Что такое декораторы в Python

Декораторы — это мощный инструмент Python, который позволяет изменять поведение функций, методов классов и других объектов без изменения их исходного кода. Они представляют собой элегантный способ добавления дополнительного функционала к существующим объектам путем их обертывания в другие функции.

Как работают декораторы Python

Декоратор в Python — это функция, которая принимает другую функцию в качестве аргумента и возвращает новую функцию. Эта новая функция обычно расширяет или изменяет поведение исходной функции, добавляя к ней дополнительный функционал.

Базовый синтаксис декоратора

```
def my_decorator(func):
    def wrapper():
        print("Дополнительный код перед вызовом функции")
        func()
        print("Дополнительный код после вызова функции")
    return wrapper
```

Применение декораторов с символом @

Для использования декоратора применяется специальный синтаксис с символом @. Декоратор размещается непосредственно перед определением функции:

```
@my_decorator
def say_hello():
    print("Привет, мир!")

say_hello()
```

Результат выполнения:

Дополнительный код перед вызовом функции

Привет, мир!

Дополнительный код после вызова функции

Декораторы с аргументами

Когда декорируемая функция принимает аргументы, необходимо использовать *args и **kwargs для передачи параметров:

```
def my_decorator(func):
    def wrapper(*args, **kwargs):
        print("Дополнительный код перед вызовом функции")
        result = func(*args, **kwargs)
        print("Дополнительный код после вызова функции")
        return result
    return wrapper

@my_decorator
def greet(name):
```

```
    print("Привет, {}!".format(name))

greet("Мир")
```

Декораторы для методов класса

Декораторы успешно применяются для декорирования методов класса, что особенно полезно для логирования, проверки прав доступа или кэширования:

```
def my_decorator(func):
    def wrapper(*args, **kwargs):
        print("Дополнительный код перед вызовом метода")
        result = func(*args, **kwargs)
        print("Дополнительный код после вызова метода")
        return result
    return wrapper

class MyClass:
    @my_decorator
    def greet(self, name):
        print("Привет, {}!".format(name))

obj = MyClass()
obj.greet("Мир")
```

Множественные декораторы

Python позволяет применять несколько декораторов к одной функции. Декораторы применяются снизу вверх:

```
def uppercase(func):
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        return result.upper()
    return wrapper

def bold(func):
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        return "<b>{}</b>".format(result)
    return wrapper

@bold
@uppercase
def greet(name):
    return "Привет, {}!".format(name)

print(greet("Мир"))  # <b>ПРИВЕТ, МИР!</b>
```

Встроенные декораторы Python

Python предоставляет несколько встроенных декораторов:

```
@property
```

Превращает метод в свойство класса:

```
class Circle:
    def __init__(self, radius):
        self._radius = radius

    @property
    def area(self):
        return 3.14159 * self._radius ** 2
```

Создает статический метод класса:

```
class MathUtils:
    @staticmethod
    def add(a, b):
        return a + b
```

Создает метод класса:

```
@classmethod
```

```
class Person:
    @classmethod
    def from_string(cls, name_str):
        return cls(name_str)
```

Декораторы с параметрами

Для создания декораторов с параметрами используется трехуровневая структура:

```
def repeat(times):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for _ in range(times):
                result = func(*args, **kwargs)
            return result
        return wrapper
    return decorator

@repeat(3)
def say_hello():
    print("Привет!")
```

Практические применения декораторов

Логирование

```python
import functools
import logging

def log_calls(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        logging.info(f"Вызов функции {func.__name__}")
        return func(*args, **kwargs)
    return wrapper
```

Измерение времени выполнения

```python
import time
import functools

def timing(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"{func.__name__} выполнилась за {end - start:.4f}
секунд")
        return result
    return wrapper
```

Кэширование результатов

```python
import functools


def cache(func):
    cache_dict = {}

    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        key = str(args) + str(kwargs)
        if key not in cache_dict:
            cache_dict[key] = func(*args, **kwargs)
        return cache_dict[key]

    return wrapper
```

Лучшие практики использования декораторов

1. Используйте functools.wraps: Это сохраняет метаданные оригинальной функции
2. Возвращайте результат: Всегда возвращайте результат выполнения оригинальной функции
3. Обрабатывайте исключения: Учитывайте возможные исключения в декораторе
4. Документируйте декораторы: Добавляйте docstring для объяснения назначения декоратора

Декораторы Python представляют собой мощный инструмент для создания чистого, читаемого кода. Они позволяют следовать принципу DRY (Don't Repeat Yourself) и создавать переиспользуемый функционал, который может быть легко применен к различным функциям и методам.