**Модуль 9: 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)**