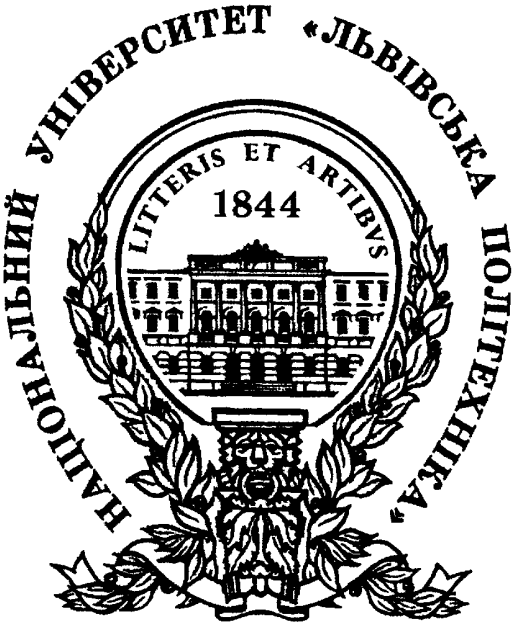
МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ЛЬВІВСЬКА ПОЛІТЕХНІКА»

Кафедра "Захист інформації"



Об'єктно-орієнтоване програмування

МЕТОДИЧНІ ВКАЗІВКИ
до лабораторної роботи № 9
з курсу «Програмування скриптовими мовами»
для студентів спеціальності
«Кібербезпека»

Затверджено
на засіданні кафедри
"Захист інформації"
протокол № 01 від 29.08.2024 р.

Львів – 2024

Об’єктно-орієнтоване програмування: Методичні вказівки до лабораторної роботи № 9 з курсу «Програмування скриптовими мовами» для студентів спеціальності «Кібербезпека» / Укл. *Я. Р. Совин* – Львів: Національний університет "Львівська політехніка", 2024. – 28 с.

Укладач: Я. Р. Совин, канд. техн. наук, доцент

Відповідальний за випуск: В. Б. Дудикевич, д.т.н., професор

Рецензенти: А. Я. Горпенюк, канд. техн. наук, доцент
Ю. Я. Наконечний, канд. техн. наук, доцент

Мета роботи – ознайомитись з реалізацією об’єктно-орієнтованого підходу в Python**.**

1. ТЕОРЕТИЧНІ ВІДОМОСТІ

1.1. Об’єктно-орієнтоване програмування

Об’єктно-орієнтоване програмування (ООП) – це спосіб організації програми, який дозволяє використовувати один і той же код багаторазово. На відміну від функцій і модулів, ООП дозволяє не тільки розділити програму на фрагменти, а й описати предмети реального світу у вигляді зручних сутностей – об'єктів, а також організувати зв'язки між цими об'єктами.

Базовими принципами ООП є інкапсуляція, успадкування та поліморфізм. Інкапсуляція - дозволяє взаємодіяти з абстракціями (об'єктами) за допомогою спеціального зовнішнього уявлення (інтерфейсу). Причому немає необхідності знати, як об'єкт утворений (що у нього всередині). Інтерфейс являє набір публічних (public) властивостей і методів, до яких користувач може звертатися. При цьому йому надається специфікація, тобто набір доступних атрибутів об'єкта.

Успадкування є принципом, який дозволяє створити новий набір об'єктів (новий клас) на базі існуючої абстракції (базовий клас). Об'єкти такої абстракції називають спадкоємцями або нащадками. Поряд з властивостями, що дісталися їм від батьків вони можуть свої специфічні властивості. При цьому немає необхідності описувати властивості батьківського класу, в новому класі досить вказати, що спадкоємець є нащадком класу вищого рівня. Таке спадкування називається простим. При цьому ієрархію можна встановлювати як завгодно великого рівня складності. Здебільшого використовують дворівневу вкладеність.

Поліморфізм - це принцип ООП, що має на увазі, що об'єкти, які мають однакові атрибути (специфікацію), можуть бути реалізовані по-різному.

Основною «цеглинкою» ООП є *клас* – складний тип даних, що включає набір змінних і функцій для управління значеннями, що зберігаються в цих змінних. Змінні називають *атрибутами* або властивостями, а функції – *методами*. Клас є фабрикою об’єктів, тобто дозволяє створити необмежену кількість *екземплярів*, заснованих на цьому класі.

1.2. Визначення класу і створення екземпляра класу

Класи містять атрибути (властивості) та методи. Клас описується за допомогою ключового слова *class* за наступною схемою:

```
class <Ім’я класу> [(<Клас1> [..., <КлассN>])]:
    ["""Рядок документування"""]
    # <Опис атрибутів і методів>
    [Змінна = Значення]
    [def <Ім’я методу>(self, [список параметрів])
        self.Змінна = Значення]
```

Інструкція створює новий об'єкт і привласнює посилання на нього ідентифікатору, вказаному після ключового слова *class*. Це означає, що назва класу має повністю відповідати правилам іменування змінних. Після назви класу в

круглих дужках можна вказати один або декілька базових класів через кому. Якщо ж клас не успадковує базові класи, то круглі дужки можна не вказувати. Слід зауважити, що всі вирази всередині інструкції *class* виконуються при створенні класу, а не його екзмепляру. Для прикладу створимо клас, всередині якого просто виводиться повідомлення:

```
class MyClass:
    """Це рядок документування"""
    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 в функції *print()*, буде відразу виведено.

Створення атрибуту класу аналогічно створенню звичайної змінної. Метод всередині класу створюється так само, як і звичайна функція, - за допомогою інструкції *def*. Методам класу в першому параметрі, який обов'язково слід вказати явно, автоматично передається посилання на екземпляр класу. Загальноприйнято цей параметр називати ім'ям *self*, хоча це і не обов'язково. Доступ до атрибутів і методів класу всередині методу, що визначається проводиться через змінну *self* за допомогою крапкової нотації - до атрибуту *x* з методу класу можна звернутися так: *self.x*.

Щоб використовувати атрибути і методи класу, необхідно створити екземпляр класу згідно наступного синтаксису:

```
<Екземпляр класу> = <Назва класу>([<Параметри>])
```

При зверненні до методів класу використовується такий формат:

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

Зверніть увагу на те, що при виклику методу не потрібно передавати посилання на екземпляр класу в якості параметра, як це робиться у визначенні методу всередині класу. Посилання на екземпляр класу інтерпретатор передає автоматично.

Звернення до атрибутів класу здійснюється аналогічно:

```
<Екземпляр класу>.<Ім'я атрибуту>
```

Визначимо клас *MyClass* з атрибутом *x* і методом *print_x()*, що виводить значення цього атрибуту, а потім створимо екземпляр класу і викличемо метод:

```
class MyClass:
    def __init__(self): # Конструктор
        self.x = 10     # Атрибут екземпляра класу
    def print_x(self):  # Метод класу. self - екземпляр класу
        print(self.x)   # Виводимо значення атрибуту

c = MyClass()           # Створення екземпляра класу
                        # Викликаємо метод print_x()
c.print_x()             # self не вказується при виклику методу
print(c.x)              # До атрибуту можна звернутися безпосередньо
```

Для доступу до атрибутів і методів можна використовувати і такі функції:

♦ *getattr()* - повертає значення атрибуту за його назвою, заданій у вигляді рядка. За допомогою цієї функції можна сформувати ім'я атрибуту динамічно під час виконання програми. Формат функції:

```
getattr(<Об'єкт>, <Атрибут> [, <Значення за замовчуванням>])
```

Якщо вказаний атрибут не знайдено, генерується виняток *AttributeError*.

Щоб уникнути виведення повідомлення про помилку, в третьому параметрі можна вказати значення, яке буде повертатися, якщо атрибут не існує;

♦ *setattr()* - задає значення атрибута. Назва атрибута вказується у вигляді рядка. Формат функції:

```
setattr(<Об'єкт>, <Атрибут>, <Значення>)
```

Другим параметром функції *setattr()* можна передати ім'я неіснуючого атрибуту - в цьому випадку атрибут з вказаним ім'ям буде створений;

♦ *delattr(<Об'єкт>, <Атрибут>)* - видаляє атрибут, чия назва вказана у вигляді рядка;

♦ *hasattr(<Об'єкт>, <Атрибут>)* - перевіряє наявність зазначеного атрибута. Якщо атрибут існує, функція повертає значення *True*.

Продемонструємо роботу функцій на прикладі:

```
class MyClass:
    def __init__(self):
        self.x = 10
    def get_x(self):
        return self.x

c = MyClass()                 # Створюємо екземпляр класу
print(getattr(c, "x"))        # Виведе: 10
print(getattr(c, "get_x")()) # Виведе: 10
print(getattr(c, "y", 0))     # Виведе: 0, бо атрибут не знайдено
setattr(c, "y", 20)           # Створюємо атрибут y
print(getattr(c, "y", 0))     # Виведе: 20
delattr(c, "y")               # Видаляємо атрибут y
print(getattr(c, "y", 0))     # Виведе: 0, бо атрибут не знайдено
print(hasattr(c, "x"))        # Виведе: True
print(hasattr(c, "y"))        # Виведе: False
```

Всі атрибути класу в мові Python є відкритими (public), тобто доступними для безпосередньої зміни як з самого класу, так і з інших класів і з основного коду програми.

Крім того, атрибути допускається створювати динамічно після створення класу – можна створити як атрибут об'єкта класу, так і атрибут екземпляра класу. Розглянемо це на прикладі:

```
class MyClass: # Визначаємо порожній клас
    pass

MyClass.x = 50                  # Створюємо атрибут об'єкта класу
c1, c2 = MyClass(), MyClass() # Створюємо два примірника класу
c1.y = 10                       # Створюємо атрибут екземпляра класу
c2.y = 20                       # Створюємо атрибут екземпляра класу
print(c1.x, c1.y)               # Виведе: 50 10
print(c2.x, c2.y)               # Виведе: 50 20
```

У цьому прикладі ми визначаємо порожній клас, розмістивши в ньому оператор *pass*. Далі створюємо атрибут об'єкта класу: *x*. Цей атрибут буде доступний всім створюваним екземплярам класу. Потім створюємо два екземпляри класу і додаємо однойменні атрибути: *y*. Значення цих атрибутів будуть різними в кожному екземплярі класу. Але якщо створити новий екземпляр (наприклад, *c3*), то атрибут *y* в ньому визначений не буде. Таким чином, за допомогою класів можна імітувати типи даних, підтримувані іншими мовами

програмування (Наприклад, тип *struct*, доступний в мові C).

Екземпляри класів можуть використовуватися як структури або записи. На відміну від структур C або класів Java, поля даних примірника необов'язково оголошувати заздалегідь, вони можуть створюватися «на ходу». У наступному прикладі визначається клас з ім'ям *Circle*, створюється екземпляр *Circle*, полю *radius* екземпляру присвоюється значення, після чого це поле використовується для обчислення довжини кола:

```
>>> class Circle:
        pass
>>> my_circle = Circle()
>>> my_circle.radius = 5
>>> print (2 * 3.14 * my_circle.radius)
31.4
```

Як і в Java (а також у багатьох інших мовах), для звернення до полів екземпляра/структури і присвоювання їм значень використовується крапкова нотація.

Дуже важливо розуміти різницю між атрибутами об'єкта класу і атрибутами екземпляра класу. *Атрибут об'єкта класу* доступний всім екземплярам класу, і після зміни атрибута значення зміниться в усіх екземплярах класу. *Атрибут екземпляра класу* може зберігати унікальне значення для кожного екземпляра, і зміна його в одному екземплярі класу не торкнеться значення однойменного атрибуту в інших примірниках того ж класу. Розглянемо це на прикладі, створивши клас з атрибутом об'єкта класу (*x*) і атрибутом екземпляра класу (*y*):

```
class MyClass:
    x = 10          # Атрибут об'єкта класу
    def __init__(self):
        self.y = 20 # Атрибут екземпляра класу
```

Тепер створимо два екземпляри цього класу:

```
c1 = MyClass()      # Створюємо екземпляр класу
c2 = MyClass()      # Створюємо екземпляр класу
```

Виведемо значення атрибуту *x*, а потім змінимо значення і знову зробимо вивід:

```
print(c1.x, c2.x) # 10 10
MyClass.x = 88    # Змінюємо атрибут об'єкта класу
print(c1.x, c2.x) # 88 88
```

Як видно з прикладу, зміна атрибута об'єкта класу торкнулася значення в двох примірниках класу відразу. Тепер зробимо аналогічну операцію з атрибутом *y*:

```
print(c1.y, c2.y) # 20 20
c1.y = 88         # Змінюємо атрибут екземпляра класу
print(c1.y, c2.y) # 88 20
```

В цьому випадку змінилося значення атрибута лише в екземплярі *c1*.

Слід також враховувати, що в одному класі можуть одночасно існувати атрибут об'єкта і атрибут реалізації з одним ім'ям. Зміна атрибута об'єкта класу ми проводили наступним чином:

```
MyClass.x = 88 # Змінюємо атрибут об'єкта класу
```

Якщо після цієї інструкції вставити інструкцію:

```
c1.x = 200 # Створюємо атрибут реалізації
```

то буде створений атрибут екземпляра класу, а не змінено значення атрибута

об'єкта класу.

Щоб побачити різницю, виведемо значення атрибутів:

```
print(c1.x, MyClass.x)      # 200 88
```

Змінна класу (class variable) це змінна, пов'язана з класом, а не з його конкретним екземпляром і доступна для всіх екземплярів класу. Змінна класу може використовуватися для відстеження інформації на рівні класу - наприклад, кількості екземплярів класу, створених в будь-який момент часу.

Змінна класу створюється присвоєнням в тілі класу, а не в функції *__init__*. Після того як вона буде створена, змінна стає видимою для всіх екземплярів класу. Наприклад, за допомогою змінної класу можна надати доступ до значення *pi* всіх примірників класу *Circle*:

```
class Circle:
    pi = 3.14159

    def __init__(self, radius):
        self.radius = radius
    def area(self):
        return self.radius * self.radius * Circle.pi

>>> Circle.pi
3.14159
>>> Circle.pi = 4
>>> Circle.pi
4
```

Цей приклад показує, як повинна працювати змінна класу; вона пов'язана з класом, що визначив її, і міститься в ньому. Зверніть увагу: в цьому прикладі ми звертаємося до *Circle.pi* до того, як буде створено хоча б один екземпляр класу. Очевидно, *Circle.pi* існує незалежно від будь-яких конкретних екземплярів класу *Circle*.

До змінної класу також можна звернутися з методу класу на ім'я класу. Ми робимо це в визначенні *Circle.area*, де функція area звертається до *Circle.pi*. Це дає бажаний ефект: правильне значення *pi* читається з класу і використовується в обчисленнях:

```
>>> c = Circle(3)
>>> c.area()
28.27431
```

Можливо, вам не сподобається, що ім'я класу жорстко фіксується в методах цього класу. Цього можна уникнути за допомогою спеціального атрибута *__class__*, доступного для всіх екземплярів класу Python. Цей атрибут повертає клас, до якого належить екземпляр, наприклад:

```
>>> Circle
<class '__main__.Circle'>
>>> c.__class__
<class '__main__.Circle'>
```

Приклад дозволяє отримати значення *Circle.pi* з *c* без явного вказання імені класу *Circle*:

```
>>> c.__class__.pi
3.14159
```

Цей код можна було використовувати в методі *area* для того, щоб позбутися

від внутрішнього згадки класу *Circle*; посилання *Circle.pi* замінюється на *self.__class__.pi*.

Існує два типи атрибутів - це *захищений* атрибут (одне нижнє підкреслення *_x*) і *приватний* атрибут (два нижніх підкреслення *__x*). Перш за все, такий поділ зроблено для розробників, щоб розуміти, що атрибут першого типу не повинен бути змінений (рекомендація), а атрибут другого типу не повинен бути доступний ззовні. Такий аргумент в інших мовах програмування називають приватним (private).

Приватний атрибут або *приватний метод* не видно за межами методів класу, в якому вони визначаються. Приватні змінні і методи корисні з двох причин: вони підвищують рівень безпеки та надійності за рахунок виборчого обмеження доступу до важливих або критичних частин реалізації об'єкта, а також запобігають конфліктам імен, які можуть виникнути через застосування успадкування. Приватні змінні спрощують читання коду, оскільки вони явно вказують, що мають використовуватися тільки усередині класу. Все інше відноситься до інтерфейсу класу.

Багато мов, що визначають приватні змінні, використовують для цього ключове слово «private». Синтаксис Python простіший, до того ж з нього відразу видно, які змінні або методи є приватними, а які ні. Будь-який метод або змінна екземпляра, ім'я якої починається (саме починається, а не закінчується!) з подвійного символу підкреслення (__), є приватними; все інше приватним не є.

```
class MyClass:
    def __init__(self):
        self.x   = 10    # Звичайна змінна
        self._y  = 20    # Захищена змінна
        self.__z = 30    # Приватна змінна

c1 = MyClass()

print(c1.x);   c1.x  = 11;  print(c1.x)            # 10 11
print(c1._y);  c1._y = 21;  print(c1._y)           # 20 21
print(c1.__z); c1.__z = 31; print(c1._MyClass__z) # Error
```

Проте захист з приватних змінних і методів можна зняти. Механізм реалізації приватності перетворює імена приватних змінних і приватних методів при компіляції в байт-код, а саме до імені змінної приєднується префікс *_classname*:

```
>>> print(c1._MyClass__z) # Виведе 30
```

У стильовому оформленні класів є кілька моментів, про які варто згадати окремо.

Імена класів рекомендується записувати за такою схемою: перша буква кожного слова записується в верхньому регістрі, слова не розділяються пробілами (*MyClass*). Імена екземплярів і модулів записуються в нижньому регістрі з поділом слів символами підкреслення (*my_class*).

Кожен клас повинен мати рядок документації, розміщений відразу ж за визначенням класу. Рядок документації повинен містити короткий опис того, що робить клас, і в ньому повинні дотримуватися ті ж угоди щодо форматування, як і у функціях. Кожен модуль також повинен містити рядок документації з описом

можливих застосувань класів в модулі.

Порожні рядки можуть використовуватися для структурування коду, але зловживати ними не варто. У класах можна розділяти методи одним порожнім рядком, а в модулях для поділу класів можна використовувати дві порожні рядки.
Якщо вам буде потрібно імпортувати модуль зі стандартної бібліотеки і модуль з сторонньої бібліотеки, почніть з команди import для модуля стандартної бібліотеки. Потім додайте порожній рядок і команду import для стороннього модуля. У програмах з декількома командами *import* виконання цієї угоди допоможе зрозуміти, звідки беруться різні модулі, використані в програмі.

1.3. Методи __init__() і __del__()

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 [, <Значення1> [, ... ,<ЗначенняN>]]):
    <Інструкції>
```

З допомогою методу *__init__()* атрибутам класу можна присвоїти початкові значення. При створенні екземпляра класу параметри цього методу вказуються після імені класу в круглих дужках:

```
<Екземпляр класу> = <Ім'я класу> ([<Значення1> [..., <ЗначенняN>]])
```

Приклад використання методу *__init__()*:

```
class MyClass:
    def __init__(self, value1, value2): # Конструктор
        self.x = value1
        self.y = value2

c = MyClass(100, 300) # Створюємо екземпляр класу
print (c.x, c.y)      # Виведе: 100 300
```

Якщо конструктор викликається при створенні екземпляра, то перед знищенням екземпляру автоматично викликається метод, званий деструктором. У мові Python деструктор реалізується у вигляді визначеного методу *__del__()*. Варто зауважити, що метод не буде викликаний, якщо на екземпляр класу існує хоча б одне посилання. Втім, оскільки інтерпретатор самостійно дбає про видалення об'єктів, використання деструктора в мові Python не має особливого сенсу.

```
class MyClass:
    def __init__(self): # Конструктор класу
        print("Викликаний метод __init__()")
    def __del__(self): # Деструкція класу
        print("Викликаний метод __del__()")

c1 = MyClass() # Виведе: Викликаний метод __init__()
del c1         # Виведе: Викликаний метод __del__()
c2 = MyClass() # Виведе: Викликаний метод __init__()
c3 = c2        # Створюємо посилання на екземпляр класу
del c2         # Нічого не виведе, т. к. існує посилання
del c3         # Виведе: Викликаний метод __del__()
```

Напишемо клас, що представляє автомобіль. Цей клас буде містити

інформацію про тип машини (рік випуску, фірму-виробника і модель), а також метод для виведення короткого опису:

```
class Car():
    """Проста модель автомобіля."""
    def __init__(self, make, model, year):
        """Ініціалізує атрибути опису автомобіля."""
        self.make = make
        self.model = model
        self.year = year

    def get_descriptive_name(self):
        """Повертає акуратно відформатований опис."""
        long_name = str(self.year) + ' ' + self.make + ' ' +
self.model
        return long_name.title()

my_new_car = Car('audi', 'a4', 2016)
print(my_new_car.get_descriptive_name())
```

Додамо атрибут, що змінюється з часом, - в ньому буде зберігатися пробіг машини в милях. Кожен атрибут класу повинен мати початкове значення, навіть якщо він дорівнює 0 або порожній рядок. У деяких випадках (наприклад, при завданні значень за замовчуванням) це початкове значення є сенс ставити в тілі методу *__init__()*; в такому випадку передавати параметр для цього атрибута при створенні об'єкта не обов'язково.

Додамо атрибут з ім'ям *odometer_reading*, початкове значення якого завжди дорівнює 0. Також в клас буде включений метод *read_odometer()* для читання поточних показань одометра:

```
class Car():
    """Проста модель автомобіля."""
    def __init__ (self, make, model, year):
        """Ініціалізує атрибути опису автомобіля."""
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        """Повертає акуратно відформатований опис."""
        long_name = str(self.year) + ' ' + self.make + ' ' +
self.model
        return long_name.title()

    def read_odometer(self):
        """Виводить пробіг машини."""
        print( "This car has " + str(self.odometer_reading) + " khm
on it.")

my_new_car = Car( 'audi', 'a4', 2016)
print(my_new_car.get_descriptive_name())
my_new_car.read_odometer()
```

Результат:

```
2016 Audi A4
```

```
This car has 0 khm on it.
```

1.4. Успадкування

Успадкування є, мабуть, найголовнішим поняттям ООП. Припустимо, у нас є клас (наприклад, *class1*). За допомогою успадкування ми можемо створити новий клас (наприклад, *class2*), в якому буде реалізований доступ до всіх атрибутів і методів класу *class1:*

```
class Class1: # Базовий клас
    def func1(self):
        print("Метод func1() класу Class1")
    def func2(self):
        print("Метод func2() класу Classl")

class Class2(Class1): # Клас Class2 успадковує клас Class1
    def func3(self):
        print("Метод func3() класу Class2")

c = Class2() # Створюємо екземпляр класу Class2
c.func1()    # Виведе: Метод funcl() класу Classl
c.func2()    # Виведе: Метод func2() класу Classl
c.func3()    # Виведе: Метод func3() класу Class2
```

Як видно з прикладу, клас *Class1* вказується всередині круглих дужок у визначенні класу *Class2*. Таким чином, клас *Class2* успадковує всі атрибути і методи класу *Class1*. Клас Class1 називається *базовим* або *суперкласом*, а клас *Class2 - похідним* або *підкласом*.

Якщо ім'я методу в класі *Class2* збігається з ім'ям методу класу *Class1*, то буде використовуватися метод з класу *Class2*. Щоб викликати однойменний метод з базового класу, перед методом слід через крапку написати назву базового класу, а в першому параметрі методу - явно вказати посилання на екземпляр класу. Розглянемо це на прикладі:

```
class Class1: # Базовий клас
    def __init__(self):
        print("Конструктор базового класу")
    def func1(self):
        print("Метод func1() класу Class1")

class Class2(Class1):         # Клас Class2 успадковує клас Classl
    def __init__(self):
        print("Конструктор похідного класу")
        Class1.__init__(self) # Виклик конструктора базового класу
    def func1(self):
        print("Метод func1() класу Class2")
        Class1.func1(self)    # Викликаємо метод базового класу

c = Class2() # Створюємо екземпляр класу Class2
c.func1()    # Викликаємо метод func1()
```

Результат:

```
Конструктор похідного класу
Конструктор базового класу
Метод func1() класу Class2
```

```
Метод func1() класу Class1
```

Конструктор базового класу автоматично не викликається, якщо він перевизначений у похідному класі.

Щоб викликати однойменний метод з базового класу, також можна скористатися функцією *super()*. Формат функції:

```
super([<Клас>, <Покажчик self>])
```

З допомогою функції super() інструкцію

```
Class1.__init__(self) # Викликаємо конструктор базового класу
```

можна записати так:

```
super().__init__() # Викликаємо конструктор базового класу
```

або так:

```
super(Class2, self).__init__() # Виклик конструктора базового класу
```

Зверніть увагу на те, що при використанні функції super () не потрібно явно передавати покажчик *self* в викликається метод. Крім того, в першому параметрі функції *super()* вказується похідний клас, а не базовий. Пошук ідентифікатора буде проводитися у всіх базових класах. Результатом пошуку стане перший знайдений ідентифікатор в ланцюжку успадкування.

Спробуємо побудувати модель електромобіля. Електромобіль є спеціалізованим різновидом автомобіля, тому новий клас *ElectricCar* можна створити на базі класу *Car*, написаного раніше. Тоді нам залишиться додати в нього код атрибутів і поведінки, що відноситься тільки до електромобілів. Почнемо з створення простої версії класу *ElectricCar*, який робить все, що робить клас *Car*:

```
class Car():
    """Проста модель автомобіля."""
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        long_name = str(self.year) + ' ' + self.make + ' ' +
self.model
        return long_name.title()

    def read_odometer (self):
        print( "This car has " + str(self.odometer_reading) + " khm
on it.")

    def update_odometer (self, mileage):
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print ("You can not roll back an odometer!")

    def increment_odometer (self, miles):
        self.odometer_reading += miles

class ElectricCar (Car):
    """Представляє аспекти машини, специфічні для електромобілів."""
```

```
    def __init__(self, make, model, year):
        """Ініціалізує атрибути класу-батька."""
        super().__init__(make, model, year)

my_tesla = ElectricCar('tesla', 'model s', 2016)
print(my_tesla.get_descriptive_name())
```

Після створення класу-нащадка можна переходити до додавання нових атрибутів і методів, необхідних для того, щоб нащадок відрізнявся від батька.

Додамо атрибут, специфічний для електромобілів (наприклад, потужність акумулятора), і метод для виведення інформації про це атрибуті:

```
class ElectricCar (Car):
    """Представляє аспекти машини, специфічні для електромобілів."""
    def __init__(self, make, model, year):
        """Ініціалізує атрибути класу-батька."""
        super().__init__(make, model, year)
        self.battery_size = 70

    def describe_battery(self):
        """Виводить інформацію про потужності акумулятора."""
        print( "This car has a " + str(self.battery_size) + "-kWh
battery.")

my_tesla = ElectricCar('tesla', 'model s', 2016)
print(my_tesla.get_descriptive_name())
my_tesla.describe_battery()
```

Результат:

```
2016 Tesla Model S
This car has a 70-kWh battery.
```

Додається новий атрибут *self.battery_size*, якому присвоюється початкове значення - скажімо, 70. Цей атрибут буде присутній у всіх екземплярах, створених на основі класу *ElectricCar* (але не у всякому екземплярі *Car*). Також додається метод з ім'ям *describe_battery ()*, який виводить інформацію про акумулятор.

Будь-який метод батьківського класу, який в моделюється ситуації робить не те, що потрібно, можна перевизначити. Для цього в класі-нащадку визначається метод з тим же ім'ям, що і у методу класу-батька. Python ігнорує метод батька і звертає увагу тільки на метод, визначений в нащадку.

Припустимо, в класі *Car* є метод *fill_gas_tank()*. Для електромобілів заправка бензином безглузда, тому цей метод логічно перевизначити. Наприклад, це можна зробити так:

```
def ElectricCar (Car):
...
    def fill_gas_tank():
        """У електромобілів немає бензобака."""
        print("This car does not need a gas tank!")
```

І якщо хтось спробує викликати метод *fill_gas_tank()* для електромобіля, Python ігнорує метод *fill_gas_tank()* класу *Car* та виконає замість нього цей код. Із застосуванням успадкування нащадок зберігає ті аспекти батька, які вам потрібні, і перекриває все непотрібне.

При моделюванні явищ реального світу в програмах класи нерідко доповнюються усе більшою кількістю подробиць. Списки атрибутів і методів

ростуть, і через якийсь час файли стають довгими і громіздкими. У такій ситуації частина одного класу нерідко можна записати у вигляді окремого класу. Великий код розбивається на менші класи, які працюють у взаємодії один з одним.

Наприклад, при подальшій доробці класу *ElectricCar* може виявитися, що в ньому з'явилося занадто багато атрибутів і методів, що відносяться до акумулятора. В такому випадку можна зупинитися і перемістити всі ці атрибути і методи в окремий клас з ім'ям *Battery*. Потім екземпляр *Battery* стає атрибутом класу *ElectricCar*:

```
class Car ():
...
class Battery ():
     """Проста модель акумулятора електромобіля."""
     def __init__(self, battery_size = 70):
          """Ініціалізує атрибути акумулятора."""
          self.battery_size = battery_size

     def describe_battery (self):
          """Виводить інформацію про потужності акумулятора."""
          print ("This car has a " + str (self.battery_size) + "-kWh
battery.")

class ElectricCar (Car):
     """Представляє аспекти машини, специфічні для електромобілів."""
     def __init__(self, make, model, year):
     """
     Ініціалізує атрибути класу-батька.
     Потім ініціалізує атрибути, специфічні для електромобіля.
     """
     super().__init__(make, model, year)
     self.battery = Battery()
```

Тепер акумулятор можна моделювати з будь-яким ступенем деталізації без захаращення класу *ElectricCar*. Додамо в *Battery* ще один метод, який виводить запас ходу на підставі потужності акумулятора:

```
class Car ():
...
class Battery ():
...
     def get_range (self):
          """Виводить приблизний запас ходу для акумулятора."""
          if self.battery_size == 70:
               range = 240
          elif self.battery_size == 85:
               range = 270
          message = "This car can go approximately " + str (range)
          message += "khm on a full charge."
          print (message)
```

1.5. Множине успадкування

У визначенні класу в круглих дужках можна вказати відразу декілька базових класів через кому. Розглянемо множинне спадкування на прикладі:

```
class Class1: # Базовий клас для класу Class2
    def func1(self):
        print("Метод func1() класу Classl")

class Class2 (Class1): # Клас Class2 успадковує клас Classl
    def func2(self):
        print("Метод func2() класу Class2")

class Class3 (Class1): # Клас Class3 успадковує клас Classl
    def func1(self):
        print ("Метод func1() класу Class3")
    def func2(self):
        print ("Метод func2() класу Class3")
    def func3(self):
        print("Метод func3() класу Class3")
    def func4(self):
        print("Метод func4() класу Class3")

class Class4 (Class2, Class3): # Множинне успадкування
    def func4 (self):
        print("Метод func4() класу Class4")

c = Class4()    # Створюємо екземпляр класу Class4
c.func3()       # Виведе: Метод func1() класу Class3
c.func2()       # Виведе: Метод func2() класу Class2
c.func3()       # Виведе: Метод func3() класу Class3
c.func4()       # Виведе: Метод func4() класу Class4
```

Метод *func1()* визначено в двох класах: *Class1* і *Class3*. Так як спочатку проглядаються всі базові класи, безпосередньо зазначені в визначені поточного класу, метод *func1()* буде знайдений в класі *Class3* (оскільки він вказаний в числі базових класів у визначенні *Class4*), а не в класі *Class1*. Метод *func2()* також визначено в двох класах: *Class2* і *Class3*. Так як клас *Class2* стоїть першим в списку базових класів, то метод буде знайдений саме в ньому. щоб наслідувати метод з класу *Class3*, слід вказати це явно. Переробимо визначення класу *Class*4 з попереднього прикладу і успадковуємо метод *func2()* з класу *Class3*:

```
class Class4 (Class2, Class3): # Множинне успадкування
    # Наслідуєм func2() з класу Class3, а не з класу Class2
    func2 = Class3.func2
    def func4(self):
        print( "Метод func4() класу Class4")
```

Повернемося до лістингу. Метод *func3()* визначений тільки в класі *Class3*, тому метод успадковується від цього класу. Метод *func4()*, визначений в класі *Class3*, перевизначається в похідному класі.

Якщо шуканий метод знайдений в похідному класі, то вся ієрархія успадкування проглядатися не буде.

Для отримання переліку базових класів можна скористатися атрибутом *__bases__*. Як значення атрибут повертає кортеж. Як приклад виведемо базові класи для всіх класів з попереднього прикладу:

```
print(Class1.__bases__)
print(Class2.__bases__)
print(Class3.__bases__)
```

```
print(Class4.__bases__)
```

виведе:

```
(<class 'object'>,)
(<class '__main__.Class1'>,)
(<class '__main__.Class1'>,)
(<class '__main__.Class2'>, <class '__main__.Class3'>)
```

Розглянемо порядок пошуку ідентифікаторів при складній ієрархії множинного наслідування:

```
class Classl: x = 10
class Class2(Classl): pass
class Class3(Class2): pass
class Class4(Class3): pass
class Class5(Class2): pass
class Class6(Class5): pass
class Class7(Class4, Class6): pass
c = Class7()
print(c.x)
```

Послідовність пошуку атрибуту *x* буде такою:

```
Class7 -> Class4 -> Class3 -> Class6 -> Class5 -> Class2 -> Classl
```

Отримати весь ланцюжок успадкування дозволяє атрибут *__mro__*:

```
print(Class7.__mro__)
```

Результат виконання:

```
(<class '__main__.Class7'>, <class '__main__.Class4'>, <class
'__main__.Class3'>, <class '__main__.Class6'>, <class
'__main__.Class5'>, <class '__main__.Class2'>, <class
'__main__.Classl'>, <class 'object'>)
```

1.6. Домішки і їх використання

Множинне спадкування, підтримуване Python, дозволяє реалізувати цікавий спосіб розширення функціональності класів за допомогою так званих *домішок* (Mixins). Домішка - це клас, що включає будь-які атрибути і методи, які необхідно додати до інших класів. Оголошуються вони точно так само, як і звичайні класи.

Як приклад оголосимо клас-домішку *Mixin*, після чого оголосимо ще два класи, додамо до їх функціональності ту, що визначена в домішці *Mixin*, і перевіримо її в дії:

```
class Mixin: # Визначаємо сам клас-домішку
    attr = 0 # Визначаємо атрибут домішки
    def mixin_method(self): # Визначаємо метод домішки
        print ( "Метод домішки")

class Class1 (Mixin):
    def method1(self):
        print("Метод класу Class1")

class Class2 (Class1, Mixin):
    def method2(self):
        print("Метод класу Class2")

c1 = Class1()
c1.method1()
```

```
c1.mixin_method() # Class1 підтримує метод домішки
c2 = Class2()
c2.method1()
c2.method2()
c2.mixin_method() # Class2 також підтримує метод домішки
```

Ось результат виконання коду, наведеного коду:

```
Метод класу Class1
Метод домішки
Метод класу Class1
Метод класу Class2
Метод домішки
```

Домішки активно застосовуються в різних додаткових бібліотеках, зокрема в популярному веб-фреймворку Django.

1.7. Спеціальні методи

Класи підтримують такі спеціальні методи:

♦ *__call__()* - дозволяє обробити виклик екземпляра класу як виклик функції. Формат методу:

```
__call__(self[, <Параметр1> [, <ПараметрN>]])
```

Приклад:

```
class MyClass:
    def __init__(self, m):
        self.msg = m
    def __call__(self):
        print(self.msg)

c1 = MyClass("Значення1") # Створення екземпляра класу
c2 = MyClass("Значення2") # Створення екземпляра класу
c1() # Виведе: Значення1
c2() # Виведе: Значення2
```

♦ *__getattr__(self, <Атрибут>)* - викликається при зверненні до неіснуючого атрибуту класу:

```
class MyClass:
    def __init__(self):
        self.i = 20
    def __getattr__(self, attr):
        print("Викликаний метод __getattr__()")
        return 0

c = MyClass()
# Атрибут i існує
print(c.i) # Виведе: 20. Метод __getattr__() не викликається
# Атрибут s не існує
print(c.s) # Виведе: Викликаний метод __getattr__() 0
```

♦ *__getattribute__(self, <Атрибут>)* - викликається при зверненні до будь-якого атрибуту класу. Необхідно враховувати, що використання крапкової нотації (для звернення до атрибуту класу) всередині цього методу приведе до зациклення. Щоб уникнути зациклення, слід викликати метод *__getattribute__()* об'єкта *object* і всередині цього методу повернути значення атрибута або згенерувати виняток *AttributeError*:

```
class MyClass:
    def __init__(self):
        self.i = 20
    def __getattribute__(self, attr):
        print("Викликаний метод __getattribute__()")
        return object.__getattribute__(self, attr) # Тільки так !!!

c = MyClass()
print(c.i) # Виведе: Викликаний метод __getattribute__() 20
```

♦ ***__setattr__(self, <Атрибут>, <Значення>)*** - викликається при спробі привласнення значення атрибуту екземпляра класу. Якщо всередині методу необхідно присвоїти значення атрибуту, слід використовувати словник ***__dict__***, оскільки при застосуванні крапкової нотації метод ***__setattr__()*** буде викликаний повторно, що призведе до зациклення:

```
class MyClass:
    def __setattr__(self, attr, value):
        print("Викликаний метод __setattr__()")
        self.__dict__[attr] = value # Тільки так !!!

c = MyClass()
c.i = 10        # Виведе: Викликаний метод __setattr__()
print(c.i)      # Виведе: 10
```

♦ ***__delattr__(self, <Атрибут>)*** - викликається при видаленні атрибута за допомогою інструкції *del <Екземпляр класу>.<Атрибут>*;

♦ ***__len__(self)*** - викликається при використанні функції *len()*, а також для перевірки об'єкта на логічне значення при відсутності методу ***__bool__()***. Метод повинен повертати позитивне число:

```
class MyClass:
    def __len__(self):
        return 50

c = MyClass()
print(len(c)) # Виведе: 50
```

♦ ***__bool__(self)*** - викликається при використанні функції *bool()*;

♦ ***__int__(self)*** - викликається при перетворенні об'єкта в ціле число за допомогою функції *int()*;

♦ ***__float__(self)*** - викликається при перетворенні об'єкта в дійсне число за допомогою функції *float()*;

♦ ***__complex__(self)*** - викликається при перетворенні об'єкта в комплексне число за допомогою функції *complex()*;

♦ ***__round__(self, n)*** - викликається при використанні функції *round()*;

♦ ***__index__(self)*** - викликається при використанні функцій *bin(), hex()* і *oct()*;

♦ ***__repr__(self)*** і ***__str__(self)*** - служать для перетворення об'єкта в рядок. Метод ***__repr__()*** викликається при виведенні в інтерактивній оболонці, а також при використанні функції *repr()*. Метод ***__str__()*** викликається при виведенні за допомогою функції *print()*, а також при використанні функції *str()*. Якщо метод ***__str__()*** відсутній, буде викликаний метод ***__repr__()***. Як значення методи ***__repr__()*** і ***__str__()*** повинні повертати рядок:

```
class MyClass:
```

```
    def __init__(self, m):
        self.msg = m
    def __repr__(self):
        return "Викликаний метод __repr__() {0}".format(self.msg)
    def __str__(self):
        return "Викликаний метод __str__() {0}".format(self.msg)

c = MyClass("Значення")
print(repr(c)) # Виведе: Викликаний метод __repr__() Значення
print(str(c))  # Виведе: Викликаний метод __str__() Значення
print(c)       # Виведе: Викликаний метод __str__() Значення
```

♦ ***__hash__(self)*** - цей метод слід перевизначити, якщо екземпляр класу планується використовувати в якості ключа словника або всередині множини:

```
class MyClass:
    def __init__(self, y):
        self.x = y
    def __hash__(self):
        return hash(self.x)

c = MyClass(10)
d = {}
d[c] = "Значення"
print(d[c]) # Виведе: Значення
```

Класи підтримують ще кілька спеціальних методів, які застосовуються лише в особливих випадках і будуть розглянуті далі.

1.8. Перевантаження операторів

Перевантаження операторів дозволяє екземплярам класів брати участь в звичайних операціях. Щоб перевантажити оператор, необхідно в класі визначити метод зі спеціальною назвою.

Перевантаження математичних операторів проводиться за допомогою таких методів:

♦ *x + y* – додавання: *x.__add__(y)*;

♦ *y + x* - додавання (екземпляр класу праворуч): *x.__radd__(y)*;

♦ *x += y* - додавання і присвоювання: *x.__iadd__(y)*;

♦ *x - y* - віднімання: *x.__sub__(y)*;

♦ *y - x* - віднімання (екземпляр класу праворуч): *x.__rsub__(y)*;

♦ *x -= y* - віднімання і присвоювання: *x.__isub__(y)*;

♦ *x * y* – множення: *x.__mul__(y)*;

♦ *y * x* - множення (екземпляр класу праворуч): *x.__rmul__(y)*;

♦ *x *= y* - множення і присвоювання: *x.__imul__(y)*;

♦ *x / y* – ділення: *x.__truediv__(y)*;

♦ *y / x* - ділення (екземпляр класу праворуч): *x.__rtruediv__(y)*;

♦ *x /= y* - ділення і присвоєння: *x.__itruediv__(y)*;

♦ *x // y* - ділення з округленням вниз: *x.__floordiv__(y)*;

♦ *y // x* - ділення з округленням вниз (екземпляр класу праворуч): *x.__rfloordiv__(y)*;

♦ *x //= y* - діленняз округленням вниз і присвоєнням: *x.__ifloordiv__(y)*;

♦ *x % y* - залишок від ділення: *x.__mod__(y)*;
♦ *y % x* - залишок від ділення (екземпляр класу праворуч): *x.__rmod__(y)*;
♦ *x %= y* - залишок від ділення і присвоєння: *x.__imod__(y)*;
♦ *x ** y* – піднесення в степінь: *x.__pow__(y)*;
♦ *y ** x* – піднесення в степінь (екземпляр класу праворуч): *x.__rpow__(y)*;
♦ *x **= y* - піднесення в степінь і присвоювання: *x.__ipow__(y)*;
♦ *-x* - унарний мінус: *x.__neg__()*;
♦ *+x* - унарний плюс: *x.__pos__()*;
♦ *abs(x)* - абсолютне значення: *x.__abs__()*.

Приклад перевантаження математичних операторів:

```
class MyClass:
    def __init__(self, y):
        self.x = y
    def __add__(self, y):
        print("Екземпляр зліва")
        return self.x + y
    def __radd__(self, y):
        print("Екземпляр справа")
        return self.x + y
    def __iadd__(self, y):
        print("Додавання з присвоєнням")
        self.x += y
        return self

c = MyClass(50)
print(c + 10)       # Виведе: Екземпляр зліва 60
print(20 + c)       # Виведе: Екземпляр справа 70
c += 30             # Виведе: Додавання з присвоєнням
print(c.x)          # Виведе: 80
```

Перевантаження двійкових операторів проводиться за допомогою таких методів:

♦ *~x* - двійкова інверсія: *x.__invert__()*;
♦ *x & y* - бінарне І: *x.__and__(y)*;
♦ *y & x* - бінарне І (екземпляр класу праворуч): *x.__rand__(y)*;
♦ *x &= y* - бінарне І та присвоєння: *x.__iand__(y)*;
♦ *x | y* - бінарне АБО: *x.__or__(y)*;
♦ *y | x* - бінарне АБО (екземпляр класу праворуч): *x.__ror__(y)*;
♦ *x |= y* - бінарне АБО і присвоєння: *x.__ior__(y)*;
♦ *x ^ y* - бінарне виключне АБО: *x.__xor__(y)*;
♦ *y ^ x* - бінарне виключне АБО (екземпляр класу праворуч): *x.__rxor__(y)*;
♦ *x ^= y* - бінарне виключне АБО і присвоєння: *x.__ixor__(y)*;
♦ *x << y* - зсув вліво: *x.__lshift__(y)*;
♦ *y << x* - зсув вліво (екземпляр класу праворуч): *x.__rishift__(y)*;
♦ x <<= y - зсув вліво і присвоєння: *x.__ilshift__(y)*;
♦ *x >> y* - зсув вправо: *x.__rshift__(y)*;
♦ *y >> x* - зсув вправо (екземпляр класу праворуч): *x._rrshift__(y)*;
♦ *x >>= y* - зсув вправо і присвоєння: *x.__irshift__(y)*.

Перевантаження операторів порівняння проводиться за допомогою таких

методів:

- ♦ *x == y* - рівне: *x.__eq__(y)*;
- ♦ *x != y* - не дорівнює: *x.__ne__(y)*;
- ♦ *x < y* - менше: *x.__lt__(y)*;
- ♦ *x> y* - більше: *x.__gt__(y)*;
- ♦ *x <= y* - менше або дорівнює: *x.__le__(y)*;
- ♦ *x>= y* - більше або дорівнює: *x.__ge__(y)*;
- ♦ *y in x* - перевірка на входження: *x.__contains__(y)*.

Приклад перевантаження операторів порівняння:

```
class MyClass:
    def __init__(self):
        self.x = 50
        self.arr = [1, 2, 3, 4, 5]
    def __eq__(self, y): # Перевантаження оператора ==
            return self.x == y
    def __contains__(self, y): # Перевантаження оператора in
        return y in self.arr

c = MyClass ()
print("Рівне" if c == 50 else "Не рівне") # Виведе: Рівне
print("Рівне" if c == 51 else "Не рівне") # Виведе: Не рівне
print("Є" if 5 in c else "Немає")         # Виведе: Є
```

Створимо клас (власний тип даних) *Point*, в якому визначимо (перевизначимо методи базового класу *object*) спеціальні методи *__init__(), __eq__(), __str__()*:

```
class Point:
    def __init__(self, x = 0, y = 0): # Конструктор
        self.x = x
        self.y = y
    def __eq__(self, other): # Метод для порівняння двох точок
        return self.x == other.x and self.y == other.y
    def __str__(self): # Метод для строкового виведення інформації
        return "({0.x}, {0.y})".format(self)

a = Point()   # Створюємо об'єкт, за замовчуванням x = 0, y = 0
print(str(a)) # Тут викликається метод __str__() класу Point
# Повна форма Point.__str__(a)
b = Point(3, 4)
print(str(b))
b.x = -19
print(str(b))
print(a == b, a != b) # Викликається метод __eq__()
# Повна форма для порівняння a == b має вигляд: Point.__eq__(a, b)
```

Результат роботи програми:

```
(0, 0)
(3, 4)
0
(-19, 4)
False True
```

1.9. Статичні методи і методи класу

Усередині класу можна створити метод, який буде доступний без створення екземпляра класу (статичний метод). Для цього перед визначенням методу всередині класу слід вказати декоратор *@staticmethod*. Виклик статичного методу без створення екземпляра класу здійснюється наступним чином:

```
<Назва класу>.<Назва методу>(<Параметри>)
```

Крім того, можна викликати статичний метод через екземпляр класу:

```
<Екземпляр класу>.<Назва методу>(<Параметри>)
```

Приклад використання статичних методів:

```
class MyClass:
    @staticmethod
    def func1(x, y): # Статичний метод
        return x + y
    def func2(self, x, y): # Звичайний метод в класі
        return x + y
    def func3(self, x, y):
        return MyClass.func1(x, y) # Виклик з методу класу

print(MyClass.func1(10, 20)) # Викликаємо статичний метод
c = MyClass()
print(c.func2(15, 6))  # Викликаємо метод класу
print(c.func1(50, 12)) # Виклик стат. методу через екземпляр класу
print(c.func3(23, 5))  # Виклик стат. методу всередині класу
```

Зверніть увагу на те, що у визначенні статичного методу немає параметра *self*. Це означає, що всередині статичного методу немає доступу до атрибутів і методів екземпляра класу.

Методи класу створюються за допомогою декоратора *@classmethod*. В якості першого параметра в метод класу передається посилання на клас. Виклик методу класу здійснюєтьсянаступним чином:

```
<Назва класу>.<Назва методу>(<Параметри>)
```

Крім того, можна викликати метод класу через екземпляр класу:

```
<Екземпляр класу>.<Назва методу>(<Параметри>)
```

Приклад використання методів класу:

```
class MyClass:
    @classmethod
    def func(cls, x): # Метод класу
        print(cls, x)

MyClass.func(10) # Викликаємо метод через назву класу
c = MyClass ()
c.func(50)       # Викликаємо метод класу через екземпляр
```

Результат:

```
<class '__main__.MyClass'> 10
<class '__main__.MyClass'> 50
```

1.10. Абстрактні методи

Абстрактні методи містять тільки визначення методу без реалізації. Передбачається, що похідний клас повинен перевизначити метод і реалізувати його функціональність. Щоб таке припущення зробити більш очевидним, часто всередині абстрактного методу генерують виняток:

```
class Class1:
    def func(self, x): # Абстрактний метод
    # Генеруємо виняток з допомогою raise
        raise NotImplementedError ("Необхідно перевизначити метод")

class Class2 (Class1): # Наслідуємо абстрактний метод
    def func(self, x): # Перевизначаємо метод
        print(x)

class Class3 (Class1): # Клас не перевизначає метод
    pass

c2 = Class2()
c2.func(50)     # Виведе: 50
c3 = Class3()

try:            # Перехоплюємо виняток
    c3.func(50) # Помилка. Метод func() не перевизначений
except NotImplementedError as msg:
    print(msg)  # Виведе: Необхідно перевизначити метод
```

До складу стандартної бібліотеки входить модуль *abc*. У цьому модулі визначено декоратор *@abstractmethod*, який дозволяє вказати, що метод, перед яким розташований декоратор, є абстрактним. При спробі створити екземпляр похідного класу, в якому не перевизначений абстрактний метод, генерується виняток *TypeError*. Розглянемо використання декоратора *@abstractmethod* на прикладі:

```
from abc import ABCMeta, abstractmethod
class Class1(metaclass = ABCMeta):
    @abstractmethod
    def func(self, x): # Абстрактний метод
        pass

class Class2(Class1):  # Наслідуємо абстрактний метод
    def func(self, x): # Перевизначаємо метод
        print (x)

class Class3(Class1): # Клас не перевизначає метод
    pass

c2 = Class2()
c2.func(50) # Виведе: 50
try:
    c3 = Class3() # Помилка. Метод func() не перевизначений
    c3.func (50)
except TypeError as msg:
    print (msg)# Виведе: Can't instantiate abstract class
               # Class3 with abstract methods func
```

Є можливість створення абстрактних статичних методів і абстрактних методів класу, для чого необхідні декоратори вказуються одночасно, один за одним. Для прикладу оголосимо клас з абстрактним статичним методом і абстрактним методом класу

```
from abc import ABCMeta, abstractmethod
```

```
class MyClass(metaclass = ABCMeta):
    @staticmethod
    @abstractmethod
    def static_func(self, x): # Абстрактний статичний метод
        pass

    @classmethod
    @abstractmethod
    def class_func(self, x): # Абстрактний метод класа
        pass
```

1.11. Обмеження доступу до ідентифікаторів всередині класу

Всі ідентифікатори всередині класу в мові Python є відкритими, тобто доступні для безпосередньої зміни. Для імітації приватних ідентифікаторів можна скористатися методами *__getattr__(), __getattribute__()* і *__setattr__()*, які перехоплюють звернення до атрибутів класу. Крім того, можна скористатися ідентифікаторами, назви яких починаються з двох символів підкреслення. такі ідентифікатори називаються *псевдоприватними*. Псевдоприватні ідентифікатори доступні лише всередині класу, але не поза ним. Проте, змінити ідентифікатор через екземпляр класу все одно можна, знаючи, яким чином спотворюється назва ідентифікатора. Наприклад, ідентифікатор *__privateVar* всередині класу *Classl* буде доступний за ім'ям *_Class1__privateVar*. Як можна бачити, тут перед ідентифікатором додається назва класу з попереднім символом підкреслення. Наведемо приклад використання псевдоприватних ідентифікаторів:

```
class MyClass:
    def __init__(self, x):
        self.__privateVar = x
    def set_var(self, x): # Зміна значення
        self.__privateVar = x
    def get_var(self): # Отримання значення
        return self.__privateVar

c = MyClass(10)                # Створюємо екземпляр класу
print(c.get_var())             # Виведе: 10
c.set_var(20)                  # Змінюємо значення
print(c.get_var())             # Виведе: 20
try:                           # Перехоплюємо помилки
    print(c.__privateVar)      # Помилка !!!
except AttributeError as msg:
    print(msg)                 # Виведе: 'MyClass' object has
                               # no attribute '__privateVar'
c._MyClass__privateVar = 50    # Значення псевдоприватних атрибутів
                               # все одно можна змінити
print(c.get_var())             # Виведе: 50
```

Можна також обмежити перелік атрибутів, дозволених для екземплярів класу. Для цього дозволені атрибути вказуються всередині класу в атрибуті *__slots__*. В якості значення атрибуту можна присвоїти рядок або список рядків з назвами ідентифікаторів. Якщо проводиться спроба звернення до атрибуту, не вказаного в *__slots__*, генерується виняток *AttributeError*:

```
class MyClass:
```

```
    __slots__ = ["x", "y"]
    def __init__(self, a, b):
        self.x, self.y = a, b

c = MyClass(1, 2)
print(c.x, c.y)        # Виведе: 1 2
c.x, c.y = 10, 20      # Змінюємо значення атрибутів
print(c.x, c.y)        # Виведе: 10 20
try:                   # Перехоплюємо винятки
    c.z = 50           # Атрибут z не вказано в __slots__,
                       # тому генерується виняток
except AttributeError as msg:
    print(msg)         # Виведе 'MyClass' object has no attribute 'z'
```

1.12. Властивості класу

Python дозволяє програмісту звертатися до змінних примірників безпосередньо - без проміжних *get*- і *set*-методів, часто використовуваних в Java і інших об'єктно-орієнтованих мовах. При відсутності *get*- і *set*-методів код класів Python стає більш лаконічним і наочним, але в деяких ситуаціях *get*- і *set*-методи виявляються зручними. Уявіть, що значення потрібно перевірити перед збереженням у змінній екземпляру або ж значення атрибуту повинно обчислюватися «на ходу». В обох випадках *get*- і *set*-методи вирішать завдання, але за рахунок втрати зручного доступу до змінних Python.

У таких ситуаціях слід використовувати *property* (властивість). Властивості об'єднують можливість передачі звернень до змінної екземпляру через аналоги *get*- і *set*-методів і прямолінійні звернення до змінних екземплярів в крапковій нотації.

Усередині класу можна створити ідентифікатор, через який в подальшому будуть проводитися операції отримання і зміни значення будь-якого атрибуту, а також його видалення. Створюється такий ідентифікатор за допомогою функції *property()*. Формат функції:

```
<Властивість> = property(<Читання> [, <Запис> [, <Видалення> [,
<Рядок документування>]]])
```

У перших трьох параметрах вказуються посилання на відповідні методи класу. При спробі отримати значення буде викликаний метод, зазначений в першому параметрі. При операції присвоювання значення буде викликаний метод, зазначений у другому параметрі, - цей метод повинен приймати один параметр. У разі видалення атрибута викликається метод, зазначений в третьому параметрі. Якщо в якості якого-небудь параметра встановлено значення *None*, то це означає, що відповідний метод не підтримується. Розглянемо властивості класу на прикладі:

```
class MyClass:
    def __init__(self, value):
        self.__var = value
    def get_var (self):        # Читання
        return self.__var
    def set_var (self, value): # Запис
        self.__var = value
```

```
    def del_var (self):         # Видалення
        del self.__var
    v = property(get_var, set_var, del_var, "Рядок документування")

c = MyClass(5)
c.v = 35    # Викликається метод set_var()
print(c.v)  # Викликається метод get_var()
del c.v     # Викликається метод del_var()
```

Python підтримує альтернативний метод визначення властивостей - за допомогою методів *getter()*, *setter()* і *deleter()*, які використовуються в декораторах. Відповідний приклад:

```
class MyClass:
    def __init__(self, value):
        self.__var = value
    @property
    def v (self):       # Читання
        return self.__var
    @v.setter
    def v(self, value): # Запис
        self.__var = value
    @v.deleter
    def v (self):       # Видалення
        del self.__var

c = MyClass(5)
c.v = 35    # Запис
print(c.v)  # Читання
del c.v     # Видалення
```

Розглянемо ще приклад:

```
class Temperature:
    def __init__(self):
        self._temp_fahr = 0

    @property
    def temp(self):
        return (self._temp_fahr - 32) * 5/9
```

Без *set*-методу така властивість доступна тільки для читання. Щоб змінити властивість, необхідно додати *set*-метод:

```
    @temp.setter
    def temp(self, new_temp):
        self._temp_fahr = new_temp * 9/5 + 32
```

Тепер стандартний синтаксис з крапкою може використовуватися як для читання, так і для запису властивості *temp*. Зверніть увагу: ім'я методу залишається незмінним, але декоратор з імені властивості (*temp* в даному випадку) і суфікса *.setter* показує, що визначається *set*-метод для властивості *temp*:

```
>>> t = Temperature()
>>> t._temp_fahr
0
>>> t.temp
-17.77777777777778
>>> t.temp = 34
>>> t._temp_fahr
```

```
93.2
>>> t.temp
34.0
```

Значення 0 в *_temp_fahr* перетворюється в шкалу Цельсія перед поверненням. Значення 34 перетвориться назад в шкалу Фаренгейта *set*-методом.

Одна з великих переваг можливості створення властивостей в Python полягає в тому, що в ході розробки можуть використовуватися звичайні змінні екземплярів, які потім будуть легко перетворені у властивості там, де це знадобиться, без зміни клієнтського коду звернення.

Є можливість визначити абстрактну властивість - в цьому випадку всі реалізуюч її методи повинні бути перевизначені в підкласі. Виконується це за допомогою знайомого нам декоратора *@abstractmethod* з модуля *abc*. Приклад визначення абстрактної властивості:

```
from abc import ABCMeta, abstractmethod
class MyClass1(metaclass = ABCMeta):
    def __init__(self, value):
        self.__var = valua
    @property
    @abstractmethod
    def v(self):              # Читання
        return self.__var
    @v.setter
    @abstractmethod
    def v(self, value):       # Запис
        self.__var = value
    @v.deleter
    @abstractmethod
    def v(self):              # Видалення
        del self.__var
```

1.13. Декоратори класів

У мові Python, крім декораторів функцій, підтримуються декоратори класів, які дозволяють змінити поведінку самих класів. Як параметр декоратор приймає посилання на об'єкт класу, поведінку якого необхідно змінити, і повинен повертати посилання на той же клас або будь-який інший. Приклад декорування класу:

```
def deco(C):                          # Приймає об'єкт класу
    print("Всередині декоратора")     # Здійснює якісь дії
    return C                          # Повертає об'єкт класу
@deco
class MyClass:
    def __init__(self, value):
        self.v = value

c = MyClass(5)
print(c.v)
```

2. ЗАВДАННЯ

2.1. Домашня підготовка до роботи

1. Вивчити теоретичний матеріал.

2.2. Виконати в лабораторії

1. Написати програму яка створює клас з атрибутами об’єкту заданими в таблиці. Для класу повинен бути перевантажені оператори порівняння за останнім атрибутом (наприклад, для автомобіля операції порівняння за швидкістю). Атрибути об’єкту повинні бути приватними з доступом через властивості класу або *get*- і *set*-методи. На базі цього класу створити клас, який зберігає список об’єктів та підтримує методи, які дозволяють через діалоговий режим виконувати такі операції:
    a. Вивести весь список.
    b. Додавати елементи до списку.
    c. Відсортувати список за заданим атрибутом.
    d. Видаляти елементи за заданим атрибутом.
    e. Видаляти елемент за заданим індексом.
    f. Виводити всі елементи за заданим атрибутом.

Табл. 3

Варіант	Об’єкт	Атрибути
1	Автомобіль	Виробник, модель, рік випуску, пробіг, макс. швидкість
2	Книга	Автор, назва, видавництво, рік видання, кількість сторінок
3	Пасажирський літак	Виробник, модель, рік випуску, кількість пасажирів, макс. Швидкість
4	Студент	ПІБ, група, рік вступу, середній бал
5	Користувач	Логін, пароль, телефон, е-мейл, кількість авторизацій
6	Фільм	Назва, режисер, рік випуску, бюджет, тривалість, розмір файлу
7	Пиво	Назва, виробник, міцність, ціна, термін зберігання в днях
8	Працівник	Назва фірми, посада, телефон, е-мейл, оклад
9	Пісня	Виконавець, назва, альбом, тривалість, розмір файлу
10	Смартфон	Виробник, модель, ціна, ємність батареї, обсяг пам’яті
11	Процесор	Виробник, модель, тактова частота, ціна, розсіювана потужність
12	Квартира	Власник, адреса, поверх, площа, кількість кімнат
13	Собака	Порода, кличка, вік, вага
14	Ноутбук	Виробник, процесор, ціна, діагональ, час роботи від акумулятора
15	Країна	Назва, столиця, населення, площа, ВВП
16	Нобелівський лауреат	Прізвище, рік народження, країна, рік вручення, галузь,

17	Купюра	Валюта, номінал, рік випуску, курс до долара
18	Дисципліна	Назва, викладач, семестр, балів за поточний контроль, балів за екзамен
19	Річка	Назва, континент, довжина, басейн, середньорічний стік
20	Місто	Назва, країна, населення, рік заснування, площа

3. ЗМІСТ ЗВІТУ

1. Мета роботи.
2. Повний текст завдання згідно варіанту.
3. Лістинг програми.
4. Результати роботи програм (у текстовій формі та скріншот).
5. Висновок.

4. КОНТРОЛЬНІ ЗАПИТАННЯ

1. Що таке клас в Python? Які його основні характеристики?
2. Опишіть базові принципи ООП.
3. Що таке екземпляр класу? Яким чином можна здійснити його створення?
4. Дайте визначення атрибута класу і опишіть його основні особливості.
5. Що таке методи класу? Які особливості створення та виклику методу?
6. Які відмінності закритих методів від звичайних?
7. У чому полягає перевага використання конструктора *__init__()* при створенні класу?
8. Як здійснюється перевантаження спеціальних методів классу в Python?
9. Як реалізується принцип успадкування в Python? Наведіть приклади.

5. СПИСОК ЛІТЕРАТУРИ

1. Learn to Program with Python 3. A Step-by-Step Guide to Programming, Second Edition / Irv Kalb. – Mountain View: Apress, 2018. – 361 p.
2. The Python Workbook. A Brief Introduction with Exercises and Solutions, Second Edition / Ben Stephenson. – Cham: Springer, 2014. – 218 p.
3. Python Pocket Reference, Fifth Edition / Mark Lutz. – Sebastopol: O'Reilly Media, Inc., 2014. – 264 p.
4. Learn Python 3 the Hard Way / Zed A. Shaw. – Boston: Addison-Wesley, 2017. – 321 p.
5. A Python Book: Beginning Python, Advanced Python, and Python Exercises / Dave Kuhlman. – Boston: MIT, 2013. – 278 p.

НАВЧАЛЬНЕ ВИДАННЯ

Об’єктно-орієнтоване програмування

МЕТОДИЧНІ ВКАЗІВКИ
до лабораторної роботи № 9
з курсу «Програмування скриптовими мовами»
для студентів спеціальності
«Кібербезпека»

Укладач: Я. Р. Совин, канд. техн. наук, доцент