**Модуль 11: «Магічні» методи**

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) 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**

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**

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**

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**

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**

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**

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).**

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**

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 self.data:

            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**

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