УРОК 31. ДЕКОРАТОРЫ

ЧТО ТАКОЕ ДЕКОРАТОРЫ 2
ЗАДАНИЕ ДЛЯ ЗАКРЕПЛЕНИЯ 4
ВИДЫ ДЕКОРАТОРОВ 5
ЗАДАНИЕ ДЛЯ ЗАКРЕПЛЕНИЯ 7
СОЗДАНИЕ И ПРИМЕНЕНИЕ ДЕКОРАТОРОВ 8
ПРАКТИЧЕСКАЯ РАБОТА 10
ПОЛЕЗНЫЕ МАТЕРИАЛЫ 11

ЧТО ТАКОЕ ДЕКОРАТОРЫ

В языке Python функции являются объектами первого класса, что означает, что они могут быть присвоены переменным, переданы в качестве аргументов другим функциям и возвращены из функций. Это открывает возможности для создания мощных инструментов, таких как декораторы.
Декораторы позволяют изменять поведение существующих функций или классов без изменения их исходного кода.

Декораторы представляют собой функции, которые принимают другую функцию в качестве аргумента, добавляют к ней некоторую функциональность и возвращают измененную функцию.

Это позволяет применять одну и ту же декорирующую функцию к различным функциям, обеспечивая единообразие в изменении их поведения.

Функция-обертка представляет собой вспомогательную функцию, которая принимает оригинальную функцию в качестве аргумента, добавляет к ней дополнительную логику и возвращает измененную функцию.

Внутри функции-обертки можно выполнять различные действия, такие как запись логов, проверка аргументов, изменение возвращаемого значения и другие.

```python
def decorator(func):
  def wrapper(*args, **kwargs):
    # Дополнительная логика перед выполнением функции
    result = func(*args, **kwargs)
    # Дополнительная логика после выполнения функции
```


```
    return result
  return wrapper
```

Декораторы применяются к функциям или классам с помощью символа @, за которым следует имя декорирующей функции. При вызове декорируемой функции синтаксис декоратора автоматически применяется, что позволяет просто и удобно изменять поведение функции.

```python
@decorator
def my_function():
  # Тело функции
  pass
```

ЗАДАНИЕ ДЛЯ ЗАКРЕПЛЕНИЯ

Разберите построчно, что происходит в этом коде:

```python
def uppercase_decorator(function):
  def wrapper():
    func = function()
    make_uppercase = func.upper()
    return make_uppercase

  return wrapper
```

ВИДЫ ДЕКОРАТОРОВ

Декораторы могут быть применены к функциям, которые принимают аргументы. Для этого в функции-обертке необходимо использовать *args и **kwargs, чтобы передать все аргументы в декорируемую функцию.

```python
def decorator(func):
  def wrapper(*args, **kwargs):
    # Дополнительная логика перед выполнением функции
    result = func(*args, **kwargs)
    # Дополнительная логика после выполнения функции
    return result
  return wrapper

@decorator
def my_function(arg1, arg2):
  # Тело функции
  pass
```

Параметрические декораторы позволяют передавать аргументы в декорирующую функцию при ее применении. Для этого можно создать дополнительную функцию-обертку, которая принимает аргументы декоратора и возвращает саму функцию-обертку, принимающую декорируемую функцию.

```python
def param_decorator(arg1, arg2):
  def decorator(func):
    def wrapper(*args, **kwargs):
      # Дополнительная логика перед выполнением функции с использованием arg1 и
arg2
      result = func(*args, **kwargs)
```


```
      # Дополнительная логика после выполнения функции с использованием arg1 и
arg2
      return result
    return wrapper
  return decorator

@param_decorator(arg1, arg2)
def my_function():
  # Тело функции
  pass
```

Декораторы могут изменять атрибуты декорируемой функции, такие как имя или документацию. Однако это может привести к проблемам, когда требуется работать с атрибутами оригинальной функции. Модуль functools предоставляет декоратор wraps, который позволяет сохранить атрибуты оригинальной функции при применении декоратора.

Python

```
import functools

def decorator(func):
  @functools.wraps(func)
  def wrapper(*args, **kwargs):
    # Дополнительная логика перед выполнением функции
    result = func(*args, **kwargs)
    # Дополнительная логика после выполнения функции
    return result
  return wrapper
```

ЗАДАНИЕ ДЛЯ ЗАКРЕПЛЕНИЯ

Объясните, что происходит в этом коде:

```python
from functools import wraps
def log_decorator(func):
  @wraps(func)
  def wrapper():
    print(f'Вызов функции: {func.__name__}')
    func()
  return wrapper
@log_decorator
def hello_world():
  print("Hello, world!")
print(hello_world.__name__)
# Вывод: hello_world
```

СОЗДАНИЕ И ПРИМЕНЕНИЕ ДЕКОРАТОРОВ

Функцию можно декорировать с помощью нескольких декораторов, применив их последовательно сверху вниз. Это позволяет комбинировать различные аспекты функциональности и изменять поведение функции с использованием разных декораторов.

```python
@decorator1
@decorator2
def my_function():
  # Тело функции
  pass
```

Вы также можете создавать собственные декораторы, которые выполняют определенную логику в зависимости от ваших потребностей. Вы можете применять декораторы к функциям или классам, чтобы добавить или изменить их функциональность, улучшить читаемость и повторное использование кода.

```python
def my_decorator(func):
  def wrapper(*args, **kwargs):
    # Дополнительная логика перед выполнением функции
    result = func(*args, **kwargs)
    # Дополнительная логика после выполнения функции
    return result
  return wrapper

@my_decorator
def my_function():
  # Тело функции
  pass
```

Декораторы также могут быть применены к классам. В этом случае они могут изменять поведение методов класса или добавлять новые методы и атрибуты.

Декораторы классов широко используются для реализации паттернов проектирования и обеспечения гибкости в работе с классами.

```python
def class_decorator(cls):
  # Логика перед созданием класса
  return cls

@class_decorator
class MyClass:
  # Тело класса
  pass
```

ПРАКТИЧЕСКАЯ РАБОТА

Создать декоратор, который замеряет время работы функции.

Предусмотреть различное число итераций для сглаживания показателя.

ПОЛЕЗНЫЕ МАТЕРИАЛЫ

1. Декораторы
2. Декораторы в Python: понять и полюбить