Декоратори у Python: Теорія та синтаксис

1.1. Функції як об'єкти першого класу (First-Class Objects)

У Python функції є "об'єктами першого класу". Це означає, що вони можуть:

- Бути присвоєні змінним.
- Передаватися як аргументи іншим функціям.
- Повертатися з інших функцій.
- Зберігатися у структурах даних (списках, словниках).

Це фундаментальна концепція для розуміння декораторів.

Приклад:

```
def greet(name):
    return f"Привіт, {name}!"

# Присвоєння функції змінній
my_function = greet
print(my_function("Олег")) # Виведе: Привіт, Олег!

# Передача функції як аргументу
def call_function(func, arg):
    return func(arg)

print(call_function(greet, "Марія")) # Виведе: Привіт, Марія!
```

1.2. Вкладені функції (Nested Functions) та Замикання (Closures)

Вкладені функції — це функції, визначені всередині інших функцій.

Замикання — це вкладена функція, яка "запам'ятовує" та має доступ до змінних зі своєї зовнішньої (охоплюючої) області видимості, навіть після того, як зовнішня функція завершила своє виконання.

Приклад:

```python
def outer_function(msg):
    # msg - змінна із зовнішньої області видимості
    def inner_function():
        print(msg)
    return inner_function

# outer_function повертає inner_function (замикання)
hello_func = outer_function("Привіт!")
bye_func = outer_function("Бувай!")

hello_func() # Виведе: Привіт!
bye_func()   # Виведе: Бувай!
```

1.3. Що таке Декоратор?

Декоратор — це функція, яка приймає іншу функцію як аргумент, розширює її функціонал (загортає її) і повертає нову функцію (або модифіковану версію оригінальної функції).

По суті, декоратори дозволяють "обернути" функцію, щоб додати до неї певну поведінку до або після її виконання, не змінюючи її вихідний код.

Основна ідея:

```python
@декоратор
def функція():
    pass

# Це еквівалентно:
def функція():
    pass
функція = декоратор(функція)
```

1.4. Синтаксис Декораторів (@)

Python надає синтаксичний цукор (@) для зручного використання декораторів.

Без синтаксичного цукру:

```
def my_decorator(func):
    def wrapper():
        print("Щось перед виконанням функції.")
        func()
        print("Щось після виконання функції.")
    return wrapper

def say_hello():
    print("Привіт, світ!")

# Декорування функції вручну
say_hello = my_decorator(say_hello)
say_hello()
```

З синтаксичним цукром (@):

```
def my_decorator(func):
    def wrapper():
        print("Щось перед виконанням функції.")
        func()
        print("Щось після виконання функції.")
    return wrapper

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

say_hello()
```

Обидва приклади дають однаковий результат, але синтаксис @ є чистішим та

більш читабельним.

1.5. Навіщо використовувати Декоратори?

- **Принцип DRY (Don't Repeat Yourself):** Уникайте дублювання коду. Якщо вам потрібно додати одну й ту саму допоміжну функціональність до багатьох функцій, декоратор ідеально підходить.
- **Розділення відповідальностей (Separation of Concerns):** Відокремлюйте основну бізнес-логіку функції від допоміжної (логінг, аутентифікація, вимірювання часу).
- **Модифікація поведінки без зміни коду:** Додавайте або змінюйте поведінку функції, не торкаючись її внутрішньої реалізації. Це робить код більш гнучким та легким для підтримки.
- **Читабельність коду:** Використання @ робить код більш декларативним, чітко показуючи, яка додаткова функціональність застосовується до функції.

1.6. Поширені випадки використання Декораторів (з фокусом на веб-додатки)

1. **Логування (Logging):** Запис інформації про виклики функцій, їхні аргументи, результати та час виконання.
   - *Веб-приклад:* Логування HTTP-запитів до певних ендпоінтів, запис інформації про користувача, що здійснив запит.
2. **Аутентифікація та Авторизація (Authentication & Authorization):** Перевірка, чи має користувач право доступу до певної функції або ресурсу.
   - *Веб-приклад:* @login_required для доступу лише зареєстрованим користувачам; @admin_only для доступу лише адміністраторам.
3. **Вимірювання часу виконання (Timing/Performance):** Вимірювання, скільки часу займає виконання функції.
   - *Веб-приклад:* Моніторинг продуктивності API-ендпоінтів.
4. **Кешування (Caching):** Зберігання результатів дорогих обчислень, щоб уникнути їх повторного виконання.
   - *Веб-приклад:* Кешування результатів запитів до бази даних або зовнішніх API.
5. **Маршрутизація (Routing) у веб-фреймворках:** Визначення, який URL-шлях відповідає певній функції обробника (view function).
   - *Веб-приклад:* @app.route('/users') у Flask, @app.get("/items/") у FastAPI.
6. **Обробка помилок (Error Handling):** Обгортання функцій для централізованого перехоплення та обробки винятків.
   - *Веб-приклад:* Автоматичне повернення 500 Internal Server Error або іншої

відповіді у випадку неочікуваної помилки в обробнику запиту.

1.7. Декоратори з аргументами

Щоб передати аргументи декоратору, вам потрібна додаткова функція-обгортка. Структура стає такою:

```
def decorator_with_args(arg1, arg2): # Це функція, яка приймає аргументи декоратора
    def actual_decorator(func):      # Це власне декоратор, який приймає функцію
        def wrapper(*args, **kwargs): # Це обгортка, яка виконує логіку
            print(f"Декоратор отримав аргументи: {arg1}, {arg2}")
            result = func(*args, **kwargs)
            return result
        return wrapper
    return actual_decorator

@decorator_with_args("значення1", "значення2")
def my_function():
    print("Функція виконана.")

my_function()
```

Тут decorator_with_args — це фабрика декораторів, яка повертає сам декоратор actual_decorator.

1.8. Важливість functools.wraps

Коли ви використовуєте декоратор, він замінює оригінальну функцію на функцію wrapper. Це призводить до втрати метаданих оригінальної функції, таких як її ім'я (__name__), docstring (__doc__), модуль тощо.

Це може ускладнити налагодження, тестування та інтроспекцію коду.

Модуль functools надає декоратор @functools.wraps(func), який копіює відповідні атрибути з оригінальної функції func до функції wrapper.

Приклад без wraps:

```
def simple_decorator(func):
    def wrapper():
        """Це docstring обгортки."""
        return func()
    return wrapper

@simple_decorator
def original_function():
    """Це docstring оригінальної функції."""
    pass

print(original_function.__name__) # Виведе: wrapper
print(original_function.__doc__)  # Виведе: Це docstring обгортки.
```

Приклад з wraps:

```
import functools

def simple_decorator_with_wraps(func):
    @functools.wraps(func) # Використовуємо @functools.wraps
    def wrapper():
        """Це docstring обгортки."""
        return func()
    return wrapper

@simple_decorator_with_wraps
def original_function_with_wraps():
    """Це docstring оригінальної функції."""
    pass

print(original_function_with_wraps.__name__) # Виведе:
original_function_with_wraps
print(original_function_with_wraps.__doc__)  # Виведе: Це docstring
оригінальної функції.
```

Завжди використовуйте @functools.wraps(func) при створенні декораторів!

Частина 2: Практичні Приклади Декораторів

```
import functools
import time
import datetime
```

Приклад 1: Базовий декоратор

 # Декоратор, який додає повідомлення до та після виконання функції.

```
def simple_logger_decorator(func):
    @functools.wraps(func) # Важливо для збереження метаданих
оригінальної функції
    def wrapper(*args, **kwargs):
        print(f"[{datetime.datetime.now().strftime('%H:%M:%S')}]
Виклик функції: {func.__name__}")
        result = func(*args, **kwargs)
        print(f"[{datetime.datetime.now().strftime('%H:%M:%S')}]
Функція {func.__name__} завершила виконання.")
        return result
    return wrapper

@simple_logger_decorator
def greet(name):
    """Привітати користувача за ім'ям."""
    print(f"Привіт, {name}!")
    return f"Привітання для {name}"

@simple_logger_decorator
def add_numbers(a, b):
    """Додає два числа."""
    print(f"Додаю {a} і {b}")
```

```
    return a + b

print("\n--- Приклад 1: Базовий декоратор ---")
greet("Олег")
result_add = add_numbers(5, 3)
print(f"Результат додавання: {result_add}")
print(f"Ім'я функції greet після декорування: {greet.__name__}")
print(f"Docstring функції greet після декорування: {greet.__doc__}")
print("-" * 30)
```

Приклад 2: Декоратор з аргументами (для імітації ролей/прав доступу)

Декоратор, який перевіряє, чи має користувач потрібну роль для доступу до функції.

```
# Імітація поточного користувача та його ролей
current_user_roles = ["user", "admin"]

def role_required(required_role):
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            if required_role in current_user_roles:
                print(f"Доступ дозволено для {func.__name__}
(потрібна роль: {required_role}).")
                return func(*args, **kwargs)
            else:
                print(f"Доступ заборонено для {func.__name__}.
Потрібна роль '{required_role}'.")
                return None # Або викликати виняток, повернути
помилку HTTP тощо.
        return wrapper
```

```
    return decorator

@role_required("admin")
def delete_user(user_id):
    """Видаляє користувача з системи."""
    print(f"Користувач {user_id} видалено.")
    return True

@role_required("user")
def view_profile(user_id):
    """Переглядає профіль користувача."""
    print(f"Перегляд профілю користувача {user_id}.")
    return {"user_id": user_id, "data": "some_data"}

print("\n--- Приклад 2: Декоратор з аргументами (ролі) ---")
delete_user(123) # Доступ заборонено, бо current_user_roles не
містить "admin"
current_user_roles.append("admin") # Додаємо роль адміністратора
print("\n(Після додавання ролі 'admin' до поточного користувача)")
delete_user(123) # Доступ дозволено
view_profile(456) # Доступ дозволено
print("-" * 30)
```

Приклад 3: Декоратор для вимірювання часу виконання (Timing Decorator)

Корисно для моніторингу продуктивності веб-ендпоінтів або дорогих операцій.

```
def timer_decorator(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        start_time = time.perf_counter() # Використовуємо
```

```
perf_counter для точності
        result = func(*args, **kwargs)
        end_time = time.perf_counter()
        execution_time = end_time - start_time
        print(f"Функція '{func.__name__}' виконана за
{execution_time:.4f} секунд.")
        return result
    return wrapper

@timer_decorator
def simulate_heavy_computation(duration_seconds):
    """Імітує важкі обчислення."""
    print(f"Починаю важкі обчислення на {duration_seconds}
секунд...")
    time.sleep(duration_seconds)
    print("Обчислення завершено.")
    return "Обчислення успішні"

print("\n--- Приклад 3: Декоратор для вимірювання часу ---")
simulate_heavy_computation(0.5)
simulate_heavy_computation(0.1)
print("-" * 30)
```

Приклад 4: Декоратор для обробки команд Telegram бота

Демонструє, як декоратори використовуються для реєстрації обробників команд у ботах.

```
# Імітація сховища для обробників команд
_BOT_COMMAND_HANDLERS = {} # Словник: команда -> функція-обробник

class TelegramBot:
```

```
    def command(self, command_name):
        """
        Декоратор для реєстрації функції як обробника команди бота.
        Приймає назву команди (наприклад, "start", "help").
        """
        def decorator(func):
            @functools.wraps(func)
            def wrapper(message):
                # Тут може бути логіка парсингу повідомлення,
                # передача аргументів команди тощо.
                print(f"[{datetime.datetime.now().strftime('%H:%M:%S')}]
Обробка команди '/{command_name}'...")
                return func(message) # Передаємо повідомлення обробнику

            # Реєструємо обгорнуту функцію як обробник для цієї команди
            _BOT_COMMAND_HANDLERS[command_name] = wrapper
            print(f"Команда '/{command_name}' зареєстрована для функції
'{func.__name__}'.")
            return wrapper # Повертаємо обгорнуту функцію
        return decorator

    def process_message(self, message_text):
        """
        Імітує отримання повідомлення та його маршрутизацію до
відповідного обробника.
        """
        if message_text.startswith('/'):
            parts = message_text.split(' ', 1) # Розділяємо команду та
можливі аргументи
            command = parts[0][1:] # Видаляємо '/'

            if command in _BOT_COMMAND_HANDLERS:
                print(f"Диспетчеризація повідомлення: '{message_text}' до
обробника команди '{command}'.")
                return _BOT_COMMAND_HANDLERS[command](message_text)
            else:
                print(f"Невідома команда: '{command}'.")
                return "Невідома команда."
```

```
        else:
            print(f"Отримано звичайне повідомлення: '{message_text}'.")
            return "Я отримав ваше повідомлення."

# Створюємо екземпляр нашого "Telegram бота"
bot = TelegramBot()

@bot.command("start")
def handle_start(message):
    """Обробник команди /start."""
    return "Привіт! Я ваш тестовий бот. Використовуйте /help для списку
команд."

@bot.command("help")
def handle_help(message):
    """Обробник команди /help."""
    return "Доступні команди: /start, /help, /echo <текст>."

@bot.command("echo")
def handle_echo(message):
    """Обробник команди /echo, що повторює текст користувача."""
    parts = message.split(' ', 1)
    if len(parts) > 1:
        return f"Ви сказали: {parts[1]}"
    else:
        return "Будь ласка, надайте текст для команди /echo."

print("\n--- Приклад 4: Декоратор для обробки команд Telegram бота
---")
print("Зареєстровані команди бота:")
for cmd, handler in _BOT_COMMAND_HANDLERS.items():
    print(f"  - /{cmd}: {handler.__name__}")

# Імітація отримання повідомлень
print("\nІмітація отримання повідомлень:")
print(f"Бот відповів: {bot.process_message('/start')}")
print(f"Бот відповів: {bot.process_message('/help')}")
print(f"Бот відповів: {bot.process_message('/echo Привіт, бот!')}")
```

```
print(f"Бот відповів: {bot.process_message('Звичайний текст')}")
print(f"Бот відповів: {bot.process_message('/unknown_command')}")
print("-" * 30)
```

Приклад 5: Декоратор для обробки помилок (Error Handling Decorator)

Декоратор, який перехоплює певні винятки та повертає стандартне повідомлення про помилку.

```
import logging

# Налаштування базового логування
logging.basicConfig(level=logging.INFO, format='%(asctime)s -
%(levelname)s - %(message)s')

def handle_exceptions(exception_type, error_message="Виникла
помилка."):
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            try:
                return func(*args, **kwargs)
            except exception_type as e:
                logging.error(f"Помилка у функції '{func.__name__}':
{e}")
                return error_message # Повертаємо стандартизоване
повідомлення
        return wrapper
    return decorator

@handle_exceptions(ValueError, "Некоректне значення вхідних даних.")
def process_user_input(value):
```

```
    """Обробляє вхідні дані користувача, може викликати
ValueError."""
    if not isinstance(value, int):
        raise ValueError("Вхідне значення має бути цілим числом.")
    if value < 0:
        raise ValueError("Значення не може бути від'ємним.")
    print(f"Обробка значення: {value}")
    return value * 2

@handle_exceptions(ZeroDivisionError, "Спроба ділення на нуль.")
def divide(numerator, denominator):
    """Виконує ділення."""
    return numerator / denominator

print("\n--- Приклад 5: Декоратор для обробки помилок ---")
print(f"Результат обробки '10': {process_user_input(10)}")
print(f"Результат обробки 'abc': {process_user_input('abc')}") #
Викличе ValueError
print(f"Результат обробки '-5': {process_user_input(-5)}")   #
Викличе ValueError
print(f"Результат ділення 10/2: {divide(10, 2)}")
print(f"Результат ділення 10/0: {divide(10, 0)}") # Викличе
ZeroDivisionError
print("-" * 30)
```