IT CAREER HUB

Урок 33.
Декораторы

IT CAREER HUB

Где применяются декораторы

Основные области применения декораторов:

1. **Логирование вызовов функций**
    - Запись информации о вызове функции, её аргументах и результатах.
    - Используется в **отладке и мониторинге работы программы**.
2. **Измерение времени выполнения**
    - Декоратор может засекать время работы функции и выводить его.
    - Полезно при **оптимизации кода**.
3. **Ограничение частоты вызовов (Throttling)**
    - Контроль количества вызовов функции в единицу времени.
    - Применяется, например, в **защите от спама** или **сетевых запросах**.
4. **Проверка и валидация входных данных**
    - Декораторы могут проверять, корректны ли переданные аргументы.
    - Полезно в **API, валидации форм и обработке ошибок**.
5. **Автоматическое повторение выполнения (Retry)**
    - Если функция неудачно выполняется (например, сбой сети), можно автоматически повторять попытки.
    - Используется в **сетевых запросах и файловых операциях**.
6. **Ограничение доступа**
    - Декораторы могут проверять, имеет ли пользователь нужные права для вызова функции.
    - Применяется в **системах аутентификации и авторизации**.
7. **Кеширование результатов**
    - Декораторы позволяют сохранять результат функции и не пересчитывать его при повторных вызовах.
    - Полезно в **сложных вычислениях и оптимизации запросов к базе данных**.
8. **Автоматическое изменение данных**
    - Можно автоматически приводить результаты функции к нужному формату, например, делать текст заглавными буквами или добавлять символы.
    - Используется в **обработке текста, форматировании вывода**.

IT CAREER HUB

Декораторы для функций с аргументами

Часто декорируемые функции принимают аргументы. Чтобы декоратор мог работать с такими функциями, его вложенная функция должна уметь **принимать и передавать аргументы**.

Пример: Запись информации о вызовах функций в файл

Python

```python
import logging

# Настраиваем логирование: записи будут сохраняться в файл "functions.log"
logging.basicConfig(
    filename="functions.log",
    level=logging.INFO,
    format="%(asctime)s - %(levelname)s - %(message)s",
    encoding="utf-8"
)

def log_decorator(func):
    def wrapper(*args, **kwargs):  # Принимаем все аргументы функции
        logging.info(f"Функция {func.__name__} вызвана с аргументами: {args},
{kwargs}")
        result = func(*args, **kwargs)  # Передаём аргументы в функцию
        logging.info(f"Функция {func.__name__} вернула: {result}")
        return result  # Возвращаем результат
    return wrapper

@log_decorator
def add(a, b):
    return a + b

@log_decorator
def say_hello():
    print("Привет!")

print(add(3, 5))  # Декорируемая функция принимает аргументами
say_hello()  # Декорируемая функция без аргументов
```

IT CAREER HUB

Возврат результата из вложенной функции

Если оригинальная функция **возвращает значение**, декоратор должен либо **вернуть его**, либо **заменить на другой результат**.
Если ничего не вернуть из вложенной функции, то результат окажется потерян, и вместо него будет возвращено None.

Пример с потерей результата

```python
def upper_decorator(func):
    def wrapper(*args, **kwargs):
        print("Выполняем функцию, но ничего не возвращаем")
        # Результат теряется
        func(*args, **kwargs).upper()
    return wrapper

@upper_decorator
def get_text():
    return "hello"

result = get_text()
print("Результат:", result)
```

Пример как правильно вернуть результат

```python
def upper_decorator(func):
    def wrapper(*args, **kwargs):
        print("Выполняем функцию и возвращаем результат")
        # Преобразуем результат в верхний регистр и возвращаем
        return func(*args, **kwargs).upper()
    return wrapper
```

```
@upper_decorator
def get_text():
    return "hello"

result = get_text()
print("Результат:", result)
```

IT CAREER HUB

Декоратор functools.wraps

При создании декораторов оригинальная функция **теряет своё имя и документацию**, так как заменяется вложенной функцией (`wrapper`). Декоратор `functools.wraps` помогает **сохранить метаданные** декорируемой функции.

Проблема без functools.wraps

```python
def simple_decorator(func):
    def wrapper(*args, **kwargs):
        """Вложенная функция wrapper"""
        print("\nДекорированная функция: ")
        print(f"Оригинальное имя функции: {func.__name__}")
        print(f"Оригинальная документация: {func.__doc__}")
        return func(*args, **kwargs)
    return wrapper

@simple_decorator
def example_function():
    """Это оригинальная функция."""
    print("Привет!")

print("Имя декорированной функции:", example_function.__name__)
print("Документация декорированной функции:", example_function.__doc__)
example_function()
```

Решение с functools.wraps

```python
import functools

def simple_decorator(func):
    @functools.wraps(func)  # Сохраняет имя и документацию оригинальной функции
    def wrapper(*args, **kwargs):
```

```
        print("\nДекорированная функция")
        return func(*args, **kwargs)
    return wrapper

@simple_decorator
def example_function():
    """Это оригинальная функция."""
    print("Привет!")

print("Имя декорированной функции:", example_function.__name__)
print("Документация декорированной функции:", example_function.__doc__)
example_function()
```

IT CAREER HUB

Задания для закрепления

1. **Что делает functools.wraps?**

a. Удаляет лишние аргументы
b. Автоматически вызывает декорируемую функцию
c. Сохраняет имя и документацию исходной функции

Посмотреть ответ

2. **Почему без functools.wraps теряется документация функции?**

Посмотреть ответ

3. **Для чего нужны *args и **kwargs в декораторах?**

a. Для совместимости с функциями без аргументов
b. Для совместимости с функциями с любыми аргументами
c. Для сокращения кода

Посмотреть ответ

Декораторы с аргументами

Обычные декораторы принимают только функцию, но иногда нужно передавать дополнительные аргументы, влияющие на их поведение. В таком случае используется **декоратор с аргументами**.

Чтобы декоратор принимал аргументы, создаётся функция **декоратор-фабрика**, которая возвращает сам декоратор.

Примеры декораторов с аргументами

Декоратор с настраиваемым сообщением

Этот декоратор добавляет перед выполнением функции произвольное сообщение, переданное в аргументе.

```python
def message_decorator(message):
    def decorator(func):
        def wrapper():
            print(message)  # Используем переданный аргумент
            return func()
        return wrapper
    return decorator

@message_decorator("Начинаем выполнение")
def analise_data():
    print("Данные проанализированы")

@message_decorator("Загрузка данных...")
def load_data():
    print("Данные загружены")

analise_data()
print()
load_data()
```

IT CAREER HUB

Автоматическое повторение функции при ошибке

Иногда функции могут завершаться с ошибкой, например, из-за нестабильного соединения. Если функция падает с ошибкой, декоратор попробует выполнить её заново несколько раз.

Python

```python
import time

def retry(attempts):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for i in range(attempts):
                try:
                    return func(*args, **kwargs)
                except Exception as e:
                    print(f"Попытка {i+1} не удалась: {e}")
                    time.sleep(5)  # Подождём перед новой попыткой
            print("Все попытки исчерпаны.")
        return wrapper
    return decorator

@retry(3)
def get_data(filename):
    """Читает данные из файла"""
    with open(filename, "r", encoding="utf-8") as file:
        return file.read()

# Пример использования
data = get_data("data.txt")
if data:
    print("Содержимое файла:")
    print(data)
```

Использование нескольких декораторов

В Python можно применять несколько декораторов к одной функции. Декораторы выполняются **снизу вверх**, то есть сначала обрабатывается декоратор, находящийся ближе к функции, затем следующий и так далее.

Особенности:

- Декораторы выполняются **снизу вверх**.
- Можно комбинировать **несколько независимых декораторов**.
- **Важно учитывать порядок выполнения**, так как один декоратор может изменить результат перед следующим.

Пример

Можно сочетать декораторы, которые выполняют разные функции.

```python
def border_decorator(func):
    def wrapper():
        print("*" * 100)  # Верхняя граница
        func()
        print("*" * 100)  # Нижняя граница
    return wrapper

def repeat_decorator(func):
    def wrapper():
        for _ in range(3):  # Повторяем вызов трижды
            func()
    return wrapper

@border_decorator  # Применяется вторым
@repeat_decorator  # Применяется первым
def print_line():
    print("-" * 100)

print_line()
```

IT CAREER HUB

Задания для закрепления

1. **Что делает следующий декоратор?**

```python
def custom_decorator(message):
    def decorator(func):
        def wrapper():
            try:
                return func()
            except Exception:
                print(message)
        return wrapper
    return decorator
```

Посмотреть ответ

2. **Какая конструкция позволит задать аргументы декоратору?**

a. `@decor("info")`
b. `@decor["info"]`
c. `@decor = "info"`
d. `@decor: "info"`

Посмотреть ответ

IT CAREER HUB

Ответы на задания

Задания на закрепление 1	Вернуться к заданиям
1. Функция `functools.wraps`	Ответ: c
2. Потеря данных без `functools.wraps`	Ответ: потому что `wrapper` заменяет функцию, и метаданные не переносятся
3. *args и **kwargs в декораторах	Ответ: b
Задания на закрепление 2	Вернуться к заданиям
1. Работа декоратора	Ответ: при возникновении ошибки выводит указанное сообщение
2. Конструкция, задающая аргументы декоратору	Ответ: a

IT CAREER HUB

Практическая работа

1. Рамка-обводка

Создайте декоратор `framed`, который **оборачивает результат** в рамку из символов = длиной 40.

Пример применения:

```python
@framed
def show_title():
    print("== Menu ==")
```

Пример вывода:

```
========================================
== Menu ==
========================================
```

Решение:

```python
def framed(func):
    def wrapper():
        print("=" * 40)
        func()
        print("=" * 40)
    return wrapper

@framed
def show_title():
    print("== Menu ==")

show_title()
```

2. Настраиваемая рамка-обводка

Доработайте декоратор `framed`, чтобы он принимал параметр `width`, определяющий ширину рамки, и параметр `symbol`, определяющий символ для рамки (по умолчанию "=").

Пример применения:

```python
@framed(30, "-")
def show_title():
    print("== Menu ==")
```

Пример вывода:

```python
------------------------------
== Menu ==
------------------------------
```

Решение:

```python
def framed(width, symbol="="):
    def decorator(func):
        def wrapper():
            print(symbol * width)
            func()
            print(symbol * width)
        return wrapper
    return decorator

@framed(30, "-")
def show_title():
    print("== Menu ==")

show_title()
```