**Типи даних у Python**

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

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/#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/#%D0%B2%D0%B8%D1%80%D0%B0%D0%B7%D0%B8-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/#%D1%82%D0%B8%D0%BF-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/#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/#%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)

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.

https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/img/lesson-01/in.jpg

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/" \l "%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" \o "Пряме посилання на Приведення типів)**

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/);
* 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.

# 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#%D1%83%D0%BC%D0%BE%D0%B2%D0%B8-%D1%83-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#%D1%86%D0%B8%D0%BA%D0%BB-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#%D1%86%D0%B8%D0%BA%D0%BB-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#%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)

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#%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)

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#%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)

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.

# 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#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.

# 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#%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)

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#%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)

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.

**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#%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)

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#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).

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#%D1%84%D1%83%D0%BD%D0%BA%D1%86%D1%96%D1%8F-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#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#%D0%BF%D1%96%D0%B4%D1%81%D1%82%D0%B0%D0%BD%D0%BE%D0%B2%D0%BA%D0%B0-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+)?` |

# 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[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson06/lesson-06#%D1%84%D0%B0%D0%B9%D0%BB%D0%BE%D0%B2%D1%96-%D0%B4%D0%B5%D1%81%D0%BA%D1%80%D0%B8%D0%BF%D1%82%D0%BE%D1%80%D0%B8)

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[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson06/lesson-06#%D0%B2%D1%96%D0%B4%D0%BA%D1%80%D0%B8%D1%82%D1%82%D1%8F-%D1%82%D0%B0-%D0%B7%D0%B0%D0%BA%D1%80%D0%B8%D1%82%D1%82%D1%8F-%D1%84%D0%B0%D0%B9%D0%BB%D1%96%D0%B2)

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[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson06/lesson-06#%D1%87%D0%B8%D1%82%D0%B0%D0%BD%D0%BD%D1%8F-%D1%82%D0%B0-%D0%B7%D0%B0%D0%BF%D0%B8%D1%81-%D1%83-%D1%84%D0%B0%D0%B9%D0%BB)

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[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson06/lesson-06#%D0%BD%D0%B0%D0%B2%D1%96%D0%B3%D0%B0%D1%86%D1%96%D1%8F-%D0%BF%D0%BE-%D1%84%D0%B0%D0%B9%D0%BB%D1%83)

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[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson06/lesson-06#%D0%BC%D0%B5%D0%BD%D0%B5%D0%B4%D0%B6%D0%B5%D1%80-%D0%BA%D0%BE%D0%BD%D1%82%D0%B5%D0%BA%D1%81%D1%82%D1%83)

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[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson06/lesson-06#%D0%B1%D0%B0%D0%B9%D1%82-%D1%80%D1%8F%D0%B4%D0%BA%D0%B8-%D0%BC%D0%B0%D1%81%D0%B8%D0%B2-%D0%B1%D0%B0%D0%B9%D1%82%D1%96%D0%B2)

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[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson06/lesson-06#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[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson06/lesson-06#%D1%81%D1%82%D0%B2%D0%BE%D1%80%D0%B5%D0%BD%D0%BD%D1%8F-%D0%B1%D0%B0%D0%B9%D1%82-%D1%80%D1%8F%D0%B4%D0%BA%D1%96%D0%B2)

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[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson06/lesson-06#%D0%B2%D0%B8%D0%B2%D0%B5%D0%B4%D0%B5%D0%BD%D0%BD%D1%8F-%D0%B1%D0%B0%D0%B9%D1%82-%D1%80%D1%8F%D0%B4%D0%BA%D1%96%D0%B2)

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)[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson06/lesson-06#%D0%BA%D0%BE%D0%B4%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F-%D1%80%D1%8F%D0%B4%D0%BA%D1%96%D0%B2-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[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson06/lesson-06#%D0%BC%D0%B0%D1%81%D0%B8%D0%B2-%D0%B1%D0%B0%D0%B9%D1%82%D1%96%D0%B2)

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[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson06/lesson-06#%D0%BF%D0%BE%D1%80%D1%96%D0%B2%D0%BD%D1%8F%D0%BD%D0%BD%D1%8F-%D1%80%D1%8F%D0%B4%D0%BA%D1%96%D0%B2)

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[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson06/lesson-06#%D1%80%D0%BE%D0%B1%D0%BE%D1%82%D0%B0-%D0%B7-%D0%BD%D0%B5%D1%82%D0%B5%D0%BA%D1%81%D1%82%D0%BE%D0%B2%D0%B8%D0%BC%D0%B8-%D1%84%D0%B0%D0%B9%D0%BB%D0%B0%D0%BC%D0%B8-%D1%83-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[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson06/lesson-06#%D1%80%D0%BE%D0%B1%D0%BE%D1%82%D0%B0-%D0%B7-%D0%B0%D1%80%D1%85%D1%96%D0%B2%D0%B0%D0%BC%D0%B8)

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 розібратися 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 представляє 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')

Обидва 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.

**Створення та встановлення власних пакетів, віртуальне оточення**

Короткі та прості скрипти поміщаються в одному не дуже великому файлі до 400 рядків. Такі скрипти не складно підтримувати і розвивати. Але рано чи пізно настає момент, коли коду стає надто багато. Розмір єдиного файлу робить навігацію файлом складною і пошук у файлі потрібного місця займає надто багато часу. У такій ситуації зручно буде розбити великий скрипт на складові частини і структурувати ваш код. Це зробить роботу з проектом простішою і зручнішою.

Хороша структура проекту значно спрощує підтримку і розробку. Погана ж — навпаки істотно уповільнює роботу. І що більший проект, то це помітніше. Не лінуйтеся приділити час на продумування структури вашого проекту при кожній значній зміні. Ваші зусилля не пройдуть даремно і обов'язково окупляться в майбутньому.

**Створення пакетів та модулів**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson07/lesson-07#%D1%81%D1%82%D0%B2%D0%BE%D1%80%D0%B5%D0%BD%D0%BD%D1%8F-%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)

Наприклад, усі функції, що мають загальне призначення, можна винести в окремий файл. "Точку входу" (скрипт, який викликається користувачем для початку роботи), допоміжні функції і налаштування в окремі файли, названі відповідно. В результаті ми отримаємо декілька відносно невеликих файлів, код в яких буде призначений для якогось одного завдання (налаштування застосунку, запуск застосунку, основна логіка). Такі файли з кодом у Python називаються модулями.

Ви можете імпортувати код (функції, змінні, що завгодно) із модулей за допомогою оператора виразу from ... import ....

Припустимо у вас є файл salary\_calculations.py і його вміст виглядає так:

def add\_bonus(salary, bonus\_percents):  
 return salary \* (1 + bonus\_percents / 100)

У тій самій папці лежить основний скрипт main.py, у якому ви б хотіли використати функцію add\_bonus з salary\_calculations.py. Щоб зробити це, нам достатньо імпортувати цю функцію:

from salary\_calculations import add\_bonus  
  
  
salary = 1000  
bonus = 15  
salary\_with\_bonus = add\_bonus(salary, bonus)  
print(salary\_with\_bonus) *# 1150*

Ви також можете імпортувати увесь модуль за допомогою оператора import і тоді у вас буде доступ до усього вмісту модуля.

import salary\_calculations  
  
  
salary = 1000  
bonus = 15  
salary\_with\_bonus = salary\_calculations.add\_bonus(salary, bonus)  
print(salary\_with\_bonus) *# 1150*

Щоб викликати функцію add\_bonus з salary\_calculations.py, коли ви імпортували увесь модуль, ви вказуєте ім'я модуля та ім'я функції через крапку.

Окремі файли з розширенням .py — це модулі. Ви можете імпортувати з них потрібні вам сутності (об'єкти, константи тощо) і використати в інших модулях. Зверніть увагу, що під час імпорту не потрібно вказувати розширення файлу .py, тільки ім'я.

Зі зростанням проекту кількість модулів (нагадаємо, що рекомендується не давати модулям занадто "розпухати") росте і вже пошук потрібного модуля починає викликати труднощі. У такій ситуації на допомогу приходять пакети Python.

Пакет — це папка, що містить модулі Python. Розташовуючи модулі по папкам, ви можете структурувати модулі за призначенням, і пошук потрібного модуля стане набагато простішим.

Робота з пакетами аналогічна роботі з модулями, ви можете імпортувати потрібну вам сутність (наприклад функцію), імпортувавши модуль з пакету. Припустимо, що модуль salary\_calculations.py ви розмістили у папці calculations поряд з main.py, де вам знадобилася функція add\_bonus з salary\_calculations.py. В такому випадку ви можете імпортувати модуль salary\_calculations.py:

from calculations import salary\_calculations  
  
  
salary = 1000  
bonus = 15  
salary\_with\_bonus = salary\_calculations.add\_bonus(salary, bonus)  
print(salary\_with\_bonus) *# 1150*

або тільки функцію:

from calculations.salary\_calculations import add\_bonus  
  
  
salary = 1000  
bonus = 15  
salary\_with\_bonus = add\_bonus(salary, bonus)  
print(salary\_with\_bonus) *# 1150*

Розбиття на пакети і модулі може тривати стільки, скільки ви вважаєте за необхідне і зручне. Будь-який пакет може містити пакети в собі, а ті — ще пакети, і так далі. Основне правило — це називати пакети та модулі так само, як і змінні Python (тільки літери, цифри та \_, ім'я не починається з цифри). Також дуже важливо продумати структуру проекту так, щоб вам було зручно потім шукати у ньому потрібні модулі.

**Навіщо потрібен init.py**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson07/lesson-07#%D0%BD%D0%B0%D0%B2%D1%96%D1%89%D0%BE-%D0%BF%D0%BE%D1%82%D1%80%D1%96%D0%B1%D0%B5%D0%BD-initpy)

У версіях Python до 3.3 в пакетах обов'язково потрібно було розмістити допоміжний файл \_\_init\_\_.py. Якщо цього не зробити, то Python не сприймав папку як пакет та імпортувати з такої папки нічого не міг. Зараз в цьому немає потреби, але часто такі файли створюються для зворотної сумісності зі старими версіями.

Файл \_\_init\_\_.py — це службовий файл, який інтерпретатор обов'язково виконає під час першого імпорту пакету. Таким чином, якщо вам потрібно виконати якісь дії під час імпорту пакету, ви можете прописати їх у \_\_init\_\_.py.

Зазвичай \_\_init\_\_.py — порожній і нічого не робить. Але, коли структура пакету не занадто проста і там багато модулів та/або пакетів, про які користувачеві знати не обов'язково, ви можете імпортувати те, що користувачеві потрібно у \_\_init\_\_.py. У такому випадку користувач зможе вже у своєму коді прописати скорочені варіанти імпортів. Наприклад, у пакеті utility є два пакети: useful та dummy. В кожному з них є модуль functions.py (у кожного свій). А в цих модулях вже є функції nice\_function та not\_bad відповідно. Користувачеві пакету utility необов'язково знати про внутрішню структуру пакету, вона зроблена для зручності розробника пакету. Розробник написав utility так, щоб надати користувачеві доступ до nice\_function та not\_bad.

Якщо файл \_\_init\_\_.py порожній, то використання nice\_function та not\_bad буде виглядати якось так:

import utility  
  
utility.useful.functions.nice\_function()  
utility.dummy.functions.not\_bad()

або:

from utility.useful.functions import nice\_function  
from utility.dummy.functions import not\_bad  
  
nice\_function()  
not\_bad()

Обидва варіанти припускають, що користувачеві пакету потрібно буде розібратися, де там що лежить.

Якщо ж розробник подумав про користувача і \_\_init\_\_.py виглядає ось так:

from utility.useful.functions import nice\_function  
from utility.dummy.functions import not\_bad  
  
\_\_all\_\_ = ['nice\_function', 'not\_bad']

Тоді можна скористатися таким імпортом:

from utility import nice\_function, not\_bad  
  
nice\_function()  
not\_bad()

або:

from utility import \*  
  
nice\_function()  
not\_bad()

Зверніть увагу на константу \_\_all\_\_ — це список модулів або пакетів, які імпортуються, якщо у виразі from ... import \* в кінці вказаний символ \*.

**Встановлення пакетів за допомогою pip**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson07/lesson-07#%D0%B2%D1%81%D1%82%D0%B0%D0%BD%D0%BE%D0%B2%D0%BB%D0%B5%D0%BD%D0%BD%D1%8F-%D0%BF%D0%B0%D0%BA%D0%B5%D1%82%D1%96%D0%B2-%D0%B7%D0%B0-%D0%B4%D0%BE%D0%BF%D0%BE%D0%BC%D0%BE%D0%B3%D0%BE%D1%8E-pip)

Встановлення додаткових пакетів в Python здійснюється менеджерами пакетів. Стандартний (але не єдиний) менеджер пакетів Python — це pip. pip може:

* встановлювати пакети <https://pypi.org/>
* встановлювати пакети з локального репозиторію або розташованого на будь-якому ресурсі, що підтримує протокол pip.
* видаляти пакети
* показувати список встановлених доступних пакетів (з версіями або без)
* показувати, які пакети застаріли (зі встановлених) і їх можна оновити
* видаляти пакети

pip написаний на Python і є пакетом Python, який входить в стандартне постачання (попередньо встановлений). Ви можете, звичайно, окремо його встановити або оновити як будь-який інший пакет, якщо потрібно.

Використання pip передбачає виконання його як скрипту Python з аргументами під час виклику. Наприклад, щоб вивести список встановлених пакетів:

python3 -m pip list

Значення -m вказує, що потрібно викликати пакет як скрипт, що виконується, list — це аргумент, за яким pip "розуміє", що потрібно зробити (вивести список встановлених пакетів).

Часто в системі прописано, що pip запускається за допомогою Python і ви можете викликати pip напряму:

pip list

Повна документація з pip доступна на [сторінці офіційної документації](https://pip.pypa.io/en/stable/).

* встановлення останньої версії пакету some\_package: pip install some\_package
* встановлення версії 1.12.3 пакету some\_package: pip install some\_package==1.12.3
* встановлення версії пакету some\_package новішого за 1.12.3: pip install some\_package>=1.12.3
* встановлення версії пакету some\_package давнішого за 1.12.3: pip install some\_package<=1.12.3
* видалення пакету some\_package: pip uninstall some\_package
* список встановлених пакетів: pip list
* список встановлених пакетів з версіями: pip freeze
* встановлення пакетів з текстового файлу requirements.txt: pip install -r requirements.txt

**Створення віртуального оточення**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson07/lesson-07#%D1%81%D1%82%D0%B2%D0%BE%D1%80%D0%B5%D0%BD%D0%BD%D1%8F-%D0%B2%D1%96%D1%80%D1%82%D1%83%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B3%D0%BE-%D0%BE%D1%82%D0%BE%D1%87%D0%B5%D0%BD%D0%BD%D1%8F)

Встановлювати пакети Python в систему глобально — не найкраща ідея, оскільки встановлені пакети захаращують систему і можуть конфліктувати. Конфлікти встановлених пакетів — це одна з проблем, що найважче виправляються, при роботі з Python. "Поламані" залежності можуть забрати багато часу навіть у дуже досвідченого розробника. Тим паче складніше впоратися з цим новачкам.

Щоб уникнути проблем із залежностями, і встановлення потрібних для якогось проекту пакетів не впливала на всю вашу операційну систему та інші проекти, рекомендується використати віртуальне оточення Python.

Оскільки пакет у Python — це папки з файлами-модулями, ми можемо створити окрему папку для пакетів-залежностей всередині кожного проекту. Якщо вказати Python, що працюючи в папці проекту, імпортувати потрібно саме з цієї папки, то тоді ми зможемо ізолювати нашу операційну систему і наші проекти один від одного. Така папка з пакетами-залежностями — це і є вуртуальне оточення.

Звичайно, існує безліч інструментів, які автоматизують для нас створення/видалення/активування віртуального оточення. Найпростіший і найстандартніший інструмент — це venv.

Почитати, що вміє venv, можна на [сторінці офіційної документації](https://docs.python.org/3/library/venv.html). Так само як і pip, venv — це консольний Python скрипт.

Щоб створити нове віртуальне оточення, виконайте: python3 -m venv /path/to/new/virtual/environment або venv /path/to/new/virtual/environment. Після виконання цієї команди у /path/to/new/virtual/environment буде скопійований Python з мінімальним набором стандартних пакетів. І ви зможете використати цей Python окремо від основного системного або будь-якого іншого.

Щоб почати використовувати Python з віртуального оточення, виконайте в консолі: \path\to\new\virtual\environment\Scripts\activate.bat або source /path/to/new/virtual/environment/bin/activate для Linux / Mac OS. Якщо ви використовуєте PowerShell, то можна виконати \path\to\new\virtual\environment\Scripts\Activate.ps1.

Після активації в цій консолі виклик python буде викликати не системний Python, а версію з /path/to/new/virtual/environment зі своїми пакетами.

Активувавши віртуальне оточення, ви можете встановлювати/видаляти пакети за допомогою pip у віртуальному оточенні. Встановлення/видалення пакетів впливає тільки на віртуальне оточення і не зачіпає систему або інші проекти.

Щоб повернутися до системного Python, виконайте: deactivate.

Щоб повністю видалити віртуальне оточення, достатньо видалити папку \path\to\new\virtual\environment\ з усім її вмістом.

**Створення встановлюваних пакетів, структура setup.py**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson07/lesson-07#%D1%81%D1%82%D0%B2%D0%BE%D1%80%D0%B5%D0%BD%D0%BD%D1%8F-%D0%B2%D1%81%D1%82%D0%B0%D0%BD%D0%BE%D0%B2%D0%BB%D1%8E%D0%B2%D0%B0%D0%BD%D0%B8%D1%85-%D0%BF%D0%B0%D0%BA%D0%B5%D1%82%D1%96%D0%B2-%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0-setuppy)

Кінцевим результатом роботи програміста є застосунок. Щоб користувач міг встановити і використати ваш застосунок, ви можете зробити встановлюваний пакет з вашого застосунку.

Щоб створити пакет, вам потрібно дотримуватися ряду вимог.

**Структура пакету:**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson07/lesson-07#%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0-%D0%BF%D0%B0%D0%BA%D0%B5%D1%82%D1%83)

Припустимо у вас є пакет useful. У цьому пакеті є модулі some\_code.py та another\_code.py. У цих модулях знаходиться код, який виконує корисну роботу.

Щоб ваш пакет можна було встановити, вам варто розмістити сам пакет всередині папки, яка називається так само:

useful/  
 useful/  
 \_\_init\_\_.py  
 some\_code.py  
 another\_code.py  
 setup.py

Поряд з самим пакетом варто розмістити модуль з інструкціями зі встановлення — setup.py. До вмісту setup.py ми повернемося трохи пізніше, а поки що продовжимо розгляд структури пакету.

Python за замовчуванням проігнорує під час встановлення усі файли, які не закінчуються на .py Якщо у вашому пакеті не буде \_\_init\_\_.py, то вміст useful/useful/ теж буде проігноровано (сподіваюся це виправлять в наступних версіях). Назва пакету повинна відповідати вимогам до імен змінних Python.

**setup.py**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson07/lesson-07#setuppy)

Модуль, що містить інструкції зі встановлення, викликає функцію setup з пакету setuptools. Функція setup здійснює встановлення пакету в системі і містить параметри, що конфігурують встановлення.

Детальні інструкції з написання setup.py ви можете отримати на [сторінці документації](https://docs.python.org/3/distutils/setupscript.html).

Приклад вмісту setup.py:

from setuptools import setup  
  
setup(name='useful',  
 version='1',  
 description='Very useful code',  
 url='http://github.com/dummy\_user/useful',  
 author='Flying Circus',  
 author\_email='flyingcircus@example.com',  
 license='MIT',  
 packages=['useful'])

В цьому прикладі ми викликаємо setup з додатковими інформаційними параметрами, які будуть доступні користувачам. А саме, ми вказали ім'я пакету, версію, короткий опис пакету, адресу, де можна подивитися початковий код, ім'я автора, його email, ліцензію, набір пакетів, які містяться у постачанні.

Що, якщо наш пакет досить великий і прописувати вручну усі модулі packages незручно, та існує ризик помилитися? Тоді у setuptools є функція find\_namespace\_packages, яка допоможе знайти всі модулі і не пропустити нічого:

from setuptools import setup, find\_namespace\_packages  
  
setup(  
 name='useful',  
 version='1',  
 description='Very useful code',  
 url='http://github.com/dummy\_user/useful',  
 author='Flying Circus',  
 author\_email='flyingcircus@example.com',  
 license='MIT',  
 packages=find\_namespace\_packages()  
)

Такий пакет можна опублікувати на PyPi і тоді його можна буде встановити за допомогою pip, або опублікувати початковий код, і тоді можна буде встановити з початкових кодів.

Щоб встановити цей пакет з початкового коду, виконайте в консолі pip install . або pip install -e . у папці, де лежить setup.py.

**Додавання не py файлів у пакет**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson07/lesson-07#%D0%B4%D0%BE%D0%B4%D0%B0%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F-%D0%BD%D0%B5-py-%D1%84%D0%B0%D0%B9%D0%BB%D1%96%D0%B2-%D1%83-%D0%BF%D0%B0%D0%BA%D0%B5%D1%82)

Коли вам потрібно додати до пакету на .py файл (зображення, README.md тощо) варто пам'ятати, що setuptools проігнорує всі не .py файли. Потрібно явно вказати додати файли в пакет, якщо вам це потрібно. Один зі способів додати файли в постачання — це вказати параметр include\_package\_data=True у виклику setup та прописати, які файли потрібно додати у постачання у MANIFEST.in.

MANIFEST.in — це файл поряд з setup.py, і в ньому вказуються шляхи до всіх файлів, які потрібно додати у постачання. Приклад MANIFEST.in для додавання файлу README.md:

include README.md

І структура пакету буде:

useful/  
 useful/  
 \_\_init\_\_.py  
 some\_code.py  
 another\_code.py  
 setup.py  
 MANIFEST.in  
 README.md

Тепер файл README.md буде доданий до пакету і буде доступний. Шлях до файлів, які потрібно включити у постачання, може бути будь-яким відносно пакету.

**Управління залежностями пакету**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson07/lesson-07#%D1%83%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D1%96%D0%BD%D0%BD%D1%8F-%D0%B7%D0%B0%D0%BB%D0%B5%D0%B6%D0%BD%D0%BE%D1%81%D1%82%D1%8F%D0%BC%D0%B8-%D0%BF%D0%B0%D0%BA%D0%B5%D1%82%D1%83)

Якщо в нашому пакеті є залежності, щоб він запрацював, потрібно встановити додаткові пакети, потрібно їх всі прописати в параметрі install\_requires:

from setuptools import setup, find\_namespace\_packages  
  
setup(  
 name='useful',  
 version='1',  
 description='Very useful code',  
 url='http://github.com/dummy\_user/useful',  
 author='Flying Circus',  
 author\_email='flyingcircus@example.com',  
 license='MIT',  
 packages=find\_namespace\_packages(),  
 install\_requires=['markdown'],  
)

В цьому прикладі наш пакет буде вимагати встановити спочатку пакет markdown перед встановленням. Порядок встановлення залежностей визначає сам менеджер пакетів (pip наприклад).

**Консольний скрипт як пакет**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson07/lesson-07#%D0%BA%D0%BE%D0%BD%D1%81%D0%BE%D0%BB%D1%8C%D0%BD%D0%B8%D0%B9-%D1%81%D0%BA%D1%80%D0%B8%D0%BF%D1%82-%D1%8F%D0%BA-%D0%BF%D0%B0%D0%BA%D0%B5%D1%82)

Якщо наш пакет містить застосунок, який можна викликати з консолі, зручно буде додати можливість виклику цього застосунку у будь-якому місці нашої системи з консолі. Для цього у виклику setup додамо ще один параметр — entry\_points. Цей параметр приймає словник, де ми можемо вказати список "точок входу" для ключа console\_scripts.

Наприклад, в нашому пакеті у модулі some\_code.py є функція hello\_world, яка виводить у консоль повідомлення Hello World!. Після встановлення пакету ми зможемо в будь-якому місці нашої системи виконати в консолі команду: helloworld і отримаємо у відповідь Hello World!.

Щоб це працювало в системі, Python повинен викликатися при виклику файлів з розширенням .py та setup.py повинен бути змінений:

from setuptools import setup, find\_namespace\_packages  
  
setup(  
 name='useful',  
 version='1',  
 description='Very useful code',  
 url='http://github.com/dummy\_user/useful',  
 author='Flying Circus',  
 author\_email='flyingcircus@example.com',  
 license='MIT',  
 packages=find\_namespace\_packages(),  
 install\_requires=['markdown'],  
 entry\_points={'console\_scripts': ['helloworld = useful.some\_code:hello\_world']}  
)

У списку точок входу console\_scripts можуть бути файли, що виконуються (.exe), скрипти Bash, cmd, PowerShell і будь-який інший файл, який операційна система зможе виконати.

**Можливості деяких вбудованих пакетів Python**

**Робота з датою і часом (datetime)**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson08/lesson-08#%D1%80%D0%BE%D0%B1%D0%BE%D1%82%D0%B0-%D0%B7-%D0%B4%D0%B0%D1%82%D0%BE%D1%8E-%D1%96-%D1%87%D0%B0%D1%81%D0%BE%D0%BC-datetime)

Робота з датою і часом у Python реалізована у пакеті datetime. Основні можливості datetime:

* визначення поточної дати і часу;
* обчислення інтервалу між двома подіями;
* визначення дня тижня, високосного року для будь-якої дати у минулому не раніше року datetime.MINYEAR або в майбутньому не пізніше року datetime.MAXYEAR;
* порівняння дати і часу декількох подій за допомогою операторів порівняння;
* робота з часовими зонами, порівняння подій з урахуванням часових зон та переходу на літній/зимовий час;
* перетворення дати/часу в рядок і навпаки.

**datetime**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson08/lesson-08#datetime)

Щоб отримати поточну дату і час без урахування часового пояса, можна викликати метод now() у datetime:

from datetime import datetime  
  
current\_datetime = datetime.now()  
print(current\_datetime) *# 2020-10-09 22:13:35.053819*

У результаті виклику now() ми отримуємо об'єкт datetime, у якого є ряд корисних атрибутів:

from datetime import datetime  
  
current\_datetime = datetime.now()  
  
print(current\_datetime.year) *# 2020*  
print(current\_datetime.month) *# 10*  
print(current\_datetime.day) *# 09*  
print(current\_datetime.hour) *# 22*  
print(current\_datetime.minute) *# 32*  
print(current\_datetime.second) *# 22*  
print(current\_datetime.microsecond) *# 819366*

В об'єкта datetime є методи, щоб отримати дату (без часу) та час (без дати):

from datetime import datetime  
  
current\_datetime = datetime.now()  
print(current\_datetime.date()) *# 2020-10-09*  
print(current\_datetime.time()) *# 22:13:35.053819*

Щоб створити об'єкт datetime з будь-якою вибраною датою, можна зробити так:

from datetime import datetime  
  
d1 = datetime(year=2012, month=1, day=7, hour=14)  
print(d1) *# 2012-01-07 14:00:00*

Щоб дізнатися день тижня, можна скористатися методом weekday:

from datetime import datetime  
  
seventh\_day\_2020 = datetime(year=2020, month=1, day=7, hour=14)  
print(seventh\_day\_2020.weekday()) *# 1*

Дні тижня у Python починаються з понеділка і він буде 0. У прикладі вище 7 Січня 2020 року було вівторком.

Щоб порівняти два об'єкти datetime, достатньо скористатися оператором порівняння:

from datetime import datetime  
  
current\_datetime = datetime.now()  
  
future\_month = (current\_datetime.month % 12) + 1  
future\_year = current\_datetime.year + int(current\_datetime.month / 12)  
future\_datetime = datetime(future\_year, future\_month, 1)  
  
print(current\_datetime < future\_datetime) *# True*

**timedelta**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson08/lesson-08#timedelta)

Якщо відняти від одного datetime об'єкту інший, то отримаємо timedelta об'єкт. Він відповідає за відрізок часу між двома датами.

from datetime import datetime  
  
seventh\_day\_2019 = datetime(year=2019, month=1, day=7, hour=14)  
seventh\_day\_2020 = datetime(year=2020, month=1, day=7, hour=14)  
  
difference = seventh\_day\_2020 - seventh\_day\_2019  
print(difference) *# 365 days, 0:00:00*  
print(difference.total\_seconds()) *# 31536000.0*

Об'єкти timedelta можна створювати самостійно, щоб отримати дату/час віддалену від початкової:

from datetime import datetime, timedelta  
  
seventh\_day\_2020 = datetime(year=2020, month=1, day=7, hour=14)  
four\_weeks\_interval = timedelta(weeks=4)  
  
print(seventh\_day\_2020 + four\_weeks\_interval) *# 2020-02-04 14:00:00*  
print(seventh\_day\_2020 - four\_weeks\_interval) *# 2019-12-10 14:00:00*

Об'єкт timedelta можна створити, задаючи тижні, дні, години, хвилини, секунди, мілісекунди і мікросекунди:

from datetime import timedelta  
delta = timedelta(  
 days=50,  
 seconds=27,  
 microseconds=10,  
 milliseconds=29000,  
 minutes=5,  
 hours=8,  
 weeks=2  
)

Якщо якийсь параметр не заданий, то він дорівнює 0 за замовчуванням.

Робота з часовими поясами припускає знання класів у Python і ми розглянемо її пізніше.

**timestamp**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson08/lesson-08#timestamp)

Окремо потрібно сказати про timestamp. timestamp — це кількість секунд, що пройшло з 00 годин 00 хвилин 1 Січня 1970 року в UTC (часовий пояс Гринвіча). Це просто прийнята константа і нічого особливого вона не означає. Просто для зручності колись почали відраховувати час в секундах з цієї миті і це виявилося дуже зручно. Адже порівняти два числа завжди легше і швидше, ніж порівняти складну структуру дат і часів.

Звичайно можна з timestamp отримати дату/час і навпаки:

from datetime import datetime  
  
seventh\_day\_2020 = datetime(year=2020, month=1, day=7, hour=14)  
ts = seventh\_day\_2020.timestamp()  
print(ts) *# 1578398400.0*  
  
ts += 100\_000  
print(datetime.fromtimestamp(ts)) *# 2020-01-08 17:46:40*

**Перетворення в рядок і з рядка**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson08/lesson-08#%D0%BF%D0%B5%D1%80%D0%B5%D1%82%D0%B2%D0%BE%D1%80%D0%B5%D0%BD%D0%BD%D1%8F-%D0%B2-%D1%80%D1%8F%D0%B4%D0%BE%D0%BA-%D1%96-%D0%B7-%D1%80%D1%8F%D0%B4%D0%BA%D0%B0)

Коли потрібно перетворити дату/час в рядок, ви можете скористатися функцією str, яка перетворить datetime у рядок. Але часто формат такого перетворення незручний і в Python є окрема мінімова для опису, як перетворити дату/час в рядок:

from datetime import datetime  
  
seventh\_day\_2020 = datetime(year=2020, month=1, day=7, hour=14)  
print(seventh\_day\_2020.strftime('%A %d %B %Y')) *# Tuesday 07 January 2020*

Та ж міні-мова використовується для конвертації вже рядків в дату/час:

from datetime import datetime  
  
s = '10 January 2020'  
print(datetime.strptime(s, '%d %B %Y')) *# 2020-01-10 00:00:00*

Деталі можна вивчити на [сторінці офіційної документації](https://docs.python.org/3/library/datetime.html#strftime-and-strptime-format-codes).

**Генерація випадкових чисел**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson08/lesson-08#%D0%B3%D0%B5%D0%BD%D0%B5%D1%80%D0%B0%D1%86%D1%96%D1%8F-%D0%B2%D0%B8%D0%BF%D0%B0%D0%B4%D0%BA%D0%BE%D0%B2%D0%B8%D1%85-%D1%87%D0%B8%D1%81%D0%B5%D0%BB)

Для генерації випадкових (псевдовипадкових) чисел у Python є пакет random. Він досить хороший для ряду побутових завдань, але не для криптографії. На жаль, вбудований генератор псевдовипадкових чисел досить скоро починає повторюватися і не є достатньо криптостійким. Проте, для прикладних завдань поза сферою криптографії його цілком вистачає.

Отримання випадкового цілого числа з рівномірного розподілу в інтервалі між 1 та 1000 включно:

import random  
  
random.randint(1, 1000)

Якщо потрібно отримати випадкове число в інтервалі 0, 1 включно:

import random  
  
random.random()

Коли у вас є список об'єктів і вам потрібно перемішати порядок елементів в цьому списку на випадковий:

import random  
  
fruits = ['apple', 'banana', 'orange']  
random.shuffle(fruits)  
print(fruits) *# ['banana', 'orange', 'apple']*

Якщо потрібно вибрати випадковий елемент зі списку:

import random  
  
fruits = ['apple', 'banana', 'orange']  
print(random.choice(fruits)) *# 'banana'*

Щоб вибрати N випадкових елементів зі списку:

import random  
  
fruits = ['apple', 'banana', 'orange']  
print(random.choices(fruits, k=4)) *# ['banana', 'orange', 'orange', 'orange']*

Щоб вибрати N елементів, що не повторюються, зі списку:

import random  
  
fruits = ['apple', 'banana', 'orange']  
print(random.sample(fruits, k=2)) *# ['banana', 'orange']*

Зверніть увагу, що k не може бути більше довжини fruits.

Якщо вам потрібно скористатися відмінним від рівномірного розподілу, можете вибрати один із [підтримуваних розподілів](https://docs.python.org/3/library/random.html#real-valued-distributions).

**Контроль точності обчислень (decimal)**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson08/lesson-08#%D0%BA%D0%BE%D0%BD%D1%82%D1%80%D0%BE%D0%BB%D1%8C-%D1%82%D0%BE%D1%87%D0%BD%D0%BE%D1%81%D1%82%D1%96-%D0%BE%D0%B1%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D1%8C-decimal)

Комп'ютер усі обчислення робить в бінарному вигляді, а десяткові числа використовуються тільки для "спілкування" з користувачем для зручності останнього. Через це і через те, що точність обчислень в комп'ютері обмежена, виникають помилки округлення під час виконання математичних операцій.

print(0.1 + 0.2 == 0.3) *# False*  
print(0.1 + 0.2) *# 0.30000000000000004*

Перший вираз може збити вас з пантелику, оскільки математика стверджує однозначно, що 0.1 + 0.2 = 0.3. Але помилка округлення під час виконання обчислювальних операцій з дійсними числами у двійковій системі обчислення призводить до такої неоднозначності.

Проблема точності обчислень так і не розв'язана остаточно, і математика продовжує розвиватися в цьому напрямі.

Щоб контролювати точність обчислень більш явно, у Python є пакет decimal.

from decimal import Decimal, getcontext  
  
  
getcontext().prec = 6  
Decimal(1) / Decimal(7) *# Decimal('0.142857')*  
  
getcontext().prec = 28  
Decimal(1) / Decimal(7) *# Decimal('0.1428571428571428571428571429')*

У цьому прикладі ми вирахували вираз 1 / 7 з точністю до 6 знаків після коми і до 28 знаків. Щоб встановити точність обчислення, ми скористалися функцією getcontext, яка повертає поточні налаштування точності, та встановили налаштування prec у 6 та 28 відповідно.

Об'єкти Decimal поводяться так само, як float, але їх і не можна використовувати в одному виразі разом. Виконання виразу на кшталт Decimal(0.1) + 0.2 призведе до помилки.

Повертаючись до нашого прикладу із додаванням 0.1 та 0.2:

from decimal import Decimal, getcontext  
  
  
getcontext().prec = 6  
float(Decimal(0.1) + Decimal(0.2)) == 0.3 *# True*

На жаль, Decimal має ще ту особливість, що при створенні Decimal із float його точність береться максимальною для цієї платформи, а не з налаштувань getcontext.

Саме тому:

Decimal(0.2) + Decimal(0.1) == Decimal(0.3) *# False*

але

Decimal(0.2) + Decimal(0.1) == Decimal(0.3) + Decimal(0.0) *# True*

При конвертації в float точність береться з налаштувань getcontext.

**Можливості пакетів math, cmath**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson08/lesson-08#%D0%BC%D0%BE%D0%B6%D0%BB%D0%B8%D0%B2%D0%BE%D1%81%D1%82%D1%96-%D0%BF%D0%B0%D0%BA%D0%B5%D1%82%D1%96%D0%B2-math-cmath)

Для математичних обчислень у Python доданий пакет math. Цей пакет містить ряд часто використовуваних математичних функцій та констант:

* тригонометричні функції
  + sin / asin
  + cos / acos
  + tan / atan
  + hypot, обчислення відстані Евкліда між точками
* Перетворення кутів та радіанів:
  + degrees
  + radians
* Гіперболічні функції:
  + sinh / asinh
  + cosh / acosh
  + tanh / atanh
* степеневі функції:
  + exp
  + pow (перетворить аргументи в float, на відміну від вбудованої pow)
  + sqrt
  + log2
  + log10
  + log логарифм із основою e, якщо основа не задана або із заданою основою.
* константи:
  + pi
  + e
  + tau
  + inf, нескінченність
  + nan, не число

import math  
  
math.sin(math.pi / 4) *# 0.7071067811865475*  
math.degrees(math.pi / 4) *# 45.0*

Якщо вам потрібна комплексна математика, то можна скористатися пакетом cmath. Він надає той самий API, але вміє працювати з комплексними числами.

import cmath  
  
cmath.sqrt(-4) *# 2j*

**Collections**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson08/lesson-08#collections)

У Python є пакет з додатковими колекціями, які можуть дуже знадобитися розробнику-початківцю і сильно розширяти його арсенал.

**Іменовані кортежі**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson08/lesson-08#%D1%96%D0%BC%D0%B5%D0%BD%D0%BE%D0%B2%D0%B0%D0%BD%D1%96-%D0%BA%D0%BE%D1%80%D1%82%D0%B5%D0%B6%D1%96)

Використання кортежів у Python для передачі даних між обробниками — це хороша та поширена практика. Але є одна незручність у кортежів, вам необхідно пам'ятати індексацію елементів у кортежі і не плутати їх порядок. Це не завжди зручно і для ситуацій, коли в кортежі є 2 або більше елементів, такий підхід ускладнює читабельність коду:

person = ('Mick', 'Nitch', 35, 'Boston', '01146')

Після створення person там, де ви його використовуєте, вам потрібно пам'ятати, що ім'я на першому місці, а вік — на третьому. Щоб не плутатися, доведеться постійно повертатися до коду, де створюється person. Це незручно і спеціально для таких випадків додали іменовані кортежі:

import collections  
  
Person = collections.namedtuple('Person', ['name', 'last\_name', 'age', 'birth\_place', 'post\_index'])  
person = Person('Mick', 'Nitch', 35, 'Boston', '01146')  
person.name *# 'Mick'*  
person.post\_index *# '01146'*  
person.age *# 35*  
person[3] *# 'Boston'*

Тепер, використовуючи Person, ви можете створювати кортежі, які обов'язково повинні містити 5 елементів і у таких кортежів, окрім індексів, є атрибути. Звертатися до елементів такого кортежу можна як за індексом, так і за ім'ям. За такого підходу вам достатньо один раз визначити Person і більше не повертатися до нього, щоб згадати, який елемент за що відповідає.

**Counter**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson08/lesson-08#counter)

Часто вам потрібно підрахувати кількість елементів у певній послідовності. Для цього зручно скористатися словником.

student\_marks = [4, 2, 4, 6, 7, 4, 2 , 3, 4, 5, 6, 6, 7, 1, 1, 1, 3, 5]  
mark\_counts = {}  
for mark in student\_marks:  
 if mark in mark\_counts:  
 mark\_counts[mark] += 1  
 else:  
 mark\_counts[mark] = 1  
print(mark\_counts) *# {4: 4, 2: 2, 6: 3, 7: 2, 3: 2, 5: 2, 1: 3}*

Таке завдання зустрічається досить часто і, щоб не писати одні й ті самі 6 рядків коду постійно, у collections додали спеціальний словник Counter:

import collections  
  
  
student\_marks = [4, 2, 4, 6, 7, 4, 2 , 3, 4, 5, 6, 6, 7 , 1, 1, 1, 3, 5]  
mark\_counts = collections.Counter(student\_marks)  
print(mark\_counts) *# Counter({4: 4, 6: 3, 1: 3, 2: 2, 7: 2, 3: 2, 5: 2})*

Але на цьому корисні властивості Counter не закінчуються. Він може вивести елементи за частотою появи:

import collections  
  
  
student\_marks = [4, 2, 4, 6, 7, 4, 2 , 3, 4, 5, 6, 6, 7 , 1, 1, 1, 3, 5]  
mark\_counts = collections.Counter(student\_marks)  
print(mark\_counts.most\_common(1)) *# [(4, 4)]*  
print(mark\_counts.most\_common(2)) *# [(4, 4), (6, 3)]*

Ще Counter може відняти кількість елементів одного Counter від другого поелементно:

from collections import Counter  
  
  
c = Counter(a=4, b=2, c=0, d=-2)  
d = Counter(a=1, b=2, c=3, d=4)  
c.subtract(d)  
print(c) *# Counter({'a': 3, 'b': 0, 'c': -3, 'd': -6})*

**defaultdict**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson08/lesson-08#defaultdict)

Це спеціальний словник, який створює значення для ключів, яких в словнику не було за запитом. Наприклад, у вас є список слів і ви хочете розбити його на декілька списків, залежно від першої літери слова.

words = ['apple', 'zoo', 'lion', 'lama', 'bear', 'bet', 'wolf', 'appendix']  
grouped\_words = {}  
  
for word in words:  
 char = word[0]  
 if char not in grouped\_words:  
 grouped\_words[char] = []  
 grouped\_words[char].append(word)  
  
print(grouped\_words)

У консолі ми побачимо:

{  
 'a': ['apple', 'appendix'],  
 'z': ['zoo'],  
 'l': ['lion', 'lama'],  
 'b': ['bear', 'bet'],  
 'w': ['wolf']  
}

Таким чином ми можемо отримати всі слова із words, що починаються на якусь літеру. Подібні завдання зустрічаються досить часто. Щоб не перевіряти, чи є список на цю літеру в словнику grouped\_words, ми можемо скористатися defaultdict із collections та задати значенням за замовчуванням порожній список:

from collections import defaultdict  
  
  
words = ['apple', 'zoo', 'lion', 'lama', 'bear', 'bet', 'wolf', 'appendix']  
grouped\_words = defaultdict(list)  
  
for word in words:  
 char = word[0]  
 grouped\_words[char].append(word)

Результат виконання буде ідентичний. defaultdict приймає у якості аргументу функцію, яка буде використовуватися для створення значення за замовчуванням. В цьому прикладі ми використали list, але ви можете передати будь-яку функцію, яка викликається без аргументів.

**deque**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson08/lesson-08#deque)

Списки у Python реалізовані таким чином, що вибір елемента за індексом відбувається за константний час (дуже швидко) і додавання/видалення елементу в кінець списку теж відбувається дуже швидко. Але ось додавання елементу в будь-яке інше місце в списку змушує Python перерахувати індекси усіх елементів списку до кінця. Для великих списків це може бути дуже невигідно. Щоб швидко додавати елементи на початок списку, в Python в пакеті collections є така колекція як deque:

from collections import deque  
  
  
d = deque()  
d.append('last')  
d.appendleft('first')  
d.insert(1, 'middle')  
print(d) *# deque(['first', 'middle', 'last'])*  
  
print(d.pop()) *# 'last'*  
print(d.popleft()) *# 'first'*  
print(d) *# deque(['middle'])*

Окрім додавання на початок за допомогою appendleft, у deque є і швидке видалення першого елементу popleft.

Ще однією особливістю deque є можливість обмежити розмір deque:

from collections import deque  
  
  
d = deque(maxlen=5)  
for i in range(10):  
 d.append(i)  
  
print(d) *# deque([5, 6, 7, 8, 9], maxlen=5)*

Як видно з прикладу, нові елементи витісняють старіші, але розмір залишається незмінним.

В іншому deque веде себе точно як список Python.

**Comprehensions**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson08/lesson-08#comprehensions)

Часто потрібно створити колекцію і відразу заповнити її елементами. Зробити це одним виразом не можна, доводиться писати цикл. Наприклад, створимо список квадратів чисел від 1 до 5:

sq = []  
for i in range(1, 5+1):  
 sq.append(i\*\*2)  
  
print(sq) *# [1, 4, 9, 16, 25]*

Подібні операції, які ми робимо зі змінюваними колекціями (list, dict, set) у циклі for, ми можемо виконати одним виразом за допомогою конструкції comprehension.

comprehensions — це синтаксична конструкція Python, створена спеціально, щоб зменшити кількість коду, коли потрібно для кожної ітерації циклу for додати один елемент у нову колекцію.

Суть синтаксису comprehensions: всередині дужок (для list — квадратні, для dict та set — круглі) записується вираз (тільки один), який потрібно виконати з кожним елементом циклу for та сам цикл, але без двокрапки у кінці.

Найпростіше зрозуміти цю конструкцію на прикладах.

**list**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson08/lesson-08#list)

Щоб записати до списку числа від 1 до 5 за допомогою comprehensions:

numbers = [i for i in range(1, 5+1)]

Попередній приклад із квадратами чисел від 1 до 5 можна переписати за допомогою comprehension:

sq = [i \*\* 2 for i in range(1, 5+1)]

**set**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson08/lesson-08#set)

Для множин ситуація абсолютно аналогічна. Збережімо множини квадратів чисел зі списку:

numbers = [1, 4, 1, 3, 2, 5, 2, 6]  
sq = {i \*\* 2 for i in numbers}  
print(sq) *# {1, 4, 36, 9, 16, 25}*

**dict**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson08/lesson-08#dict)

Для словників синтаксис comprehension трохи відрізняється, оскільки потрібно явно вказати ключ та значення:

numbers = [1, 4, 1, 3, 2, 5, 2, 6]  
sq = {i: i \*\* 2 for i in numbers}  
print(sq) *# {1: 1, 4: 16, 3: 9, 2: 4, 5: 25, 6: 36}*

В цьому прикладі ми записали в словник sq числа від 1 до 5 у якості ключів, а їх квадрати як значення. Основна відмінність за синтаксисом — це двокрапка між ключем і значенням в лівій частині виразу всередині дужок.

**Функції (декоратори, замикання)**

**Функція як об'єкт першого класу**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson09/lesson-09#%D1%84%D1%83%D0%BD%D0%BA%D1%86%D1%96%D1%8F-%D1%8F%D0%BA-%D0%BE%D0%B1%D1%94%D0%BA%D1%82-%D0%BF%D0%B5%D1%80%D1%88%D0%BE%D0%B3%D0%BE-%D0%BA%D0%BB%D0%B0%D1%81%D1%83)

Об'єктами першого класу (англ. first-class object, first-class entity, first-class citizen) у контексті конкретної мови програмування називаються елементи, які можуть бути передані як параметр, повернуті із функції, присвоєні змінній [wiki](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82_%D0%BF%D0%B5%D1%80%D0%B2%D0%BE%D0%B3%D0%BE_%D0%BA%D0%BB%D0%B0%D1%81%D1%81%D0%B0).

На практиці це означає, що ми можемо працювати з функціями у Python точно так, як з будь-якими іншими типами даних:

* можемо створювати змінні і записувати в них функції;

def func(x, y):  
 return x + y  
  
 func\_alias = func  
 result = func\_alias(2, 3)  
 print(result) *# 5*

* можемо передавати функцію як аргументи для інших функцій;

def sum\_func(x, y):  
 return x + y  
  
 def subtraction\_func(x, y):  
 return x - y  
  
 def tricky\_func(x, y, func):  
 return func(x, y)  
  
 sum\_result = tricky\_func(2, 3, sum\_func)  
 min\_result = tricky\_func(2, 3, subtraction\_func)  
  
 print(sum\_result) *# 5*  
 print(min\_result) *# -1*

* можемо повертати з функції інші функції.

def sum\_func(x, y):  
 return x + y  
  
 def subtraction\_func(x, y):  
 return x - y  
  
 def get\_operator(operator):  
 if operator == '+':  
 return sum\_func  
 elif operator == '-':  
 return subtraction\_func  
 else:  
 print('Unknown operator')  
  
 sum\_action\_function = get\_operator("+")  
 print(sum\_action\_function(2, 3)) *# 5*  
  
 sub\_action\_function = get\_operator("-")  
 print(sub\_action\_function(2, 3)) *# -1*

Таким чином з функціями у Python можна працювати так само, як і з будь-якими іншими об'єктами. Це відкриває перед розробником безліч можливостей, про які ми поговоримо далі.

**Області видимості (LEGB)**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson09/lesson-09#%D0%BE%D0%B1%D0%BB%D0%B0%D1%81%D1%82%D1%96-%D0%B2%D0%B8%D0%B4%D0%B8%D0%BC%D0%BE%D1%81%D1%82%D1%96-legb)

Область видимості — це область у програмі (коді), в межах якої ви можете звернутися за ім'ям до вмісту змінної (або як ми вже з'ясували — функції). У Python можна виділити дві великі області видимості:

* глобальну (Global Scope). Все, що оголошено в цій області, буде доступне в будь-якій частині вашого застосунку;
* локальну (Local Scope). Все, що оголошено в локальній області видимості, буде доступно тільки всередині неї.

Глобальна область видимості лише одна і в ній міститься все, що ви оголосили на рівні модулю (змінні, функції, інші модулі тощо).

Локальна область видимості створюється, коли інтерпретатор потрапляє всередину функції.

SOME\_VAR = 3  
  
  
def func(x):  
 SOME\_VAR = x  
 print(SOME\_VAR)  
  
  
def procedure()  
 print(SOME\_VAR)  
  
  
procedure() *# 3*  
func(5) *# 5*  
print(SOME\_VAR) *# 3*

Давайте розберемо на цьому прикладі, як працюють області видимості. Змінна SOME\_VAR оголошена на рівні модулю у глобальній області видимості (так само як і функції func та procedure).

Коли ми викликаємо procedure всередині неї, створюється локальна область видимості. У локальній для procedure області видимості немає SOME\_VAR, і коли ми виконуємо вираз print(SOME\_VAR), Python спочатку шукає SOME\_VAR у локальній області видимості, не знаходить, і йде шукати у глобальній, де і знаходить SOME\_VAR рівне 3.

Коли ми викликаємо func, всередині неї створюється локальна область видимості. В цій локальній області ми створюємо змінну SOME\_VAR і присвоюємо їй значення 5. У цей момент глобальна область видимості жодним чином не змінюється, нова SOME\_VAR існує тільки всередині func. На підтвердження цього ми виконуємо print(SOME\_VAR) на рівні модуля й отримуємо 3, що каже про те, що всередині func існувала своя окрема змінна з ім'ям SOME\_VAR.

Щоб краще розуміти, що таке зона видимості, варто зрозуміти як працює їх механіка.

У Python області видимості реалізовані як словники, в яких ім'я змінної або функції (рядок) — це ключ, а сам об'єкт — це значення в цьому словнику.

Таким чином, коли ви у коді виконали x = 2, в словнику простору імен (те саме, що й область видимості) додалася пара: 'x': 2. І коли десь далі вам потрібно отримати значення x з цього словника, буде взято значення 2, знайдене за ключем 'x'.

Коли створюється нова зона видимості (простір імен), то насправді створюється новий подібний словник (старий нікуди не дівається звичайно ж).

Тут можна зробити важливий висновок, у вас в коді може одночасно існувати декілька просторів імен, які не взаємодіють один з одним. Це у свою чергу означає, що пошук об'єктів за ім'ям повинен відбуватися за якимось правилом.

У Python правила пошуку імен дуже прості:

1. спочатку пошук йде в локальному просторі імен (LOCAL);
2. якщо не знайдено в локальному, то в локальному на наступному рівні (ENCLOSED) і так далі, доки локальні не закінчаться;
3. далі буде перевірено глобальний (GLOBAL) простір імен (рівень модуля);
4. і в останню чергу простір вбудованих імен (BUILT INS) — це ключові слова і функції, що є частиною мови Python.

Якщо ніде не знайдено сутності з таким ім'ям, то отримаємо виключення.

Це правило можна запам'ятати за його акронімом: (LOCAL, ENCLOSED, GLOBAL, BUILT INS).

У цьому правилі ми не розглянули, що таке ENCLOSED.

GLOBAL\_SCOPE\_VAR = 1  
  
def func():  
 enclosed\_scope\_var = 2  
 def inner():  
 inner\_var = 3

В цьому коді всередині функції func оголошена функція inner. Для функції func enclosed\_scope\_var та inner — знаходиться в LOCAL області видимості. Для функції inner змінна inner\_var буде в її локальній області видимості, а вже enclosed\_scope\_var буде в ENCLOSED, в локальному просторі, але на рівень вище.

**Замикання**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson09/lesson-09#%D0%B7%D0%B0%D0%BC%D0%B8%D0%BA%D0%B0%D0%BD%D0%BD%D1%8F)

Особливість існування вкладених локальних просторів імен і той факт, що вони створюються динамічно, дає можливість використати механізм замикань у Python.

Найпростіше це пояснити на прикладі:

def adder(val):  
 def inner(x):  
 return x + val  
 return inner  
  
  
two\_adder = adder(2)  
print(two\_adder(3)) *# 5*  
print(two\_adder(5)) *# 7*  
  
three\_adder = adder(3)  
print(three\_adder(5)) *# 8*  
print(three\_adder(-3)) *# 0*  
  
id(two\_adder) == id(three\_adder) *# False*

В цьому прикладі ми створили функцію adder, яка повертає внутрішню функцію inner. Коли інтерпретатор заходить в adder (наприклад, у виразі two\_adder = adder(2)) створюється новий локальний простір імен в adder, де створюється функція inner, яка і повертається з функції. Що не очевидно, так це те, що новостворена inner, коли її викликають, буде використовувати те значення val, яке було в adder на момент її створення.

Коли ми виконали two\_adder = adder(2), в two\_adder записали функцію inner, яка буде додавати 2 до x. А коли виконали three\_adder = adder(3), то в three\_adder записали inner, яка завжди буде додавати 3 до x.

two\_adder та three\_adder — це дві різні функції, про що каже результат перевірки їх ідентифікаторів (вираз id(two\_adder) == id(three\_adder)).

**Каррування**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson09/lesson-09#%D0%BA%D0%B0%D1%80%D1%80%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F)

Каррування — це перетворення функції від багатьох аргументів у набір функцій, кожна з яких є функцією від одного аргументу. Ми можемо передати частину аргументів у функцію і отримати назад функцію, що очікує інші аргументи.

Наприклад, для застосунку калькулятора ми можемо створити функцію, яка буде приймати операнди та оператор:

def handle\_operation(x, y, operator):  
 if operator == '-':  
 return x - y  
 elif operator == '+':  
 return x + y  
  
  
handle\_operation(2, 3, '+') *# 5*  
handle\_operation(2, 3, '-') *# -1*

Це рішення страждає тим, що додавання нових і нових математичних операторів постійно ускладнюватиме конструкцію if ... elif ....

Щоб уникнути цього, ми можемо скористатися карруванням:

def sum\_func(x, y):  
 return x + y  
  
  
def sub\_func(x, y):  
 return x - y  
  
  
OPERATIONS = {  
 '-': sub\_func,  
 '+': sum\_func  
}  
  
  
def get\_handler(operator):  
 return OPERATIONS[operator]  
  
  
handler = get\_handler('-')  
handler(2, 3) *# -1*  
  
get\_handler('+')(2, 3) *# 5*

За такого підходу get\_handler повертатиме нам потрібну функцію двох аргументів зі словника з функціями OPERATIONS. Це швидший і елегантніший спосіб уникнути величезних if ... elif ... конструкцій. Крім того, так ви можете зменшувати кількість аргументів, які передаєте у функцію до прийнятної кількості (не більше двох), створюючи "на льоту" нові функції, які вже "знають" про передані аргументи. Як ми робили з two\_adder та three\_adder вище.

**Декоратори**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson09/lesson-09#%D0%B4%D0%B5%D0%BA%D0%BE%D1%80%D0%B0%D1%82%D0%BE%D1%80%D0%B8)

Є такий шаблон проектування — Декоратор. Цей шаблон полягає в тому, щоб розширювати існуючий функціонал, не вносячи змін в код цього самого функціоналу.

Наприклад, у нас є якась дуже складна і важлива функція complicated:

def complicated(x, y):  
 return x / y

І ми не хочемо міняти її код з якоїсь причини. Але нам потрібно додати логування до цієї функції, виводити в консоль щоразу, коли вона викликається, з якими аргументами її викликали і що вона повернула в результаті.

Пам'ятаючи про те, що функція — це об'єкт першого класу, можна зробити щось подібне:

def complicated(x, y):  
 return x / y  
  
  
def logged\_func(func):  
 def inner(x, y):  
 print(f'called with {x}, {y}')  
 result = func(x, y)  
 print(f'result: {result}')  
 return result  
 return inner  
  
  
complicated = logged\_func(complicated)

Тепер, викликаючи complicated, ми побачимо у консолі, з якими аргументами її викликали і що вона повернула. При цьому, код самої complicated жодним чином не змінився і спосіб роботи з нею також.

Проте такий код не занадто легкий для читання і досить об'ємний. Крім того, в коді легко пропустити рядок complicated = logged\_func(complicated) і не занадто просто зрозуміти, звідки виходитиме доданий до complicated функціонал.

Щоб спростити застосування цього шаблону проектування, в Python є спеціальний синтаксис декоратора. Точно той самий код, який робить в точності те саме, можна записати у вигляді:

def logged\_func(func):  
 def inner(x, y):  
 print(f'called with {x}, {y}')  
 result = func(x, y)  
 print(f'result: {result}')  
 return result  
 return inner  
  
  
@logged\_func  
def complicated(x, y):  
 return x / y

Тепер у коді явно видно, що complicated була задекорована logged\_func у тому самому місці, де complicated була оголошена.

**Ітератори/генератори (ключове слово yield)**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson09/lesson-09#%D1%96%D1%82%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80%D0%B8%D0%B3%D0%B5%D0%BD%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80%D0%B8-%D0%BA%D0%BB%D1%8E%D1%87%D0%BE%D0%B2%D0%B5-%D1%81%D0%BB%D0%BE%D0%B2%D0%BE-yield)

Функції — це ізольовані набори інструкцій з однією точкою входу і однією точкою виходу. Іншими словами, коли ви викликаєте функцію, то інтерпретатор виконає увесь її код від початку і до кінця при кожному виклику.

Але, що якщо потрібно отримати такий ізольований набір інструкцій з декількома точками входу? Або, що якщо ми хочемо продовжити виконання функції з якогось місця при наступному її виклику?

Прикладом таких ситуацій може слугувати генератор випадкових чисел, який при кожному виклику видає нове "випадкове" число і не повторює вже виданих. Або, коли нам потрібна велика послідовність, яку можна легко обчислити і було б невигідно зберігати в пам'яті повністю.

Подібні завдання у Python вирішують ітератори і генератори. Один із способів створити ітератор або генератор у Python — це створити особливу функцію з декількома точками входу. Для цього використовується ключове слово yield.

yield поводиться схожим чином з return, повертає управління потоком виконання програмою з тіла функції. Але, на відміну від return, yield при наступному зверненні не розпочинає виконання функції з початку, а продовжує з місця зупинки.

Звичайно така поведінка припускає, що десь в пам'яті повинна зберігатися інформація про те, на чому виконання зупинилося і про стан локального простору імен.

def interval\_generator(x, y):  
 while x <= y:  
 yield x  
 x += 1  
  
  
five\_to\_ten\_generator = interval\_generator(5, 10)  
  
next(five\_to\_ten\_generator) *# 5*  
next(five\_to\_ten\_generator) *# 6*  
next(five\_to\_ten\_generator) *# 7*  
next(five\_to\_ten\_generator) *# 8*  
next(five\_to\_ten\_generator) *# 9*  
next(five\_to\_ten\_generator) *# 10*

В цьому прикладі ми створили генератор цілих чисел від x до y interval\_generator. Щоб створити генератор чисел від 5 до 10, ми викликали interval\_generator, передавши як числа 5 та 10. Результат (сам генератор) зберегли в five\_to\_ten\_generator. Генератор — це не звичайна функція і, щоб отримати з нього значення, можна скористатися вбудованою функцією next. Але це не найзручніший спосіб. Найчастіше генератори використовуються безпосередньо в циклах for ...:

def interval\_generator(x, y):  
 while x <= y:  
 yield x  
 x += 1  
  
  
five\_to\_ten\_generator = interval\_generator(5, 10)  
for i in five\_to\_ten\_generator:  
 print(i)

Результат виконання цього коду буде абсолютно ідентичний. Ви вже стикалися з генераторами у Python, наприклад range — це генератор.

Принципова відмінність генератора від ітератора в тому, що генератор обчислює наступне значення деякої послідовності у момент виклику, а ітератор перебирає збережену в пам'яті послідовність. З точки зору використання генератор та ітератор абсолютно ідентичні.

**Лямбда-функції (анонімні функції)**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson09/lesson-09#%D0%BB%D1%8F%D0%BC%D0%B1%D0%B4%D0%B0-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D1%96%D1%97-%D0%B0%D0%BD%D0%BE%D0%BD%D1%96%D0%BC%D0%BD%D1%96-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D1%96%D1%97)

Буває, що нам потрібна функція, суть якої можна викликати, передавши їй аргументи, але сама функція дуже проста і її всю можна описати одним виразом. У таких ситуаціях немає особливого сенсу створювати функцію, використовуючи стандартний синтаксис і захаращувати простір імен.

Спеціально для таких випадків у Python є лямбда-функції або анонімні функції.

sum\_lambda = lambda x, y: x + y

Синтаксис лямбда функцій:

* розпочинається з ключового слова lambda, після якого йде список позиційних аргументів функції через кому (аргументів може і не бути);
* потім йде двокрапка;
* далі йде тіло функції, суворо один вираз;
* результат виразу буде повернений як результат лямбди (return не потрібен).

У прикладі вище ми створили лямбда-функцію sum\_lambda, яка повертає суму двох чисел. Насправді це "поганий тон" зберігати лямбда-функції у змінних, вони повинні створюватися там, де будуть використовуватися і більше ніде у коді не залишають слідів.

**map**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson09/lesson-09#map)

Один з прикладів використання лямбда-функцій — це генератор map.

map приймає на вхід функцію і послідовність, повертає генератор, який застосовує до кожного елементу послідовності функцію.

Давайте напишемо за допомогою map генератор, який підносить числа із списку numbers до квадрату:

numbers = [1, 2, 3, 4, 5]  
  
for i in map(lambda x: x \*\* 2, numbers):  
 print(i)

в результаті виконання цього коду ви побачите:

1  
4  
9  
16  
25

В якості першого аргументу у map ми передали лямбда-функцію lambda x: x \*\* 2, яка повертає x у степені 2.

В результаті виконання виразу map(lambda x: x \*\* 2, numbers) ми отримаємо генератор, яким пройшлися в циклі for та вивели значення на кожній ітерації функцією print.

У map можна передавати будь-яку функцію, яка приймає один аргумент і будь-яку послідовність, за якою можна ітеруватися (список, кортеж, словник, множини, другий генератор/ітератор).

**filter**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson09/lesson-09#filter)

Аналогічний map синтаксис у filter. filter теж приймає на вхід функцію та послідовність і повертає генератор. Головна відмінність у тому, що filter застосовує функцію до кожного елемента послідовності та повертає тільки елементи, які дають "truthy" результати.

Нагадаємо, що у Python можна будь-який тип привести до boolean. До False приводяться 0, None та порожні контейнери (рядки, списки, словники, множини, кортежі, всі інші випадки приводяться до True (їх ще називають "truthy").

Наприклад, виведемо список чисел, які діляться на 2 із залишком в інтервалі від 1 до 10:

for i in filter(lambda x: x % 2, range(1, 10+1)):  
 print(i)

У цьому прикладі ми передали у filter лямбда функцію lambda x: x % 2, яка повертає залишок від ділення. Для кожного елемента послідовності з генератора range(1, 10+1) filter виконав перевірку лямбда-функцією та, якщо вона (лямбда) повернула "truthy" результат, то цей результат потрапляв у відфільтровану послідовність на виході.

Інший приклад, давайте відфільтруємо з рядка літери верхнього регістру, щоб залишилися лише літери нижнього:

some\_str = 'aaAbbB C F DDd EEe'  
for i in filter(lambda x: x.islower(), some\_str):  
 print(i)

**Визначення та створення класу**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson10/lesson-10#%D0%B2%D0%B8%D0%B7%D0%BD%D0%B0%D1%87%D0%B5%D0%BD%D0%BD%D1%8F-%D1%82%D0%B0-%D1%81%D1%82%D0%B2%D0%BE%D1%80%D0%B5%D0%BD%D0%BD%D1%8F-%D0%BA%D0%BB%D0%B0%D1%81%D1%83)

Класи — це структура мови програмування, яка дозволяє об'єднати в рамках однієї сутності змінні різних типів (поля) та функції (методи). Ви вже знайомі з тим, як працювати з класами у Python і неодноразово використовували їх. Річ у тім, що майже все у Python — це об'єкт класу.

Наочною метафорою для класів і об'єктів може бути форма для випічки печива і печиво. Клас — це форма, яка задає суть об'єкта, які поля він містить і як можна оперувати з цими полями. А об'єкт класу — це печиво, воно обов'язково буде заданої класом форми, але начинка кожного печивка може бути різною.

class User:  
 name = 'UserName'  
 age = 15

В цьому прикладі ми створили клас User з полями name та age.

Створення класу починається з ключового слова class, після якого йде назва класу і перелік класів, які він наслідує у дужках через кому, якщо такі є. Після цього ставиться двокрапка і з наступного рядка описуються атрибути класу.

Існує нестрога домовленість називати клас з великої літери і, якщо назва складається з декількох слів, то усі слова з великої літери без додаткових символів між ними.

**Створення об'єктів**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson10/lesson-10#%D1%81%D1%82%D0%B2%D0%BE%D1%80%D0%B5%D0%BD%D0%BD%D1%8F-%D0%BE%D0%B1%D1%94%D0%BA%D1%82%D1%96%D0%B2)

Таким чином клас — це форма для випікання, а ось готове печиво — це об'єкти класу.

class User:  
 name = 'UserName'  
 age = 15  
  
user1 = User()  
print(user1.name) *# 'UserName'*  
print(user1.age) *# '15'*  
  
user2 = User()  
user2.name = "John"  
user2.age = 90  
  
print(user2.name) *# 'Jhon'*  
print(user2.age) *# '90'*

В цьому прикладі ми створили два об'єкти класу User: user1 та user2. Обидва цих об'єкти належать класу User (у них однакова форма), але вони відмінні один від одного (начинка у них різна). Таким чином ви можете працювати з об'єктами класів (наприклад User) знаючи, що у них є поля та методи, визначені в цьому класі (наприклад name та age).

**Атрибути класу (поля та методи)**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson10/lesson-10#%D0%B0%D1%82%D1%80%D0%B8%D0%B1%D1%83%D1%82%D0%B8-%D0%BA%D0%BB%D0%B0%D1%81%D1%83-%D0%BF%D0%BE%D0%BB%D1%8F-%D1%82%D0%B0-%D0%BC%D0%B5%D1%82%D0%BE%D0%B4%D0%B8)

Поля та методи класів називаються одним загальним словом "атрибути" класу. У визначенні класів у блоці коду під ім'ям класу відбувається визначення атрибутів класу.

Поля класу — це змінні, в яких міститься інформація. Полем може бути будь-який об'єкт Python. Зазвичай це змінна, або контейнер (словник, список, рядок тощо).

Метод класу — це функція, яка оперує з полями класу та/або аргументами, які передаються у метод.

У кожного об'єкта значення в полях можуть відрізнятися або можуть бути спільними для всіх об'єктів класу, залежно від потрібної вам поведінки.

Методи класу — це функції, які описують поведінку класу та як він взаємодіє з іншими об'єктами. Щоб метод класу міг працювати з іншими методами та полями класу, першим аргументом будь-якого методу завжди виступає сам об'єкт класу (ну майже завжди).

Для першого аргументу можна використовувати будь-яку назву, яка не викликає синтаксичної помилки, однак, є домовленість завжди використовувати self.

У будь-якого методу класу завжди повинен бути, принаймні, один аргумент (self), це вимога синтаксису Python, оскільки інтерпретатор під час виклику методу обов'язково передасть першим аргументом сам об'єкт, а потім уже всі аргументи, які були передані під час виклику.

class User:  
 name = 'UserName'  
 age = 15  
  
 def say\_name(self):  
 print(f'Hi! I am {self.name} and I am {self.age} years old.')  
  
 def set\_age(self, age):  
 self.age = age  
  
  
bob = User()  
bob.name = 'Bob'  
  
bob.say\_name() *# Hi! I am Bob and I am 15 years old.*  
  
bob.set\_age(25)  
bob.say\_name() *# Hi! I am Bob and I am 25 years old.*

У цьому прикладі ми додали два методи класу User: say\_name та set\_age. say\_name виводить у консоль рядок привітання від користувача. Для формування цього рядку використовуються поля name та age.

Метод set\_age приймає на вхід число та записує його в поле age.

class Human:  
 name = ""  
  
 def hello(self, val):  
 if self.name:  
 return f"Hello {val}! I am {self.name}."  
 return f"Hello {val}!"  
  
bill = Human()  
print(bill.hello("John")) *# Hello John!*  
  
bill.name = "Bill"  
print(bill.hello("John")) *# Hello John! I am Bill.*

**Конструктор класу**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson10/lesson-10#%D0%BA%D0%BE%D0%BD%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D0%BE%D1%80-%D0%BA%D0%BB%D0%B0%D1%81%D1%83)

Підіб'ємо короткий підсумок, об'єкти можуть зберігати дані у звичайних змінних, які належать об'єкту. Змінні, що належать об'єкту або класу, називають **полями**. Об'єкти можуть мати функції, що належать класу. Такі функції називають **методами** класу. Всі разом (поля та методи) заведено називати **атрибутами** класу.

За допомогою атрибутів класу ми виконуємо *інкапсуляцію* — приховуємо деталі реалізації під інтерфейсом класу.

ІНКАПСУЛЯЦІЯ

Це здатність об'єктів приховувати частину свого стану та поведінки від інших об'єктів, надаючи зовнішньому світу лише певний інтерфейс взаємодії із собою.

Як бачимо — методи класу мають одну відмінність від звичайних функцій, вони повинні мати додатково ім'я self, яке додається до початку списку параметрів. Однак, під час виклику методу жодного значення цьому параметру надавати не потрібно — його додасть сам Python. Ця змінна вказує на сам об'єкт екземпляра класу, і за традицією вона називається self.

Розглянемо наступний код:

class Person:  
 def \_\_init\_\_(self, name, age):  
 self.name = name  
 self.age = age  
  
 def greeting(self):  
 return f"Hi {self.name}"  
  
  
p = Person("Boris", 34)  
print(p.name) *# Boris*  
print(p.age) *# 34*  
print(p.greeting()) *# Hi Boris*

Тут метод \_\_init\_\_() — спеціальний метод конструктор, який автоматично виконується під час створення кожного нового екземпляра на базі класу Person. Ім'я методу починається і закінчується двома символами підкреслення. Причина в тому, що так ми запобігаємо конфліктам імен стандартних методів Python та методів ваших класів. Насправді метод \_\_init\_\_() є магічним методом і ми повернемося до нього у наступному модулі.

Коли ми створюємо екземпляр Person, Python викликає метод \_\_init\_\_() із класу Person. Ми передаємо у виклик Person("Boris", 34) ім'я "Boris" та вік 34 в аргументах. Пам'ятаємо, що значення self буде передано автоматично, явно передавати його не потрібно.

Зверніть увагу, що метод класу greeting не приймає параметрів, проте має self у визначенні функції.

**Наслідування**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson10/lesson-10#%D0%BD%D0%B0%D1%81%D0%BB%D1%96%D0%B4%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F)

Основна сила об'єктно-орієнтованого програмування полягає саме в можливості наслідування класів. Наслідування дає можливість створювати нові класи, що містять атрибути батьківських класів.

class Human:  
 name = ''  
 def voice(self):  
 print(f"Hello! My name is {self.name}")  
  
  
class Developer(Human):  
 field\_description = "My Programming language"  
 language = ""  
 def make\_some\_code(self):  
 return f"{self.field\_description} is {self.value}"  
  
  
class PythonDeveloper(Developer):  
 value = "Python"  
  
  
class JSDeveloper(Developer):  
 value = "JavaScript"  
  
  
p\_dev = PythonDeveloper()  
p\_dev.name = 'Bob'  
p\_dev.voice() *# Hello! My name is Bob*  
p\_dev.make\_some\_code() *# My Programming language is Python*  
  
  
js\_dev = JSDeveloper()  
js\_dev.make\_some\_code() *# My Programming language is JavaScript*

У цьому прикладі ми створили батьківський клас Human, який визначив, що у всіх є ім'я та метод voice.

Далі ми розширили функціонал класу Human та створили клас Developer, який наслідується від Human і додає йому поля field\_description та language і метод make\_some\_code. І щоб реалізувати функціонал розробника конкретною мовою, ми зробили два класи PythonDeveloper та JSDeveloper. Тепер, якщо ми захочемо додати функціонал у всі дочірні для Human класи, то для цього можна додати потрібні атрибути в Human і вони автоматично з'являться і в PythonDeveloper, і JSDeveloper. Якщо потрібно розширити функціонал всіх розробників, то можна додати потрібні атрибути до класу Developer. І якщо потрібно додати особливу поведінку класу розробників конкретною мовою, то можна додати або змінити атрибути класів PythonDeveloper або JSDeveloper.

Таким чином у коді повинно бути лише одне місце, де визначено поведінку об'єкта. І якщо нам потрібно отримати інший об'єкт, який має цю поведінку і якісь свої особливості, ми можемо наслідуватись від класу з потрібними нам спільними атрибутами та додати унікальні.

Цей підхід дозволяє писати менше коду та структурувати дані, створюючи моделі реальних об'єктів з їх характеристиками (полями) та поведінкою (методами).

**Method Resolution Order (MRO).**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson10/lesson-10#method-resolution-order-mro)

Наслідування — це дуже потужний інструмент. Наслідуватися можна не тільки від одного класу, а можна одразу від кількох. Таким чином можна отримувати об'єкти, що поєднують у собі властивості багатьох класів. Тут повинно виникнути питання, а що буде, якщо кілька класів мають атрибути з однаковим ім'ям?

Для відповіді на це питання потрібно зрозуміти, як Python шукає атрибути (поля або методи) в об'єктах. Власне, це є MRO (Method Resolution Order). MRO у Python працює наступним чином:

1. Шукає атрибут серед атрибутів самого класу. Саме завдяки цьому ви можете "перевизначати" батьківські атрибути.
2. Шукає атрибут у першого з батьків (той, що вказаний першим у списку батьків).
3. Шукає атрибут у наступного батька у списку батьків, доки такі є.
4. Шукає атрибут у батьках першого батька.
5. Повторює п.4 для всіх батьків.
6. Викликає виняток, що атрибут не знайдено.

Пошуки закінчуються, як тільки атрибут знайдено.

class A:  
 x = 'I am A class'  
  
  
class B:  
 x = 'I am B class'  
 y = 'I exist only in B'  
  
  
class C(A, B):  
 z = "This exists only in C"  
  
  
c = C()  
print(c.z) *# This exists only in C*  
print(c.y) *# I exist only in B*  
print(c.x) *# I am A class*

З цього прикладу видно, що у класі C поле x береться з A класу. Якщо ж в цьому самому прикладі змінити список батьків, то отримаємо:

class A:  
 x = 'I am A class'  
  
  
class B:  
 x = 'I am B class'  
 y = 'I exist only in B'  
  
  
class C(B, A):  
 z = "This exists only in C"  
  
  
c = C()  
print(c.z) *# This exists only in C*  
print(c.y) *# I exist only in B*  
print(c.x) *# I am B class*

Тепер у класі C поле x береться з B класу.

**Контейнери, створені за допомогою наслідування (UserList, UserDict, UserString).**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson10/lesson-10#%D0%BA%D0%BE%D0%BD%D1%82%D0%B5%D0%B9%D0%BD%D0%B5%D1%80%D0%B8-%D1%81%D1%82%D0%B2%D0%BE%D1%80%D0%B5%D0%BD%D1%96-%D0%B7%D0%B0-%D0%B4%D0%BE%D0%BF%D0%BE%D0%BC%D0%BE%D0%B3%D0%BE%D1%8E-%D0%BD%D0%B0%D1%81%D0%BB%D1%96%D0%B4%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F-userlist-userdict-userstring)

Часто для роботи потрібно створити об'єкти, які поводяться як стандартні контейнери Python, але з модифікованою поведінкою. Ви, звичайно, можете спробувати наслідувати dict, str, list, але це може призвести до ряду непередбачених помилок. Правильний спосіб отримати модифікований контейнер — це використовувати пакет collections та класи UserList, UserDict, UserString, які в ньому є.

Всі ці класи поводяться точно як вбудовані контейнери з тією лише відмінністю, що самі дані лежать у полі data у цих класів і ви можете використовувати це поле на свій розсуд.

from collections import UserDict  
  
  
class ValueSearchableDict(UserDict):  
 def has\_in\_values(self, value):  
 return value in self.data.values()  
  
  
as\_dict = ValueSearchableDict()  
as\_dict['a'] = 1  
as\_dict.has\_in\_values(1) *# True*  
as\_dict.has\_in\_values(2) *# False*

У цьому прикладі ми створили клас, який поводиться як словник, але в ньому є додатковий метод, який перевіряє наявність деякого value серед значень у цьому словнику.

from collections import UserList  
  
class CountableList(UserList):  
 def sum(self):  
 return sum(map(lambda x: int(x), self.data))  
  
  
countable = CountableList([1, '2', 3, '4'])  
countable.append('5')  
countable.sum() *# 15*

У цьому прикладі ми створили клас, який поводиться як список, але в ньому є метод sum , який повертає суму всього вмісту цього класу, при цьому перетворюючи рядки на цілі числа.

from collections import UserString  
  
  
class TruncatedString(UserString):  
 MAX\_LEN = 7  
 def truncate(self):  
 self.data = self.data[:self.MAX\_LEN]  
  
  
ts = TruncatedString('abcdefghjklmnop')  
ts.truncate()  
print(ts) *# abcdefg*

Останній приклад показує модифікований рядок з методом truncate, який обмежує розмір рядка до MAX\_LEN символів.

**Власні винятки**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson10/lesson-10#%D0%B2%D0%BB%D0%B0%D1%81%D0%BD%D1%96-%D0%B2%D0%B8%D0%BD%D1%8F%D1%82%D0%BA%D0%B8)

У Python широко використовується механізм винятків (Exceptions) для того, щоб дати зрозуміти коду, що викликає, що саме пішло не так і що з цим робити. На винятках також будують розгалуження коду, наприклад, ми очікуємо, що користувач введе саме число, але він може ввести що завгодно:

def input\_number():  
 while True:  
 try:  
 num = input("Enter integer number: ")  
 return int(num)  
 except:  
 print(f'"{num}" is not a number. Try again')  
  
  
num = input\_number()

У цьому прикладі функція input\_number вийде з нескінченного циклу тільки, коли користувач введе ціле число. Це приклад використання винятків у Python з метою управління потоком виконання.

Коли ж ви пишете свій застосунок вам може знадобитися створити свої власні винятки, щоб обробляти їх на вищому рівні. Наприклад, ви очікуєте, що користувач повинен ввести ім'я, і це ім'я не повинно бути коротшим трьох символів і починатися з великої літери. Ви можете створити власний виняток, який буде викликатися, якщо введення користувача не пройшло цю перевірку. Тоді будь-який код, який викликатиме цю функцію, зможе коректно обробити цей конкретний випадок.

import string  
  
  
class NameTooShortError(Exception):  
 pass  
  
  
class NameStartsFromLowError(Exception):  
 pass  
  
  
def enter\_name():  
 name = input("Enter name: ")  
 if len(name) < 3:  
 raise NameTooShortError  
 if name[0] not in string.ascii\_uppercase:  
 raise NameStartsFromLowError  
  
  
  
while True:  
 try:  
 name = enter\_name()  
 break  
 except NameTooShortError:  
 print('Name is too short, need more than 3 symbols. Try again.')  
 except NameStartsFromLowError:  
 print('Name should start from capital letter. Try again.')

У цьому прикладі ми створили власні винятки, наслідуючи батьківський клас для всіх винятків у Python — клас Exception. Далі у коді коректно обробили два випадки, коли користувач ввів занадто коротке ім'я, або коли ім'я починається не з великої літери.

**Поліморфізм / Качина типізація**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson10/lesson-10#%D0%BF%D0%BE%D0%BB%D1%96%D0%BC%D0%BE%D1%80%D1%84%D1%96%D0%B7%D0%BC--%D0%BA%D0%B0%D1%87%D0%B8%D0%BD%D0%B0-%D1%82%D0%B8%D0%BF%D1%96%D0%B7%D0%B0%D1%86%D1%96%D1%8F)

Качина типізація — це механізм властивий Python, який дозволяє використовувати будь-які об'єкти один замість іншого, аби в обох були потрібні методи та поля. Качиною ця типізація називається від приказки: "Якщо крякає як качка, плаває як качка і літає як качка, — це качка". Це добре відображає суть підходу, реалізованого у Python. Ні, інтерпретатор не перевіряє, що у функцію або метод був переданий об'єкт потрібного або дочірнього класу, достатньо щоб в об'єкта були потрібні методи і все буде працювати.

class Mammal:  
 phrase = ''  
 def voice(self):  
 return self.phrase  
  
  
class Dog(Mammal):  
 phrase = 'Bark!'  
  
  
  
class Cat(Mammal):  
 phrase = 'Meow!'  
  
  
class Chupakabra:  
 def voice(self):  
 return 'Whooooo!!!'  
  
  
class Recorder:  
 def record\_animal(self, animal):  
 voice = animal.voice()  
 print(f'Recorded "{voice}"')  
  
  
r = Recorder()  
cat = Cat()  
dog = Dog()  
strange\_animal = Chupakabra()  
  
r.record\_animal(cat) *# Recorded "Meow!"*  
r.record\_animal(dog) *# Recorded "Bark!"*  
r.record\_animal(strange\_animal) *# Recorded "Whooooo!!!"*

В цьому прикладі ми створили батьківський клас Mammal, у якого є метод voice та два дочірніх до нього Dog та Cat. Клас Record приймає на вхід методу record\_animal об'єкт animalта викликає в нього метод voice, щоб вивести результати виконання voice у консоль. При цьому є клас Chupakabra, у якого також є метод voice, та хоч він і не наслідується від Mammal, але об'єкти цього класу так само можна передавати в record\_animal. Головне, щоб атрибут називався так само і приймав ті самі аргументи (якщо це метод).

**"Магічні" методи**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson11/lesson-11#%D0%BC%D0%B0%D0%B3%D1%96%D1%87%D0%BD%D1%96-%D0%BC%D0%B5%D1%82%D0%BE%D0%B4%D0%B8)

Як ми вже говорили, практично все у Python є об'єктом. Поводження ж будь-якого об'єкта можна модифікувати, модифікуючи його поля з даними та/або методи. Коли ми застосовуємо до об'єкта оператор, наприклад + або -, ми, насправді, викликаємо деякий метод у цього об'єкта. Метод, який відповідає за те, як повинен цей об'єкт поводитися з цим оператором, що він повинен робити. Це стосується переважної більшості операторів Python. Таким чином, знаючи, який метод відповідає за поведінку об'єкта, коли до нього застосовують, наприклад +, ми можемо цей спосіб перевизначити. Методи, які відповідають за поведінку об'єктів, коли до них застосовуються синтаксичні конструкції Python, заведено називати "магічними". Звичайно нічого магічного в них немає. Магічними вони називаються, оскільки немає способу дізнатися, що, наприклад, метод \_\_add\_\_ відповідає за те, як поведеться об'єкт, якщо до нього застосувати оператор +. Ці та подібні методи можна знайти в довідниках та підручниках з Python, але жодним чином із самої структури мови. Вираз a + b "під капотом" перетворюється на a.\_\_add\_\_(b) і цей факт сам собою виглядає як "магія". Така поведінка — це частина мови Python і знати, які "магічні" методи за що відповідають — дуже важливо та обов'язково для розробника.

Досить детальний та докладний перелік "магічних" методів з прикладами та описом можна знайти за [посиланням](https://minhhh.github.io/posts/a-guide-to-pythons-magic-methods).

На цьому уроці ми з вами познайомимося з деякими найчастіше використовуваними представниками "магічних" методів.

Щодо цих методів є дві суворі домовленості: їх імена завжди складаються тільки з літер нижнього регістру та символів \_, і починаються та закінчуються \_\_ (двома символами нижнього підкреслення).

**Метод init**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson11/lesson-11#%D0%BC%D0%B5%D1%82%D0%BE%D0%B4-init)

Метод, що найчастіше використовується, — це метод \_\_init\_\_. Цей метод відповідає за ініціалізацію об'єкта. Коли ви створюєте об'єкт класу, то спочатку створюється порожній об'єкт, який містить лише обов'язкові службові атрибути. Після цього (об'єкт вже створено) автоматично викликається метод \_\_init\_\_, який ви можете модифікувати під ваші потреби.

class Human:  
 def \_\_init\_\_(self, name, age=0):  
 self.name = name  
 self.age = age  
  
 def say\_hello(self):  
 return f'Hello! I am {self.name}'  
  
  
bill = Human('Bill')  
print(bill.say\_hello()) *# Hello! I am Bill*  
print(bill.age) *# 0*  
  
jill = Human('Jill', 20)  
print(jill.say\_hello()) *# Hello! I am Jill*  
print(jill.age) *# 20*

В цьому прикладі ми створили клас Human, у якому визначили метод \_\_init\_\_. У цьому методі ми додаємо об'єктам цього класу поля name та age. Зверніть увагу, що метод \_\_init\_\_ може приймати аргументи позиційні та/або іменні, як і будь-який інший метод. Коли ми створюємо об'єкт класу Human, ми повинні класу передати обов'язково хоча б один аргумент, оскільки метод \_\_init\_\_ повинен приймати обов'язково name.

\_\_init\_\_ не обов'язково приймає аргументи та містить лише створення полів. Цей метод можна використовувати для реалізації будь-яких дій, які вам потрібні на етапі, коли об'єкт вже створений та його потрібно ініціалізувати.

**Методи str та repr**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson11/lesson-11#%D0%BC%D0%B5%D1%82%D0%BE%D0%B4%D0%B8-str-%D1%82%D0%B0-repr)

Коли ви в інтерактивному режимі роботи з Python хочете побачити вміст деякого об'єкта, ви просто пишете його ім'я в консолі та інтерпретатор виводить рядком представлення цього об'єкта.

l = [1, 2]  
l

У консолі ви побачите [1, 2].

За цей механізм внутрішнього читабельного представлення об'єктів відповідає магічний метод \_\_repr\_\_. Цей метод приймає лише один аргумент (self звичайно) і повинен повертати рядок.

Якщо ви хочете виводити у випадках, коли застосунок повинен відобразити об'єкт, якусь корисну інформацію, ви можете модифікувати цей метод. Наприклад, клас точки на площині в Декартових координатах:

class Point:  
 def \_\_init\_\_(self, x, y):  
 self.x = x  
 self.y = y  
  
 def \_\_repr\_\_(self):  
 return f'Point ({self.x}, {self.y})'  
  
  
a = Point(1, 9)  
a

Виконайте цей код у консолі Python і ви побачите Point(1, 9).

Дуже схожий на нього метод, який відповідає за те, як об'єкт конвертується в рядок — це метод \_\_str\_\_. Коли ви викликаєте функцію str та передаєте їй якийсь об'єкт, то насправді цей об'єкт викликається методом \_\_str\_\_.

class Human:  
 def \_\_init\_\_(self, name, age=0):  
 self.name = name  
 self.age = age  
  
 def \_\_str\_\_(self):  
 return f'Hello! I am {self.name}'  
  
  
bill = Human('Bill')  
bill\_str = str(bill)  
print(bill\_str) *# Hello! I am Bill*

**Методи getitem та setitem**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson11/lesson-11#%D0%BC%D0%B5%D1%82%D0%BE%D0%B4%D0%B8-getitem-%D1%82%D0%B0-setitem)

Квадратні дужки дозволяють вам звертатися до елементів послідовності за індексом або до елементів словника за ключем. Коли ви хочете отримати значення, використовуючи квадратні дужки, в об'єкта викликається метод \_\_getitem\_\_. Для запису значення з індексом або ключем викликається метод \_\_setitem\_\_. Обидва ці методи приймають першим аргументом self. \_\_getitem\_\_ другим аргументом приймає індекс або ключ, за яким потрібно знайти елемент, а \_\_setitem\_\_ другим аргументом приймає ключ/індекс, а третім значення, яке потрібно записати за цим ключем/індексом.

class ListedValuesDict:  
 def \_\_init\_\_(self):  
 self.data = {}  
  
 def \_\_setitem\_\_(self, key, value):  
 if key in self.data:  
 self.data[key].append(value)  
 else:  
 self.data[key] = [value]  
  
 def \_\_getitem\_\_(self, key):  
 result = str(self.data[key][0])  
 for value in self.data[key][1:]:  
 result += ", " + str(value)  
 return result  
  
  
l\_dict = ListedValuesDict()  
l\_dict[1] = 'a'  
l\_dict[1] = 'b'  
print(l\_dict[1]) *# a, b*

У цьому прикладі ми створили власний клас, який поводиться як словник. ListedValuesDict значення зберігає у список і вже цей список зберігає як значення для ключа. Головна відмінність від словника у тому, що ListedValuesDict не дозволяє перезаписувати значення, завжди додаватиме нове значення в кінець списку. І при отриманні значення повертає рядок, складений зі значень у списку.

**Функтори, метод call**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson11/lesson-11#%D1%84%D1%83%D0%BD%D0%BA%D1%82%D0%BE%D1%80%D0%B8-%D0%BC%D0%B5%D1%82%D0%BE%D0%B4-call)

Функтори — це об'єкти, які поводяться як функції у тому сенсі, що їх можна викликати та передавати їм аргументи. Функція у Python — це такий самий об'єкт, але у ньому реалізований метод \_\_call\_\_, який відповідає за синтаксис виклику з круглими дужками.

class Adder:  
 def \_\_init\_\_(self, add\_value):  
 self.add\_value = add\_value  
  
 def \_\_call\_\_(self, value):  
 return self.add\_value + value  
  
  
two\_adder = Adder(2)  
print(two\_adder(5)) *# 7*  
print(two\_adder(4)) *# 6*  
  
three\_adder = Adder(3)  
print(three\_adder(5)) *# 8*  
print(three\_adder(4)) *# 7*

В цьому прикладі ми створили клас Adder, у якого є метод \_\_call\_\_. Тепер об'єкти цього класу можна викликати як функцію, передаючи їм аргументи. Ці виклики будуть викликати метод \_\_call\_\_ в об'єктів класу Adder.

**Створення власних менеджерів контексту**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson11/lesson-11#%D1%81%D1%82%D0%B2%D0%BE%D1%80%D0%B5%D0%BD%D0%BD%D1%8F-%D0%B2%D0%BB%D0%B0%D1%81%D0%BD%D0%B8%D1%85-%D0%BC%D0%B5%D0%BD%D0%B5%D0%B4%D0%B6%D0%B5%D1%80%D1%96%D0%B2-%D0%BA%D0%BE%D0%BD%D1%82%D0%B5%D0%BA%D1%81%D1%82%D1%83)

Популярне завдання — це створення власних менеджерів контексту. Наприклад, ми пишемо клієнта для сервера, який повинен закрити сесію з сервером у будь-якому випадку. Для цього нам зручно буде створити менеджер контексту, який закриє з'єднання в будь-якому випадку.

У цьому нам допоможуть магічні методи, які відповідають за синтаксис with ... as ...:.

* \_\_enter\_\_ викликається, коли інтерпретатор заходить у контекст і те, що він поверне, буде записано в змінну після as;
* \_\_exit\_\_ викликається, коли інтерпретатор виходить із блоку менеджера контексту. Буде викликаний в будь-якому випадку.

class Session:  
 def \_\_init\_\_(self, addr, port=8080):  
 self.connected = True  
 self.addr = addr  
 self.port = port  
  
 def \_\_enter\_\_(self):  
 print(f"connected to {self.addr}:{self.port}")  
 return self  
  
 def \_\_exit\_\_(self, exception\_type, exception\_value, traceback):  
 self.connected = False  
 if exception\_type is not None:  
 print("Some error!")  
 else:  
 print("No problem")

Об'єкт класу **Session** буде менеджером контексту. Ми можемо створити його та використовувати повторно за потребою:

localhost\_session = Session("localhost")  
  
with localhost\_session as session:  
 print(session is localhost\_session) *# True*  
 print(localhost\_session.connected) *# True*  
  
print(localhost\_session.connected) *# False*

Коли інтерпретатор всередині менеджера контексту \_\_enter\_\_ вже викликаний та self.connected == True. Зверніть увагу, session is localhost\_session повертає True, оскільки метод \_\_enter\_\_ повертає self. Ви можете створювати об'єкт Session всередині виразу with ... as ...::

with Session("localhost") as session:  
 print(session.connected) *# True*

Метод \_\_exit\_\_ буде викликаний при виході з контексту помилково або штатно. \_\_exit\_\_ обов'язково повинен приймати аргументи exception\_type, exception\_value, traceback, окрім self. Це тип винятку, значення та увесь **traceback** помилки. Зверніть увагу, що \_\_exit\_\_ не дозволяє перехоплювати помилку, але він дає можливість щось виконати, незалежно від того, чи виникла помилка. Якщо контекст завершився без помилок, то в цих аргументах буде None.

Наприклад, покажемо ситуацію, коли менеджер контексту завершився з помилкою.

with Session("host", "port") as session:  
 raise Exception("OH NO!")

В результаті ви побачите у консолі:

connected to host:port  
Some error!  
Traceback (most recent call last):  
 File "<stdin>", line 2, in <module>  
 raise Exception("OH NO!")  
Exception: OH NO!

На виході з контексту обов'язково буде виконано \_\_exit\_\_ і, навіть якщо помилка трапилася, ми можемо щось зробити (наприклад, встановити self.connected = False). Метод \_\_exit\_\_ не дозволяє перехоплювати винятки, він потрібен лише для того, щоб правильно завершити контекст (закрити відкриті файли та з'єднання, повернути ресурси системі тощо).

**Створення об'єкта ітератора/генератора**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson11/lesson-11#%D1%81%D1%82%D0%B2%D0%BE%D1%80%D0%B5%D0%BD%D0%BD%D1%8F-%D0%BE%D0%B1%D1%94%D0%BA%D1%82%D0%B0-%D1%96%D1%82%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80%D0%B0%D0%B3%D0%B5%D0%BD%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80%D0%B0)

Протокол ітератора у Python реалізований за допомогою методу \_\_iter\_\_. Цей метод повинен повертати ітератор. Ітератором може бути будь-який об'єкт, у якого є метод \_\_next\_\_, який за кожного виклику повертає значення. Щоб створити ітератор, достатньо реалізувати метод \_\_next\_\_.

Наприклад, створимо клас, яким можна ітеруватися Iterable та клас ітератор:

class Iterable:  
 MAX\_VALUE = 10  
 def \_\_init\_\_(self):  
 self.current\_value = 0  
  
 def \_\_next\_\_(self):  
 if self.current\_value < self.MAX\_VALUE:  
 self.current\_value += 1  
 return self.current\_value  
 raise StopIteration  
  
  
class CustomIterator:  
 def \_\_iter\_\_(self):  
 return Iterable()  
  
  
c = CustomIterator()  
for i in c:  
 print(i)

Зверніть увагу, що метод \_\_next\_\_ повинен викликати виняток StopIteration, щоб вказати, що ітерування завершено, інакше цикл for за таким об'єктом буде нескінченний.

**Інкапсуляція у Python (property, setter).**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson11/lesson-11#%D1%96%D0%BD%D0%BA%D0%B0%D0%BF%D1%81%D1%83%D0%BB%D1%8F%D1%86%D1%96%D1%8F-%D1%83-python-property-setter)

У Python неможливо інкапсулювати (зробити недоступними) атрибути класу. Ви завжди можете отримати доступ до будь-якого атрибуту. Щоб якось вказати розробнику, що доступ до атрибута напряму небажаний, прийнято називати такі поля або методи, починаючи з одного нижнього підкреслення. Якщо ж назвати атрибут так, що спочатку буде два нижні підкреслення, то включиться механізм "приховування" імен. Це не означає, що доступ до цього поля буде закрито, просто дещо ускладнений.

class Secret:  
 public\_field = 'this is public'  
 \_private\_field = 'avoid using this please'  
 \_\_real\_secret = 'I am hidden'  
  
s = Secret()  
print(s.public\_field) *# this is public*  
print(s.\_private\_field) *# avoid using this please*  
print(s.\_Secret\_\_real\_secret) *# I am hidden*

Як видно з цього прикладу, доступу за допомогою s.\_\_real\_secret немає, але можна отримати доступ до цього самого поля через s.\_Secret\_\_real\_secret, що загалом нічого не захищає.

Цей механізм можна використовувати для реалізації механізму setter та getter. Буває виникає необхідність перевірити, що користувач хоче записати в поле. Для цього можна написати окремий метод, який буде перед збереженням значення в полі реалізовувати перевірку, але саме поле, як і раніше, залишиться доступним. Можна ж скористатися декоратором setter. Для обчислення значення "на льоту" або як пару для setter можна скористатися декоратором property, який перетворює будь-який метод на поле. Наприклад, ми хочемо перевірити, що користувач вводить лише додатні числа.

class PositiveNumber:  
 def \_\_init\_\_(self):  
 self.\_\_value = None  
  
 @property  
 def value(self):  
 return self.\_\_value  
  
 @value.setter  
 def value(self, new\_value):  
 if new\_value > 0:  
 self.\_\_value = new\_value  
 else:  
 print('Only numbers greater zero accepted')  
  
  
p = PositiveNumber()  
p.value = 1  
print(p.value) *# 1*  
p.value = -1 *# Only numbers greater zero accepted*  
p.\_PositiveNumber\_\_value = -1  
print(p.value) *# -1s*

У цьому прикладі поле \_\_value можна вважати прихованим, воно певною мірою інкапсульовано. Проте значення в цьому полі може бути отримане і модифіковане напряму. Ще декоратор property зручний, коли значення у полі потрібно обчислювати у момент звернення.

**Перевизначення математичних операторів**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson11/lesson-11#%D0%BF%D0%B5%D1%80%D0%B5%D0%B2%D0%B8%D0%B7%D0%BD%D0%B0%D1%87%D0%B5%D0%BD%D0%BD%D1%8F-%D0%BC%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D1%87%D0%BD%D0%B8%D1%85-%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80%D1%96%D0%B2)

Усі математичні оператори можна перевизначити. Для цього є методи, відповідальні за кожний оператор:

* \_\_add\_\_ додавання
* \_\_sub\_\_ віднімання
* \_\_mul\_\_ множення
* \_\_div\_\_ ділення
* \_\_pow\_\_ піднесення до степеня

та інші. Перевизначення математичних операторів може стати зручним інструментом. Наприклад, створимо клас словників, які підтримують операції додавання та віднімання:

from collections import UserDict  
  
  
class MyDict(UserDict):  
 def \_\_add\_\_(self, other):  
 self.data.update(other)  
 return self  
  
 def \_\_sub\_\_(self, other):  
 for key in other:  
 if key in other:  
 self.data.pop(key)  
 return self  
  
  
d1 = MyDict({1: 'a', 2: 'b'})  
d2 = MyDict({3: 'c', 4: 'd'})  
  
d3 = d1 + d2  
print(d3) *# {1: 'a', 2: 'b', 3: 'c', 4: 'd'}*  
  
d4 = d3 - d2  
print(d4) *# {1: 'a', 2: 'b'}*

Синтаксис простий і код досить виразний, але потрібно бути акуратним з перевизначенням математичних операторів, зазвичай така поведінка неочевидна і може навпаки заплутати.

**Перевизначення операцій порівняння**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson11/lesson-11#%D0%BF%D0%B5%D1%80%D0%B5%D0%B2%D0%B8%D0%B7%D0%BD%D0%B0%D1%87%D0%B5%D0%BD%D0%BD%D1%8F-%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D1%96%D0%B9-%D0%BF%D0%BE%D1%80%D1%96%D0%B2%D0%BD%D1%8F%D0%BD%D0%BD%D1%8F)

Операції порівняння, як і інші оператори, мають свої "магічні" методи:

* \_\_eq\_\_(self, other) — визначає поведінку під час перевірки на відповідність (==).
* \_\_ne\_\_(self, other) — визначає поведінку під час перевірки на невідповідність. !=.
* \_\_lt\_\_(self, other) — визначає поведінку під час перевірки на менше <.
* \_\_gt\_\_(self, other) — визначає поведінку під час перевірки на більше >.
* \_\_le\_\_(self, other) — визначає поведінку під час перевірки на менше-дорівнює <=.
* \_\_ge\_\_(self, other) — визначає поведінку під час перевірки на більше-дорівнює >=.

Якщо вам потрібно, щоб ваш об'єкт був порівнянний, ви можете реалізувати ці шість методів і тоді будь-яка перевірка на порівняння працюватиме:

class Point:  
 def \_\_init\_\_(self, x, y):  
 self.x = x  
 self.y = y  
  
 def \_\_eq\_\_(self, other):  
 return self.x == other.x and self.y == other.y  
  
 def \_\_ne\_\_(self, other):  
 return self.x != other.x or self.y != other.y  
  
 def \_\_lt\_\_(self, other):  
 return self.x < other.x and self.y < other.y  
  
 def \_\_gt\_\_(self, other):  
 return self.x > other.x and self.y > other.y  
  
 def \_\_le\_\_(self, other):  
 return self.x <= other.x and self.y <= other.y  
  
 def \_\_ge\_\_(self, other):  
 return self.x >= other.x and self.y >= other.y  
  
  
Point(0, 0) == Point(0, 0) *# True*  
Point(0, 0) != Point(0, 0) *# False*  
Point(0, 0) < Point(1, 0) *# False*  
Point(0, 0) > Point(0, 1) *# False*  
Point(0, 2) >= Point(0, 1) *# True*  
Point(0, 0) <= Point(0, 0) *# True*

**Серіалізація об'єктів Python**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson12/lesson-12#%D1%81%D0%B5%D1%80%D1%96%D0%B0%D0%BB%D1%96%D0%B7%D0%B0%D1%86%D1%96%D1%8F-%D0%BE%D0%B1%D1%94%D0%BA%D1%82%D1%96%D0%B2-python)

Серіалізація (у програмуванні) — процес перетворення будь-якої структури даних у послідовність байтів. Зворотна до операції серіалізації є операція десеріалізації (структуризації) — відновлення початкового стану структури даних із бітової послідовності.

Завжди, коли ви хочете зберегти інформацію для подальшого використання в зрозумілої машині (комп'ютеру) формі, ви робите серіалізацію. Найочевидніший приклад — це збереження даних у текстовий файл. Ви можете зберегти, наприклад, перелік витрат у текстовий файл:

expenses = {  
 "hotel": 150,  
 "breakfast": 30,  
 "taxi": 15,  
 "lunch": 20  
}  
  
  
file\_name = "expenses.txt"  
with open(file\_name, "w") as fh:  
 for key, value in expenses.items():  
 fh.write(f"{key}|{value}\n")

Цей файл буде повністю читабельним і, якщо вам знадобиться потім завантажити цей перелік назад у Python, ви завжди зможете це зробити:

file\_name = "expenses.txt"  
expenses = {}  
with open(file\_name, "r") as fh:  
 raw\_expenses = fh.readlines()  
 for line in raw\_expenses:  
 key, value = line.split("|")  
 expenses[key] = int(value)

У цьому примітивному прикладі ми серіалізували та десеріалізували словник expenses. Такий підхід цілком виправданий, особливо, якщо вам корисно зберегти людське подання інформації.

Однак, далеко не завжди такий підхід виправданий. Зауважте, що у цьому прикладі довелося винайти свій протокол серіалізації. А саме:

* символ нового рядка вказує на початок нової пари ключ-значення;
* символ | розділяє ключ і значення
* значення потрібно перетворити з рядкового представлення на числову форму.

Для роботи з даними, серіалізованими за цим протоколом, необхідно буде пам'ятати про нього та розширювати протокол за потребою.

Можна піти іншим шляхом та скористатися одним із стандартних протоколів. За такого підходу ви значно економите, використовуючи готове та протестоване рішення. Найчастіше у Python використовуються три підходи до упакування та розпакування об'єктів:

* об'єкти, що мають рядкове представлення, приводяться до рядків і використовуються в такому вигляді (як ми зробили у прикладі вище);
* вбудований пакет pickle дозволяє працювати з вбудованими типами (словники, списки, кортежі, рядки, множини та ін.) і навіть з нескладними класами;
* протокол JSON підтримується Python і з невеликими обмеженнями дозволяє працювати з рядками, числами, списками, кортежами та словниками.

**Серіалізація об'єктів Python за допомогою pickle**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson12/lesson-12#%D1%81%D0%B5%D1%80%D1%96%D0%B0%D0%BB%D1%96%D0%B7%D0%B0%D1%86%D1%96%D1%8F-%D0%BE%D0%B1%D1%94%D0%BA%D1%82%D1%96%D0%B2-python-%D0%B7%D0%B0-%D0%B4%D0%BE%D0%BF%D0%BE%D0%BC%D0%BE%D0%B3%D0%BE%D1%8E-pickle)

Для серіалізації/десеріалізації об'єктів Python, коли важлива швидкість, коректність і невеликий розмір пам'яті, що використовується, найкраще підійде пакет pickle.

У пакета pickle є дві пари парних методів:

**упакування у byte-рядки та розпакування із byte-рядків**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson12/lesson-12#%D1%83%D0%BF%D0%B0%D0%BA%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F-%D1%83-byte-%D1%80%D1%8F%D0%B4%D0%BA%D0%B8-%D1%82%D0%B0-%D1%80%D0%BE%D0%B7%D0%BF%D0%B0%D0%BA%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F-%D1%96%D0%B7-byte-%D1%80%D1%8F%D0%B4%D0%BA%D1%96%D0%B2)

dumps запаковує в byte-рядок об'єкт, loads розпаковує з byte-рядок в об'єкт. Ці методи потрібні, коли ми хочемо контролювати, що робити з byte представленням, наприклад, відправити його мережею або прийняти з мережі.

import pickle  
  
  
some\_data = {  
 (1, 3.5): 'tuple',  
 2: [1, 2, 3],  
 'a': {'key': 'value'}  
}  
  
byte\_string = pickle.dumps(some\_data)  
unpacked = pickle.loads(byte\_string)  
  
print(unpacked == some\_data) *# True*  
print(unpacked is some\_data) *# False*

У цьому прикладі упакований у byte\_string словник some\_data розпакований в unpacked та unpacked суворо дорівнює some\_data, але це все ж таки не той самий об'єкт.

**упакування у відкритий файл та розпакування з файлу**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson12/lesson-12#%D1%83%D0%BF%D0%B0%D0%BA%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F-%D1%83-%D0%B2%D1%96%D0%B4%D0%BA%D1%80%D0%B8%D1%82%D0%B8%D0%B9-%D1%84%D0%B0%D0%B9%D0%BB-%D1%82%D0%B0-%D1%80%D0%BE%D0%B7%D0%BF%D0%B0%D0%BA%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F-%D0%B7-%D1%84%D0%B0%D0%B9%D0%BB%D1%83)

dump, load упаковує у відкритий для byte-запису файл та розпаковує із відкритого для byte-читання файлу.

import pickle  
  
  
some\_data = {  
 (1, 3.5): 'tuple',  
 2: [1, 2, 3],  
 'a': {'key': 'value'}  
}  
  
file\_name = 'data.bin'  
  
with open(file\_name, "wb") as fh:  
 pickle.dump(some\_data, fh)  
  
  
with open(file\_name, "rb") as fh:  
 unpacked = pickle.load(fh)  
  
  
print(unpacked == some\_data) *# True*  
print(unpacked is some\_data) *# False*

Результат аналогічний попередньому прикладу. Головна відмінність у тому, що під час виконання цього коду в робочій папці з'явився файл data.bin.

**Робота з класами користувача**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson12/lesson-12#%D1%80%D0%BE%D0%B1%D0%BE%D1%82%D0%B0-%D0%B7-%D0%BA%D0%BB%D0%B0%D1%81%D0%B0%D0%BC%D0%B8-%D0%BA%D0%BE%D1%80%D0%B8%D1%81%D1%82%D1%83%D0%B2%D0%B0%D1%87%D0%B0)

Ви можете зберігати об'єкти для подальшого використання, але з умовою. Самі класи та функції pickle зберігати не вміє і, якщо вам потрібно розпакувати упакований об'єкт класу, то сам клас повинен бути оголошений раніше у коді.

import pickle  
  
  
class Human:  
 def \_\_init\_\_(self, name):  
 self.name = name  
  
  
bob = Human("Bob")  
encoded\_bob = pickle.dumps(bob)  
  
decoded\_bob = pickle.loads(encoded\_bob)  
  
bob.name == decoded\_bob.name *# True*

Але, якби ви захотіли передати об'єкт bob мережею іншому комп'ютеру, який нічого не знає про клас Human, то ви отримаєте помилку. Якщо ж на обох кінцях каналу зв'язку оголошено клас Human, то такий обмін працюватиме.

**Серіалізація об'єктів Python за допомогою json**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson12/lesson-12#%D1%81%D0%B5%D1%80%D1%96%D0%B0%D0%BB%D1%96%D0%B7%D0%B0%D1%86%D1%96%D1%8F-%D0%BE%D0%B1%D1%94%D0%BA%D1%82%D1%96%D0%B2-python-%D0%B7%D0%B0-%D0%B4%D0%BE%D0%BF%D0%BE%D0%BC%D0%BE%D0%B3%D0%BE%D1%8E-json)

Протокол JSON (розшифровується як JavaScript Object Notation) — дуже популярний в інтернеті протокол передачі. Цей протокол має низку переваг:

* простий, його легко реалізувати;
* читабельний;
* відносно компактний (є набагато економніші протоколи).

Перша перевага зробила JSON універсальним, будь-яка сучасна мова програмування підтримує JSON. А якщо ні, то ви самі можете легко реалізувати підтримку JSON.

Недоліки у JSON теж є:

* обмежений набір типів;
* ресурсомісткий (є й вимогливіші до ресурсів протоколи).

JSON підтримує такі типи даних:

* запис (як словник у Python), в якості ключа можуть бути **тільки** рядки, значення — будь-який JSON тип;
* масив (як список у Python);
* число (немає різниці між цілими або дробовими);
* літерал (True, False, None);
* рядок.

Як і у Python запис та масив можуть містити вкладені записи та/або словники будь-якої глибини вкладеності.

Варто бути обережним з конвертацією типів під час роботи з JSON у Python. Кортежі під час розпакування з JSON стають списками, ключі словника, якщо вони були числами, стають рядками ж.

Python підтримує JSON і в стандартному постачанні є пакет json, в якому є все необхідне для роботи з JSON.

**упакування у byte-рядки та розпакування із byte-рядків**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson12/lesson-12#%D1%83%D0%BF%D0%B0%D0%BA%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F-%D1%83-byte-%D1%80%D1%8F%D0%B4%D0%BA%D0%B8-%D1%82%D0%B0-%D1%80%D0%BE%D0%B7%D0%BF%D0%B0%D0%BA%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F-%D1%96%D0%B7-byte-%D1%80%D1%8F%D0%B4%D0%BA%D1%96%D0%B2-1)

dumps запаковує в byte-рядок об'єкт, loads розпаковує з byte-рядка в об'єкт. Ці методи потрібні, коли ми хочемо контролювати, що робити з byte представленням, наприклад, відправити його мережею або прийняти з мережі.

import json  
  
  
some\_data = {'key': 'value', 2: [1, 2, 3], 'tuple': (5, 6), 'a': {'key': 'value'}}  
  
byte\_string = json.dumps(some\_data)  
unpacked = json.loads(byte\_string)  
  
unpacked is some\_data *# False*  
unpacked == some\_data *# False*  
  
unpacked['key'] == some\_data['key'] *# True*  
unpacked['a'] == some\_data['a'] *# True*  
unpacked['2'] == some\_data[2] *# True*  
unpacked['tuple'] == [5, 6] *# True*

В цьому прикладі запакований в byte\_string словник some\_data розпакований в unpacked та unpacked не дорівнює some\_data, але це все ж таки не той самий об'єкт. Окрім того:

* ключ 2 був неявно перетворений на '2';
* кортеж (5, 6) — у список [5, 6]

**упакування у відкритий файл та розпакування з файлу**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson12/lesson-12#%D1%83%D0%BF%D0%B0%D0%BA%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F-%D1%83-%D0%B2%D1%96%D0%B4%D0%BA%D1%80%D0%B8%D1%82%D0%B8%D0%B9-%D1%84%D0%B0%D0%B9%D0%BB-%D1%82%D0%B0-%D1%80%D0%BE%D0%B7%D0%BF%D0%B0%D0%BA%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F-%D0%B7-%D1%84%D0%B0%D0%B9%D0%BB%D1%83-1)

dump, load упаковує у відкритий для byte-запису файл та розпаковує із відкритого для byte-читання файлу.

import json  
  
  
some\_data = {'key': 'value', 2: [1, 2, 3], 'tuple': (5, 6), 'a': {'key': 'value'}}  
file\_name = 'data.json'  
  
with open(file\_name, "w") as fh:  
 json.dump(some\_data, fh)  
  
  
with open(file\_name, "r") as fh:  
 unpacked = json.load(fh)  
  
  
unpacked is some\_data *# False*  
unpacked == some\_data *# False*  
  
unpacked['key'] == some\_data['key'] *# True*  
unpacked['a'] == some\_data['a'] *# True*  
unpacked['2'] == some\_data[2] *# True*  
unpacked['tuple'] == [5, 6] *# True*

Результат аналогічний попередньому прикладу. Головна відмінність у тому, що під час виконання цього коду в робочій папці з'явився файл data.json.

**Робота з таблицями CSV у Python**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson12/lesson-12#%D1%80%D0%BE%D0%B1%D0%BE%D1%82%D0%B0-%D0%B7-%D1%82%D0%B0%D0%B1%D0%BB%D0%B8%D1%86%D1%8F%D0%BC%D0%B8-csv-%D1%83-python)

Ще один формат обміну інформацією, що дуже часто використовується — це табличне представлення. Відкритий формат для зберігання табличних даних, який підтримується будь-яким редактором — це формат csv. Формат csv є, по суті, тим самим текстовим файлом, але з умовою, що вся інформація в ньому розділена на колонки та рядки символами роздільниками. За замовчуванням колонки розділяють комою, а рядки — символом нового рядка. Але можна використати будь-яку іншу комбінацію символів.

Python підтримує роботу з табличними даними у форматі csv. Для цього у стандартному постачанні йде пакет csv.

import csv  
  
  
with open('eggs.csv', 'w', newline='') as fh:  
 spam\_writer = csv.writer(fh)  
 spam\_writer.writerow(['Spam'] \* 5 + ['Baked Beans'])  
 spam\_writer.writerow(['Spam', 'Lovely Spam', 'Wonderful Spam'])  
  
  
with open('eggs.csv', newline='') as fh:  
 spam\_reader = csv.reader(fh)  
 for row in spam\_reader:  
 print(', '.join(row))

В результаті виконання цього коду в робочій папці з'явився файл *eggs.csv*. Якщо відкриєте його табличним редактором, він відкриється як таблиця.

Є два допоміжні класи в пакеті csv, які виконують роботу з табличними даними трохи зручніше:

import csv  
  
  
with open('names.csv', 'w', newline='') as fh:  
 field\_names = ['first\_name', 'last\_name']  
 writer = csv.DictWriter(fh, fieldnames=field\_names)  
 writer.writeheader()  
 writer.writerow({'first\_name': 'Baked', 'last\_name': 'Beans'})  
 writer.writerow({'first\_name': 'Lovely', 'last\_name': 'Spam'})  
 writer.writerow({'first\_name': 'Wonderful', 'last\_name': 'Spam'})  
  
  
with open('names.csv', newline='') as fh:  
 reader = csv.DictReader(fh)  
 for row in reader:  
 print(row['first\_name'], row['last\_name'])

Класи DictWriter та DictReader дозволяють працювати з рядками таблиці як зі словниками, де як ключі використовуються назви колонок (перший рядок).

Таким чином за допомогою Python можна генерувати табличні дані та імпортувати дані з таблиць.

**Управління порядком серіалізації**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson12/lesson-12#%D1%83%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D1%96%D0%BD%D0%BD%D1%8F-%D0%BF%D0%BE%D1%80%D1%8F%D0%B4%D0%BA%D0%BE%D0%BC-%D1%81%D0%B5%D1%80%D1%96%D0%B0%D0%BB%D1%96%D0%B7%D0%B0%D1%86%D1%96%D1%97)

Не всі об'єкти Python можна серіалізувати. Наприклад, не можна серіалізувати файловий дескриптор або системний ресурс. Але що робити, коли у вас є клас, об'єкт якого ви хочете запакувати, використовуючи pickle, але у нього є атрибути, що не серіалізуються? У такій ситуації ви можете скористатися магічними методами, які управляють серіалізацією та десеріалізацією за допомогою pickle.

**Упакування за допомогою \_\_getstate\_\_**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson12/lesson-12#%D1%83%D0%BF%D0%B0%D0%BA%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F-%D0%B7%D0%B0-%D0%B4%D0%BE%D0%BF%D0%BE%D0%BC%D0%BE%D0%B3%D0%BE%D1%8E-__getstate__)

Магічний метод \_\_getstate\_\_ викликається, коли pickle намагається отримати представлення об'єкта у вигляді byte-рядка. У звичайній реалізації \_\_getstate\_\_ повертає \_\_dict\_\_ словник, де зберігаються всі атрибути класу. Але ви можете змінити цей метод.

import pickle  
  
  
class Reader:  
 def \_\_init\_\_(self, file):  
 self.file = file  
 self.fh = open(self.file)  
 self.position = 0  
  
 def close(self):  
 self.fh.close()  
  
 def read(self, size=1):  
 data = self.fh.read(size)  
 self.position = self.fh.tell()  
 return data  
  
 def \_\_getstate\_\_(self):  
 attributes = self.\_\_dict\_\_.copy()  
 attributes['fh'] = None  
 return attributes

У цьому прикладі клас Reader можна серіалізувати, помилки через неможливість упакувати файловий дескриптор не виникне.

**Розпакування за допомогою \_\_setstate\_\_**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson12/lesson-12#%D1%80%D0%BE%D0%B7%D0%BF%D0%B0%D0%BA%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F-%D0%B7%D0%B0-%D0%B4%D0%BE%D0%BF%D0%BE%D0%BC%D0%BE%D0%B3%D0%BE%D1%8E-__setstate__)

Цей магічний метод отримує на вхід словник, розпакований з файлу або byte-рядка. Поведінка за замовчуванням — це записати отримане значення в self.\_\_dict\_\_. Доопрацюймо клас Reader так, щоб він міг після розпакування продовжити читання з того самого місця.

import pickle  
  
  
class Reader:  
 def \_\_init\_\_(self, file):  
 self.file = file  
 self.fh = open(self.file)  
 self.position = 0  
  
 def close(self):  
 self.fh.close()  
  
 def read(self, size=1):  
 data = self.fh.read(size)  
 self.position = self.fh.tell()  
 return data  
  
 def \_\_getstate\_\_(self):  
 attributes = {\*\*self.\_\_dict\_\_}  
 attributes['fh'] = None  
 return attributes  
  
 def \_\_setstate\_\_(self, value):  
 self.\_\_dict\_\_ = value  
 self.fh = open(value['file'])  
 self.fh.seek(value['position'])

**Створення копій об'єктів Python**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson12/lesson-12#%D1%81%D1%82%D0%B2%D0%BE%D1%80%D0%B5%D0%BD%D0%BD%D1%8F-%D0%BA%D0%BE%D0%BF%D1%96%D0%B9-%D0%BE%D0%B1%D1%94%D0%BA%D1%82%D1%96%D0%B2-python)

Python намагається заощаджувати пам'ять і не копіювати дані з однієї області пам'яті в іншу. Натомість інтерпретатор створює нове посилання (ще один псевдонім) на існуючий об'єкт, замість копіювання вмісту. Така поведінка може бути небажаною, наприклад:

my\_list = [1, 2, 3]  
copy\_list = my\_list  
copy\_list.append(4)  
print(my\_list) *# [1, 2, 3, 4]*

Виходить, що copy\_list — це просто ще одне ім'я для того самого списку my\_list і, змінюючи copy\_list, ми змінюємо й my\_list. Це неочевидно і може збивати з пантелику.

Така поведінка може призводити до помилок, коли справа стосується типів, словників, списків, класів користувача, що змінюються. Для списків та словників можна скористатися явним копіюванням:

my\_list = [1, 2, 3]  
copy\_list = my\_list[:]  
copy\_list.append(4)  
print(my\_list) *# [1, 2, 3]*  
  
d = {1: 'a'}  
d\_copy = {\*\*d}  
d\_copy[2] = 'b'  
print(d) *# {1: 'a'}*

Але з типами користувача так не зробиш. Щоб вирішити цю проблему, у Python є механізм копіювання — це функції із пакету copy.

**Створення поверхневих копій об'єктів Python**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson12/lesson-12#%D1%81%D1%82%D0%B2%D0%BE%D1%80%D0%B5%D0%BD%D0%BD%D1%8F-%D0%BF%D0%BE%D0%B2%D0%B5%D1%80%D1%85%D0%BD%D0%B5%D0%B2%D0%B8%D1%85-%D0%BA%D0%BE%D0%BF%D1%96%D0%B9-%D0%BE%D0%B1%D1%94%D0%BA%D1%82%D1%96%D0%B2-python)

Щоб створити "поверхневу" копію об'єкта, у пакеті copy є функція copy. Ця функція створює новий об'єкт такого самого типу і потім створює посилання на увесь вміст старого об'єкта в новий. Такий механізм досить хороший для роботи з об'єктами, де вже на першому рівні вкладеності немає змінних об'єктів, і він працює досить швидко. Але для об'єктів із глибокою вкладеністю така функція все ж таки не дасть потрібного ефекту:

import copy  
  
  
my\_list = [1, 2, {1: 'a'}]  
copy\_list = copy.copy(my\_list)  
copy\_list.append(4)  
print(my\_list) *# [1, 2, {1: 'a'}]*  
print(copy\_list) *# [1, 2, {1: 'a'}, 4]*  
  
copy\_list[2][2] = 'b'  
print(my\_list) *# [1, 2, {1: 'a', 2: 'b'}]*

З цього прикладу видно, що хоча copy\_list вже є новим об'єктом (не посилання на my\_list), але вкладений у нього словник з індексом 2 — це один і той самий словник і в copy\_list, і в my\_list.

**Створення глибоких копій об'єктів Python**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson12/lesson-12#%D1%81%D1%82%D0%B2%D0%BE%D1%80%D0%B5%D0%BD%D0%BD%D1%8F-%D0%B3%D0%BB%D0%B8%D0%B1%D0%BE%D0%BA%D0%B8%D1%85-%D0%BA%D0%BE%D0%BF%D1%96%D0%B9-%D0%BE%D0%B1%D1%94%D0%BA%D1%82%D1%96%D0%B2-python)

Для ситуацій, коли нам потрібно, щоб на будь-якому рівні вкладеності створювалися нові об'єкти, а не посилання на існуючі, у пакеті copy є функція deepcopy. Ця функція рекурсивно створює нові об'єкти.

import copy  
  
  
my\_list = [1, 2, {1: 'a'}]  
copy\_list = copy.deepcopy(my\_list)  
copy\_list.append(4)  
print(my\_list) *# [1, 2, {1: 'a'}]*  
print(copy\_list) *# [1, 2, {1: 'a'}, 4]*  
  
copy\_list[2][2] = 'b'  
print(my\_list) *# [1, 2, {1: 'a'}]*

**Управління порядком копіювання об'єктів Python**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/lesson12/lesson-12#%D1%83%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D1%96%D0%BD%D0%BD%D1%8F-%D0%BF%D0%BE%D1%80%D1%8F%D0%B4%D0%BA%D0%BE%D0%BC-%D0%BA%D0%BE%D0%BF%D1%96%D1%8E%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F-%D0%BE%D0%B1%D1%94%D0%BA%D1%82%D1%96%D0%B2-python)

Ще одна проблема вирішується за допомогою пакету copy — це копіювання об'єктів користувача. Щоб створити об'єкт, який буде коректно оброблятися функціями copy та deepcopy, ваш клас повинен реалізувати два магічних методи: \_\_copy\_\_ та \_\_deepcopy\_\_ для поверхневого та глибокого копіювання відповідно.

from copy import deepcopy, copy  
  
  
class Expenses:  
 def \_\_init\_\_(self):  
 self.data = {}  
 self.places = []  
  
 def spent(self, place, value):  
 self.data[str(place)] = value  
 self.places.append(place)  
  
 def \_\_copy\_\_(self):  
 copy\_obj = Expenses()  
 copy\_obj.data = self.data  
 copy\_obj.places = self.places  
 return copy\_obj  
  
 def \_\_deepcopy\_\_(self, memo):  
 copy\_obj = Expenses()  
 memo[id(self)] = copy\_obj  
 copy\_obj.data = deepcopy(self.data, memo)  
 copy\_obj.places = deepcopy(self.places, memo)  
 return copy\_obj  
  
  
e = Expenses()  
e.spent('hotel', 100)  
e.spent('taxi', 10)  
print(e.places) *# ['hotel', 'taxi']*  
  
e\_copy = copy(e)  
print(e\_copy is e) *# False*  
e\_copy.spent('bar', 30)  
print(e.places) *# ['hotel', 'taxi', 'bar']*  
  
e\_deep\_copy = deepcopy(e)  
print(e\_deep\_copy is e) *# False*  
e\_deep\_copy.spent(  
 'airport',  
 300  
)  
print(e.places) *# ['hotel', 'taxi', 'bar']*  
print(e\_deep\_copy.places) *# ['hotel', 'taxi', 'bar', 'airport']*

Використовуючи методи \_\_copy\_\_ та \_\_deepcopy\_\_, ви можете керувати як саме буде створюватися копія вашого об'єкта. Метод \_\_deepcopy\_\_ обов'язково повинен приймати один аргумент — словник, в який записуються усі об'єкти, які піддаються копіюванню. Це потрібно, щоб уникнути нескінченної рекурсії, якщо якийсь об'єкт є спільним для кількох копійованих. У такому випадку алгоритм глибокого копіювання може зайти в нескінченний цикл, копіюючи поперемінно об'єкти із посиланнями один на одного. Словник memo зберігає як ключі id об'єктів і самі об'єкти як значення. Коли перевизначаємо як повинно відбуватися копіювання, ми можемо і не використовувати memo, якщо точно знаємо, що рекурсії не виникне.

# Єдина точка входу в проект

## Використання main.py як єдиної точки входу[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/additional/main#%D0%B2%D0%B8%D0%BA%D0%BE%D1%80%D0%B8%D1%81%D1%82%D0%B0%D0%BD%D0%BD%D1%8F-mainpy-%D1%8F%D0%BA-%D1%94%D0%B4%D0%B8%D0%BD%D0%BE%D1%97-%D1%82%D0%BE%D1%87%D0%BA%D0%B8-%D0%B2%D1%85%D0%BE%D0%B4%D1%83)

main.py це файл, який є точкою входу для вашого проекту. Це означає, що коли ви запускаєте ваш проект, інтерпретатор Python починає виконувати код цього файлу. Це може бути корисним, тому що вам не потрібно запускати кожен файл окремо, а можете використовувати одну точку входу для запуску всієї програми.

Щоб об'єднати файли в проекті, можна імпортувати функції або класи з інших файлів у main.py. Це може виглядати так:

from my\_module import my\_function  
  
  
def main():  
 my\_function()  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

У цьому прикладі ми імпортуємо функцію my\_function з файлу my\_module.py і викликаємо її в функції main(). Потім ми перевіряємо, чи запущено цей файл як основний скрипт \_\_name\_\_ == "\_\_main\_\_", і якщо так, ми запускаємо функціюmain(). Це дозволяє використовувати цей файл як модуль для інших скриптів, не запускаючи функцію main(), якщо він імпортований як модуль.

Ви також можете використовувати main.py для об'єднання та організації вашого коду, наприклад, імпортуючи різні класи та функції з різних файлів та використовуючи їх для створення програмного інтерфейсу або для запуску різних функцій у певному порядку. Це може допомогти вам зберегти ваш код у порядку і легко знайти потрібний код у майбутньому.

## Приклад[​](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/additional/main#%D0%BF%D1%80%D0%B8%D0%BA%D0%BB%D0%B0%D0%B4)

Допустимо, у вас є простий проект, який складається з трьох файлів: main.py, data.py та processing.py.

data.py містить функцію для завантаження та обробки даних, processing.py містить функцію для аналізу даних, а main.py служить як точка входу для проекту.

У файлі main.py ви можете імпортувати функції з data.py та processing.py та використовувати їх для обробки даних та аналізу:

from data import load\_data  
from processing import analyze\_data  
  
  
def main():  
 data = load\_data()  
 analyze\_data(data)  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

У цьому прикладі функція load\_data з data.py використовується для завантаження та обробки даних, а функція analyze\_data з processing.py використовується для аналізу даних. Потім функція main викликається для запуску проекту.

Це простий приклад, але він демонструє, як ви можете використовувати main.py для об'єднання різних функцій та модулів в одному місці та організації вашого коду. Ви можете додавати додаткові функції та модулі до вашого проекту і використовувати main.py для об'єднання їх всього в одному місці. Це допоможе вам зберегти ваш код у порядку і легко знайти потрібний код у майбутньому.

ВИСНОВОК

Коротше кажучи, коли створюєш код, думай як через main.py ти шукатимеш свої файли в проекті, переходячи по ним як по сторінкам веб-сайту.

**Телеграм канали**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/additional/materials#%D1%82%D0%B5%D0%BB%D0%B5%D0%B3%D1%80%D0%B0%D0%BC-%D0%BA%D0%B0%D0%BD%D0%B0%D0%BB%D0%B8)

Перелік актуальних телеграм каналів з корисною інформацією:

1. [Python etc](https://t.me/pythonetc) – Канал з корисними прикладами коду
2. [Python Books](https://t.me/pythonbooks) – Книги з програмування для майбутнього Python Developera. Всі в PDF
3. [Python Textbooks](https://t.me/python_textbooks) – Велика база книг з Python
4. [Бібліотека Python розробника](https://t.me/BookPython) – Книги з програмування на Python
5. [Pythonist - освіта з пітону](https://t.me/pythonist_ru) – допомога в підготовці до співбесіди на позицію Python

**Книги**[**​**](https://textbook.edu.goit.global/python/core-pz9qu8/v1/uk/docs/additional/materials#%D0%BA%D0%BD%D0%B8%D0%B3%D0%B8)

Перелік актуальних книг для програміста початківця:

1. [Марк Лутц. Програмування на Python](https://www.amazon.com/Learning-Python-5th-Mark-Lutz/dp/1449355730) – Настільна книга python розробника-початківця. У двох томах.
2. [Python: Master the Art of Design Patterns](https://www.amazon.com/Python-Master-Art-Design-Patterns/dp/1787125181) – Дуже корисна на етапі освоєння ООП та патернів програмування.
3. [Python Tricks: The Book](https://www.amazon.com/Python-Tricks-Buffet-Awesome-Features-ebook/dp/B0785Q7GSY) – Може бути корисна на початковому етапі та коли є бажання і розуміння писати код елегантніше.
4. [Чистий Python](https://www.bookovka.ua/uk/piznavalna-literatura/19975-chistyj-python-tonkosti-programmirovaniya-dlya-profi.html) – аналог книги 'Python Tricks: The Book', але в перекладі на російську мову.
5. [Daniel Arbuckle's Mastering Python](https://www.amazon.com/Daniel-Arbuckles-Mastering-Python-Arbuckle/dp/1787283690) – Настільна книга. Корисна як на початковому, так і на просунутому рівні.
6. [A Byte of Python (Russian)](http://wombat.org.ua/AByteOfPython/AByteofPythonRussian-2.02.pdf) – Безкоштовна книга для новачка у Python. Дуже доступно і без зайвих подробиць.
7. [The Little Book of Python Anti-Patterns](https://docs.quantifiedcode.com/python-anti-patterns/) – Це невелика книга про антипатерни Python та найгірших практиках.