skypro

Шпаргалка. Декораторы

Замыкания

Замыкание (`closure`) — это функция, внутри которой объявлены другие функции, она может «запоминать» значения переменных, которые были ей доступны в момент создания, даже если внешняя функция уже завершила выполнение.

```
# Внешняя функция принимает параметр x
def make_multiplier(x):

    # Внутренняя функция использует значение x из внешней функции
    def multiplier(y):
        return x * y

    # Возвращаем внутреннюю функцию
    return multiplier

# Создаем замыкание
double = make_multiplier(2)
print(double(5))

>>> 10
```

Простые декораторы

Декоратор — это функция, которая принимает другую функцию в качестве аргумента и изменяет ее поведение без изменения самой функции.

Синтаксис написания декоратора

```
def my_decorator(func):
    def wrapper():
        print("Что-то происходит перед вызовом функции.")
        func()  # Вызов исходной функции
        print("Что-то происходит после вызова функции.")
    return wrapper
```

Синтаксис применения декоратора

```
@my_decorator
def say_hello():
    print("Hello!")

say_hello()
```

Пример простого декоратора

```
def printing(function):
    def inner(*args, **kwargs):
        result = function(*args, **kwargs)
        print('result =', result)
        return result
    return inner

@printing
def add_one(x):
    return x + 1

y = add_one(10)
>>> result = 11
print(y)
>>> 11
```

Применение нескольких декораторов

```
@logging
@printing
@cached
def foo():
    # ...
```

> Если к функции применяется несколько декораторов, то они выполняются в порядке «снизу вверх».

Пример с несколькими декораторами:

```
def printing(func):
    def wrapper(*args, **kwargs) :
          print(f'Function {func} started')
          result = func(*args, **kwargs)
          print(f'Function {func} finished')
          return result
    return wrapper


def timer(func):
  def wrapper(*args, **kwargs):
      time_1 = time()
      result = func(*args, **kwargs)
      time_2 = time()
      print(f'Time for work: {time_2 - time_1} )
      return result
  return wrapper


@printing
@timer
def example():
    for i in range (100000000):
        continue

example()
```

Декораторы с параметрами

Декоратор с параметрами — это расширенный тип декоратора, который позволяет передавать аргументы непосредственно в декоратор для настройки его поведения.

Синтаксис создания декоратора с параметрами

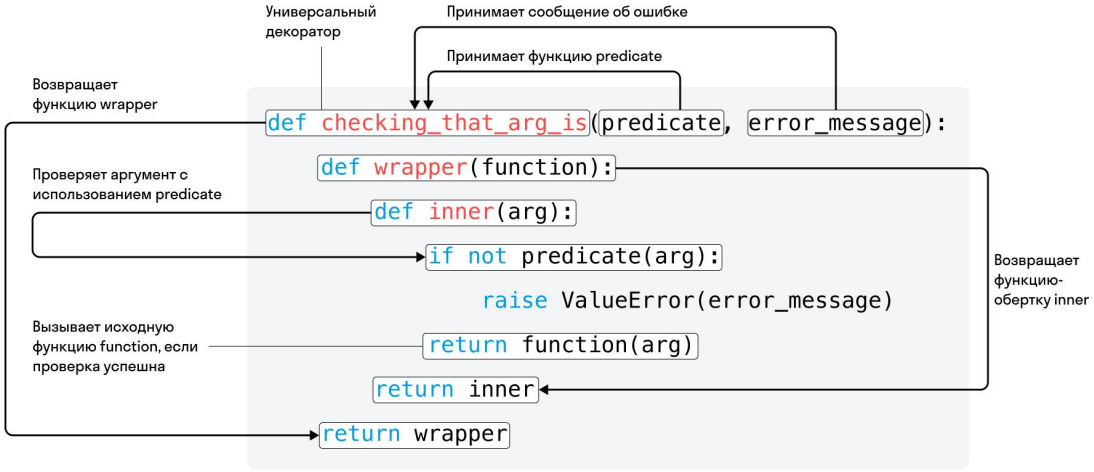
```
def decorator_with_args(arg1, arg2):
    def my_decorator(func):
        def wrapper(*args, **kwargs):
            print(f"Аргументы декоратора: {arg1}, {arg2}")
            result = func(*args, **kwargs)
            print("После выполнения функции")
            return result
        return wrapper
    return my_decorator
```

Синтаксис применения декоратора с параметрами

```
@decorator_with_args("значение1", "значение2")
def my_function(x, y):
    print(f"Выполнение функции с аргументами {x} и {y}")

my_function(10, 20)
```

Пример декоратора с параметрами




```
def checking_that_arg_is(predicate, error_message):
        def wrapper(function):
            def inner(arg):
                if not predicate(arg):
                    raise ValueError(error_message)
                return function(arg)
             return inner
        return wrapper


@checking_that_arg_is(lambda x: x > 0, "Value must be greater than 0!")
def example_function(value):
    return value * 2
```

Декоратор wraps

Декоратор `wraps` — это способ передать метаданные, в частности docstring, новому объекту, используемому под именем декорированной функции.

```
from functools import wraps

def wrapped(function):
    @wraps(function)
    def inner(arg):
        return function(arg)
    return inner

def foo(_):
    """Bar."""
    return 42

foo = wrapped(foo)
foo
>>> <function foo at 0x7f1057b15048>
help(foo)
>>> foo()
    Bar.
```

Тестирование декораторов с использованием pytest

Тестирование декораторов без параметров

Декоратор:

```
def double_decorator(func):
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        return result * 2
    return wrapper
```

Функция для декорирования:

```
def add_numbers(a, b):
    return a + b
```

Пример теста для декоратора без параметров:

```
import pytest

def test_double_decorator():
    @double_decorator
    def add_numbers(a, b):
        return a + b

    result = add_numbers(3, 5)
    assert result == 16  # (3 + 5) * 2
```

Тестирование декораторов с параметрами

Декоратор:

```
def retry_decorator(max_retries):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for _ in range(max_retries):
                try:
                    result = func(*args, **kwargs)
                    return result
                except Exception as e:
                    print(f"Retrying... ({e})")
            raise Exception(f"Max retries exceeded")
        return wrapper
    return decorator
```

У нас есть функция, которую мы декорируем и работу которой будем тестировать:

```
@retry_decorator(max_retries=3)
def example_function():
    # some potentially failing operation
    raise ValueError("Something went wrong!")
```

Пример теста для декоратора с параметрами:

```
import pytest

def test_retry_decorator():
    with pytest.raises(Exception, match="Max retries exceeded"):
        example_function()
```

Важно отметить, что при тестировании декораторов мы проверяем работу самого декоратора, а не декорируемой функции. То есть тест мы выстраиваем так, чтобы проверить работу декоратора. В качестве одного из случаев для проверки декоратора может выступать проверка, что результат работы функции корректно возвращается при декорировании.

Тестирование вывода в консоль

Фикстура `capsys` — позволяет перехватывать вывод в стандартный поток вывода `stdout` и стандартный поток ошибок `stderr` во время тестирования. Это полезно, когда вам нужно проверить, что ваша функция или приложение выводит ожидаемые данные в консоль.

Пример:

```
def hello_world():
    print("Hello, world!")
```

Тест с использованием `capsys`:

```
def test_hello_world(capsys):
    hello_world()
    captured = capsys.readouterr()
    assert captured.out == "Hello, world!\n"
```

Обратите внимание, что `print` в Python по умолчанию добавляет символ новой строки (`\n`), поэтому мы включаем его в наше ожидаемое значение.