**Модуль 10: Основи роботи з класами**

**Визначення та створення класу**

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**

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)**

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**

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**

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).**

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).**

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**

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**

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 **animal** та 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).