Лекція 21

Об'єктно-орієнтоване програмування

Основні поняття про ООП

Об'єктно-орієнтоване програмування (ООП) – це спосіб організації програми, що дозволяє використовувати той самий код багаторазово.

ООП дозволяє:

розділити програму на фрагменти

описати предмети реального світу у вигляді об'єктів,

організувати зв'язки між цими об'єктами.

Кожен об'єкт належить до певного класу (типу), який задає поведінку об'єктів, створених на його основі.

Тому основною «цеглинкою» ООП є клас.

Мова Python - об'єктно-орієнтована мова програмування.

Це означає, що Python побудований з урахуванням наступних принципів:

1. Усі дані в ньому представлені як об'єкти.
2. Програму можна створити як набір взаємодіючих об'єктів, що посилають один одному повідомлення.
3. Кожний об'єкт має власну частину пам'яті і може складатися з інших об'єктів.
4. Кожний об'єкт має тип.
5. Усі об'єкти одного типу можуть приймати ті самі повідомлення (і виконувати ті самі дії).

Основні поняття

При процедурному програмуванні програму розбивають на частини відповідно до алгоритму: кожна частина є складовою частиною одного алгоритму.

При об'єктно-орієнтованому програмуванні програму будують як сукупність взаємодіючих об'єктів.

Об'єкт - це щось, що має
значення (стан),
тип (поведінка)
індивідуальність.

Визначити об'єкт в предметній області – це абстрагуватися від більшості його властивостей, концентруючись на істотних для задачі властивостях.

Стан і поведінка об’єкта

Об'єкти створюють на основі **класів**.
Клас є шаблоном об'єкта.
Усі об'єкти, які визначені користувачем, є екземплярами класу.
Об'єкт, створений на основі деякого класу, називають **екземпляром класу**.

Кожний об'єкт зберігає свій **стан, який визначається атрибутами.** Об'єкт також має певний набір **методів**.
Методи визначають **поведінку** об'єкта.
Об'єкти одного і того ж класу мають спільну поведінку.

Можуть одночасно існувати кілька екземплярів класу, створених на основі одного і того ж класу.

Формальні визначення

Об'єкт - це будь-яка сутність в Python
(число, рядок, список, кортеж, словник, функція, тобто ВСЕ).

```
123, "string", [1,2,3,4], (5,6,7),{"a":1,"b":2},
```

Клас це об'єкт, типом якого є тип type:

```
class MyClass:
    <атрибути>
    <методи>
print(type(MyClass))
Результат: <class 'type'>
```

Екземпляр деякого класу A - це об'єкт, у якого в атрибуті __class__ є посилання на клас A.

```
mc = MyClass()
print(c.__class__)
Результат: <class '__main__.MyClass'>
```

Визначення класу

У мові Python для визначення класу використовується оператор class:

Клас описують за допомогою ключового слова **class** за наступною схемою:

```
class <Назва класу>[(<Клас1>[, ... , <Класn>])]:

["""  Рядок документування """]

<Опис атрибутів>

<Опис методів>
```

`class` опертор класу

Назва класу повинна повністю відповідати правилам іменування змінних.

Круглі дужки після назви і двокрапка - ():

Після назви класу в круглих дужках можна вказати один або кілька базових класів через кому.

При відсутності базових класів дужки не вказують.

`<Опис атрибутів>` Змінні даного класу

`<Опис методів>` Функції даного класу

Функції мають обов'язковий параметр **`self`**

Отже клас визначає **тип** об'єкта, тобто його можливі стани й набір операцій.

Виконання інструкцій класу

Всі вирази всередині інструкції **class** виконуються при створенні класу, а не його екземпляра.

Приклад 1. Створення класу

```
class MyClass:

 """ Це рядок документування """
    A=20+10
    print("Інструкції виконуються відразу")
```

Цей приклад містить лише визначення класу `MyClass` і не створює екземпляр класу.

Як тільки потік виконання досягне інструкції **class**, повідомлення, зазначене у функції **print**(), буде відразу виведене.

Створення атрибутів і методів класу

Створення атрибута класу аналогічно створенню звичайної змінної.

Створення метода класу. Метод всередині класу створюється так само, як і звичайна функція, за допомогою інструкції **`def`**.

У методах класу перший параметр обов'язково слід указати явно, який автоматично передає посилання на екземпляр класу.

Загально прийнято цей параметр називати ім’ям **`self`**, хоча це й не обов'язково.

Доступ до атрибутів і методів класу всередині методу проводиться через змінну `self` за допомогою точкової нотації. до атрибута `x` з методу класу можна звернутися так: `self.x`.

Створення екземпляру класу

Щоб використовувати атрибути й методи класу, необхідно створити екземпляр класу згідно з наступним синтаксисом:

```
<Екземпляр класу> = <Назва класу>(<Параметри>])
                    a=MyClas()
```

Доступ до методів класу використовує такий формат:

```
<Екземпляр класу>.<Ім'я методу>(<Параметри>])
                  a.mymethod()
```

Доступ до атрибутів класу здійснюється аналогічно:

```
<Екземпляр класу>.<Ім'я атрибута>
                  a.myattribute
```

Посилання на екземпляр класу при доступі до атрибутів та методів класу інтерпретатор передає автоматично.

Приклад задавання класу

Визначимо клас `MyClass` з атрибутом `x` і методом `out_x()`, що виводять значення цього атрибута, а потім створимо екземпляр класу й викличемо метод.

Приклад 2.

```
class MyClass:
    x = 10    # Атрибут екземпляра класу
    def out_x(self):# self-це посилання на екземпляр класу
        print("Attribute x =",self.x)
c = MyClass ()  # Створення екземпляра класу
                # Викликаємо метод- out_x()
c.out_x()   # self не вказують при виклику методу
print("Stright print x =",c.x)
```

```
  Результат виконання:
  Atribut x = 10
  Stright print x = 10
```

Конструктор класу: метод `__init__()`

При створенні екземпляра класу інтерпретатор автоматично викликає метод ініціалізації **`__init__()`**. В інших мовах програмування такий метод прийнято називати конструктором класу.

Формат методу:

```
def__init__(self[,<Знач1>[,... , <Значn>]]):
    <Інструкції>
```

За допомогою методу `__init__()` можна присвоїти початкові значення атрибутам класу. При створенні екземпляра класу параметри цього методу вказують після імені класу в круглих дужках:

```
<Екз.класу>=<Ім'я класу>([<Знач1>[,...,<Значn>]])
```

Приклад використання методу __init__() :

Приклад 3.

```
#Створюємо клас
class MyClass:
    def __init__(self, value1, value2):#Конструктор
        self.x = value1
        self.y = value2
        print("Спрацював конструктор")
    print("Створено клас MyClass")
# Створюємо екземпляр класу
c = MyClass(100, 300)
print("Створено екземпляр класу MyClass")
print(c.x, c.y)
```

Результат виконання:

```
  Створено клас MyClass
  Спрацював конструктор
  Створено екземпляр класу MyClass
  100 300
```

Приклад класу `Person` з інформацією про людину

Приклад 4.

```
class Person:
    def __init__(self, name, job, pay):
        # Конструктор вимагає трьох аргументів
        self.name = name
        # Атрибути потрібно заповнити при створенні екземпляра
        self.job = job  # self – це новий екземпляр класу
        self.pay = pay
```

```
petro = Person("Petrenko Petro","Python developer", 10000)
print(petro.name, petro.job, petro.pay)
```

Ми бачимо, що екземпляр класу **`petro`** був успішно створений, оскільки всі параметри для методу **`__init__`** задані коректно.

Зробимо спробу створити екземпляр класу Person з неповною кількістю параметрів.

Приклад 5.

```
#Не заповнили job і name
ivan = Person("Ivanov Ivan")
print(ivan.name)
Результат виконання:
Traceback (most recent call last):
  TypeError: __init__() missing 2 required positional arguments: 'job' and 'pay'
```

При створенні екземпляра класу `ivan` виникло виключення `Typeerror`, оскільки не всі обов'язкові параметри були задані на етапі ініціалізації.

Для того, щоб уникнути таких помилок, деяким параметрам методу **`__init__`** можна **задати значення за замовчуванням:**

Приклад 5. Person з параметрами за замовчуванням

```
class Person:
    def __init__(self, name, job = None, pay = 0):
        # Конструктор не вимагає трьох аргументів
        self.name = name
        self.job = job
        self.pay = pay
petro = Person("Petrenko Petro","Python developer", 10000 )
print(petro.name, petro.job, petro.pay)
ivan = Person("Ivanov Ivan")
print(ivan.name, ivan.job, ivan.pay)
Результат виконання:
  Petrenko Petro Python developer 10000
  Ivanov Ivan None 0
```

Деструктор класу: метод `__del__()`

Перед знищенням об'єкта (екземпляра класу) автоматично викликається метод, називаний деструктором. У мові `Python` деструктор реалізується у вигляді визначеного методу **`__del__()`.**

Метод не буде викликатися, якщо на екземпляр класу існує хоча б одне посилання.

Інтерпретатор самостійно опікується видаленням об'єктів.

Тому використання деструктора в мові Python необхідно тільки в тому випадку, коли Ви плануєте спеціальний порядок видалення об'єкта

(наприклад, збереження значень його атрибутів).

Приклад 6. Приклад явного виклику деструктора

```
class MyClass:
    def __init__(self):
        print( "Виконано метод __init__() ")
    def __del__(self):   # Деструктор класу
        print("Виконано метод __del__()")
c1 = MyClass()
del c1
c2 = MyClass()
c3 = c2  # Створюємо посилання на екземпляр класу
del c2  #Нічого не виведе, оскільки існує посилання
print("C2 не видалено. Оскільки є C3")
del c3
```

Результат виконання:

```
Виконано метод __init__()
Виконано метод __del__()
Виконано метод __init__()
C2 не видалено. Оскільки є C3
Виконано метод __del__()
```

Системні атрибути та атрибути користувача

Атрибути об’єкта будемо розділяти на:

- системні атрибути,
- атрибути користувача.

Системні атрибути – це атрибути, які створюються Пайтоном.
Приклади системних атрибутів:
__dict__ , __class__ , __bases__ , __doc__,__module__

Атрибути користувача – це атрибути, які створює користувач під час створення класів.
Приклади атрибутів користувача:

```
class MyClass:
    my_attribute="Це атрибут користувача"
    value = 12
```

Атрибут dict
(відображення атрибутів екземпляру)

```
class Person:
    Inner= True
    def __init__(self, name, job = None, pay = 0):
        # Конструктор не вимагає трьох аргументів
        self.name = name
        self.job = job
        self.pay = pay
m= Person("Michael","programmer", 100)
print(m.__dict__)
```

Результат:

{'name': 'Michael', 'job': 'programmer', 'pay': 100}

- Атрибут Inner не належить до атрибутів екземпляру m

Атрибут dict
(відображення атрибутів класу)

```
class Person:
    Inner= True
    def __init__(self,name,job=None, pay = 0):
                self.name = name
        self.job = job
        self.pay = pay
m= Person("Michael","programmer", 100)
print(Person.__class__)
print(Person.__dict__)
print(m.__class__.__dict__)
<class 'type'>
{'__dict__': <attribute '__dict__' of 'Person' objects>, '__doc__': None,
'Inner': True, '__module__': '__main__', ……}
{'__dict__': <attribute '__dict__' of 'Person' objects>, '__doc__': None,
'Inner': True, '__module__': '__main__', ……}
```

Створення атрибутів для конкретного екземпляру та доступ до внутрішніх атрибутів класу

```
class Car:
    wheels=4
    engine=1

my=Car()
my.seats='Leather'
print("Атрибут екземпляру:", my.seats)
print("Атрибут класу:", my.wheels)
print(my.__dict__)
print(my.__class__.__dict__)
```

```
Результат:
Атрибут екземпляру: Leather
Атрибут класу: 4
{'seats': 'Leather'}
{'wheels': 4, 'engine': 1,…}
```