ММП 2025/2026

Викладач Канцедал Георгій Олегович

Об’єктно-орієнтоване програмування (ООП)

Об’єктно-орієнтоване програмування (ООП) — це парадигма програмування, зосереджена навколо об'єктів, які є екземплярами класів.
Принципи ООП включають інкапсуляцію, успадкування та поліморфізм, що сприяє модульності та повторному використанню коду.

Python підтримує ООП завдяки системі класів та об'єктів, що дозволяє створювати класи та ініціалізувати їхні екземпляри.

Класи інкапсулюють дані та методи, а об'єкти представляють конкретні екземпляри цих класів.

Об’єктно-орієнтоване програмування (ООП)

Переваги:

- **Модульність** – код розбивається на окремі класи та об'єкти, що покращує організацію та спрощує підтримку.
- **Повторне використання коду** – завдяки **успадкуванню** можна створювати нові класи на основі вже існуючих, що зменшує дублювання коду.
- **Інкапсуляція** – приховує внутрішню реалізацію класів, забезпечуючи безпеку даних і спрощуючи взаємодію між об'єктами.
- **Поліморфізм** – дозволяє використовувати один і той самий інтерфейс для різних типів об'єктів, що підвищує гнучкість коду.
- **Зручність тестування** – розбиття на класи та об'єкти полегшує тестування окремих компонентів.

Недоліки:

- **Складність** – проєктування об'єктно-орієнтованої системи може бути складнішим, ніж використання простих імперативних підходів.
- **Витрати на продуктивність** – динамічне створення об'єктів та виклики методів можуть мати більші накладні витрати порівняно з процедурним програмуванням.
- **Надмірність коду** – у деяких випадках ООП вимагає більше коду, ніж імперативний або функціональний підхід.
- **Проблеми з багатопотоковістю** – взаємодія між об'єктами, що змінюють стан, може ускладнювати паралельне програмування.
- **Не завжди виправдане використання** – для невеликих скриптів або простих завдань ООП може бути надмірним, ускладнюючи реалізацію.

Інкапсуляція (Encapsulation)

Інкапсуляція – це механізм приховування деталей реалізації класу від зовнішнього світу та надання доступу до них лише через спеціально визначені методи.

Переваги:

- Захист внутрішніх даних від несанкціонованої модифікації.
- Спрощення роботи з об'єктами через визначені інтерфейси.

```python
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance  # Приватна змінна

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
        else:
            print("Недостатньо коштів")

    def get_balance(self):
        return self.__balance  # Доступ до прихованого атрибута через метод

# Використання класу
account = BankAccount(1000)
account.deposit(500)
print(account.get_balance())  # Виведе: 1500
account.withdraw(2000)  # Недостатньо коштів
```

Інкапсуляція (Encapsulation)

Ключові моменти:

- Атрибут __balance є приватним (з двома підкресленнями).
- Доступ до балансу можливий тільки через метод get_balance().

Успадкування (Inheritance)

Успадкування дозволяє створювати нові класи на основі вже існуючих, що допомагає уникати дублювання коду.

Переваги:

- Повторне використання коду.
- Спрощення структури програмного коду.

```python
# Базовий клас
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return "Звуки тварини"

# Підклас (дочірній клас)
class Dog(Animal):
    def speak(self):
        return "Гав-гав!"

class Cat(Animal):
    def speak(self):
        return "Мяу-мяу!"

# Використання класів
dog = Dog("Бобік")
cat = Cat("Мурка")

print(dog.name, "говорить:", dog.speak())  # Бобік говорить: Гав-гав!
print(cat.name, "говорить:", cat.speak())  # Мурка говорить: Мяу-мяу!
```

Успадкування (Inheritance)

Ключові моменти:

- Dog та Cat успадковують методи і змінні класу Animal.
- Вони перевизначають метод speak(), що є проявом **поліморфізму**.

Поліморфізм (Polymorphism)

Поліморфізм дозволяє об'єктам різних класів мати однакові методи, що виконують різні дії.

Переваги:

- Гнучкість у роботі з об'єктами.
- Дозволяє використовувати один інтерфейс для різних реалізацій.

```python
class Bird:
    def speak(self):
        return "Пташка співає"

class Parrot(Bird):
    def speak(self):
        return "Привіт! Я папуга!"

class Crow(Bird):
    def speak(self):
        return "Кар-кар!"

# Функція, яка використовує поліморфізм
def make_sound(bird):
    print(bird.speak())

# Використання
parrot = Parrot()
crow = Crow()

make_sound(parrot)  # Виведе: Привіт! Я папуга!
make_sound(crow)    # Виведе: Кар-кар!
```

Поліморфізм (Polymorphism)

Ключові моменти:

- Метод speak() перевизначений у дочірніх класах Parrot і Crow.
- Функція make_sound() працює з будь-яким об'єктом, що має метод speak().

Абстракція (Abstraction)

Абстракція приховує складність реалізації та дозволяє працювати лише з необхідними деталями.

Переваги:

- Спрощує використання об'єктів.
- Зменшує залежність користувача від деталей реалізації.

```python
from abc import ABC, abstractmethod

# Абстрактний клас
class Vehicle(ABC):
    @abstractmethod
    def move(self):
        pass  # Абстрактний метод, який має бути реалізований у дочірньому класі

# Дочірні класи, які реалізують метод move()
class Car(Vehicle):
    def move(self):
        return "Автомобіль їде по дорозі"

class Boat(Vehicle):
    def move(self):
        return "Човен пливе по воді"

# Використання класів
car = Car()
boat = Boat()

print(car.move())  # Виведе: Автомобіль їде по дорозі
print(boat.move())  # Виведе: Човен пливе по воді
```

Абстракція (Abstraction)

Ключові моменти:

- Абстрактний клас Vehicle містить метод move(), який не має реалізації.
- Класи Car та Boat реалізують move() по-різному.
- Абстракція запобігає створенню об'єктів без визначених методів.

Абстракція (Abstraction)

Хтось вже здогадався що попередній приклад не показує всієї її краси тому едванс рішення:

```python
from abc import ABC, abstractmethod


3 usages  Георгий Канцедал
class BasePipeline(ABC):
    1 usage  Георгий Канцедал
    @abstractmethod
    def setup(self):
        pass

    1 usage  Георгий Канцедал
    @abstractmethod
    def tear_down(self):
        pass
```

```python
    1 usage  Георгий Канцедал
    def __get_pipeline(self, pipeline_class):
        if self.start_from_saved_state and is_file_exist(self.pipeline_path):
            pipeline = self.__load_pipeline()
            if isinstance(pipeline, BasePipeline):
                pipeline.setup()
            return pipeline
        else:
            return pipeline_class()

    1 usage  Георгий Канцедал
    def __save_pipeline_if_needed(self):
        if self.save_state_on_every_step:
            timer = Timer("Saving pipeline").start()
            if isinstance(self.pipeline, BasePipeline):
                self.pipeline.tear_down()
            save_to_pickle(self.pipeline, self.pipeline_path)
            timer.end()
```

Кінець