



ПАРАДИГМИ ПРОГРАМУВАННЯ ТА ООП

A decorative graphic on the left side of the slide, consisting of a network of white lines and small circles on a blue gradient background, resembling a circuit board or a neural network.

План:

1. Види парадигм програмування;
2. Декоратори;
3. Основні принципи об'єктно-орієнтованого програмування;
4. Замикання;
5. Анонімні функції



Парадигми програмування:

1. Імперативне програмування;
2. Процедурне (структурне) програмування;
3. Об'єктно-орієнтоване програмування (ООП);
4. Функціональне програмування;
5. Декларативне програмування;
6. Метaproграмування.

Імперативне програмування є однією з базових парадигм, підтримуваних мовою Python. Його сутність полягає у визначенні алгоритму у вигляді послідовності інструкцій, що безпосередньо змінюють стан обчислювальної системи. У такому підході програміст формулює не лише кінцеву мету, але й конкретні кроки, які необхідно виконати для її досягнення.

```
x = 0
for i in range(5):
    x += i
print(x)
```

Керування потоком виконання здійснюється за допомогою явних конструкцій, таких як:

- Умовні оператори (if-else): дозволяють виконувати різні блоки коду залежно від певних умов.
- Цикли (for, while): забезпечують багаторазове виконання набору інструкцій.
- Підпрограми (процедури та функції): інкапслюють послідовності команд для їх повторного використання.

```
#Процедурне
def factorial(n):
    if n == 1:
        return 1
    return n * factorial(n - 1)

print(factorial(5))
```

Процедурне програмування - це підпарадигма імперативного програмування, де основна увага приділяється структуризації коду за допомогою **процедур** (procedures) або **функцій** (functions). Це еволюція простого імперативного підходу, яка дозволяє організувати код у логічні блоки, які можуть бути викликані з будь-якої точки програми.

Процедурне (структурне) програмування

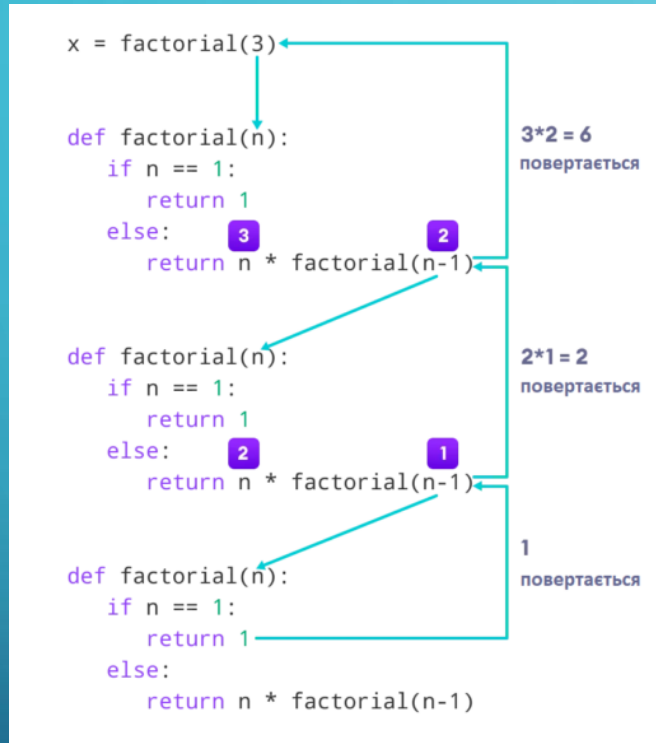
- Програма організовується у вигляді функцій (процедур).
- Використовуються функції для поділу коду на логічні блоки.

РЕКУРСІЯ

Рекурсія — це процес визначення чогось у межах самого себе. Прикладом у фізичному світі може бути розміщення двох дзеркал одне проти одного. Будь-який об'єкт між ними відображатиметься рекурсивно.

Наша рекурсія закінчується, коли число зменшується до 1. Це називається базовою умовою. Кожна рекурсивна функція повинна мати базову умову, яка зупиняє рекурсію, інакше функція буде нескінченно викликати саму себе. Інтерпретатор Python обмежує глибину рекурсії, щоб уникнути нескінченних рекурсій, що призводять до переповнення стека.

За замовчуванням максимальна глибина рекурсії дорівнює 1000. Якщо межа перевищена, то виникає помилка `RecursionError`. Розглянемо приклад нескінченної рекурсії:



• Переваги рекурсії

- Рекурсивні функції роблять код чистішим.
- Складну задачу можна розбити на простіші підзадачі за допомогою рекурсії.
- Генерація послідовності простіша з рекурсією, ніж з використанням будь-якої вкладеної ітерації.

Недоліки рекурсії

- Іноді логіку рекурсії важко зрозуміти.
- Рекурсивні виклики дорогі (неефективні), тому що забирають багато пам'яті та часу.
- Рекурсивні функції важко відлагоджувати.

```
#ООП
class Animal:
    def speak(self):
        pass

class Dog(Animal):
    def speak(self):
        return "Woof!"

d = Dog()
print(d.speak())
```

Об'єктно-орієнтоване програмування (ООП) — це парадигма програмування, яка базується на концепції об'єктів. На відміну від процедурного підходу, де логіка і дані існують окремо, в ООП вони об'єднуються в єдині сутності, що робить код більш структурованим, модульним та гнучким.

ООП базується на чотирьох ключових принципах, які визначають його структуру і поведінку:

- Інкапсуляція
- Наслідування
- Поліморфізм
- Абстракція

```
#Функціональне програмування
import math
def calculate_circle_area(radius):
    """
    Функція для обчислення площі кола.
    """
    return math.pi * radius**2

# Використовуємо функцію без зміни стану
radius = 5
area = calculate_circle_area(radius)
print("Площа кола з функціональним підходом:", area)
```

Функціональне програмування — це парадигма програмування, що розглядає обчислення як обчислення математичних функцій, уникаючи зміни стану та мутабельних даних. На відміну від імперативного підходу, який зосереджений на послідовних командах, функціональне програмування акцентує увагу на **виразах** та **функціях**, які не мають побічних ефектів.

Функціональне програмування ґрунтується на кількох ключових принципах:

- **Чисті функції (Pure Functions).** Це функції, які при однакових входних даних завжди повертають однаковий результат, і не мають побічних ефектів (наприклад, не змінюють глобальні змінні або не здійснюють операції введення/виведення). Це робить код передбачуваним, легшим для тестування та паралелізації.
- **Незмінність даних (Immutability).** Усі дані розглядаються як незмінні. Замість того, щоб змінювати існуючі змінні, ви створюєте нові, які є результатом застосування функцій. Цей підхід усуває побічні ефекти та робить програми безпечнішими в багатопотоковому середовищі.
- **Функції як об'єкти першого класу (First-Class Functions).** Функції можуть бути присвоєні змінним, передані як аргументи іншим функціям і повернені як результат. Це дозволяє створювати функції вищого порядку (Higher-Order Functions), які приймають інші функції як аргументи або повертають їх.

#Функціональний відмінності від імперативного

```
def add_one_to_list_functional(numbers):
```

```
    # Ця функція не змінює оригінальний список.
```

```
    # Вона створює новий список і повертає його.
```

```
    # Ми використовуємо "спискове включення" (list comprehension)
```

```
    return [number + 1 for number in numbers]
```

```
# Наш початковий список
```

```
my_numbers_immutable = [1, 2, 3, 4]
```

```
# Викликаємо функцію, яка повертає новий список
```

```
new_numbers = add_one_to_list_functional(my_numbers_immutable)
```

```
# Тепер перевіряємо обидва списки
```

```
print(f"Оригінальний список (незмінний): {my_numbers_immutable}")
```

```
# Результат: [1, 2, 3, 4]
```

```
print(f"Новий список: {new_numbers}")
```

```
# Результат: [2, 3, 4, 5]
```

#імперативний відмінності

```
def add_one_to_list(numbers):
```

```
    # Ця процедура змінює оригінальний список "numbers"
```

```
    for i in range(len(numbers)):
```

```
        numbers[i] += 1
```

```
# Наш початковий список
```

```
my_numbers = [1, 2, 3, 4]
```

```
# Викликаємо процедуру, яка змінює список
```

```
add_one_to_list(my_numbers)
```

```
# Тепер перевіряємо, що сталося з "my_numbers"
```

```
print(f"Оригінальний список після зміни: {my_numbers}")
```

```
# Результат: [2, 3, 4, 5]
```

```
# Список змінився! Це і є побічний ефект.
```

Декларативне програмування — це парадигма, що фокусується на "що" потрібно зробити, а не на "як". На відміну від імперативного підходу, де ви надаєте покрокові інструкції, декларативне програмування дозволяє вам просто **описати бажаний результат**, а система сама визначає, як його досягти. Ця парадигма широко застосовується в мовах, де синтаксис більше схожий на опис, а не на команди.

```
SELECT name FROM users WHERE age > 30;
```

SQL (Structured Query Language): Ви описуєте, які дані ви хочете отримати. Наприклад, замість того, щоб вказувати комп'ютеру "пройдися по кожному рядку таблиці, перевірити, чи вік більший за 30, і якщо так, скопіювати ім'я в новий список", ви просто пишете:

HTML (HyperText Markup Language): Ви описуєте, як має виглядати веб-сторінка. Ви не даєте інструкцій, як "намалювати" заголовок чи абзац. Ви просто заявляєте, що це має бути заголовок (<h1>) або абзац (<p>), а браузер сам вирішує, як це відобразити.

```
import re

text = "Зв'яжіться з нами за адресою info@example.com або support@anotherdomain.net. Наша пошта: my.email@test.co.uk."

# Це і є регулярний вираз. Він описує, як виглядає email-адреса.
# Тут ми описуємо, що email складається з:
#   - набору символів, цифр, крапок чи підкреслень ([\w.-]+)
#   - символу @ (@)
#   - ще одного набору символів ([\w.-]+)
#   - крапки (\.)
#   - і, нарешті, 2-3 літер ([\w]{2,3})
email_pattern = r'[\w.-]+@[ \w.-]+\.[\w]{2,3}'

# Використовуємо функцію re.findall() для пошуку всіх збігів
# Ця функція знаходить усі підрядки, що відповідають шаблону
found_emails = re.findall(email_pattern, text)

# Виводимо знайдені email-адреси
print("Знайдені email-адреси:")
for email in found_emails:
    print(email)
```

Регулярні вирази (Regular Expressions): Ви описуєте шаблон тексту, який шукаєте, і не надаєте інструкцій, як його знайти. Наприклад, щоб знайти всі email-адреси, ви описуєте їхню структуру, а не пишете алгоритм посимвольного пошуку.

Метапрограмування — це техніка програмування, що дозволяє програмам **писати або змінювати інші програми** як під час компіляції, так і під час виконання. Це програмний код, який оперує іншим кодом, розглядаючи його як дані. Основна мета — підвищити гнучкість, зменшити дублювання коду та автоматизувати рутинні завдання, які інакше довелося б писати вручну.

Метапрограмування може бути реалізоване двома основними способами:

- Під час виконання (Run-time). Код генерується або модифікується під час роботи програми. Це дозволяє створювати гнучкі системи, які можуть адаптуватися до мінливих умов. Приклад: динамічна генерація функцій на основі даних, отриманих від користувача. Python, Ruby і JavaScript відомі своєю підтримкою цього типу метапрограмування.
- Під час компіляції (Compile-time). Код генерується або модифікується перед запуском програми. Це дозволяє створювати дуже ефективний та оптимізований код, оскільки вся робота виконується на етапі компіляції. Приклади: C++-шаблони (templates) та макроси в Rust і Lisp.

```
def my_decorator(func):
    def wrapper():
        print("Щось відбувається до виклику функції.")
        func()
        print("Щось відбувається після виклику функції.")
    return wrapper

@my_decorator
def say_hello():
    print("Привіт!")

say_hello()
```

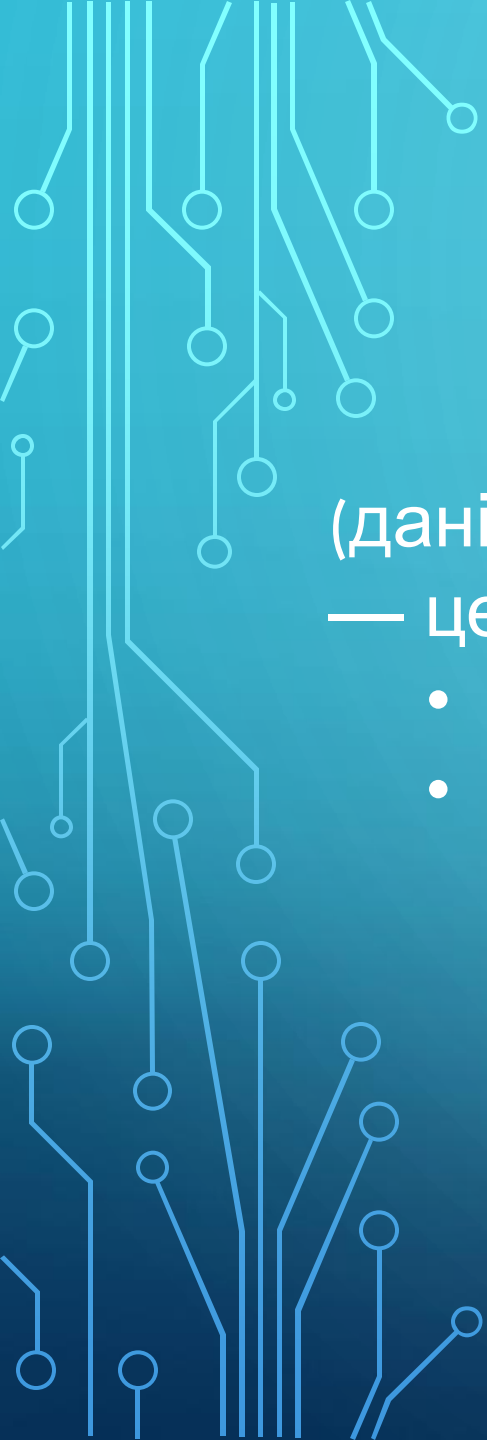
ДЕКОРАТОРИ З АРГУМЕНТАМИ У PYTHON

- Декоратори з аргументами у Python — це функції, які самі приймають аргументи, а потім повертають іншу функцію (декоратор), що вже буде використана для декорування. Цей механізм дозволяє створювати гнучкіші та багаторазові декоратори, які можна налаштовувати для різних потреб.

```
def message_decorator(message_prefix):  
    # 2. Середня функція: це сам декоратор, який приймає функцію для декорування  
    def decorator(func):  
        # 3. Внутрішня функція: обгортка, що містить нову логіку  
        def wrapper(*args, **kwargs):  
            # Використовуємо аргумент 'message_prefix' із зовнішньої функції  
            print(f"[{message_prefix}] Функція '{func.__name__}' починає роботу...")  
            result = func(*args, **kwargs)  
            print(f"[{message_prefix}] Функція '{func.__name__}' завершила роботу.")  
            return result  
        return wrapper  
    return decorator  
  
# Використання декоратора з аргументом  
@message_decorator(message_prefix="INFO")  
def add_numbers(a, b):  
    return a + b  
  
@message_decorator(message_prefix="WARNING")  
def subtract_numbers(a, b):  
    return a - b  
  
# Виклики функцій, які демонструють різні префікси  
print("Приклад з додаванням:")  
add_numbers(10, 5)  
  
print("\nПриклад з відніманням:")  
subtract_numbers(20, 8)
```

Основна особливість декораторів з аргументами полягає в їхній **трирівневій структурі**, яка дозволяє їм зберігати стан. Ця структура включає:

- **Зовнішня функція (фабрика декораторів):** Ця функція приймає аргументи, які налаштовують поведінку декоратора. Вона не є самим декоратором, а лише створює його.
- **Середня функція (власне декоратор):** Ця функція приймає декоровану функцію як аргумент. Вона створюється і повертається зовнішньою функцією.
- **Внутрішня функція (обгортка):** Це функція, яка безпосередньо замінює оригінальну функцію. Вона отримує аргументи, які передаються при виклику оригінальної функції, і містить нову логіку. Вона **замикає** навколо себе аргументи, передані в зовнішню функцію, та саму декоровану функцію.



Об'єктно-орієнтоване програмування в Python

Об'єкт — це будь-яка сутність, яка має атрибути (дані) та поведінку (методи/функції). Наприклад, кошеня — це об'єкт. В нього є:

- атрибути — ім'я, вік, колір тощо.
- поведінка — бігає, нявкає, спить і т.д.

Клас — це креслення (план) цього об'єкта.

Об'єкт — це набір даних (змінних) та методів (функцій).

Клас — це креслення (план) об'єкта. Ми можемо уявити клас як ескіз (прототип) будинку. Він містить всі деталі про поверхи, двері, вікна тощо. На основі цих описів ми будуємо будинок. Будинок — це об'єкт. Оскільки з одного й того самого опису можна побудувати багато будинків, ми можемо створити багато об'єктів з одного класу.

Для створення класу в Python використовується ключове слово `class`:

```
class ClassName:  
    # Визначення класу
```

Приклад:

```
class Bike:  
    name = ""  
    gear = 0
```

Об'єкт — це екземпляр класу. Наприклад, `Bike` — це клас і ми можемо створити об'єкти типу `bike1`, `bike2` з цього класу.

```
class Bike:  
    name = ""  
    gear = 0  
  
# Створюємо об'єкт класу  
bike1 = Bike()
```

ДОСТУП ДО АТРИБУТІВ КЛАСУ ЧЕРЕЗ ОБ'ЄКТИ

Оператор `.` використовується для доступу до атрибутів класу.

```
# Створюємо клас
class Bike:
    name = ""
    gear = 0

# Створюємо об'єкт класу
bike1 = Bike()

# Отримуємо доступ до атрибутів об'єкта та присвоюємо нові значення
bike1.gear = 11
bike1.name = "Mountain Bike"

# Виводимо дані об'єкта на екран
print(f"Name: {bike1.name}, Gears: {bike1.gear} ")
```

Методи в Python

Функція, визначена всередині класу, називається **методом**.

```
class Cat:
    name = ""
    age = 0

    def showOldCat(self):
        print(f"{cat1.name} is {cat1.age} years old")

# Створення об'єкта cat1
cat1 = Cat()
cat1.name = "Blu"
cat1.age = 10

# Створення іншого об'єкта — cat2
cat2 = Cat()
cat2.name = "Woo"
cat2.age = 15
cat2.showOldCat()

# Доступ до атрибутів класу Cat
print(f"{cat1.name} is {cat1.age} years old")
print(f"{cat2.name} is {cat2.age} years old")
```

Аргумент **self** обов'язково передається в методи класу, тому що він є **посиланням на конкретний об'єкт**, для якого викликається метод. Він дозволяє методам отримувати доступ до даних (атрибутів) та інших методів саме того екземпляра класу, з яким вони працюють.

Посилання, а не змінна: **self** — це не ключове слово, а лише загальноприйняте ім'я для першого параметра. Ви могли б назвати його як завгодно (наприклад, **this**, як в інших мовах), але використання **self** є стандартом Python.

Чому не глобальна змінна?: Без **self**, метод не знав би, з яким об'єктом йому працювати. Якби метод **bark()** звертався до глобальної змінної **cat1** або **cat2**, він був би жорстко прив'язаний до них і не міг би використовуватися для інших об'єктів. **self** робить методи універсальними, дозволяючи їм працювати з будь-яким екземпляром класу.

```
class Cat:|
    name = ""
    age = 0

# Створення об'єкта cat1
cat1 = Cat()
cat1.name = "Blu"
cat1.age = 10

# Створення іншого об'єкта – cat2
cat2 = Cat()
cat2.name = "Woo"
cat2.age = 15

# Доступ до атрибутів класу Cat
print(f"{cat1.name} is {cat1.age} years old")
print(f"{cat2.name} is {cat2.age} years old")
```

Ми визначили клас `Cat` із двома атрибутами: `name` та `age`. Далі створили його екземпляри — об'єкти `cat1` і `cat2`, які є посиланнями на нові об'єкти цього класу. Доступ до атрибутів та їх зміна виконуються через ім'я екземпляра з використанням оператора `.`

Конструктори в Python

```
class Dog:
    # Конструктор
    def __init__(self, name, age):
        # Атрибути (дані) об'єкта ініціалізуються тут
        self.name = name # Присвоюємо ім'я, передане в конструктор
        self.age = age    # Присвоюємо вік, переданий в конструктор

    def greet(self):
        print(f"Привіт! Я пес {self.name}, мені {self.age} років.")

# Створення об'єкта Dog. Це автоматично викликає __init__
my_dog = Dog("Рекс", 5)

my_dog.greet() # Результат: Привіт! Я пес Рекс, мені 5 років.
```

Конструктор в Python — це особливий метод, який автоматично викликається при **створенні нового екземпляра класу**. Його основна мета — **ініціалізувати (налаштувати)** об'єкт, присвоївши початкові значення його атрибутам (даним).

Тут `__init__()` — це функція-конструктор, яка викликається щоразу, коли створюється новий об'єкт цього класу.

Переваги використання конструкторів в Python

Ініціалізація об'єктів. Конструктори використовуються для ініціалізації об'єктів класу. Вони дозволяють встановити значення за замовчуванням для членів класу, а також ініціалізувати об'єкт користувацькими даними.

Простота реалізації. Конструктори легко реалізувати в Python за допомогою методу `__init__()`.

Покращення читабельності коду. Конструктори покращують читабельність коду, оскільки дають зрозуміти, які значення ініціалізуються і як вони ініціалізуються.

Інкапсуляція. Конструктори можуть використовуватися для забезпечення інкапсуляції, гарантуючи, що члени об'єкта ініціалізуються коректно та контрольовано.

Деструктори в Python

```
class FileHandler:
    def __init__(self, filename):
        print(f"Об'єкт {filename} створено.")
        self.file = open(filename, 'w')
        self.file.write("Дані записано.")

    def __del__(self):
        # Деструктор для закриття файлу
        print("Деструктор викликано.")
        if not self.file.closed:
            self.file.close()
            print(f"Файл закрито.")

# Створюємо об'єкт
handler = FileHandler("example.txt")

# Видаляємо посилання на об'єкт
del handler

# Тут, після видалення останнього посилання, __del__ буде викликано
# збирачем сміття.
print("Програма завершується.")
```

Деструктор в Python — це особливий метод `__del__`, який викликається, коли об'єкт знищується або видаляється зі сховища пам'яті. Його основна роль — виконувати операції "очищення" або звільнення ресурсів, які були виділені об'єктом під час його життєвого циклу.

На відміну від конструктора `__init__`, який викликається негайно при створенні об'єкта, деструктор `__del__` викликається тоді, коли **лічильник посилань** на об'єкт досягає нуля. Це означає, що немає жодної змінної, яка б посилалася на цей об'єкт, і він стає доступним для **збирання сміття (garbage collection)**.

```
# Батьківський клас
class Animal:

    def eat(self):
        print( "I can eat!")

    def sleep(self):
        print("I can sleep!")

# Дочірній клас
class Dog(Animal):

    def bark(self):
        print("I can bark! Woof woof!!")

# Створюємо об'єкт класу Dog
dog1 = Dog()

# Викликаємо методи батьківського класу
dog1.eat()
dog1.sleep()

# Викликаємо метод дочірнього класу
dog1.bark();
```

Спадкування — це спосіб створення нового класу, використовуючи дані вже існуючого класу без внесення змін до нього. Створюваний клас є дочірнім (або “похідним”) класом, а існуючий клас є батьківським (або “базовим”) класом.

```
#Super()
class Animal:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def show_info(self):
        print(f"Ім'я: {self.name}, Вік: {self.age} років.")

class Dog(Animal):
    def __init__(self, name, age, breed):
        # Виклик конструктора батьківського класу за допомогою super()
        super().__init__(name, age)

        # Додаткова ініціалізація для класу Dog
        self.breed = breed

    def show_info(self):
        # Виклик методу show_info() з батьківського класу
        super().show_info()
        print(f"Порода: {self.breed}.")

# Створюємо об'єкт Dog
my_dog = Dog("Рекс", 5, "Німецька вівчарка")

# Викликаємо метод, який використовує super()
my_dog.show_info()
```

Принцип роботи `super()` в Python полягає в **зверненні до батьківського класу**. Це дозволяє дочірньому класу викликати методи, які були перевизначені або існують лише у його батьківському класі. Найчастіше `super()` використовується в конструкторі `__init__`, щоб ініціалізувати атрибути, успадковані від батьківського класу.

`super()` — це вбудована функція Python, яка повертає тимчасовий проксі-об'єкт, що дозволяє делегувати виклики методів до батьківського або до іншого спорідненого класу в ієрархії. Це критично важливий інструмент для коректного успадкування та поліморфізму, особливо в ієрархіях зі складним множинним успадкуванням.

МНОЖИННЕ УСПАДКУВАННЯ В PYTHON

```
#Множинне успадкування
class Swimmer:
    def swim(self):
        print("Я вмію плавати.")

class Flyer:
    def fly(self):
        print("Я вмію літати.")

class Duck(Swimmer, Flyer):
    def quack(self):
        print("Кря-кря!")

# Створюємо об'єкт класу Duck
duck = Duck()

# Качка успадковує методи від обох батьківських класів
duck.swim()
duck.fly()
duck.quack()
```

Множинне успадкування (multiple inheritance) — це можливість дочірнього класу успадковувати атрибути та методи від **кількох батьківських класів** одночасно. Ця функція дозволяє створювати складні ієрархії, де один клас може поєднувати поведінку та властивості різних батьківських класів, що робить код більш гнучким.

БАГАТОРІВНЕВЕ СПАДКУВАННЯ В PYTHON

Багаторівневе успадкування (multilevel inheritance) — це тип успадкування, при якому один клас є батьківським для іншого, який, у свою чергу, є батьківським для третього класу. Це створює ієрархію, схожу на ланцюжок або сімейне дерево.

Організація коду: Багаторівневе успадкування дозволяє організувати код у логічну, ієрархічну структуру, що може бути корисною для моделювання складних відносин.

Повторне використання: Ви можете додавати нові класи до ланцюжка, розширюючи функціонал, не змінюючи існуючі базові класи.

Збереження логіки: Завдяки `super()`, кожен рівень ланцюжка успадковує та зберігає логіку попередніх, що допомагає уникнути дублювання коду.

```
#Багаторівневе успадкування
class Grandparent:
    def __init__(self, grandparent_name):
        self.grandparent_name = grandparent_name
        print(f"Grandparent {self.grandparent_name} created.")

class Parent(Grandparent):
    def __init__(self, grandparent_name, parent_name):
        super().__init__(grandparent_name)
        self.parent_name = parent_name
        print(f"Parent {self.parent_name} created.")

class Child(Parent):
    def __init__(self, grandparent_name, parent_name, child_name):
        super().__init__(grandparent_name, parent_name)
        self.child_name = child_name
        print(f"Child {self.child_name} created.")

    def show_lineage(self):
        print(f"Я {self.child_name}, син {self.parent_name}, онук {self.grandparent_name}.")

# Створюємо об'єкт Child
my_family = Child("Bob", "Дмитро", "Сергій")

# Викликаємо метод, який використовує атрибути всіх трьох рівнів
my_family.show_lineage()
```

ПОРЯДОК ВИКЛИКУ МЕТОДІВ (MRO) В PYTHON

```
class A:
    def method(self):
        print("Метод класу A")

class B(A):
    def method(self):
        print("Метод класу B")
        super().method()

class C(A):
    def method(self):
        print("Метод класу C")
        super().method()

class D(B, C):
    def method(self):
        print("Метод класу D")
        super().method()

# Створюємо об'єкт класу D
d = D()
d.method()

# Перевіряємо порядок MRO
print("\nПорядок вирішення методів (MRO) для D:")
print(D.mro())
```

Метод класу D
Метод класу B
Метод класу C
Метод класу A

Порядок вирішення методів (MRO) — це алгоритм, який Python використовує для визначення того, в якій послідовності слід шукати методи в ієрархії успадкування. Це особливо важливо для класів, що використовують **множинне успадкування**, оскільки MRO вирішує, який метод буде викликаний, якщо він існує в кількох батьківських класах.

```
# Інкапсуляція
class Car:
    def __init__(self, make, model):
        self.make = make      # Публічні атрибути
        self.model = model
        self.__engine_running = False

    def start_engine(self):
        # Публічний метод, який керує прихованим атрибутом
        self.__engine_running = True
        print(f"Двигун {self.make} {self.model} запущено.")

    def stop_engine(self):
        # Ще один публічний метод
        self.__engine_running = False
        print(f"Двигун {self.make} {self.model} вимкнено.")

# Створення об'єкта
my_car = Car("Toyota", "Camry")

# Використання публічного інтерфейсу
my_car.start_engine()

# Спроба прямого доступу до прихованого атрибута призведе до
# print(my_car.__engine_running) # Це спричинить помилку At

# Замість цього ми повинні використовувати методи
my_car.stop_engine()
```

Інкапсуляція — це принцип об'єктно-орієнтованого програмування, який означає об'єднання **даних** (атрибутів) і **методів**, що з ними працюють, у єдину сутність — клас, а також **приховування внутрішньої реалізації** від зовнішнього доступу.

Інакше кажучи, інкапсуляція забезпечує:

- 1. Приховування деталей реалізації** (користувач працює з інтерфейсом, а не з внутрішнім кодом).
- 2. Контроль доступу** до атрибутів і методів (через модифікатори доступу).
- 3. Захист даних** від некоректних змін.

```

class BankAccount:
    def __init__(self, owner, balance=0):
        self.owner = owner          # public
        self._balance = balance     # protected
        self.__pin = "1234"         # private

    def deposit(self, amount):
        self._balance += amount

    def get_balance(self, pin):
        if self.__verify_pin(pin):
            return self._balance
        else:
            return "Невірний PIN"

    def __verify_pin(self, pin):
        return pin == self.__pin

acc = BankAccount("Bob", 1000)

print(acc.owner)          # доступний (public)
print(acc._balance)       # можна, але не рекоменду
# print(acc.__pin)        # AttributeError (private)

# виклик через метод (інтерфейс)
print(acc.get_balance("1231")) # 1000
# print(acc.__verify_pin("1234"))

```

public — атрибути й методи без підкреслення (доступні всюди).

_protected — одинарне підкреслення (_var), сигналізує, що атрибут "для внутрішнього використання".

__private — подвійне підкреслення (__var), виконується name mangling (зміна імені всередині класу), що ускладнює прямий доступ.

```
class Animal:
    def make_sound(self):
        # Базовий метод, який може бути перевизначений
        pass

class Dog(Animal):
    def make_sound(self):
        # Перевизначення методу для собаки
        return "Гав!"

class Cat(Animal):
    def make_sound(self):
        # Перевизначення методу для кота
        return "Мяу!"

# Створюємо об'єкти
dog = Dog()
cat = Cat()

# Функція, яка приймає будь-який об'єкт типу Animal
def animal_sound(animal):
    return animal.make_sound()

# Використовуємо один і той же виклик, але отримуємо різні результати
print(f"Собака каже: {animal_sound(dog)}")
print(f"Кіт каже: {animal_sound(cat)}")
```

Поліморфізм — це принцип об'єктно-орієнтованого програмування, що дозволяє об'єктам різних класів **реагувати по-різному на один і той же виклик**. Це дає можливість використовувати спільний інтерфейс для роботи з різними типами даних, що значно спрощує та уніфікує код.

Переваги:

- Зниження зв'язаності (Coupling): Код стає менш залежним від конкретних реалізацій, що дозволяє легко додавати нові класи без зміни існуючого коду.
- Спрощення коду: Замість використання складних конструкцій if-elif-else для кожного типу об'єкта, ви можете просто викликати один і той же метод.
- Гнучкість та розширюваність: Ви можете писати загальний код, який працюватиме з будь-якими майбутніми класами, що наслідують ваш базовий клас.

```
from abc import ABC, abstractmethod

# Абстрактний клас
class Vehicle(ABC):
    @abstractmethod
    def start_engine(self):
        # Це абстрактний метод, який повинен бути реалізований
        # у всіх дочірніх класах.
        pass

    def drive(self):
        print("Починаємо рух.")

# Конкретний клас, що реалізує абстракцію Vehicle
class Car(Vehicle):
    def start_engine(self):
        # Тут ми реалізуємо, як саме запускається двигун автомобіля.
        print("Запуск двигуна за допомогою запалювання.")

# Ще один конкретний клас
class ElectricCar(Vehicle):
    def start_engine(self):
        # У цьому класі запуск двигуна реалізований інакше.
        print("Запуск електродвигуна.")

# Створення об'єктів
my_car = Car()
my_electric_car = ElectricCar()

# Ми взаємодіємо з об'єктами через їхні публічні методи,
# не знаючи про внутрішню реалізацію.
my_car.start_engine()
my_car.drive()
print("-" * 20)
my_electric_car.start_engine()
my_electric_car.drive()
```

Абстракція — це один із чотирьох основних принципів об'єктно-орієнтованого програмування, що дозволяє нам **спростити складні системи**, приховуючи їхні внутрішні деталі та показуючи лише найважливіші аспекти.

В програмуванні абстракція дозволяє нам створювати **класи**, які представляють собою спрощені моделі реального світу. Ми визначаємо, які дані (атрибути) та поведінку (методи) є важливими для зовнішнього світу, і приховуємо всі інші деталі.

ПЕРЕВАНТАЖЕННЯ ОПЕРАТОРІВ В PYTHON

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    # Метод для перевантаження оператора +
    def __add__(self, other):
        # Ми визначаємо, як саме додаються два вектори
        new_x = self.x + other.x
        new_y = self.y + other.y
        return Vector(new_x, new_y)

    def __str__(self):
        # Метод для зручного виведення
        return f"Vector({self.x}, {self.y})"

# Створюємо два об'єкти Vector
v1 = Vector(2, 3)
v2 = Vector(5, 7)

# Використовуємо оператор +, як для звичайних чисел
v3 = v1 + v2

print(f"Результат додавання: {v3}")
```

Перевантаження операторів (operator overloading) — це можливість **перевизначати поведінку стандартних операторів** (наприклад, +, -, *, ==) для ваших власних класів. Це дозволяє використовувати звичний синтаксис для роботи з об'єктами, роблячи код більш інтуїтивно зрозумілим і читабельним.

Таблиця основних операторів Python

Оператор	Приклад	Magic-метод
Додавання	<code>p1 + p2</code>	<code>p1.__add__(p2)</code>
Віднімання	<code>p1 - p2</code>	<code>p1.__sub__(p2)</code>
Множення	<code>p1 * p2</code>	<code>p1.__mul__(p2)</code>
Піднесення до степеня	<code>p1 ** p2</code>	<code>p1.__pow__(p2)</code>
Ділення	<code>p1 / p2</code>	<code>p1.__truediv__(p2)</code>
Цілочисельне ділення	<code>p1 // p2</code>	<code>p1.__floordiv__(p2)</code>
Залишок від ділення	<code>p1 % p2</code>	<code>p1.__mod__(p2)</code>
Зсув вліво	<code>p1 << p2</code>	<code>p1.__lshift__(p2)</code>
Зсув вправо	<code>p1 >> p2</code>	<code>p1.__rshift__(p2)</code>
Побітове І	<code>p1 & p2</code>	<code>p1.__and__(p2)</code>
Побітове АБО	<code>p1 p2</code>	<code>p1.__or__(p2)</code>
Побітове виключне АБО	<code>p1 ^ p2</code>	<code>p1.__xor__(p2)</code>
Побітове НЕ	<code>~p1</code>	<code>p1.__invert__()</code>

ЗАМИКАННЯ В PYTHON

```
def create_counter(start_value=0):
    count = start_value # Змінна із зовнішньої області видимості

    def increment():
        nonlocal count # Використовуємо nonlocal, щоб змінити 'count'
        count += 1
        return count

    return increment # Повертаємо внутрішню функцію, а не її результат

# Створюємо два окремі лічильники
counter1 = create_counter(10)
counter2 = create_counter(100)

# Кожен лічильник має свій власний "заморожений" стан
print(f"Перший лічильник: {counter1()}") # 11
print(f"Перший лічильник: {counter1()}") # 12
print(f"Другий лічильник: {counter2()}") # 101
print(f"Другий лічильник: {counter2()}") # 102
```

Замикання (closure) — це функція, яка **запам'ятовує та зберігає доступ** до змінних із області видимості, в якій вона була створена, навіть після того, як ця зовнішня область видимості завершила своє виконання.

Замикання виникає, коли ви виконуєте дві умови:

- 1.Вкладеність функцій:** Ви визначаєте функцію всередині іншої функції.
- 2.Доступ до зовнішньої змінної:** Внутрішня функція посилається на змінну, яка належить зовнішній (батьківській) функції.

Коли зовнішня функція завершує своє виконання та повертає внутрішню функцію, внутрішня функція "замикає" навколо себе змінну з батьківської області видимості.

АНОНІМНІ ФУНКЦІЇ В PYTHON

- Анонімна функція функція , яка оголошується без імені.
- Зазвичай оголошення функції супроводжується ключовим словом `def`. Анонімні функції оголошуються за допомогою `lambda`.
- Саме тому анонімні функції ще називають лямбда-функціями.

`lambda` аргументи: вираз

```
double = lambda x: x * 2

print(double(5))
# Звичайни запис
def doubledef(x):
    return x * 2
print(doubledef(5))
```

Лямбда-функції використовуються у випадках, коли потрібні безіменні функції на невеликий проміжок часу.

Зазвичай у Python вони використовуються як аргумент для функцій вищого порядку (функцій, які приймають як аргументи інші функції). Часто їх використовують у комбінації з наступними вбудованими функціями: `filter()`, `map()` і т.д.

```
my_list = [1, 5, 4, 6, 8, 11, 3, 12]

new_list = list(filter(lambda x: (x%2 == 0) , my_list))

print(new_list)
```