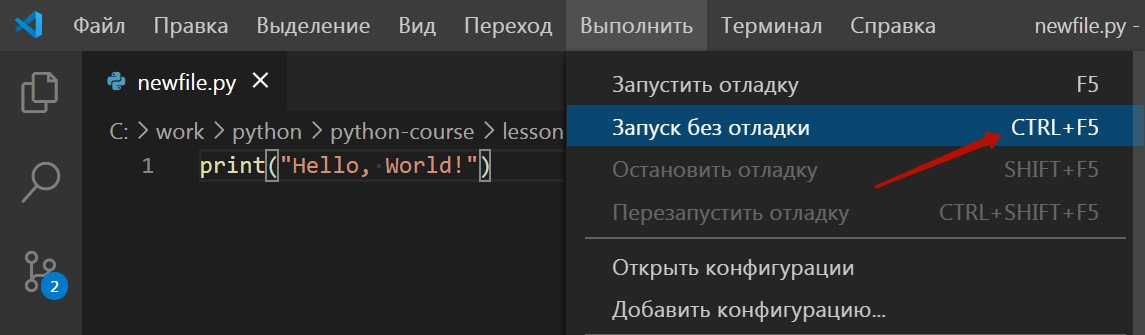
*print*("Hello, World!")

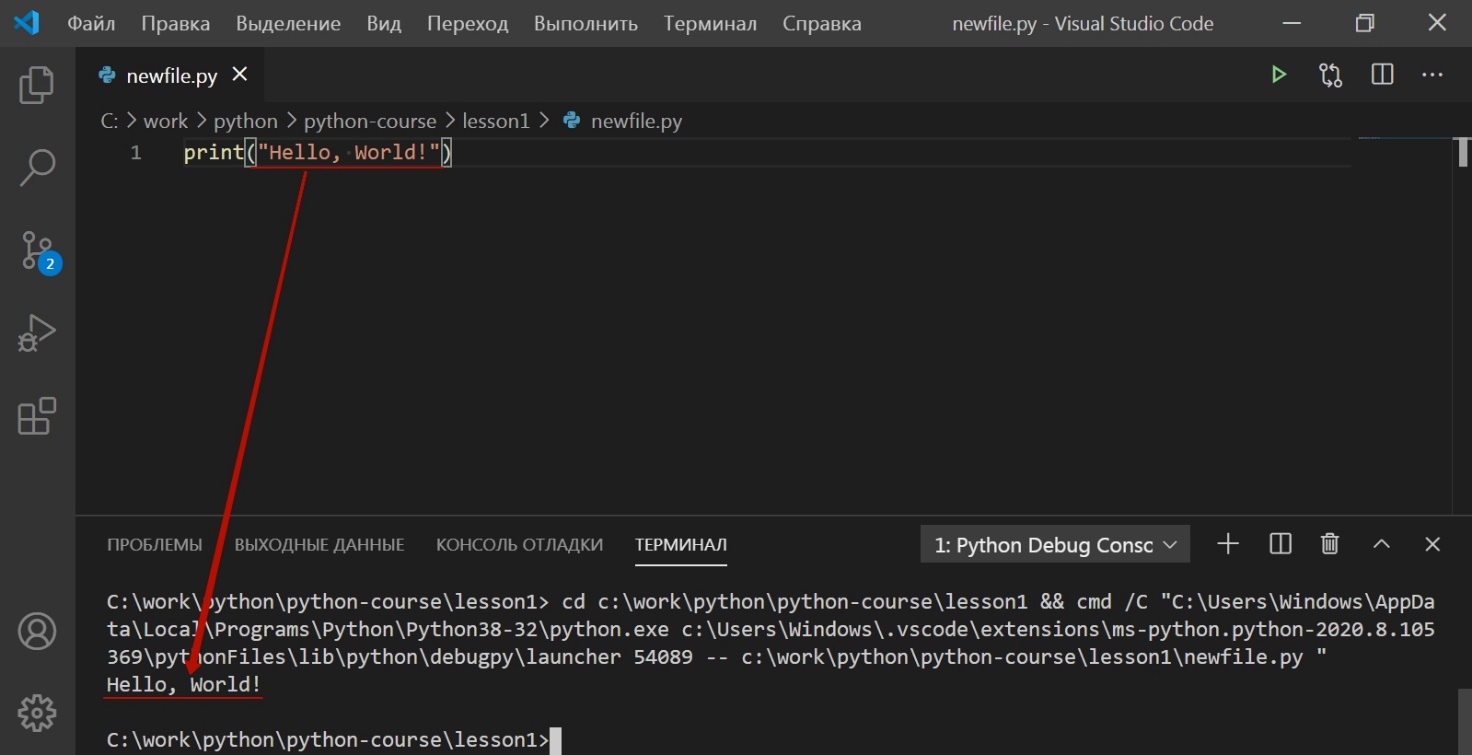
Для її запуску необхідно зробити наступні кроки:









Програма print("Hello, World!") виводить результат (текст "Hello, World!" у консоль)

Також виведення може здійснюватися у файл на комп'ютері, у вигляді відправлення запиту на сервер, та іншими способами.

З цими можливостями ми ознайомимося на наступних заняттях.

**Модуль 1: Типи даних у Python**

### **Вступ**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson01/lesson-01#%D0%B2%D1%81%D1%82%D1%83%D0%BF)

Коли ми говоримо про програмування, перше, що спадає на думку — це набір інструкцій у файлі — початковий код.

*print*("Hello World!")

Наведений вище приклад початкового коду скрипта, який під час запуску виводить у консоль текст "Hello World!".

**Початковий код (source code)** — це набір фраз, слів та спеціальних символів, специфічних для мови програмування, що описують набір інструкцій для комп'ютера.

Але символи, слова та фрази, які складають програму, насправді незрозумілі для машини. Є крок, який виконується після написання програми, який конвертує вихідний код у файлі у набір інструкцій, зрозумілих комп'ютеру. Цим займається спеціальна програма: компілятор або інтерпретатор.

### **Системний підхід у програмуванні**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson01/lesson-01#%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%BD%D0%B8%D0%B9-%D0%BF%D1%96%D0%B4%D1%85%D1%96%D0%B4-%D1%83-%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%96)

Спрощено — у будь-якої програми є:

* Введення (отримання даних)
* Перетворення (обчислення, обробка даних)
* Виведення (виведення даних)

Такі елементи є в будь-якій програмі:

* **Без виведення** програма не має сенсу, тому що ви не дізнаєтесь про те що вона зробила.
* **Немає обчислень** — це не програма, а обчислювальний пристрій (комп'ютер не потрібен).
* **А якщо немає введення** — програма не підключена до реальності. Програми отримують інформацію від користувачів, інших програм, та можуть зчитувати її з файлів на диску.

### **Python**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson01/lesson-01" \l "python" \o "Пряме посилання на Python)

**Python** — високорівнева мова програмування загального призначення, стандартна бібліотека якого містить великий набір корисних функцій.

Сьогодні **Python** використовують у таких сферах:

* Web-розробка
* Data Science
  + Data Mining (отримання даних)
  + Machine Learning (машинне навчання)
  + Deep Learning (глибинне навчання)

### **Змінні**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson01/lesson-01#%D0%B7%D0%BC%D1%96%D0%BD%D0%BD%D1%96)

Змінна — це ім'я (псевдонім) для деякої області пам'яті комп'ютера. Передбачається, що в цій пам'яті лежить корисна для розробника інформація, до якої є необхідність звертатися неодноразово. Python, як і будь-яка інша мова програмування, працює з даними в пам'яті, до яких звертається за допомогою імен (змінних).

**Змінна** — це елемент пам'яті, у якого є ім'я і в якому можуть зберігатися данні.

age = 20  
user1\_age = 30  
user2\_age = 30  
ADULT\_THR = 18  
  
\_do\_not\_use\_this = 0

Стосовно іменування змінних у Python є три суворих правила:

* ім'я змінної у Python може складатися лише з цифр, літер та знаків підкреслення \_;
* ім'я змінної не може починатися з цифри, але може зі знака нижнього підкреслення;
* використання зарезервованих слів у якості імені змінної буде призводити до помилки.

Перелік зарезервованих слів: False, None, True, and, as, assert, async, await, break, class, continue, def, del, elif, else, except, finally, for, from, global, if, import, in, is, lambda, nonlocal, not, or, pass, raise, return, try, while, with, yield

### **Інструкція**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson01/lesson-01#%D1%96%D0%BD%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%86%D1%96%D1%8F)

**Інструкція (statement)** — це пов'язаний набір слів і символів з синтаксису мови, які об'єднуються, щоб виразити одну ідею, одну інструкцію для машини.

x = 2  
y = x + 10

Це приклад інструкцій. У Python інструкції розділяються символом нового рядка (починаються з нового рядка). У прикладі вище дві інструкції: x = 2 та y = x + 10.

Зазвичай, один рядок — одна інструкція. Інструкції можна відокремлювати одну від одної символом ;, але такий синтаксис небажаний, хоча і не викликає помилки.

Python виконує інструкції одну за одною по черзі зліва направо та зверху вниз. У нашому прикладі спочатку виконається x = 2, потім y = x + 10.

### **Вирази Python**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson01/lesson-01" \l "%D0%B2%D0%B8%D1%80%D0%B0%D0%B7%D0%B8-python" \o "Пряме посилання на Вирази Python)

a = 1  
b = 2  
c = a + b + 10

Виразом називається сукупність змінних, операцій, імен функцій, дужок, який може бути обчислений відповідно до синтаксису Python.

Наприклад, a + b + 10 — це вираз, a — вираз, b — вираз.

### **Оператори та операнди**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson01/lesson-01#%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80%D0%B8-%D1%82%D0%B0-%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D0%BD%D0%B4%D0%B8)

Оператори та операнди — це частини виразу. Оператори — визначають дію, операнди — те, з чим ця дія буде вчинена. У виразі 2 + 3 — числа 3 та 2 будуть операндами, а знак + — оператором.

Для Python визначені такі арифметичні оператори:

| **Оператор** | **Значення** | **Приклади** |
| --- | --- | --- |
| + | Додавання | 3 + 3 дає результат 6 4 + 5 дає результат 9 |
| - | Віднімання | 3 - 2 дає результат 1 5 - 5 дає результат 0 |
| \* | Множення | 2 \* 2 дає результат 4 6 \* 3 дає результат 18 |
| / | Ділення | 2 / 2 дає результат 1 5 / 2 дає результат 2.5 6 / 3 дає результат 2 |
| % | Залишок від ділення | 5 % 2 дає результат 1 30 % 3 дає результат 0 |
| // | Ділення без остачі | 5 // 2 дає результат 2 26 // 8 дає результат 3 |
| \*\* | Піднесення до степеня | 2 \*\* 6 дає результат 64 5 \*\* 2 дає результат 25 |
| Операндами для арифметичних операторів є числа. |  |  |

### **Порядок виконання операцій у виразі**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson01/lesson-01#%D0%BF%D0%BE%D1%80%D1%8F%D0%B4%D0%BE%D0%BA-%D0%B2%D0%B8%D0%BA%D0%BE%D0%BD%D0%B0%D0%BD%D0%BD%D1%8F-%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D1%96%D0%B9-%D1%83-%D0%B2%D0%B8%D1%80%D0%B0%D0%B7%D1%96)

Python виконує операції у виразі у тому ж порядку, в якому виконуються математичні операції. Спочатку виконується вираз у дужках, потім піднесення до степеня, потім множення, потім додавання та віднімання.

При цьому, будь-який операнд, який є виразом Python, буде виконаний, коли виникне необхідність дізнатися його значення.

x = 8\*\*3 + 4\*(2 + 2)

В результаті виконання цієї інструкції в x буде записано значення 528. Спочатку Python вичислить 8\*\*3, потім (2 + 2), далі результат (2 + 2) помножить на 4 та додасть результат до результату 8\*\*3.

### **Коментарі**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson01/lesson-01#%D0%BA%D0%BE%D0%BC%D0%B5%D0%BD%D1%82%D0%B0%D1%80%D1%96)

Навіть ви у своєму коді через місяць мало що згадаєте. Тому, дуже важливим інструментом розробника є коментарі. Також коментарі використовуються у тих випадках, коли виконувати частину програми не потрібно, але видаляти цю частину поки що рано.

Коментарі у Python позначаються символом #, все, що після цього символу, і до початку нового рядка, інтерпретатор просто проігнорує.

# user\_age = 7  
user\_age = 18  
user\_status = "adult" # 'adult' for users older 18 and 'child' for younger

### **Типи даних**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson01/lesson-01#%D1%82%D0%B8%D0%BF%D0%B8-%D0%B4%D0%B0%D0%BD%D0%B8%D1%85)

Змінні можуть бути різного типу (зберігати інформацію у різноманітних форматах):

1. None — порожнє значення і "жодний" тип даних.
2. Числа (Numeric Type)
3. Boolean — бульовий (логічний) тип. Є підтипом цілих чисел.
4. Рядки (Text Sequence Type)

### **Тип None**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson01/lesson-01" \l "%D1%82%D0%B8%D0%BF-none" \o "Пряме посилання на Тип None)

У Python для позначення порожнього значення використовується None

a = None

None використовується тоді, коли потрібно явно повернути якесь значення або створити його (зарезервувати ім'я для чогось), але по суті та за змістом жодної корисної інформації поки що зберегти в цьому значені не можна, навіть якого роду ця інформація (рядок або число, або щось інше) поки неясно. В таких випадках застосовують None.

### **Числа**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson01/lesson-01#%D1%87%D0%B8%D1%81%D0%BB%D0%B0)

int\_number = 3  
float\_number = 3.3  
complex\_number = 3.3 + 2j

Числа у Python представлені цілими, дробовими і комплексними числами. Python — мова з динамічною типізацією, а значить, при створенні змінної не потрібно визначати її тип.

Інтерпретатор сам визначить, до якого типу вона належить, виходячи з її значення, та визначить, скільки пам'яті для неї виділити.

У прикладі вище інструкція int\_number = 3 створить змінну **int\_number** зі значенням **3** та **int\_number** буде тип int.

Інструкція float\_number = 3.3 створить змінну **float\_number** зі значенням **3.3**, **float\_number** буде мати тип float.

А інструкція complex\_number = 3.3 + 2j створить змінну **complex\_number** зі значенням **3.3 + 2j** — комплексне число.

### **Рядки**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson01/lesson-01#%D1%80%D1%8F%D0%B4%D0%BA%D0%B8)

hello\_string = "Hello"  
world\_string = 'World!'

Рядкові змінні — це впорядковані незмінні набори символів. "Впорядковані" означає, що можна звертатися до символів рядків за індексом, копіювати їх, порівнювати, шукати. "Незмінні" означає, що одного разу створивши рядок, не можна змінити його вміст, можна тільки створити новий.

Для того щоб Python зрозумів, що ви хочете створити рядкову змінну, необхідно взяти символи рядка в лапки. Для цього підійдуть як одинарні лапки ', так і подвійні ".

Наприклад, створимо змінну s, в якій зберігається рядок "Hello, World!":

s = "Hello, World!"

### **Операції над рядками**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson01/lesson-01#%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D1%96%D1%97-%D0%BD%D0%B0%D0%B4-%D1%80%D1%8F%D0%B4%D0%BA%D0%B0%D0%BC%D0%B8)

s1 = "Hello,"  
s2 = " World!"  
joined\_string = s1 + s2

Основна операція, яка реалізована для рядків, — це об'єднання рядків (конкатенація). Конкатенація рядків реалізована з використанням оператора додавання + Якщо "додати" два або більше рядків, то в результаті отримаємо об'єднаний рядок. У прикладі вище joined\_string буде дорівнювати "Hello, World!".

Для зручності виведення тексту у Python застосовують спеціальну конструкцію f-рядка.

name = "Oleg"  
hello\_string = f"Hello, {name}!"

f-рядок — це такий шаблон, який дозволяє зручним чином генерувати рядок, підставляючи результат виконання виразів в потрібне місце у шаблоні.

Синтаксично f-рядок відрізняється від звичайного тим, що на початку рядка стоїть символ f. Інтерпретатор зрозуміє, що якщо в такому рядку він зустріне символи фігурних дужок {}, то всередині них міститься вираз, який потрібно виконати і результат підставити в рядок.

У прикладі hello\_string буде дорівнювати "Hello, Oleg!".

**Boolean[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson01/lesson-01" \l "boolean" \o "Пряме посилання на Boolean)**

Boolean тип даних — підтип цілих чисел, який може приймати тільки два значення, True або False.

### **Створення змінних типу bool**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson01/lesson-01" \l "%D1%81%D1%82%D0%B2%D0%BE%D1%80%D0%B5%D0%BD%D0%BD%D1%8F-%D0%B7%D0%BC%D1%96%D0%BD%D0%BD%D0%B8%D1%85-%D1%82%D0%B8%D0%BF%D1%83-bool" \o "Пряме посилання на створення-змінних-типу-bool)

Є 2 прості способи створити змінну з типом bool:

* Присвоїти змінній значення True або False

a = True  
b = False

* Присвоїти змінній результат виконання логічного виразу, наприклад порівняння:

age = 18  
adult1 = age >= 18 # True  
  
age = 15  
adult2 = age >= 18 # False

### **Вирази порівняння**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson01/lesson-01#%D0%B2%D0%B8%D1%80%D0%B0%D0%B7%D0%B8-%D0%BF%D0%BE%D1%80%D1%96%D0%B2%D0%BD%D1%8F%D0%BD%D0%BD%D1%8F)

Для порівняння у Python є оператори < (менше), <= (менше або дорівнює), > (більше), >= (більше або дорівнює), == (дорівнює), != (не дорівнює).

a = 3, b = 5  
c = a < b # True  
d = a > b # False  
a == b # False  
a != b # True

### **Вбудовані функції**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson01/lesson-01#%D0%B2%D0%B1%D1%83%D0%B4%D0%BE%D0%B2%D0%B0%D0%BD%D1%96-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D1%96%D1%97)

Стандартна бібліотека Python реалізує ряд функцій, які одразу доступні розробнику без необхідності їх якось імпортувати. Повний перелік можна знайти на [сторінці офіційної документації](https://docs.python.org/3/library/functions.html).

У міру освоєння мови ви знайомитиметеся з цими функціями і вивчатимете їх можливості.

Інструкція виклику функції у Python (не важливо вбудованої або власної) — ім'я функції і круглі дужки, в яких перераховані через кому аргументи, якщо такі є.

*print*("Hello World!")

В цьому прикладі ми викликали функцію print, яка виводить в консоль те, що їй передадуть як аргумент. В такому випадку аргумент — рядок "Hello World!".

Функція print приймає довільну кількість аргументів і все намагається перетворити в рядок і вивести в консоль. print зручно використовувати, щоб дізнатися вміст тієї або іншої змінної у міру виконання інструкцій.

### **Введення даних**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson01/lesson-01#%D0%B2%D0%B2%D0%B5%D0%B4%D0%B5%D0%BD%D0%BD%D1%8F-%D0%B4%D0%B0%D0%BD%D0%B8%D1%85)

Для введення даних з консолі використовується функція input Значення аргументу функції (рядок) буде виводитися в консоль, а далі — буде активовано спеціальний курсор, який означає, що програма чекає введення даних.



Інтерпретатор буде чекати до тих пір, доки не зустріне символ нового рядка (Enter). Після цього весь введений з клавіатури текст буде повернений як результат роботи функції input.

a = input("Рядок запрошення: ")  
# На екрані ви побачите: Рядок запрошення:

Змінна a отримає те значення, яке ввів користувач.

### **Приведення типів**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson01/lesson-01#%D0%BF%D1%80%D0%B8%D0%B2%D0%B5%D0%B4%D0%B5%D0%BD%D0%BD%D1%8F-%D1%82%D0%B8%D0%BF%D1%96%D0%B2)

Python — це мова з динамічною суворою типізацією. Це означає, що одна й та сама змінна може змінювати свій тип у міру виконання нових інструкцій, але автоматично інтерпретатор не змінюватиме тип даних.

age = input("How old are you? ")

Функція input повертає str — рядок, та порівняти значення age з числом 18 — не можна, бо незрозуміло, яким чином повинно відбуватися таке порівняння.

Але можна перетворити тип змінної age у int за допомогою вбудованої функції int (функція називається так само, як і тип):

age = input("How old are you? ")  
age = int(age)

Для перетворення рядків в числа з дробовою частиною можна використати функцію float:

pi = float('3.14')

Також можна перетворити практично будь-який Python об'єкт в рядок функцією str:

pi\_str = str(3.14)  
age\_str = str(29)

### **Запуск програми**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson01/lesson-01#%D0%B7%D0%B0%D0%BF%D1%83%D1%81%D0%BA-%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%B8)

Для того щоб запустити програму мовою програмування Python на локальному комп'ютері, необхідно:

* Завантажити та встановити інтерпретатор [Python](https://www.python.org/downloads/" \t "_blank);
* Створити текстовий файл з кодом скрипта, наприклад:

*print*("Hello World!")  
input()

* Зберегти створений файл з ім'ям hello.py;
* Запустити скрипт hello.py за допомогою python інтерпретатора.

Скрипт hello.py виводить результат (текст "Hello, World!") у консоль.

**Модуль 2: Керуючі конструкції. Винятки**

### **Керуючі конструкції. Винятки**

name = input("What is your name? ")  
*print*(f"Hello {name}")

За замовчуванням у Python інструкції виконуються одна за одною зверху вниз. У прикладі дві інструкції — спочатку виконується name = input("What is your name? "), потім print(f"Hello {name}").

Послідовність виконання виразів в програмі називається «потік виконання» (Flow of execution).

У Python існує три способи управління потоком виконання:

* умовне виконання — виконання блоку інструкцій тільки при настанні деякої умови;
* цикли — повторення виконання блоку інструкцій, доки виконується деяка умова;
* винятки — виконання блоку інструкцій у разі помилки.

### **Умовне виконання**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson02/lesson-02#%D1%83%D0%BC%D0%BE%D0%B2%D0%BD%D0%B5-%D0%B2%D0%B8%D0%BA%D0%BE%D0%BD%D0%B0%D0%BD%D0%BD%D1%8F)

age = input("How old are you? ")  
  
*if* int(age) >= 18:  
 *print*("You are adult already.")  
*else*:  
 *print*("You are infant yet.")

У Python реалізований оператор контролю виконання (умовний оператор) if ... elif ... else.

Оператор контролю виконання дозволяє виконувати блоки інструкцій не завжди, а тільки тоді, коли буде виконана умова.

Синтаксис умовного оператора:

* починається з ключового слова if, за яким йде умова;
* після умови ставиться двокрапка і з нового рядка з відступом йде блок інструкцій, які будуть виконані, якщо умова виконується;
* після блоку if може бути нуль або більше блоків elif, інтерпретатор послідовно перевірятиме усі умови elif зверху вниз, доки не знайде той, який виконується;
* потім може бути один блок else, який виконується, якщо всі попередні умови не виконуються.

a = input('Введіть число')  
a = int(a)  
*if* a > 0:  
 *print*('Число додатне')  
*elif* a < 0:  
 *print*("Число від'ємне")  
*else*:  
 *print*('Це число - нуль')

Під час виконання умовного оператора інтерпретатор Python перевіряє умови зверху вниз, доки не знайде те, яке виконується, потім виконає вираз для цієї умови та вийде з перевірки умов.

a = input('Введіть число')  
a = int(a)  
*if* a > 0:  
 *print*('Число додатне')  
*elif* a == 1:  
 *print*('Число дорівнює 1')  
*else*:  
 *print*("a <= 0")

В такому випадку код для умови a == 1 ніколи не виконається.

### **Умови у Python**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson02/lesson-02" \l "%D1%83%D0%BC%D0%BE%D0%B2%D0%B8-%D1%83-python" \o "Пряме посилання на Умови у Python)

Умовний оператор if ... elif ... else у Python у якості умов може приймати змінні типу bool або будь-який вираз, який він виконає і результат перетворить в bool.

### **Логічні вирази**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson02/lesson-02#%D0%BB%D0%BE%D0%B3%D1%96%D1%87%D0%BD%D1%96-%D0%B2%D0%B8%D1%80%D0%B0%D0%B7%D0%B8)

Коли в якості умови в умовний оператор ми передаємо вираз, то вираз виконається, а результат його виконання буде перетворений в тип bool.

Для зручності у Python є механізм неявного приведення будь-якого типу до типу bool. Правила приведення до bool — інтуїтивні:

* число 0 приводиться до False (ціле, дробове або комплексне);

money = 0  
*if* money:  
 *print*(f"You have {money} on your bank account")  
*else*:  
 *print*("You have no money and no debts")

* None приводиться до False;

result = None  
*if* result:  
 *print*(result)  
*else*:  
 *print*("Result is None, do something")

* порожній контейнер (порожній рядок тощо) приводиться до False

user\_name = input("Enter your name: ")  
  
*if* user\_name:  
 *print*(f"Hello {user\_name}")  
*else*:  
 *print*("Hi Anonym!")

* все інше приводиться до True

Правила приведення до bool дозволяють писати умовні вирази у Python практично літературною англійською. В будь-якому разі, такий код стає дуже зрозумілим.

### **Булева алгебра**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson02/lesson-02#%D0%B1%D1%83%D0%BB%D0%B5%D0%B2%D0%B0-%D0%B0%D0%BB%D0%B3%D0%B5%D0%B1%D1%80%D0%B0)

Що робити, якщо у нас складна умова, яка поєднує в собі декілька вкладених умов? Наприклад, щоб користувач міг орендувати автомобіль, потрібно, щоб у користувача обов'язково було вказане ім'я, користувач був старший 18 і були водійські права.

name = "Taras"  
age = 22  
has\_driver\_licence = True  
  
*if* name *and* age >= 18 *and* has\_driver\_licence:  
 *print*(f"User {name} can rent a car")

Для побудови логічних умов з декількох, використовується булева алгебра.

**Булева алгебра — це розділ математичної логіки, в якому вивчаються логічні операції над висловлюваннями.**

У програмуванні застосовують бінарну логіку, можливі значення для бінарної логіки можуть бути True та False.

Булева алгебра будується на трьох основних операціях: "І", "АБО", "НЕ". Є ще допоміжні, але давайте поки що розглянемо основні.

У Python оператори булевої алгебри — це оператори not, and, or.

**and (І) вираз виконується, якщо обидві умови виконуються**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson02/lesson-02#and-%D1%96-%D0%B2%D0%B8%D1%80%D0%B0%D0%B7-%D0%B2%D0%B8%D0%BA%D0%BE%D0%BD%D1%83%D1%94%D1%82%D1%8C%D1%81%D1%8F-%D1%8F%D0%BA%D1%89%D0%BE-%D0%BE%D0%B1%D0%B8%D0%B4%D0%B2%D1%96-%D1%83%D0%BC%D0%BE%D0%B2%D0%B8-%D0%B2%D0%B8%D0%BA%D0%BE%D0%BD%D1%83%D1%8E%D1%82%D1%8C%D1%81%D1%8F)

| **A** | **B** | **A and B** |
| --- | --- | --- |
| True | True | True |
| True | False | False |
| False | True | False |
| False | False | False |

a = True *and* False # False

**or (АБО) вираз виконується, якщо хоча б одна з умов виконується**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson02/lesson-02#or-%D0%B0%D0%B1%D0%BE-%D0%B2%D0%B8%D1%80%D0%B0%D0%B7-%D0%B2%D0%B8%D0%BA%D0%BE%D0%BD%D1%83%D1%94%D1%82%D1%8C%D1%81%D1%8F-%D1%8F%D0%BA%D1%89%D0%BE-%D1%85%D0%BE%D1%87%D0%B0-%D0%B1-%D0%BE%D0%B4%D0%BD%D0%B0-%D0%B7-%D1%83%D0%BC%D0%BE%D0%B2-%D0%B2%D0%B8%D0%BA%D0%BE%D0%BD%D1%83%D1%94%D1%82%D1%8C%D1%81%D1%8F)

| **A** | **B** | **A or B** |
| --- | --- | --- |
| True | True | True |
| True | False | True |
| False | True | True |
| False | False | False |

a = True *or* False # True

**not (НІ) заперечення, вираз виконується, якщо операнд — неправда**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson02/lesson-02#not-%D0%BD%D1%96-%D0%B7%D0%B0%D0%BF%D0%B5%D1%80%D0%B5%D1%87%D0%B5%D0%BD%D0%BD%D1%8F-%D0%B2%D0%B8%D1%80%D0%B0%D0%B7-%D0%B2%D0%B8%D0%BA%D0%BE%D0%BD%D1%83%D1%94%D1%82%D1%8C%D1%81%D1%8F-%D1%8F%D0%BA%D1%89%D0%BE-%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D0%BD%D0%B4--%D0%BD%D0%B5%D0%BF%D1%80%D0%B0%D0%B2%D0%B4%D0%B0)

| **A** | **not A** |
| --- | --- |
| True | False |
| False | True |

a = *not* 2 < 0 # True

### **Тернарні операції**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson02/lesson-02#%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%80%D0%BD%D1%96-%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D1%96%D1%97)

Тернарні оператори — це ті самі умовні вирази, але в скороченій формі. Ці оператори повертають щось, залежно від того, чи є умова істиною або брехнею.

is\_nice = True  
state = "nice" *if* is\_nice *else* "not nice"

У цьому прикладі у state буде рядок 'nice'.

Такий підхід дозволяє швидко перевірити умову, а не писати декілька рядків оператора if ... else ....

У Python також існує коротший варіант тернарного оператора.

some\_data = None  
msg = some\_data *or* "Не було повернено даних"

У цьому прикладі msg містить рядок 'Не було повернено даних', це зручно, коли потрібно швидко перевірити значення та показати повідомлення, якщо значення None.

**Зверніть увагу**, що для скороченої форми використовується саме оператор or (**АБО**).

### **Блоки інструкцій**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson02/lesson-02#%D0%B1%D0%BB%D0%BE%D0%BA%D0%B8-%D1%96%D0%BD%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%86%D1%96%D0%B9)

x = int(input("X: "))  
y = int(input("Y: "))  
  
*if* x == 0:  
 *print*("X can`t be equal to zero")  
 x = int(input("X: "))  
  
result = y / x

У Python особливий синтаксис стосовно виділення блоків інструкцій. Щоб інтерпретатор сприйняв набір інструкцій як окремий блок, достатньо виділити всі інструкції цього блоку однаковою кількістю відступів зліва. У Python рекомендується для виділення одного рівня вкладеності для блоку інструкцій використовувати **4 пробіли**.

Ви можете використати символи табуляції для виділення блоку інструкцій, це не помилка, але такий спосіб не рекомендується.

Синтаксичною **помилкою** буде змішати в одному файлі виділення блоків за допомогою **табуляцій та пробілів одночасно**.

Ви також можете виділяти декілька рівнів вкладеності, додаючи ще 4 пробіли зліва для всіх інструкцій блоку:

x = int(input("X: "))  
y = int(input("Y: "))  
  
*if* x == 0:  
 *print*("X can`t be equal to zero")  
 x = int(input("X: "))  
  
 *if* x == 0:  
 *print*("X can`t be equal to zero")  
 x = int(input("X: "))  
  
 *if* x == 0:  
 *print*("X can`t be equal to zero")  
 x = int(input("X: "))  
  
result = y / x

В цьому прикладі тричі повторюється перевірка на нерівність x нулю, і на кожну перевірку блок інструкцій виділяється додатковими 4-ма пробілами.

Приклад вкладеності для визначення чвертей для координатної площини.

*if* x >= 0:  
 *if* y >= 0: # x > 0, y > 0  
 *print*("Перша чверть")  
 *else*: # x > 0, y < 0  
 *print*("Четверта чверть")  
*else*:  
 *if* y >= 0: # x < 0, y > 0  
 *print*("Друга чверть")  
 *else*: # x < 0, y < 0  
 *print*("Третя чверть")

### **Цикли**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson02/lesson-02#%D1%86%D0%B8%D0%BA%D0%BB%D0%B8)

Для того, щоб повторити якийсь блок коду кілька разів або повторювати, доки виконується деяка умова, у Python реалізовані цикли. Існує два види циклів:

* цикл for, який ще називають ітеруючим, він перебирає усі елементи деякої послідовності;
* цикл while, який виконується, доки виконується деяка умова.

**Ітерація (лат. iteratio «повторювання») — повторювання будь-якої дії.**Ітерація у програмуванні — організація обробки даних, за якої дії повторюються багаторазово, не призводячи, при цьому, до викликів самих себе.

Одна ітерація — це одне повторювання.

### **Цикл for**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson02/lesson-02" \l "%D1%86%D0%B8%D0%BA%D0%BB-for" \o "Пряме посилання на Цикл for)

У Python цикл for використовується для перебору усіх елементів контейнерів або ітерованих об'єктів, наприклад, списків. Інструкції, які знаходяться у тілі циклу, будуть виконані стільки разів, скільки елементів у списку.

При цьому на кожній ітерації спеціальна змінна набуває значення одного з елементів списку.

Роботу циклу for можна порівняти з тим, що ви по черзі візьмете кожну літеру з фрази й промовите її. Фразою буде виступати рядок 'apple', а аналогом вимовлення вголос буде виступати виведення відповідної літери в консоль.

fruit = 'apple'  
*for* char *in* fruit:  
 *print*(char)

У результаті виконання цього коду ви побачите в консолі:

a  
p  
p  
l  
e

Синтаксис циклу for:

1. цикл розпочинається з ключового слова for;
2. за яким обов'язково йде назва змінної, куди записуватиметься значення, що отримується з ітерованого об'єкту на кожній ітерації;
3. далі слідує ключове слово in;
4. за яким обов'язково йде вираз або об'єкт, по якому, власне, буде ітеруватися for;
5. далі ставиться :;
6. і з нового рядка з відступом йде набір виразів, які повторюватимуться на кожній ітерації

### **Цикл while**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson02/lesson-02" \l "%D1%86%D0%B8%D0%BA%D0%BB-while" \o "Пряме посилання на Цикл while)

Цикл while дозволяє виконати інструкції, які знаходяться у тілі циклу до тих пір, доки виконується умова, вказана в циклі. Наприклад, цикл while, який виводить числа від 1 до 5:

a = 1  
*while* a <= 5:  
 *print*(a)  
 a = a + 1

### **«Нескінченні цикли» та break**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson02/lesson-02" \l "%D0%BD%D0%B5%D1%81%D0%BA%D1%96%D0%BD%D1%87%D0%B5%D0%BD%D0%BD%D1%96-%D1%86%D0%B8%D0%BA%D0%BB%D0%B8-%D1%82%D0%B0-break" \o "Пряме посилання на нескінченні-цикли-та-break)

Бувають ситуації, коли необхідно вийти з циклу до завершення ітерації, не дочекавшись, доки станеться чергова перевірка умови. Для цього є команда break. Команда break зупиняє цикл в момент виклику і не завершує ітерацію.

a = 0  
*while* True:  
 *print*(a)  
 *if* a >= 20:  
 *break*  
 a = a + 1

В цьому прикладі умова циклу буде виконуватися завжди, адже True завжди буде True. Це приклад нескінченного циклу. Але через перевірку, що a >= 20, цей цикл завершиться, щойно в a буде значення 20 або більше.

Нескінченні цикли часто застосовуються там, де потрібно взаємодіяти з клієнтом, чекаючи введення від нього, і завершується тільки при настанні деякої умови.

Наприклад echo скрипт, який виводить в консоль те, що ви введете, доки ви не введете exit:

*while* True:  
 user\_input = input()  
 *print*(user\_input)  
 *if* user\_input == "exit":  
 *break*

### **Завершення ітерації за допомогою continue**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson02/lesson-02" \l "%D0%B7%D0%B0%D0%B2%D0%B5%D1%80%D1%88%D0%B5%D0%BD%D0%BD%D1%8F-%D1%96%D1%82%D0%B5%D1%80%D0%B0%D1%86%D1%96%D1%97-%D0%B7%D0%B0-%D0%B4%D0%BE%D0%BF%D0%BE%D0%BC%D0%BE%D0%B3%D0%BE%D1%8E-continue" \o "Пряме посилання на Завершення ітерації за допомогою continue)

Також для того, аби одразу перейти до наступної ітерації циклу без виконання виразів, що залишилися, є команда continue. Виклик цієї команди у тілі циклу призводить до того, що вирази цієї ітерації, що залишилися, не будуть виконані, а інтерпретатор одразу перейде до наступної ітерації або перевірки умови.

a = 0  
*while* a < 6:  
 a = a + 1  
 *if* *not* a % 2:  
 *continue*  
 *print*(a)

У консолі ви побачите:

1  
3  
5

Інструкція print(a) не виконувалась, коли a ділилося на 2 без залишку, оскільки ітерація завершувалася за допомогою continue.

В цьому прикладі використовувався оператор отримання залишку від ділення %, він повертає таке число p, що якщо його відняти від r, то результат буде ділитися на x націло:

(r - p) / x = a,

де а, x, r — цілі числа.

Оператори continue та break працюють тільки всередині одного циклу. В ситуації вкладених циклів немає способу вийти з усіх циклів одразу.

*while* True:  
 number = input("number = ")  
 number = int(number)  
 *while* True:  
 *print*(number)  
 number = number - 1  
 *if* number < 0:  
 *break*

В цьому прикладі користувач вводить число та отримує зворотний відлік від цього числа до 0 в консолі. При цьому, зовнішній нескінченний цикл жодним чином не перервати і break вийде тільки з внутрішнього циклу.

Також використання continue або break поза циклом призводить до синтаксичної помилки.

number = int(input("number = "))  
*if* number < 0:  
 *break*

Такий код призводить до помилки SyntaxError. Такі помилки називаються виключеннями.

### **Винятки**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson02/lesson-02#%D0%B2%D0%B8%D0%BD%D1%8F%D1%82%D0%BA%D0%B8)

Перетворити в int або float можна не будь-який рядок. Наприклад, якщо користувач введе 'a', то інтерпретатор не зможе визначити, як перетворити символ a в ціле число, і викличе виняток ValueError.

int("a")  
---------------------------------------------------------------------------  
ValueError Traceback (most recent call last)  
<ipython-input-6-d9136db7b558> *in* <module>  
----> 1 int("a")  
  
ValueError: invalid literal *for* int() *with* base 10: 'a'

Виключення у Python — це помилка на рівні інтерпретатора, викликана неможливістю виконати той або інший оператор з будь-яких причин (змінна не існує, синтаксична помилка, відсутній атрибут, операція ділення на нуль тощо).

У нашому прикладі (ввели 'а') інтерпретатор намагається перетворити рядок в тип int (ціле число), але як перетворити рядок 'a' у число не визначено і буде викликаний виняток із цього приводу.

### **Механізм обробки винятків**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson02/lesson-02#%D0%BC%D0%B5%D1%85%D0%B0%D0%BD%D1%96%D0%B7%D0%BC-%D0%BE%D0%B1%D1%80%D0%BE%D0%B1%D0%BA%D0%B8-%D0%B2%D0%B8%D0%BD%D1%8F%D1%82%D0%BA%D1%96%D0%B2)

Для обробки винятків існує оператор try ... except .... Синтаксично, цей оператор розпочинається з ключового слова try: (спробувати) та продовжується блоком коду, в якому ми чекаємо, що може статися помилка.

Далі йде блок обробки винятків except (крім), де можна вказати один або більше винятків. Якщо один із зазначених винятків станеться, то виконається наступний блок коду.

Цей блок не обов'язковий, але найчастіше потрібний. Він виконається, якщо станеться зазначений виняток (один із них, якщо їх декілька).

Якщо помилки можуть бути різні і обробляти їх потрібно теж по-різному, то можна додати декілька блоків except, у кожному вказати свою помилку і що робити, якщо вона станеться.

Потім йде необов'язковий блок, який розпочинається з ключового слова else. Цей код виконається, тільки якщо винятків не сталося.

Останнім йде необов'язковий блок коду, який розпочинається з ключового слова finally, він виконається у будь-якому разі, незалежно від того, були помилки або ні.

У нашому прикладі обробка призначеного для користувача введення виглядатиме таким чином:

val = 'a'  
*try*:  
 val = int(val)  
*except* ValueError:  
 *print*(f"val {val} is not a number")  
*else*:  
 *print*(val > 0)  
*finally*:  
 *print*("This will be printed anyway")

Винятки у Python — це дуже потужний інструмент, який часто використовується для управління потоком виконання, а не тільки для обробки помилок. У динамічних мовах ніколи не можна бути на 100% впевненим у тому, що користувач ввів значення коректного типу або, що інший застосунок не повернув None, замість int, наприклад.

Наївним розв'язання цієї проблеми буде повсюдне використання перевірок if на коректність введеного користувачем або іншим застосунком значення. Просунутішим, зручнішим і прозорішим рішенням є використання механізму обробки винятків там, де вони можуть статися через некоректні вхідні дані.

age = input("How old are you? ")  
*try*:  
 age = int(age)  
 *if* age >= 18:  
 *print*("You are adult.")  
 *else*:  
 *print*("You are infant")  
*except* ValueError:  
 *print*(f"{age} is not a number")

### **Основні типи виключень у Python**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson02/lesson-02" \l "%D0%BE%D1%81%D0%BD%D0%BE%D0%B2%D0%BD%D1%96-%D1%82%D0%B8%D0%BF%D0%B8-%D0%B2%D0%B8%D0%BA%D0%BB%D1%8E%D1%87%D0%B5%D0%BD%D1%8C-%D1%83-python" \o "Пряме посилання на Основні типи виключень у Python)

SyntaxError — синтаксична помилка.

IndentationError — помилка, яка виникає, якщо у виділенні блоків інструкцій пробілами припущена помилка.

TabError виникає, якщо в одному файлі використовувати пробіли і табуляції для виділення блоків інструкцій.

TypeError виникає, коли операція зі змінною цього типу неможлива.

2 / 'a'

ValueError виникає, коли тип операнда відповідний, але значення таке, що операцію неможливо виконати.

int("a")

ZeroDivisionError — ділення на нуль.

**Модуль 3: Функції**

### **Функції**

Функції – це фрагменти програми, які багаторазово використовуються. Вони дозволяють дати ім'я певному блоку команд для того, щоб згодом запускати цей блок за вказаним ім'ям у будь-якому місці програми і скільки завгодно разів. Це називається викликом функції.

### **Створення та виклик функцій**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson03/lesson-03#%D1%81%D1%82%D0%B2%D0%BE%D1%80%D0%B5%D0%BD%D0%BD%D1%8F-%D1%82%D0%B0-%D0%B2%D0%B8%D0%BA%D0%BB%D0%B8%D0%BA-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D1%96%D0%B9)

Функції визначаються за допомогою зарезервованого слова def. Після цього слова вказується ім'я функції, за яким йде пара дужок, у яких можна вказати імена деяких змінних, та заключна двокрапка в кінці рядка. Далі слідує блок команд, що складають функцію. На прикладі можна бачити, що насправді це дуже просто:

*def* say\_hello():  
 *print*('Привіт, Світ!') # блок, що належить функції  
 # Кінець функції say\_hello()  
  
# виклик функції  
say\_hello()  
  
# ще один виклик функції  
say\_hello()

Ми визначили функцію з ім'ям say\_hello, використовуючи описаний вище синтаксис. Ця функція не приймає параметрів, тому в дужках не оголошені якісь змінні. Параметри функції – це деякі вхідні дані, які ми можемо передати функції, щоб отримати результат, що відповідає їм.

### **Аргументи функції**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson03/lesson-03#%D0%B0%D1%80%D0%B3%D1%83%D0%BC%D0%B5%D0%BD%D1%82%D0%B8-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D1%96%D1%97)

Функції можуть приймати параметри, тобто деякі значення, що передаються функції для того, щоб вона щось зробила з ними. Ці параметри схожі на змінні, за виключенням того, що значення цих змінних вказуються при виклику функції, та під час роботи функції їм вже присвоєні їх значення. Параметри вказуються в дужках при оголошенні функції та розділяються комами. Аналогічно ми передаємо значення, коли викликаємо функцію. Зверніть увагу на термінологію: імена, вказані при оголошенні функції, називаються параметрами, тоді як значення, які ви передаєте у функцію при її виклику, – аргументами.

*def* print\_max(a, b):  
 *if* a > b:  
 *print*(a, 'максимально')  
 *elif* a == b:  
 *print*(a, 'дорівнює', b)  
 *else*:  
 *print*(b, 'максимально')  
  
print\_max(3, 4) # пряма передача значень  
  
x = 5  
y = 7  
print\_max(x, y) # передача змінних у якості аргументів

Тут ми визначили функцію з ім'ям print\_max, яка використовує два параметри з іменами a та b. Ми знаходимо найбільше число із застосуванням простого оператора if..else та виводимо це число. При першому виклику функції print\_max ми напряму передаємо числа як аргументи. У другому випадку ми викликаємо функцію зі змінними як аргументи, print\_max(x, y) призначає значення аргументу x параметру a, а значення аргументу y – параметру b. В обох випадках функція print\_max працює однаково.

### **Повернення результату**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson03/lesson-03#%D0%BF%D0%BE%D0%B2%D0%B5%D1%80%D0%BD%D0%B5%D0%BD%D0%BD%D1%8F-%D1%80%D0%B5%D0%B7%D1%83%D0%BB%D1%8C%D1%82%D0%B0%D1%82%D1%83)

У Python немає синтаксичної різниці між функціями і процедурами. По суті, функція вміє повертати деякий результат своєї роботи, а процедура нічого не повертає та результатом її роботи може бути зміна стану вже існуючих змінних. Така форма використання функцій максимально наближена до функцій, з якими ми знайомі з уроків математики.

Для повернення значення з функції необхідно вказати, що повернути після ключового слова return. Наприклад, функція, що виконує операцію додавання.

*def* plus(a, b):  
 c = a + b  
 *return* c  
  
res = plus(3, 4)  
*print*(res) # Виведе 7

Або ще коротше:

*def* plus(a, b):  
 *return* a + b  
*print*(plus(3, 4)) # Виведе 7

### **Локальні змінні**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson03/lesson-03#%D0%BB%D0%BE%D0%BA%D0%B0%D0%BB%D1%8C%D0%BD%D1%96-%D0%B7%D0%BC%D1%96%D0%BD%D0%BD%D1%96)

При оголошенні змінних всередині визначення функції вони жодним чином не пов'язані з іншими змінними з таким самим ім'ям за межами функції, тобто імена змінних є локальними у функції. Це називається областю видимості змінної. Область видимості усіх змінних обмежена блоком, в якому вони оголошені, починаючи з точки оголошення імені.

x = 50  
  
*def* func():  
 x = 2  
 *print*('Зміна локального x на', x) # Зміна локального x на 2  
  
func()  
*print*('x як і раніше', x) # x як і раніше 50

Далі ми назначаємо x значення 2 всередині функції func, але це "локальний" x, який жодним чином не впливає на "глобальний" x поза тілом функції. Тому коли ми заміняємо значення x у функції, x, оголошений в основному блоці, залишається незачепленим. Останнім викликом функції print ми виводимо значення x, вказане в основному блоці, підтверджуючи таким чином, що воно не змінилося при локальному присвоюванні значення в раніше викликаній функції.

### **Global**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson03/lesson-03" \l "global" \o "Пряме посилання на Global)

Щоб присвоїти деяке значення змінній, визначеній на вищому рівні програми (тобто не у якійсь зоні видимості, як функції), необхідно вказати Python, що її ім'я не локальне, а глобальне (global). Зробимо це за допомогою зарезервованого слова global. Без застосування зарезервованого слова global неможливо присвоїти значення змінній, визначеній за межами функції.

x = 50  
*def* func():  
 *global* x  
 *print*('x дорівнює', x) # x дорівнює 50  
 x = 2  
 *print*('Змінюємо глобальне значення x на', x) # Змінюємо глобальне значення x на 2  
func()  
*print*('Значення x складає', x) # Значення x складає 2

Зарезервоване слово global використовується для того, щоб оголосити, що x – це глобальна змінна, а значить, коли ми присвоюємо значення імені x всередині функції, ця зміна відобразиться на значенні змінної x в основному блоку програми. Використовуючи одне зарезервоване слово global, можна оголосити відразу декілька змінних: global x, y, z. Використовувати global варто з великою обережністю і не потрібно зловживати такою можливістю.

### **Ключові аргументи**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson03/lesson-03#%D0%BA%D0%BB%D1%8E%D1%87%D0%BE%D0%B2%D1%96-%D0%B0%D1%80%D0%B3%D1%83%D0%BC%D0%B5%D0%BD%D1%82%D0%B8)

Якщо є деяка функція з великою кількістю параметрів, і при її виклику вимагається вказати тільки деякі з них, значення цих параметрів можуть задаватися за їх ім'ям – це називається ключові параметри. В цьому випадку для передачі аргументів функції використовується ім'я (ключ), замість позиції (як було досі). Є дві переваги такого підходу: по-перше, використання функції стає легшим, оскільки немає потреби відстежувати порядок аргументів; по-друге, можна задавати значення тільки деяким обраним аргументам, за умови, що інші параметри мають значення аргументу за замовчуванням.

*def* func(a, b=5, c=10):  
 *print*('a дорівнює', a,', b дорівнює', b,', а c дорівнює', c)  
  
func(3, 7) # a дорівнює 3, b дорівнює 7, а c дорівнює 10  
func(25, c=24) # a дорівнює 25, b дорівнює 5, а c дорівнює 24  
func(c=50, a=100) # a дорівнює 100, b дорівнює 5, а c дорівнює 50

Функція з ім'ям func має один параметр без значення за замовчуванням, за яким слідують два параметри зі значеннями за замовчуванням. При першому виклику func(3, 7), параметр a отримує значення 3, параметр b отримує значення 7, а c отримує своє значення за замовчуванням, що дорівнює 10.

При другому виклику func(25, c=24) змінна a отримує значення 25 на підставі позиції аргументу. Після цього параметр c отримує значення 24 за ім'ям, тобто як ключовий параметр. Змінна b отримує значення за замовчуванням, що дорівнює 5.

При третьому зверненні func(c=50, a=100) ми використовуємо ключові аргументи для усіх вказаних значень. Зверніть увагу на те, що ми вказуємо значення для параметра c перед значенням для a, навіть попри те, що у визначенні функції параметр a вказаний раніше c.

*def* say(message, times=1):  
 *print*(message \* times)  
  
say('Привіт') # Привіт  
say('Світ', 5) # СвітСвітСвітСвітСвіт

Функція з ім'ям say використовується для виведення на екран рядка, вказаного число разів. Якщо ми не вказуємо значення за замовчуванням, рядок виводиться один раз. Ми досягаємо цього зазначенням значення аргументу за замовчуванням, що дорівнює 1 для параметра times. При першому виклику say ми вказуємо тільки рядок, і функція виводить його один раз. При другому виклику say ми вказуємо також і аргумент, позначаючи таким чином, що ми хочемо сказати фразу 5 разів.

Значеннями за замовчуванням можуть бути забезпечені тільки параметри, що знаходяться у кінці списку параметрів. Таким чином, у списку параметрів функції параметр зі значенням за замовчуванням не може передувати параметру без значення за замовчуванням. Це пов'язано з тим, що значення надаються параметрам відповідно до їх положення. Наприклад, def func(a, b=5) — допустимо, а def func(a=5, b) – не допустимо.

### **Змінна кількість параметрів**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson03/lesson-03#%D0%B7%D0%BC%D1%96%D0%BD%D0%BD%D0%B0-%D0%BA%D1%96%D0%BB%D1%8C%D0%BA%D1%96%D1%81%D1%82%D1%8C-%D0%BF%D0%B0%D1%80%D0%B0%D0%BC%D0%B5%D1%82%D1%80%D1%96%D0%B2)

Іноді буває необхідно визначити функцію, здатну приймати будь-яку кількість параметрів. Цього можна досягти за допомогою зірочок:

*def* total(a=5, \*numbers, \*\*phone\_book):  
 *print*('a', a)  
 # прохід по всіх елементах кортежу  
 *for* single\_item *in* numbers:  
 *print*('single\_item', single\_item)  
  
 #прохід по всіх елементах словника  
 *for* first\_part, second\_part *in* phone\_book.items():  
 *print*(first\_part,second\_part)  
  
*print*(total(10, 1, 2, 3, Jack=1123, John=2231, Inge=1560))

В результаті в консолі ми побачимо:

a 10  
single\_item 1  
single\_item 2  
single\_item 3  
Jack 1123  
John 2231  
Inge 1560  
None

Коли ми оголошуємо параметр із зірочкою (наприклад, \*numbers), всі позиційні аргументи, починаючи з цієї позиції до кінця, будуть зібрані в кортеж під ім'ям numbers. Аналогічно, коли ми оголошуємо параметри із двома зірочками (\*\*phone\_book), всі ключові аргументи, починаючи з цієї позиції до кінця, будуть зібрані в словник під ім'ям phone\_book.

### **Контейнери для зберігання аргументів функцій**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson03/lesson-03#%D0%BA%D0%BE%D0%BD%D1%82%D0%B5%D0%B9%D0%BD%D0%B5%D1%80%D0%B8-%D0%B4%D0%BB%D1%8F-%D0%B7%D0%B1%D0%B5%D1%80%D1%96%D0%B3%D0%B0%D0%BD%D0%BD%D1%8F-%D0%B0%D1%80%D0%B3%D1%83%D0%BC%D0%B5%D0%BD%D1%82%D1%96%D0%B2-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D1%96%D0%B9)

Для зберігання "зайвих" аргументів застосовуються контейнери Python, кортежі і словники. У ці контейнери збираються усі аргументи, які не визначені вами явно, але вони були передані при виклику функції.

Звичайно, в контейнери Python збирає аргументи, тільки якщо ви вкажете зробити так за допомогою синтаксису з \* і/або \*\*.

У прикладі вище numbers — це кортеж, а phone\_book — словник.

### **Кортежі**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson03/lesson-03#%D0%BA%D0%BE%D1%80%D1%82%D0%B5%D0%B6%D1%96)

Кортежі у Python — це впорядковані незмінні множини елементів. Елементом кортежу може бути будь-який тип даних. Кортежі не можна змінювати, не можна додавати/видаляти/переставляти елементи.

Щоб створити порожній кортеж, існують два способи, хоча і не зовсім зрозуміло навіщо потрібний порожній кортеж :-)

my\_tuple = tuple()  
another\_tuple = ()

Створення ж непорожніх кортежів відбувається наступним чином:

not\_empty = (1, 2, 3)

Доступ до елементів кортежу відбувається за індексом за допомогою синтаксису квадратних дужок:

not\_empty = (2, 4, 6)  
not\_empty[0] # 2  
not\_empty[1] # 4  
not\_empty[2] # 6

Індексом слугує суворо ціле число. Індексація починається з 0. Ще з кортежами вміє працювати цикл for та може одразу перебрати усі елементи кортежу, як в прикладі з total.

### **Словники**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson03/lesson-03#%D1%81%D0%BB%D0%BE%D0%B2%D0%BD%D0%B8%D0%BA%D0%B8)

Словник — це контейнер, який зберігає пари ключ-значення. Ключем може бути будь-який незмінний тип даних Python (число, рядок, кортеж тощо). Значенням словника може бути будь-який тип даних Python, включаючи призначені для користувача типи.

Порожній словник можна створити одним з двох способів:

empty\_dict = {}  
another\_empty\_dict = dict()

Створити непорожній словник можна наступним чином:

some\_dict = {  
 "key": "value",  
 1: "one",  
}

Для створення заповненого деякими значеннями словника, достатньо перелічити пари ключ-значення через кому всередині фігурних дужок, ключ йде першим, потім двокрапка та значення.

У прикладі показано застосування в якості ключів рядка і числа.

Запис пари ключ-значення у вже існуючий словник відбувається за допомогою квадратних дужок і оператора присвоєння =:

not\_empty = {"key": "value"}  
not\_empty["new\_key"] = "new value"  
*print*(not\_empty) # {"key": "value", "new\_key": "new value"}

### **Рекурсія**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson03/lesson-03#%D1%80%D0%B5%D0%BA%D1%83%D1%80%D1%81%D1%96%D1%8F)

Рекурсивна функція — це функція, що визначається в термінах самої себе і здатна викликати саму себе. Це означає, що функція викликатиме себе і повторюватиме свою поведінку до тих пір, доки не буде виконана деяка умова для повернення результату.

Найчастіший приклад використання рекурсивних функцій — це обчислення факторіалу. Спершу нагадаємо визначення факторіалу з математики: факторіал натурального числа n визначається як добуток усіх натуральних чисел від 1 до n включно. Наприклад: 5! = 1 · 2 · 3 · 4 · 5 = 120. Це саме визначення можна записати рекурсивно:

* 5! = 5 · 4!
* 4! = 4 · 3!
* 3! = 3 · 2!
* 2! = 2 · 1!
* 1! = 1

*def* factorial(n):  
 *if* n <= 1:  
 *return* 1  
 *else*:  
 *return* n \* factorial(n - 1)  
  
factorial(5) # 120

Ми визначили функцію factorial, яка приймає як аргумент деяке число n. Якщо n менше або дорівнює 1, то факторіал такого числа вже відомий та дорівнює 1. В іншому випадку , щоб знайти факторіал числа n, потрібно помножити n на факторіал n - 1, а факторіал n - 1 ми знаходимо, використовуючи функцію factorial. Викликати ж factorial ми будемо до тих пір, доки на черговій ітерації n не стане менше або дорівнює 1.

Рекурсивні функції зручні в ситуаціях, коли ми не знаємо заздалегідь, скільки разів потрібно буде викликати функцію, наприклад, при розборі папок на диску. Застосунок не знає заздалегідь, наскільки глибока структура папок і який у них рівень вкладеності. І щоб перебрати усі файли в усіх вкладених папках, функція повинна викликати сама себе, коли зустрічає чергову папку. Така функція, яка викликає сама себе за деяких умов, називається рекурсивною.

### **Імпорт пакетів та модулів**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson03/lesson-03#%D1%96%D0%BC%D0%BF%D0%BE%D1%80%D1%82-%D0%BF%D0%B0%D0%BA%D0%B5%D1%82%D1%96%D0%B2-%D1%82%D0%B0-%D0%BC%D0%BE%D0%B4%D1%83%D0%BB%D1%96%D0%B2)

У Python є великий набір пакетів та модулів з готовими корисними функціями та інструментами. Якщо вам потрібно зробити щось, чого немає в стандартному невеликому наборі функцій, ви можете імпортувати пакет, де є необхідна функція, щоб скористатися нею. Імпортування у Python відбувається за допомогою ключового слова import, після якого ви можете вказати один або декілька пакетів, які ви хочете імпортувати.

*import* math  
sin\_pi = math.sin(math.pi)

У цьому прикладі ми імпортували пакет математичних функцій і констант math та зберегли значення синуса π у sin\_pi

Для того щоб викликати функцію з імпортованого пакету, потрібно вказати ім'я пакету і через крапку ім'я функції або константи в цьому пакеті. Цей синтаксис дуже схожий на виклик метода, ми робили так зі словниками та списками раніше.

Є й інший спосіб: можна імпортувати з пакета тільки те, що нам необхідно за допомогою виразу from ... import ...:

*from* math *import* pi, sin  
sin\_pi = sin(pi)

Цей код робить те саме, що і код вище. Використовуйте той синтаксис, який вам здається зручнішим.

Імпортувати код можна не лише із стандартних або встановлених пакетів (модулей), але й з власноручно написаних модулів Python.

Модулем Python є будь-який текстовий файл з розширенням py, який містить код мовою Python. Це означає, що коли ви створюєте скрипт hello.py, який містить ось такий код:

*def* say\_hello(name):  
 *print*(f'Hello {name}')

ви можете імпортувати функцію say\_hello з hello.py за домопогою виразу: from hello import say\_hello у будь-якому модулі в тій самій папці, що і hello.py.

Це дуже зручно і дозволяє структурувати ваш код, розділяючи його на окремі файли (модулі).

Важливо розуміти, що під час імпорту модуля Python виконує увесь код, що міститься в модулі. Саме через це модуль з ось таким вмістом:

*def* say\_hello(name):  
 *print*(f'Hello {name}')  
  
*print*("You imported hello.py")  
say\_hello('user')

Під час імпорту (виконання виразу from hello import say\_hello) виведе у консоль:

You imported hello.py  
Hello user

Далеко не завжди така поведінка бажана.

### **Точка входу**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson03/lesson-03#%D1%82%D0%BE%D1%87%D0%BA%D0%B0-%D0%B2%D1%85%D0%BE%D0%B4%D1%83)

Що ж робити, коли ми хочемо зробити скрипт, що виконується (який можна викликати із консолі командою python [ім'я скрипта]), але зберегти можливість імпорту з цього модуля, не викликаючи його? В таких випадках нам може допомогти службова змінна Python: \_\_name\_\_. Річ у тому, що якщо скрипт викликаний безпосередньо, то він є "точкою входу" та \_\_name\_\_ == "\_\_main\_\_". Якщо ж цей модуль виконується під час імпорту, то \_\_name\_\_ == "hello" (наприклад для модуля, який називається hello.py). Таким чином ми можемо модифікувати наш модуль hello.py:

*def* say\_hello(name):  
 *print*(f'Hello {name}')  
  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 *print*("You imported hello.py")  
 say\_hello('user')

Тоді під час імпорту функції say\_hello із hello.py код у блоці if ... не буде виконаний, а якщо ж в консолі виконати python hello.py, то буде.

Для зручності прийнято весь код, який потрібно виконати, коли модуль викликається із консолі (не імпортується), поміщати у функцію main:

*def* say\_hello(name):  
 *print*(f'Hello {name}')  
  
*def* main():  
 *print*("You imported hello.py")  
 say\_hello('user')  
  
*for* arg *in* sys.argv:  
 *print*(arg)  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 main()

Так заведено і функцію main ще називають "точкою входу", оскільки робота застосунку починається з виклику цієї функції. Ви, звичайно, можете назвати цю функцію як завгодно, але називати її саме main вважається хорошим тоном.

**Модуль 4: Структури даних**

### **Колекції**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson04/lesson-04/#%D0%BA%D0%BE%D0%BB%D0%B5%D0%BA%D1%86%D1%96%D1%97)

Колекція у Python — програмний об'єкт (змінна-контейнер), що зберігає набір значень одного або різних типів.

**Простою мовою**, колекція — це сховище інформації, каталог, побудований за якимось принципом.

Сховища бувають різними: може бути мішок, з якого ти дістаєш вміст навмання, може бути розкладений за абеткою каталог, в якому дуже легко щось знайти, або труба, з якої можна дістати тільки те, що лежить з краю тощо

Залежно від цілей, можна і треба застосовувати різні колекції. Наприклад, щоб швидко знайти щось, використовують словники; для зберігання впорядкованих даних, коли порядок має значення — списки та кортежі; для зберігання тільки унікальних елементів — множини; для зберігання символів — рядки тощо.

У цьому уроці ми вивчимо основні колекції Python і навчимося працювати з ними. Це основа мови і знання можливостей колекцій є не лише дуже корисною навичкою, але й визначальною для розробника на Python.

Основні властивості колекцій:

* Впорядкованість. Впорядкований контейнер (колекція) дає можливість звертатися до своїх елементів за індексом (номером) і гарантує, що порядок елементів зберігається.
* Змінність. Якщо колекція змінювана, то її вміст можна змінювати, не створюючи нову колекцію. Наприклад додавати, видаляти, заміняти елементи колекції.
* Унікальність. Унікальність стосується вмісту колекції, чи допускається зберігання однакових об'єктів.

### **Списки**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson04/lesson-04/#%D1%81%D0%BF%D0%B8%D1%81%D0%BA%D0%B8)

Список — **впорядкований змінюваний** контейнер даних. Списки не диктують розробнику тип даних, який можна помістити в цей контейнер, і можуть містити будь-які типи даних у будь-якому зручному порядку.

Для створення порожнього списку існує два способи:

my\_list = list()  
  
empty\_list = []

Щоб створити заповнений список:

not\_empty = [1, 2, 'user']

### **Впорядковані контейнери**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson04/lesson-04/#%D0%B2%D0%BF%D0%BE%D1%80%D1%8F%D0%B4%D0%BA%D0%BE%D0%B2%D0%B0%D0%BD%D1%96-%D0%BA%D0%BE%D0%BD%D1%82%D0%B5%D0%B9%D0%BD%D0%B5%D1%80%D0%B8)

### **Доступ за індексом**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson04/lesson-04/#%D0%B4%D0%BE%D1%81%D1%82%D1%83%D0%BF-%D0%B7%D0%B0-%D1%96%D0%BD%D0%B4%D0%B5%D0%BA%D1%81%D0%BE%D0%BC)

Під впорядкованістю варто розуміти властивість контейнера зберігати порядок елементів при роботі з ним, видаляючи елемент, додаючи новий, роблячи вставку/видалення з кінця/початку/середини, ви гарантуєте, що усі елементи, окрім тих, яких безпосередньо торкнулася операція, зберігають порядок.

Найкориснішою властивістю впорядкованості є можливість доступу до елементів контейнера за індексом цього елемента в контейнері.

У Python синтаксис доступу за індексом виглядає так:

some\_iterable = ["a", "b", "c"]  
first\_letter = some\_iterable[0]  
middle\_one = some\_iterable[1]  
last\_letter = some\_iterable[2]

У першому рядку ми створили список з трьох перших літер англійського алфавіту.

У другому рядку ми зберегли у змінну first\_letter літеру "a" — перший елемент some\_iterable. **Індекс** у Python починається з 0, як і в більшості мов програмування, та індексом "a" є 0.

Третій рядок — це звернення до другого елементу some\_iterable, його індекс дорівнює 1 — це літера "b" і ми зберігаємо її у middle\_one.

Четвертий рядок — це звернення до останнього елементу some\_iterable, літери "c", ми збережемо її у last\_letter і її індекс дорівнює 2.

Python підтримує індексування елементів з кінця. Для цього потрібно додати - і вказати номер елементу з кінця. Оскільки у Python -0 == 0, то перший елемент з кінця — це -1, другий — -2 і так далі. Наш приклад можна переписати, використовуючи індексування з кінця, ось так:

some\_iterable = ["a", "b", "c"]  
first\_letter = some\_iterable[-3]  
middle\_one = some\_iterable[-2]  
last\_letter = some\_iterable[-1]

Найкориснішою властивістю списків є змінність списків, ви можете змінити значення будь-якого елементу списку:

some\_iterable = ["a", "b", "c"]  
some\_iterable[1] = "Z"  
print(some\_iterable) *# ["a", "Z", "c"]*

В цьому прикладі ми змінили другий елемент списку some\_iterable (другий елемент — це елемент з індексом 1) на 'Z'.

### **Зрізи у Python (Slice)**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson04/lesson-04/#%D0%B7%D1%80%D1%96%D0%B7%D0%B8-%D1%83-python-slice)

Для впорядкованих контейнерів є спеціальний синтаксис, коли нам необхідно отримати деяку послідовність елементів з контейнера. Наприклад, якщо ми хочемо отримати перші 5 літер рядку:

some\_str = "This is awesome string"  
first\_five = some\_str[0:5]

first\_five в цьому прикладі буде містити рядок 'This '.

Синтаксис полягає у зазначенні індексу першого елементу зрізу, індексу, до якого (не включно) брати елементи в нову послідовність, та кроку, з яким брати елементи між початковим та кінцевим індексом у квадратних дужках, розділивши їх двокрапкою.

Візьмімо список чисел від 1 до 10 і збережемо окремо парні, не парні та кратні 3.

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
  
odd\_numbers = numbers[0:9:2]  
even\_numbers = numbers[1:9:2]  
three\_numbers = numbers[2:9:3]

У odd\_numbers ми беремо числа, починаючи з індексу 0 до 10 з кроком 2 (отримаємо [1, 3, 5, 7, 9]). У even\_numbers ми беремо числа, починаючи з індексу 1 до 10 з кроком 2 (отримаємо [2, 4, 6, 8]) У three\_numbers ми беремо числа, починаючи з індексу 2 до 10 з кроком 3 (отримаємо [3, 6, 9]).

Ви можете не вказувати початковий, кінцевий індекс або крок, пропускаючи його. За замовчуванням Python візьме зріз з початку до останнього елемента з кроком 1. Перепишемо попередній приклад у скороченому синтаксисі:

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
  
odd\_numbers = numbers[::2]  
even\_numbers = numbers[1::2]  
three\_numbers = numbers[2:9:3]  
  
numbers\_copy = numbers[:]

numbers\_copy в цьому прикладі — це зріз, який бере всі елементи numbers від початку і до кінця з кроком 1.

Також важливо пам'ятати, що в зріз не входить елемент з індексом, до якого брати елементи.

numbers = [0, 1, 2, 3]  
first\_three = numbers[0:3] *# [0, 1, 2]*

В цьому прикладі елемент з індексом 3 не увійде у first\_three.

### **Використання методів об'єктів**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson04/lesson-04/#%D0%B2%D0%B8%D0%BA%D0%BE%D1%80%D0%B8%D1%81%D1%82%D0%B0%D0%BD%D0%BD%D1%8F-%D0%BC%D0%B5%D1%82%D0%BE%D0%B4%D1%96%D0%B2-%D0%BE%D0%B1%D1%94%D0%BA%D1%82%D1%96%D0%B2)

Об'єкт в програмуванні — деяка сутність у цифровому просторі, що має певний стан і поведінку, має певні властивості (атрибути) та операції над ними (методи).

Доступ до методів об'єктів у Python синтаксично відбувається за допомогою символу крапки після імені об'єкту і зазначення імені методу або атрибуту, до якого потрібно отримати доступ.

numbers = ['a', 'b']  
numbers.append('c')  
print(numbers) *# ['a', 'b', 'c']*

В цьому прикладі ми використали метод append, який є у списків і він є у списку numbers. Цей метод додає новий елемент в кінець списку. Як аргумент цей метод отримує елемент, який потрібно додати до списку. Аргументи вказуються в дужках.

Якщо метод не вимагає аргументів (наприклад метод clear), то дужки будуть порожніми:

num = [1, 2]  
num.clear()  
print(num) *# []*

### **Методи списків**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson04/lesson-04/#%D0%BC%D0%B5%D1%82%D0%BE%D0%B4%D0%B8-%D1%81%D0%BF%D0%B8%D1%81%D0%BA%D1%96%D0%B2)

* Додавання елементу в кінець списку: my\_list.append(element)

chars = ['a', 'b']  
chars.append('c')  
print(chars) *# ['a', 'b', 'c']*

* видалення елементу зі списку викличе помилку, якщо такого елементу немає в списку: my\_list.remove(element)

chars = ['a', 'b']  
chars.remove('b')  
print(chars) *# ['a']*

* Повернути i-ий елемент та видалити його зі списку i\_element = my\_list.pop(i). За замовчуванням i = -1

chars = ['a', 'b']  
last = chars.pop(1)  
print(chars) *# ['a']*  
print(last) *# 'b'*

* Розширити список a\_list елементами з b\_list: a\_list.extend(b\_list)

chars = ['a', 'b']  
numbers = [1, 2]  
  
chars.extend(numbers)  
print(chars) *# ['a', 'b', 1, 2]*

* Вставити x на позицію з індексом i: my\_list.insert(i, x)

chars = ["a", "b"]  
chars.insert(1, "c")  
print(chars) *# ['a', 'c', 'b']*

* Очистити список: my\_list.clear()

chars = ['a', 'b']  
last = chars.clear() print(chars) *# []*

* Знайти індекс першого елемента у списку, що дорівнює x: index = my\_list.index(x)

chars = ['a', 'b', 'c', 'd']  
c\_ind = chars.index('c') print(c\_ind) *#2*

* Повернути кількість елементів у списку, що дорівнюють x: x\_number = my\_list.count(x)

chars = ['a', 'b', 'c', 'a']  
a\_count = chars.count('a')  
print(a\_count) *# 2*

* Відсортувати список за зростанням: my\_list.sort(key=None, reverse=False)

chars = ['z', 'a', 'b']  
chars.sort()  
print(chars) *# ['a', 'b', 'z']*

* Змінити порядок елементів у списку на зворотний: my\_list.reverse()

chars = ['a', 'b']  
chars.reverse()  
print(chars) *# ['b', 'a']*

* Повернути копію списку: copy\_of\_my\_list = my\_list.copy()

chars = ['a', 'b']  
chars\_copy = chars.copy()  
chars == chars\_copy *# True*

### **Словники**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson04/lesson-04/#%D1%81%D0%BB%D0%BE%D0%B2%D0%BD%D0%B8%D0%BA%D0%B8)

Словник — це контейнер, який зберігає пари ключ-значення. Ключем може бути будь-який незмінний тип даних Python (число, рядок, кортеж тощо). Неможливо використовувати у якості ключа списки, словники, множини або будь-які інші змінювані типи даних. Значенням словника може бути будь-який тип даних Python, включаючи призначені для користувача типи.

### **Методи словників**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson04/lesson-04/#%D0%BC%D0%B5%D1%82%D0%BE%D0%B4%D0%B8-%D1%81%D0%BB%D0%BE%D0%B2%D0%BD%D0%B8%D0%BA%D1%96%D0%B2)

Деякі методи словників, що найчастіше використовуються:

* pop(key) — повертає значення елементу і видаляє пару ключ-значення зі словника

chars = {'a': 1, 'b': 2}  
b\_num = chars.pop('b')  
print(chars) *# {'a': 1}*  
print(b\_num) *# 2*

* update(another\_dict) — розширює словник значеннями з іншого словника

chars = {'a': 1, 'b': 2}  
chars.update({"c": 3})  
print(chars) *# {'a': 1, 'b': 2, "c": 3}*

* clear() — очищає словник, не створюючи нового

chars = {'a': 1, 'b': 2}  
chars.clear()  
print(chars) *# {}*

* copy() — повертає копію словника

chars = {'a': 1, 'b': 2}  
chars\_copy = chars.copy()  
chars\_copy == chars *# True*

* get(key[, default]) — не викликає виключення, якщо ключа немає в словнику, повертає default, за замовчуванням default=None.

chars = {'a': 1, 'b': 2}  
c\_idx = chars.get('c', -1)  
print(c\_idx) *# -1*

### **Цикли та словники**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson04/lesson-04/#%D1%86%D0%B8%D0%BA%D0%BB%D0%B8-%D1%82%D0%B0-%D1%81%D0%BB%D0%BE%D0%B2%D0%BD%D0%B8%D0%BA%D0%B8)

Ітерування за словником — це блок коду, що дуже часто зустрічається, і корисно вміти це робити.

Спершу варто сказати, що словник сам по собі — це ітерований контейнер і за ним можна ітеруватися в циклі for без необхідності заводити якийсь зовнішній лічильник тощо. Створимо словник, в якому ключами будуть числа, а значеннями — числівники англійською:

numbers = {  
 1: "one",  
 2: "two",  
 3: "three"  
}

Тепер давайте просто пройдемо словником в циклі та виведемо, що нам повертає ітератор на кожній ітерації:

for key in numbers:  
 print(key)

У виведенні ви побачите:

1  
2  
3

Ітеруючи за словником, ви перебираєте ключі словника. Таку саму поведінку можна отримати, використовуючи метод keys, але так ви явно вкажете, що хочете перебрати ключі:

for key in numbers.keys():  
 print(key)

Відповідь буде точно такою самою:

1  
2  
3

Часто необхідно перебрати саме значення словника, для цього скористаємося методом values:

for val in numbers.values():  
 print(val)

У виведенні буде:

one  
two  
tree

І переберемо пари ключ значення, використовуючи метод items. На кожній ітерації ми отримаємо пару (ключ, значення):

for key, value in numbers.items():  
 print(key, value)

Виведення:

1 one  
2 two  
3 three

Що **не можна** робити, поки ітеруєтеся за словником: не можна видаляти елементи із словника, не можна додавати елементи у словник. Але можна перезаписувати значення, якщо ви ітеруєтеся за ключами.

### **Множини**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson04/lesson-04/#%D0%BC%D0%BD%D0%BE%D0%B6%D0%B8%D0%BD%D0%B8)

Множини — це неврегульований контейнер, який містить тільки унікальні елементи. У множину можна додавати тільки незмінні типи даних.

Є тільки один спосіб створити порожню множину:

a = set()  
print(a) *# set()*

Для створення заповненої множини достатньо передати будь-який об'єкт, що ітерується, в функцію set:

a = set('hello')  
print(a) *# {'e', 'h', 'l', 'o'}*

Або ж скористатися синтаксисом з фігурними дужками (як у словниках), але елементи у фігурних дужках просто перелічити через кому без двокрапок:

b = {1, 2, 3, 4}

Унікальність має на увазі, що якщо множина вже містить такий елемент, то спроба додати ще один такий самий нічого не змінить.

numbers = {1, 2, 3, 1, 2, 3}  
print(numbers) *# {1, 2, 3}*

### **Методи множин**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson04/lesson-04/#%D0%BC%D0%B5%D1%82%D0%BE%D0%B4%D0%B8-%D0%BC%D0%BD%D0%BE%D0%B6%D0%B8%D0%BD)

Множини підтримують наступні методи:

* add(elem) — додає елемент в множину

numbers = {1, 2, 3}  
numbers.add(4)  
print(numbers) *# {1, 2, 3, 4}*

* remove(elem) — видаляє елемент з множини, викликає виняток, якщо такого елементу немає

numbers = {1, 2, 3}  
numbers.remove(3)  
print(numbers) *# {1, 2}*

* discard(elem) — видаляє елемент з множини і не викликає виняток, якщо його немає

numbers = {1, 2, 3}  
numbers.discard(2)  
print(numbers) *# {1, 3}*

### **Математичні операції над множинами**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson04/lesson-04/#%D0%BC%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D1%87%D0%BD%D1%96-%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D1%96%D1%97-%D0%BD%D0%B0%D0%B4-%D0%BC%D0%BD%D0%BE%D0%B6%D0%B8%D0%BD%D0%B0%D0%BC%D0%B8)

Давайте детальніше розглянемо, які корисні математичні операції можна робити над множинами. Спершу створимо множини a та b:

a = set('hello')  
print(a) *# {'e', 'h', 'l', 'o'}*  
  
b = set('hi there!')  
print(b) *# {'r', ' ', 'i', 'e', '!', 'h', 't'}*

Щоб знайти загальні елементи для двох множин, виконаємо над ними операцію & (AND):

a & b *# {'e', 'h'}*

Знайдемо усі елементи з двох множин, окрім загальних, за допомогою оператора ^ (XOR):

a ^ b *# {' ', '!', 'i', 'l', 'o', 'r', 't'}*

Об'єднання множин, або просто усі елементи з обох множин знаходяться за допомогою оператора | (OR):

a | b *# {' ', '!', 'e', 'h', 'i', 'l', 'o', 'r', 't'}*

Множини — це дуже потужний інструмент, коли необхідно знайти унікальні елементи в якомусь наборі і прибрати дублікати. Множини — це також найшвидший спосіб знайти загальні або відмінні елементи з декількох наборів.

### **Використання кортежів як ключів**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson04/lesson-04/#%D0%B2%D0%B8%D0%BA%D0%BE%D1%80%D0%B8%D1%81%D1%82%D0%B0%D0%BD%D0%BD%D1%8F-%D0%BA%D0%BE%D1%80%D1%82%D0%B5%D0%B6%D1%96%D0%B2-%D1%8F%D0%BA-%D0%BA%D0%BB%D1%8E%D1%87%D1%96%D0%B2)

Незмінність кортежів обмежує їх застосування, у порівнянні зі списками, але дає можливість використати кортежі як ключі для словника, або елементи множини. Наприклад, розглянемо набір точок на площині (кортежі). Їх можна використовувати як ключі у словнику:

points = {  
 (0, 0): "O"  
 (1, 1): "A"  
 (2, 2): "B"  
}

У словнику points використовуються кортежі з координатами точок як ключі, а значення — це імена (назва) точок.

При спробі змінити кортеж ви отримаєте повідомлення про помилку:

not\_empty[3] = 's'  
---------------------------------------------------------------------------  
TypeError Traceback (most recent call last)  
<ipython-input-6-0e33ea041f91> in <module>()  
----> 1 not\_empty[3] = 's'  
  
TypeError: 'tuple' object does not support item assignment

Є одна важлива особливість створення кортежів — це створення кортежів з одним елементом. Річ у тому, що Python використовує круглі дужки як математичні символи. Якщо написати вираз на кшталт a = (1), виникає невизначеність. Інтерпретатор розуміє такий вираз, як суто математичний, і просто прибирає зайві дужки, присвоюючи a значення 1. Щоб вказати, що це не математична операція, а саме кортеж, потрібно додати кому після значення. Тоді інтерпретатор однозначно зрозуміє, що ви хочете створити кортеж з одним елементом:

a = (1, )

### **Рядки**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson04/lesson-04/#%D1%80%D1%8F%D0%B4%D0%BA%D0%B8)

Робота з текстовими даними у Python реалізована через str-об'єкти або рядки.  
**Рядок** — це незмінна впорядкована послідовність символів в деякому кодуванні. За замовчуванням використовується кодування UTF-8, але можна працювати майже з усіма відомими таблицями кодування символів. Для того щоб створити змінну типу "рядок", необхідно певний набір символів взяти в лапки.

* Варіант 1. Одинарні лапки (апостроф) 'some text'
* Варіант 2. Подвійні лапки "some text".

Різні варіанти використання лапок обумовлені тим, що при використанні одинарних лапок, можна в рядку вказати подвійні і навпаки.

game\_string = 'My "Game"'

**Впорядкована** послідовність означає, що до елементів рядку можна звертатися за індексом:

s = "Hello world!"  
print(s[0]) *# H*  
print(s[-1]) *# !*

**Незмінна** послідовність означає, що якщо рядок вже створений, то змінити його не можна, можна тільки створити новий.

s = "Hello world!"  
s[0] = "Q" *# Тут буде викликано виняток (помилка) TypeError*

**Малі методи**

Для того, щоб усі літери рядка перевести у верхній регістр, використовується метод upper:

s = "Hello"  
s.upper()  
print(s) *# Виведе 'HELLO'*

Для переведення в нижній регістр використовується метод lower():

s = "Some Text"  
print(s.lower()) *# Виведе 'some text'*

Щоб перевірити, що рядок починається з підрядка, є метод startswith:

s = "Bill Jons"  
print(s.startswith("Bi")) *# Виведе True*

Щоб перевірити, що рядок закінчується підрядком, використовується метод endswith:

s = "hello.jpg"  
print(s.endswith("jpg")) *# Виведе True*

Цей метод зручно використовувати для перевірки розширення файлів.

### **Загальні для усіх колекцій операції**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson04/lesson-04/#%D0%B7%D0%B0%D0%B3%D0%B0%D0%BB%D1%8C%D0%BD%D1%96-%D0%B4%D0%BB%D1%8F-%D1%83%D1%81%D1%96%D1%85-%D0%BA%D0%BE%D0%BB%D0%B5%D0%BA%D1%86%D1%96%D0%B9-%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D1%96%D1%97)

Такі різні списки, кортежі, словники, множини і рядки об'єднують в одну групу — колекції, тому що в них є загальні властивості:

* перевірка на входження;
* кількість елементів;
* перебір усіх елементів в циклі for.

### **Перевірка на входження**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson04/lesson-04/#%D0%BF%D0%B5%D1%80%D0%B5%D0%B2%D1%96%D1%80%D0%BA%D0%B0-%D0%BD%D0%B0-%D0%B2%D1%85%D0%BE%D0%B4%D0%B6%D0%B5%D0%BD%D0%BD%D1%8F)

Будь-яка колекція дозволяє перевірити, чи містить колекція цей елемент (чи є там такий самий). Для цього використовується оператор in.

Наприклад, перевірка на те, що користувач не використовує простий пароль і в паролі не зустрічається послідовність "qwerty" або "123":

password = "qwerty123"  
if "qwerty" in password or "123" in password:  
 print("This password is too weak!")

Оператор in перевіряє наявність елементу ('qwerty', '123') у контейнері (password) та повертає True або False.

prime\_numbers = {2, 3, 5, 7, 11, 13, 17, 19, 23}  
is\_prime = 3 in prime\_numbers

Перевірка на те, що число 3 присутнє у множині перших дев'яти простих чисел prime\_numbers.

Із словниками перевірка на входження перевіряє, що елемент присутній серед ключів словника.

user = {  
 "name": "Bill",  
 "surname": "Bosh",  
 "age": 22  
}  
  
if "age" in user:  
 print(f"User is {user['age']} years old.")

### **Кількість елементів**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson04/lesson-04/#%D0%BA%D1%96%D0%BB%D1%8C%D0%BA%D1%96%D1%81%D1%82%D1%8C-%D0%B5%D0%BB%D0%B5%D0%BC%D0%B5%D0%BD%D1%82%D1%96%D0%B2)

Будь-яка колекція дозволяє дізнатися кількість елементів у ній за допомогою функції len.

password = input("Password: ")  
if len(password) < 8:  
 print("Your password is too short")

Наприклад, перевірка довжини пароля введеного користувача може бути реалізована за допомогою функції len.

### **Перебір усіх елементів колекції в циклі for**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson04/lesson-04/" \l "%D0%BF%D0%B5%D1%80%D0%B5%D0%B1%D1%96%D1%80-%D1%83%D1%81%D1%96%D1%85-%D0%B5%D0%BB%D0%B5%D0%BC%D0%B5%D0%BD%D1%82%D1%96%D0%B2-%D0%BA%D0%BE%D0%BB%D0%B5%D0%BA%D1%86%D1%96%D1%97-%D0%B2-%D1%86%D0%B8%D0%BA%D0%BB%D1%96-for" \o "Пряме посилання на перебір-усіх-елементів-колекції-в-циклі-for)

alphabet = "abcdefghijklmnopqrstuvwxyz"  
for char in alphabet:  
 print(char)

По будь-якій колекції можна пройти за допомогою циклу for і на кожній ітерації в циклі буде отриманий один з елементів цієї колекції. У прикладі літери алфавіту з alphabet виводяться в консоль по черзі в циклі for.

Давайте проітеруємо список some\_iterable у циклі та виведемо в консоль, що ми отримуємо на кожній ітерації:

some\_iterable = ["a", "b", "c"]  
  
for i in some\_iterable:  
 print(i)

У консолі ми побачимо:

a  
b  
c

odd\_numbers = [1, 3, 5, 7, 9]  
for i in odd\_numbers:  
 print(i \*\* 2)

Код з цього прикладу виведе в консоль квадрати перших п'яти непарних чисел.

### **Робота з файловою системою, пакет pathlib**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson04/lesson-04/" \l "%D1%80%D0%BE%D0%B1%D0%BE%D1%82%D0%B0-%D0%B7-%D1%84%D0%B0%D0%B9%D0%BB%D0%BE%D0%B2%D0%BE%D1%8E-%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%BE%D1%8E-%D0%BF%D0%B0%D0%BA%D0%B5%D1%82-pathlib" \o "Пряме посилання на Робота з файловою системою, пакет pathlib)

Для роботи з файловою системою в стандартному постачанні Python йде пакет pathlib, він містить безліч корисних функцій для роботи з файловою системою. Детальний перелік можливостей pathlib рекомендуємо вивчити на [сторінці офіційної документації](https://docs.python.org/3/library/pathlib.html).

Основний інструмент у pathlib — Path, об'єкт, який є шляхом (адреса у файловій системі). В основному робота з файловою системою ведеться через Path. Path варто сприймати як вказівку на файл або папку. Щоб створити такий Path, достатньо викликати Path як функцію та передати у якості аргументу рядок-адресу у файловій системі:

from pathlib import Path  
  
p = Path('/home/user/Downloads') *# p Вказує на папку /home/user/Downloads*

Можна викликати Path без аргументів, тоді ви отримаєте вказівник на папку, в якій ви зараз знаходитеся.

from pathlib import Path  
  
p = Path() *# p Вказує на папку, з якої був запущений Python*

У Path є ряд корисних методів та атрибутів:

* p.parent вказує на батьківську папку;
* p.name повертає лише ім'я (рядок) папки або файлу, на який вказує p;
* p.suffix повертає рядком розширення файлу, на який вказує p, починаючи з крапки;

p = Path('setup.py')  
p.suffix *# '.py'*

* p.exists() повертає True або False, залежно від того, чи існує такий файл або папка;
* p.is\_dir() повертає True, якщо p вказує на папку, та False, якщо на файл, або такий шлях не існує;
* p.is\_file() повертає True, якщо p вказує на файл, та False, якщо на папку, або такий шлях не існує;
* p.iterdir() повертає ітератор за всіма файлам та папками всередині папки p;

from pathlib import Path  
  
p = Path('/home/user/Downloads') *# p Вказує на папку /home/user/Downloads*  
for i in p.iterdir():  
 print(i.name) *# Виведе у циклі імена всіх папок та файлів у /home/user/Downloads*

Варто розуміти, що i в цьому прикладі також будуть об'єктами Path, але вказувати вони будуть на файли та папки всередині '/home/user/Downloads'.

З концепцією об'єктів ми познайомимося дещо пізніше, поки достатньо знати, що i — це як і p теж вказівники на папки і файли, але кожен i вказує на якийсь свій файл або папку.

### **Обробка аргументів командного рядка**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson04/lesson-04/#%D0%BE%D0%B1%D1%80%D0%BE%D0%B1%D0%BA%D0%B0-%D0%B0%D1%80%D0%B3%D1%83%D0%BC%D0%B5%D0%BD%D1%82%D1%96%D0%B2-%D0%BA%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4%D0%BD%D0%BE%D0%B3%D0%BE-%D1%80%D1%8F%D0%B4%D0%BA%D0%B0)

Коли запускаємо збережений у файлі Python скрипт, є можливість передати йому при запуску якісь аргументи, як у функцію. Тоді наш скрипт може ці аргументи прийняти і якось змінити свою поведінку. Зробити це можна за допомогою пакета sys, в якому є список argv, де з'являються всі аргументи, з якими був запущений скрипт.

Цікавою особливістю sys.argv є те, що першим елементом цього списку буде назва самого файлу скрипту. Всі аргументи будуть у sys.argv у вигляді рядків в тому самому порядку, в якому вони були передані під час виклику. Python розділяє аргументи пробілами і в sys.argv пробіли не потрапляють.

Щоб зрозуміти як працює sys.argv, можете зробити простий скрипт echo.py, який буде виводити у консоль всі передані при виклику аргументи.

import sys

for arg in sys.argv:  
 print(arg)

Якщо викликати скрипт з таким вмістом (припустимо він називається echo.py) командою: python echo.py test --user -hello some text, то у консолі ви побачите:

echo.py  
test  
--user  
-hello  
some  
text

Таким чином, якщо вам потрібно обробити перший аргумент запуску:

import sys  
def main():  
 print(sys.argv[1])

Під час виклику такого скрипту командою python echo.py 123 ви побачите у консолі 123. Зверніть увагу, що всі елементи списку sys.argv — це рядки. Якщо ви очікуєте, що користувач повинен ввести число (ціле або дробове), то вам потрібно перетворити рядок в потрібний вам тип самостійно.

## Модуль 5: Просунута робота з рядками

Ми вже поверхнево знайомилися з рядками у Python. Ви вмієте створювати рядки, об'єднувати рядки, міняти регістр рядків, перевіряти, що рядок закінчується або починається деякою послідовністю. Але можливості Python по роботі з рядками не вичерпуються цим. На цьому зайнятті ми дізнаємося більше про те, як працювати з рядками у Python, про роботу з регулярними виразами та перетвореннями рядків.

### **Ще варіанти створення рядків**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson05/lesson-05#%D1%89%D0%B5-%D0%B2%D0%B0%D1%80%D1%96%D0%B0%D0%BD%D1%82%D0%B8-%D1%81%D1%82%D0%B2%D0%BE%D1%80%D0%B5%D0%BD%D0%BD%D1%8F-%D1%80%D1%8F%D0%B4%D0%BA%D1%96%D0%B2)

Для створення рядків можна скористатися одинарними або подвійними лапками:

this\_is\_string = "Hi there!"  
  
the\_same\_string = 'Hi there!'  
  
this\_is\_string == the\_same\_string # True

Але що робити, якщо нам потрібен текст із перенесенням рядків (коли в тексті більше одного рядка)? Для цього можна скористатися потрійним повторенням лапок:

text = """This is first line  
And second line  
Last third line"""  
  
song = '''Jingle bells, jingle bells  
Jingle all the way  
Oh, what fun it is to ride  
In a one horse open sleigh'''

В цьому прикладі змінна text містить три рядки, а song — чотири рядки.

Коли інтерпретатор виявляє лапки, повторені тричі, він сприймає усі символи до наступних трьох закриваючих лапок, як символи рядка.

Зворотна ситуація, у вас є довгий рядок, який не повинен містити перенесень, але в коді його незручно відобразити одним рядком.

one\_line\_text = "Textual data in Python is handled with str objects, or strings. Strings are immutable sequences of Unicode code points. String literals are written in a variety of ways: single quotes, double quotes, triple quoted.

Щоб структурувати код і не додавати зайвих перенесень, ви можете розбити одну рядкову змінну на декілька частин:

one\_line\_text = "Textual data in Python is handled with str objects, or strings. "\  
 "Strings are immutable sequences of Unicode code points. "\  
 "String literals are written in a variety of ways: single quotes, double quotes, triple quoted."

Зверніть увагу на символ \ в кінці першого та другого рядка коду, він вказує інтерпретатору ігнорувати закінчення рядка і продовжити відразу з наступного.

one\_line\_text в обох прикладах буде містити один і той самий текст без перенесень.

Python сприймає рядкові змінні, між якими немає нічого, крім пробілів та/або символів нового рядка, як один рядок:

("spam " "eggs") == "spam eggs" # True

Вираз ліворуч та вираз праворуч — це два рівнозначні записи одного і того самого тексту 'spam eggs' і, з погляду Python, — вони нерозрізнені.

### **Спеціальні символи**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson05/lesson-05#%D1%81%D0%BF%D0%B5%D1%86%D1%96%D0%B0%D0%BB%D1%8C%D0%BD%D1%96-%D1%81%D0%B8%D0%BC%D0%B2%D0%BE%D0%BB%D0%B8)

Ми неодноразово згадували символ перенесення рядка. Це один із "спеціальних" або "керуючих" символів. Керуючі символи — це символи перенесення рядка, табуляції, повернення каретки та інші символи, які не можна або незручно ввести з клавіатури. Для того щоб можна було вводити символи, які незручно або не можна ввести з клавіатури , придумали додавати екрануючий символ '\' який означає, що наступний за ним знак потрібно сприймати як спеціальний символ, а не буквально.

Ось деякі керуючі символи:

| Позначення у коді | Опис |
| --- | --- |
| \n | Перенесення рядка |
| \f | Перенесення сторінки |
| \r | Повернення каретки |
| \t | Горизонтальна табуляція |
| \v | Вертикальна табуляція |

Наприклад, текст з явним розбиттям на рядки:

jingle\_bells = "Jingle bells, jingle bells\nJingle all the way\nOh, what fun it is to ride\nIn a one horse open sleigh"

Вибирайте ту форму запису, яка більше підходить під конкретну ситуацію, і керуйтеся зручністю для запису і читання коду, який містить рядкову змінну.

### **Методи рядків**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson05/lesson-05#%D0%BC%D0%B5%D1%82%D0%BE%D0%B4%D0%B8-%D1%80%D1%8F%D0%B4%D0%BA%D1%96%D0%B2)

Ми вже познайомилися з деякими методами рядків, зараз розберемо методи, пов'язані з пошуком в рядках і створенням нових рядків.

### **Пошук у рядку**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson05/lesson-05#%D0%BF%D0%BE%D1%88%D1%83%D0%BA-%D1%83-%D1%80%D1%8F%D0%B4%D0%BA%D1%83)

Для пошуку деякого символа або підрядка у рядку можна скористатися методом find:

s = "Hi there!"  
  
start = 0  
end = 7  
  
print(s.find("er", start, end)) # 5  
print(s.find("q")) # -1

Цей метод виводить індекс початку першого збігу в рядку s, починаючи з start до end. Якщо start та end не вказані, то з початку і до кінця рядку. Повертає -1, якщо послідовність не знайдена.

Ще один метод пошуку підрядка у рядку дуже схожий на find — це index. Основна відмінність полягає в тому, що якщо index не знайде підрядок, то викличе виняток ValueError.

Якщо вам потрібно здійснити пошук підрядка у рядку справа, а не зліва як у find, то для цього є метод rfind:

s = 'Some words'  
s.rfind('o') # 6

І "правий" аналог index — rindex:

s = 'Some words'  
s.rindex('o') # 6

### **Розбиття рядка на декілька підрядків**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson05/lesson-05" \l "%D1%80%D0%BE%D0%B7%D0%B1%D0%B8%D1%82%D1%82%D1%8F-%D1%80%D1%8F%D0%B4%D0%BA%D0%B0-%D0%BD%D0%B0-%D0%B4%D0%B5%D0%BA%D1%96%D0%BB%D1%8C%D0%BA%D0%B0-%D0%BF%D1%96%D0%B4%D1%80%D1%8F%D0%B4%D0%BA%D1%96%D0%B2" \o "Пряме посилання на Розбиття рядка на декілька підрядків)

Часта ситуація, коли необхідно розбити рядок на підрядки за деяким символом, наприклад, розбити текст на речення за символом крапки та пробілу після крапки, або речення за словами.

s = "I am learning strings in Python. Some new methods got now."  
sentences = s.split(". ")  
  
print(sentences[0]) # I am learning strings in Python  
print(sentences[1]) # Some new methods got now.

Для такого завдання можна скористатися методом split, який приймає як аргумент підрядок-маркер, який буде межею, по якій потрібно розбити рядок на частини. В результаті виклику повертається список рядків.

### **Нові рядки на основі рядків**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson05/lesson-05#%D0%BD%D0%BE%D0%B2%D1%96-%D1%80%D1%8F%D0%B4%D0%BA%D0%B8-%D0%BD%D0%B0-%D0%BE%D1%81%D0%BD%D0%BE%D0%B2%D1%96-%D1%80%D1%8F%D0%B4%D0%BA%D1%96%D0%B2)

Усі рядки незмінні, якщо ми хочемо модифікувати рядок, то є тільки один спосіб — створити новий рядок на основі вихідного.

Усі методи, які якось "модифікують" рядки, повертають нові рядки, ніяк не змінюючи оригінальний.

Для об'єднання декількох рядків в один через деякий роздільник використовується метод join:

sentences = ["I am learning strings in Python", "Some new methods got now."]  
text = ". ".join(sentences)  
print(text) # I am learning strings in Python. Some new methods got now.

По суті, це зворотна операція split

Якщо потрібно видалити зайві пробіли на початку і в кінці рядка, є спеціальний метод strip:

clean = ' spacious '.strip()  
print(clean) # spacious

У цього метода є два "брати":

"лівий", lstrip, видаляє тільки пробіли на початку рядка;

та "правий", rstrip, видаляє тільки пробіли в кінці рядка.

Коли ж нам потрібно замінити деякий підрядок в рядку, ми можемо скористатися методом replace:

dogs\_text = "All dogs bark like dogs."  
cats\_text = dogs\_text.replace("dogs", "cats")  
print(cats\_text) # All cats bark like cats.

Для видалення фіксованої послідовності на початку рядка є метод removeprefix:

print('TestHook'.removeprefix('Test')) # Hook  
print('TestHook'.removeprefix('Hook')) # TestHook

Є парний метод для видалення послідовності в кінці рядка, removesuffix:

print('TestHook'.removesuffix('Hook')) # Test  
print('TestHook'.removesuffix('Test')) # TestHook

### **Translate**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson05/lesson-05" \l "translate" \o "Пряме посилання на Translate)

В рядках є метод translate, цей метод дозволяє замінити символ в рядку на інший з мапи відповідності, яку можна задати.

map = {ord('з'): 'z', ord('ю'): 'u')}  
translated = 'зю'.translate(map)  
print(translated) # zu

В цьому прикладі мапою відповідності виступає словник map і в цій мапі ми встановлюємо відповідність між символами 'з' та 'z', та 'ю', та 'u'.

Функція ord (від order) дозволяє дізнатися, яким числом кодується символ. Кодування символів ми розглянемо на іншому зайнятті, а поки що достатньо знати, що усі символи кодуються у пам'яті комп'ютера числами, і функція ord дозволяє дізнатися, яким саме числом кодується символ, щоб однозначно визначити, що це за символ.

Неочевидний факт — багато складних символів, емодзі, наприклад, можна закодувати декількома способами, а відображатися вони будуть однаково.

Наприклад, символ 'a' кодується числом 97:

ord('a') # 97

Найчастіше застосування методу translate пов'язане з транслітерацією рядків.

### **Форматування рядків (метамова форматування)**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson05/lesson-05#%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F-%D1%80%D1%8F%D0%B4%D0%BA%D1%96%D0%B2-%D0%BC%D0%B5%D1%82%D0%B0%D0%BC%D0%BE%D0%B2%D0%B0-%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F)

Будь-яке число можна записати кількома варіантами запису:

* десятковий запис
* двійкове представлення
* шістнадцяткове представлення
* наукова нотація
* з фіксованою точністю (кількістю знаків після коми)

та інші.

Наприклад, вивести числа від 1 до 15 в десятковому, шістнадцятковому, вісімковому і двійковому представленні:

for i in range(16):  
 s = "int: {0:d}; hex: {0:#x}; oct: {0:#o}; bin: {0:#b}".format(i)  
 print(s)

Крім того, при створенні рядків буває корисним відформатувати рядок так, щоб знаки на різних рядках були один під одним (додати пробілів), додати заповнення в рядки для того, щоб результат був завжди однієї і тієї самої довжини.

Для таких і подібних завдань в Python вбудована [міні-мова форматування рядків](https://docs.python.org/3/library/string.html#format-specification-mini-language).

Або вивести квадрати та куби чисел до 12 у вигляді таблиці, відцентрувавши значення у стовпцях по 10 символів шириною:

width = 5  
for num in range(12):  
 print('{:^10} {:^10} {:^10}'.format(num, num\*\*2, num\*\*3))

Мета роботи з метамовою форматування полягає у зазначенні у фігурних дужках, яким чином варто перетворити значення перед підстановкою. Форматування працює і з f-рядками, але для більшої читабельності краще скористатися методом format, який приймає як аргументи значення для підстановки в рядок, замість виразів у фігурних дужках.

Самі вирази у фігурних дужках можуть складатися з опису, в якому вигляді варто вивести число (формат запису, кількість знаків після коми та ін.), і зазначення, чим доповнити рядок (додати пробілів на початку або в кінці, щоб рядок був N знаків завдовжки тощо).

### **Модифікатори**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson05/lesson-05#%D0%BC%D0%BE%D0%B4%D0%B8%D1%84%D1%96%D0%BA%D0%B0%D1%82%D0%BE%D1%80%D0%B8)

Міні-мова форматування рядків складається з таких модифікаторів, як:

* ім'я поля — необов'язковий елемент, можемо вказати, яку саме змінну сюди підставити за її ім'ям:

s = "{name} {last\_name}".format(last\_name="Dilan", name="Bob")  
print(s) # Bob Dilan

* перетворення — необов'язковий аргумент, вказується після символу ! і може бути або r або s. Відповідає за те, чи потрібно спробувати перетворити елемент, або відобразити елемент "як є":

s = "{name!r} {last\_name!s}".format(last\_name="Dilan", name="Bob")  
print(s) # 'Bob' Dilan

* специфікація вказується після : і відповідає за те, як відобазити значення.

### **Специфікація**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson05/lesson-05#%D1%81%D0%BF%D0%B5%D1%86%D0%B8%D1%84%D1%96%D0%BA%D0%B0%D1%86%D1%96%D1%8F)

Специфікація – набагато складніший модифікатор. З її допомогою можна:

* змінювати розрядність представлення цілих чисел (десяткові, вісімкові, шістнадцяткові та ін.);

print('dec: {:d} hex: {:x} bin: {:b}'.format(15, 15, 15)) # dec: 15 hex: f bin: 1111

* змінювати точність представлення дробових чисел (округлювати до вказаної кількості знаків);

print('pi: {:0.3}'.format(3.1415)) *# pi: 3.14*

* як відображати знак чисел:

print('"{}" "{:+}" "{:-}" "{: }"'.format(1, 2, -3, 4)) # "1" "+2" "-3" " 4"

* як вирівняти положення елементу і чим (якими символами) доповнити;

print("|{:<10}|{:\*^10}|{:>10}|".format('left', 'center', 'right')) # |left |\*\*center\*\*| right|

Знання про цей інструмент не є критично важливим, але використання форматування рядків в домашній роботі для конструювання "красивішого" результату виконання скрипта, буде плюсом.

### **Регулярні вирази**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson05/lesson-05#%D1%80%D0%B5%D0%B3%D1%83%D0%BB%D1%8F%D1%80%D0%BD%D1%96-%D0%B2%D0%B8%D1%80%D0%B0%D0%B7%D0%B8)

У програмуванні регулярний вираз (від англ. Regular expression, скорочено regex або regexp) — це рядок, який описує деяку множину рядків відповідно до набору спеціальних синтаксичних правил [wiki](https://uk.wikipedia.org/wiki/%D0%A0%D0%B5%D0%B3%D1%83%D0%BB%D1%8F%D1%80%D0%BD%D0%B8%D0%B9_%D0%B2%D0%B8%D1%80%D0%B0%D0%B7" \t "_blank).

Регулярні вирази — це окрема псевдомова програмування, яка широко використовується у багатьох мовах програмування і у Python також.

Детальне вивчення синтаксису регулярних виразів виходить за рамки цього курсу. Кому цікаво, можете продовжити знайомство з регулярними виразами за [посиланням](https://docs.python.org/3/library/re.html) або детальніший опис з прикладами [тут](https://docs.python.org/3/howto/regex.html#regular-expression-howto).

Основне завдання регулярних виразів — це пошук рядка, або підрядка, який відповідає опису в термінах регулярних виразів.

Використовуючи цей механізм можна:

* перевіряти, що рядок відповідає деяким вимогам (це номер телефону або email);
* розділяти рядки на підрядки за деяким виразом (розбити текст на речення, використовуючи усі розділові знаки, а не тільки якийсь один);
* замінювати підрядок в рядку (замінити усі слова, що починаються на деяку послідовність);
* знаходити підрядок в рядку, який відповідає виразу.

Регулярний вираз або коротко "регулярка" складається зі звичайних символів і спеціальних командних послідовностей. Наприклад, \d задає будь-яку цифру, а \d+ — задає будь-яку послідовність з однієї або більше цифр.

[Ось](https://www.programiz.com/python-programming/regex) гарна стаття на тему регулярних виразів, там багато прикладів і корисних посилань.

Для роботи з регулярними виразами у Python є стандартний модуль re. Щоб скористатися цим модулем, його потрібно спочатку імпортувати:

import re

Загальним для усіх функцій модуля re є те, що першим аргументом йде регулярний вираз у вигляді рядка.

### **Функція search[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson05/lesson-05" \l "%D1%84%D1%83%D0%BD%D0%BA%D1%86%D1%96%D1%8F-search" \o "Пряме посилання на Функція search)**

Пошук на відповідність регулярному виразу виконує функція search модуля re. В результаті її виконання повертається спеціальний об'єкт Match або None, якщо нічого не знайшлося.

s = "I am 25 years old"  
age = re.search('\d+', s)  
print(age) # <re.Match object; span=(5, 7), match='25'>

Щоб витягнути власне знайдене значення із age, можна скористатися його методом group

s = "I am 25 years old."  
age = re.search('\d+', s)  
print(age.group()) # 25

Функція search — "ледача" і знаходить тільки першу відповідність заданій умові.

### **Findall**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson05/lesson-05" \l "findall" \o "Пряме посилання на Findall)

Коли потрібно знайти всі відповідні шаблону значення, можна скористатися функцією findall.

s = "I bought 7 nuts for 6$ and 10 bolts for 3$."  
numbers = re.findall('\d+', s)  
print(numbers) # ['7', '6', '10', '3']

findall повертає список усіх відповідностей шаблону.

### **Підстановка, sub**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson05/lesson-05" \l "%D0%BF%D1%96%D0%B4%D1%81%D1%82%D0%B0%D0%BD%D0%BE%D0%B2%D0%BA%D0%B0-sub" \o "Пряме посилання на Підстановка, sub)

Щоб замінити всі підрядки, що відповідають регулярному виразу, можна скористатися функцією sub. Замінимо кольори blue, red, white на слово colour:

p = re.sub(r'(blue|white|red)', 'colour', 'blue socks and red shoes')  
print(p) # colour socks and colour shoes

### **Складання регулярних виразів**[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson05/lesson-05#%D1%81%D0%BA%D0%BB%D0%B0%D0%B4%D0%B0%D0%BD%D0%BD%D1%8F-%D1%80%D0%B5%D0%B3%D1%83%D0%BB%D1%8F%D1%80%D0%BD%D0%B8%D1%85-%D0%B2%D0%B8%D1%80%D0%B0%D0%B7%D1%96%D0%B2)

Складання регулярних виразів — це окрема велика тема варта окремого вивчення. Регулярні вирази складаються з блоків та модифікаторів.

Прикладом блоку може бути:

* [a,b,c,z] у квадратні дужки беруть набір символів, з яких повинен складатися рядок
* . будь-який символ
* \d число або [0-9]
* \D не число, або [^0-9]
* \s будь-який символ, позначає пробіл або табуляцію, перенесення рядка та ін.
* \w будь-яке число або літера, включаючи нижнє підкреслення, або [a-zA-Z0-9\_]
* \W не літера, не число і не нижнє підкреслення

Модифікатори можуть вказувати на кількість повторень блоку у виразі, наприклад:

* ? 0 або 1 раз
* + 1 або більше разів
* \* 0 або більше разів
* {n} суворо n разів (n ціле число)
* {n, m} від n до m разів

Комбінуючи блоки та вирази, можна скласти вирази для ваших потреб:

| **Регулярка** | **Її призначення** |
| --- | --- |
| simple text | В точності текст «simple text» |
| \d{5} | Послідовність із 5 цифр \d означає будь-яку цифру {5} — рівно 5 разів |
| \d\d/\d\d/\d{4} | Дати у форматі ДД/ММ/РРРР (та інші шматки схожі на них, наприклад, 98/76/5432) |
| \b\w{3}\b | Слова в точності з трьох літер \b означає межу слова (з однієї сторони літера, а з другої — ні) \w — будь-яка літера, {3} — рівно три рази |
| [-+]?\d+ | Ціле число, наприклад, 7, +17, -42, 0013 (можливі провідні нулі) [-+]? — або -, або +, або порожньо \d+ — послідовність з 1 або більше цифр |
| `[-+]?(?:\d+(?:.\d\*)? | .\d+)(?:[eE][-+]?\d+)?` |

**Модуль 6: Робота з файлами**

Коли застосунок завершує свою роботу, результат його роботи, який зберігався у змінних в застосунку, буде втрачений. Оперативна пам'ять, яка належала застосунку, буде очищена і результат роботи втрачений. Часто нам необхідно зберігати результат роботи, щоб пізніше можна було до нього повернутися. Найпростіший спосіб зберегти корисну інформацію — це записати її у файл на диск.

Python надає функціонал роботи з будь-якими файлами.

**Файлові дескриптори**

У Python є абстракція над файлами — це вказівник на файл або файловий дескриптор. Нічого складного в цьому немає, це навпаки спрощує роботу з багатьма системними ресурсами.

**Файловий дескриптор** — це системний ресурс, доступ до якого надає операційна система. Зазвичай файловий дескриптор можна відкрити (отримати/створити), закрити (повідомити системі, що робота з ним завершена), можна записати у нього щось і прочитати щось.

**Відкриття та закриття файлів**

Таким чином, робота з файлами у Python починається з відкриття файлу або отримання від системи доступу до файлу, отримання того самого файлового дескриптора. Для цього є вбудована функція open, в яку потрібно обов'язково передати ім'я файлу, який ми хочемо відкрити, і можна вказати, як саме ми хочемо відкрити файл:

fh = open('test\_file.txt')

В цьому прикладі fh — це файловий дескриптор, спеціальний об'єкт, через який ми можемо працювати з файлом.

Після того, як робота з файлом завершена, потрібно повернути ресурс (файл) системі. Для цього у файлового дескриптора потрібно викликати метод close:

fh = open('test.txt')

fh.close()

Закривати файл обов'язково. Незакриті дескриптори можуть стати причиною безлічі неочевидних проблем і складнощів. Найпростіший випадок — це зіпсований файл і повністю втрачена інформація, яка в ньому могла міститися.

Крім того, варто пам'ятати, що оскільки файловий дескриптор — це ресурс, який надається операційною системою, то будь-яке завершення роботи застосунку (аварійне або штатне) не означає закриття усіх відкритих фалів і, знову ж таки, може призвести до помилок.

Якщо не вказати, як ми хочемо відкрити файл, то він відкривається тільки для читання і за допомогою fh можна буде тільки читати з файлу. Якщо файлу з ім'ям test\_file.txt в системі немає, то ви отримаєте виняток.

Режими відкриття файлів в Python вибираються за допомогою другого аргументу функції open.

**Можливі режими відкриття файлів:**

Режим Значення

'r' відкриття на читання (є значенням за замовчуванням).

'w' відкриття на запис, вміст файлу видаляється, якщо файлу не існує, створюється новий.

'x' відкриття на запис, якщо файлу не існує, інакше виняток.

'a' відкриття на дозапис, інформація додається в кінець файлу.

'b' відкриття у двійковому режимі.

't' відкриття в текстовому режимі (є значенням за замовчуванням).

'+' відкриття на читання та запис

Наприклад, щоб відкрити файл для запису або створити новий, якщо його немає, або перезаписати файл, можна вказати значення w:

fh = open('test.txt', 'w')

symbols\_written = fh.write('hello!')

print(symbols\_written) # 6

fh.close()

В цьому прикладі ми відкрили файл test.txt для запису та записали туди рядок 'hello!' завдовжки 6 символів.

**Читання та запис у файл**

Для запису у файл використовується метод write у дескриптора fh. Цей метод повертає кількість записаних у файл символів.

Парний до нього метод — це метод read, який дозволяє прочитати деяку кількість символів із файлу.

fh = open('test.txt', 'w+')

fh.write('hello!')

fh.seek(0)

first\_two\_symbols = fh.read(2)

print(first\_two\_symbols) # 'he'

fh.close()

В цьому прикладі ми відкрили файл для читання та запису. Записали у файл рядок 'hello!' та прочитали перші два символи із файлу за допомогою методу read, вказавши у якості аргументу двійку.

Щоб повернути курсор на початок файлу, викликали метод seek та передали йому позицію, куди потрібно переміститися (0).

Щоб прочитати увесь вміст файлу за раз, можна викликати метод read без аргументів:

fh = open('test.txt', 'w')

fh.write('hello!')

fh.close()

fh = open('test.txt', 'r')

all\_file = fh.read()

print(all\_file) # 'hello!'

fh.close()

Доки файловий дескриптор не закритий, ви можете читати із нього частинами, продовжуючи читання з того самого місця, на якому зупинилися:

fh = open('test.txt', 'w')

fh.write('hello!')

fh.close()

fh = open('test.txt', 'r')

while True:

symbol = fh.read(1)

if len(symbol) == 0:

break

print(symbol)

fh.close()

В цьому прикладі у циклі ми зчитували та виводили у консоль вміст файлу по одному символу за раз. В результаті ви отримаєте у консолі:

h

e

l

l

o

!

Ще є зручний спосіб читати файл порядково, по одному рядку за раз, для цього можна скористатися методом readline:

fh = open('test.txt', 'w')

fh.write('first line\nsecond line\nthird line')

fh.close()

fh = open('test.txt', 'r')

while True:

line = fh.readline()

if not line:

break

print(line)

fh.close()

В консолі буде виведення:

first line

second line

third line

Та аналогічний метод readlines, який читає увесь файл повністю, але повертає список рядків, де елемент списку — це один рядок

fh = open('test.txt', 'w')

fh.write('first line\nsecond line\nthird line')

fh.close()

fh = open('test.txt', 'r')

lines = fh.readlines()

print(lines)

fh.close()

Виведення у консолі буде:

['first line\n', 'second line\n', 'third line']

Зверніть увагу, що всі методи, які читають файли порядково, не опускають (видаляють) символ перенесення рядка.

**Навігація по файлу**

Python дає можливість управляти положенням курсора у файлі, можна довільно переміщатися файлом за допомогою методу seek. Цей метод приймає один аргумент — це кількість символів, на які потрібно змістити курсор у файлі:

fh = open('test.txt', 'w+')

fh.write('hello!')

fh.seek(1)

second = fh.read(1)

print(second) # 'e'

fh.close()

В цьому прикладі після запису у файл курсор зупинений на останньому символі. У виразі fh.seek(1) ми перемістили курсор на другий символ у файлі.

Переміщаючи курсор, можна перезаписувати символи файлу або читати записане.

Метод seek може приймати опціонально другий аргумент, який буде вказувати, звідки потрібно зчитати зміщення. За замовчуванням зміщення задається від початку файлу, але можна задати з кінця або з поточного положення. На жаль, зазначення зміщення з кінця файлу не на всіх платформах підтримується і краще використовувати варіант за замовчуванням.

Щоб дізнатися положення курсора в цей момент, можна скористатися методом tell, він повертає позицію (номер) символу з початку файлу, де зараз знаходиться курсор.

fh = open('test.txt', 'w+')

fh.write('hello!')

position = fh.tell()

print(position) # 6

fh.seek(1)

position = fh.tell()

print(position) # 1

fh.read(2)

position = fh.tell()

print(position) # 3

fh.close()

**Менеджер контексту**

Застосунок може виконати багато операцій між відкриттям та закриттям файлу. В будь-якому місці може статися помилка та застосунок завершиться аварійно, не повернувши файловий дескриптор системі. Така поведінка, як вже згадувалося, небажана і може призводити до втрати даних.

Щоб уникнути цього, можна взяти блок коду, в якому відбувається робота з файлом, у блок try ... except:

fh = open('text.txt')

try:

some\_useful\_function(fh)

finally:

fh.close()

У цьому прикладі ми викликали функцію some\_useful\_function всередині блоку try ... except і, якщо станеться виняток, то обов'язково виконається блок finally, в якому файл буде закритий. Цей підхід гарантує, що файловий дескриптор буде обов'язково повернений системі.

Але такий підхід не надто елегантний та читабельний.

Для покращення читабельності коду при збереженні функціоналу можна скористатися менеджером контексту open. Менеджер контексту — це синтаксична конструкція, яка покращує читабельність коду, але не вносить жодного додаткового функціонала.

with open('text.txt', 'w+') as fh:

some\_useful\_function(fh)

Менеджер контексту складається з ключового слова with, після якого викликається сам менеджер і, якщо щось потрібно повернути з менеджера, то це щось можна передати у змінну, оголошену після ключового слова as. Далі ставиться двокрапка і блок коду, який буде виконаний всередині менеджера. У прикладі з try ... finally — це код, який йде всередині блоку try. Коли код виконається, менеджер контексту виконає те, що повинен зробити в будь-якому випадку, закрити файл, наприклад (це те, що відбувається в блоку finally).

Менеджер контексту open синтаксично повністю повторює свого класичного тезку open, вони повністю ідентичні з точки зору використання.

З точки зору роботи, цей приклад робить у точності те саме, що і попередній з блоком try ... finally. Але замість п'яти рядків коду, ви можете написати два, і код виглядає читабельнішим.

Такий спосіб роботи з файлами є рекомендованим у Python, оскільки гарантує, що програміст не забуде закрити файл у будь-якому разі.

**Байт-рядки, масив байтів**

Є ще один контейнер, з яким ми раніше не працювали. Це bytes — байтові рядки.

Дані у пам'яті комп'ютера зберігаються у вигляді послідовності байтів. Відповідно, будь-які дані можна представити у вигляді послідовності байтів. Для роботи з "сирими" даними у Python є два типи даних: bytes та bytearray.

Дуже багато протоколів досі працюють із "сирими" даними або просто потоком байтів, наприклад TCP/IP, послідовний порт, telnet і багато інших.

Щоб працювати з послідовністю байтів у Python є вбудовані типи даних байт-рядків (bytes) та масиви байтів bytearray.

**bytes**

За своєю суттю байт-рядки або простіше байти — це звичайні рядки, але для запису одного символу використовується суворо один байт.

Байт — це одиниця зберігання та обробки цифрової інформації, що містить 8 біт інформації. Один біт — це 0 або 1. За допомогою одного байта можна записати будь-яке число від 0 до 255 включно.

Для байт-рядків застосовуються ті самі обмеження і правила, що і для звичайних рядків:

байт-рядки — незмінні;

байт-рядки — послідовні і до їх елементів можна звертатися за індексом:

s = b'Hello!'

print(s[1]) # b'e'

байт-рядки підтримують більшість методів рядків, таких як upper, startswith, index, find та інші.

**Створення байт-рядків**

byte\_string = b'Hello world!'

В цьому прикладі у byte\_string міститься послідовність символів по одному байту на кожного. Від оголошення звичайного рядку байт-рядок відрізняє наявність символу b перед самим рядком.

Другий спосіб створення байт рядків — це перетворення у байт-рядок.

Для перетворення рядка у байт-рядок можна скористатися методом рядків encode:

byte\_str = 'some text'.encode()

У byte\_str буде записана послідовність байтів.

В один байт поміщається число від 0 до 255 включно, відповідно можна перетворити послідовність чисел від 0 до 255 у байт-рядок за допомогою функції bytes:

numbers = [0, 128, 255]

byte\_numbers = bytes(numbers)

**Виведення байт-рядків**

Для виведення байтів найзручніше скористатися шістнадцятковим записом, в якому для запису чисел від 0 до 255 достатньо двох символів. Саме такий запис використовує Python "за замовчуванням" для байтів.

some\_numbers = bytes([127, 255, 156])

print(some\_numbers) # b'\x7f\xff\x9c'

Символ \x вказує на шістнадцятковий формат запису. Щоб перевірити правильність представлення, можна скористатися вбудованою функцією hex, яка перетворить ціле число в рядок — представлення числа в шістнадцятковій формі:

for num in [127, 255, 156]:

print(hex(num))

В результаті виконання ви побачите:

0x7f

0xff

0x9c

В такому вигляді перші два символи 0x вказують на шістнадцяткову форму запису .

**Кодування рядків (ASCII, UTF-8, CP1251)**

Перші комп'ютери для роботи з текстом використовували так зване ASCII кодування. У цьому кодуванні для запису одного символу використовується один байт.

Зручність цього кодування в тому, що будь-які дані на комп'ютері можна спробувати представити у вигляді тексту в цьому кодуванні. ASCII містить 256 символів. Це не дуже багато і деякий час цього було достатньо. Але з часом алфавіту з 256 символів стало мало, виникла необхідність додати все більше символів (кирилиця, діакритичні знаки, коди валют, ієрогліфи тощо). Щоб задовольнити потребу у додаванні нових символів, придумали використати кодування, де більше одного байту на символ. Python за замовчуванням використовує UTF-8, в якій один символ може займати від 1 до 4 байт, і всього в алфавіті може бути до 1 112 064 знаків. Це не єдине кодування, на різних платформах можуть бути присутні власні, наприклад CP-1251 (кирилиця на ОС сімейства Windows), UTF-16, UTF-32 та інші.

Щоб дізнатися, якому елементу в UTF-8 відповідає символ, є функція ord (від order).

Наприклад, символ 'a' кодується числом 97:

ord('a') # 97

Зворотна операція, коли потрібно дізнатися, який символ закодований числом, наприклад 100, є функція chr (скорочено від character):

chr(128) # 'd'

Python може працювати з дуже великою кількістю різних кодувань.

s = "Привіт!"

utf8 = s.encode()

print(utf8) # b'\xd0\x9f\xd1\x80\xd0\xb8\xd0\xb2\xd0\xb5\xd1\x82!'

utf16 = s.encode('utf-16')

print(utf16) # b'\xff\xfe\x1f\x04@\x048\x042\x045\x04B\x04!\x00'

s\_from\_utf16 = utf16.decode('utf-16')

print(s\_from\_utf16 == s) # True

Спроба перетворити байт-рядок в неправильному кодуванні, призводить або до помилки, або до досить непередбачуваного результату:

print(b'Hello world!'.decode('utf-16')) # 效汬⁯潷汲Ⅴ

**Масив байтів**

Робота з рядками обмежена тим, що рядки і байт-рядки незмінні. Якщо потрібно замінити навіть один символ, потрібно, по суті, створити копію початкового рядка з єдиним відмінним символом. Щоб зменшити накладні витрати при роботі з "сирими" даними, в Python є такий контейнер як bytearray.

byte\_array = bytearray(b'Kill Bill')

byte\_array[0] = ord('B')

byte\_array[5] = ord('K')

print(byte\_array) # bytearray(b'Bill Kill')

Основна відмінність від байт-рядків — це змінність, щоб змінити масив байтів, не потрібно створювати новий. Друга важлива відмінність — це те, що масив байтів сприймається системою як послідовність чисел від 0 до 255, а не як послідовність символів в ASCII кодуванні. Саме тому не можна написати byte\_array[0] = b'B'. Елементи масиву байтів сприймаються саме як цілі числа.

В іншому ж bytearray може використовуватися як заміна байт-рядків і у нього є ті самі методи з тією самою поведінкою.

**Порівняння рядків**

Порівняння рядків у Python може давати неоднозначний результат внаслідок того, що в UTF-8 кодуванні один і той самий символ можна представити декількома кодами, наприклад, символ 'ê' можна представити кодом U+00EA, або як послідовність двох кодів U+0065 та U+0302. З цієї причини порівняння одного і того самого символу може повернути False через відмінності у записі.

Щоб розв'язати цю проблему при роботі з не ASCII символами для порівняння рядків, їх необхідно нормалізувати за допомогою методу casefold, який повертає рядок, де всі символи у нижньому регістрі і без неоднозначностей, коли будь-який символ матиме тільки одну можливу форму запису.

**Робота з нетекстовими файлами у Python**

Поки що ми розглядали тільки роботу з текстовими фалами в кодуванні UTF-8. Це режим роботи за замовчуванням. Якщо ж потрібно працювати не з текстовими файлами, то можна вказати режим відкриття файлів з b, скорочено від bytes. У такому режимі ви отримаєте файловий дескриптор для роботи з файлом в режимі байт-рядків.

with open('raw\_data.bin', 'wb') as fh:

fh.write(b'Hello world!')

В цьому прикладі ми відкрили файл raw\_data.bin у режимі для запису "сирих" даних, на що вказує значення wb. В цьому режимі можна писати у файл тільки байт-рядки або байт-масиви.

У режимі роботи з "сирими" даними можна відкрити та прочитати вміст будь-якого файлу, в тому числі й архіву.

**Робота з архівами**

Архіви по своїй суті — це ті самі файли, але інформація в них розташована з використанням алгоритмів стискання, які дозволяють записати інформацію в меншому об'ємі.

Ви можете відкрити будь-який архів як файл в режимі роботи з байт-рядками, реалізувати алгоритм стискання і розпаковування на Python, і отримати власний архіватор/деархіватор. Ця досить цікава вправа і ви, звичайно, можете її виконати, якщо хочете глибше в алгоритмах стискання.

Одна з головних причин популярності Python — в наявності великої кількості пакетів та модулів зі всіляким функціоналом, які можна використати для своїх потреб. Є простіший спосіб роботи з популярними безкоштовними архівами у Python — це пакет shutil, який є просунутіший файловий менеджер та вміє працювати з архівами.

shutil підтримує архіви zip, tar, gz. Для цього він використовує пакети zipfile та tarfile. Ви можете використовувати їх напряму, якщо хочете.

Щоб запакувати в архів поточну папку, достатньо викликати функцію make\_archive пакету shutil:

import shutil

archive\_name = shutil.make\_archive('backup', 'zip')

Якщо потрібно запакувати іншу папку, можна вказати шлях до папки третім аргументом:

import shutil

archive\_name = shutil.make\_archive('backup', 'zip', 'some\_folder/inner')

Обідва виклика створять файл backup.zip в поточній робочій папці, а в archive\_name буде рядок з повним шляхом до архіву.

Звичайно пакет shutil підтримує розпаковування архівів. Для цього є функція unpack\_archive, яка розпакує архів у поточну папку або куди вкаже другий аргумент:

import shutil

archive\_name = shutil.make\_archive('backup', 'zip', 'some\_folder/inner')

shutil.unpack\_archive(archive\_name, 'new\_folder\_for\_data')

В цьому прикладі спочатку папка 'some\_folder/inner' була упакована у backup.zip, а потім backup.zip був розпакований у папку 'new\_folder\_for\_data'.

Щоб дізнатися, які формати підтримує пакет і які для них використовуються позначення, можна викликати функцію get\_archive\_formats.

import shutil

for shortcut, description in shutil.get\_archive\_formats():

print('{:<10} | {:<10}'.format(shortcut, description))

У виведенні ви отримаєте таблицю зі скороченням формату архіву та його коротким описом виду:

"""

bztar | bzip2'ed tar-file

gztar | gzip'ed tar-file

tar | uncompressed tar file

xztar | xz'ed tar-file

zip | ZIP file

"""

Основна перевага використання shutil — це зручний інтерфейс, який візьме на себе рекурсивний прохід по усіх вкладених файлах і папках та збереже структуру файлів та папок, як при архівації, так і при розпаковуванні архіву.

## Апрпп

**Зззз**

Оооо