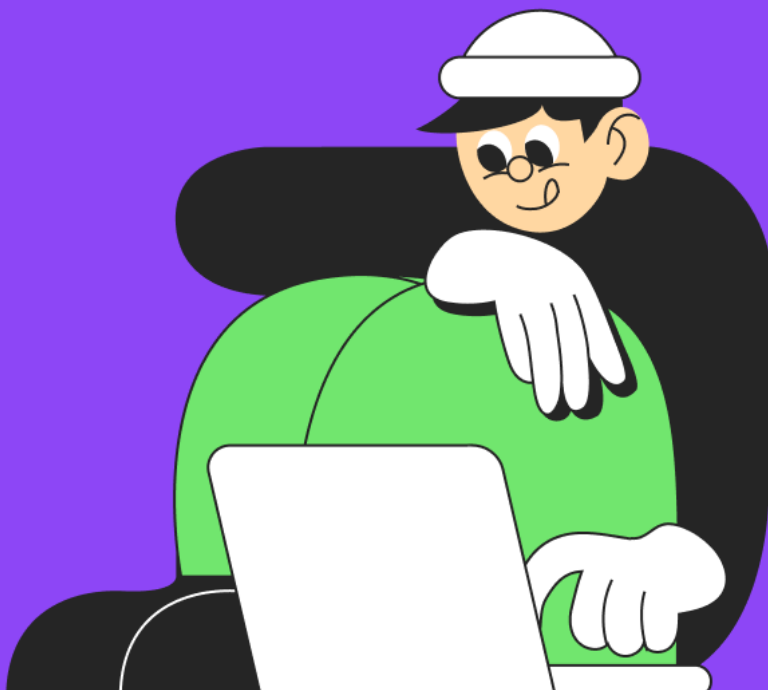
GeekBrains

Лекция 9. Декораторы

Погружение в Python



Оглавление

На этой лекции мы

1. Разберём замыкания в программировании
2. Изучим возможности Python по созданию декораторов
3. Узнаем как создавать декораторы с параметрами
4. Разберём работу некоторых декораторов из модуля functools

Краткая выжимка, о чём говорилось в предыдущей лекции

На прошлой лекции мы:

1. Разобрались в сериализации и десериализации данных
2. Изучили самый популярный формат сериализации — JSON
3. Узнали о чтении и записи таблиц в формате CSV
4. Разобрались с внутренним сериализатором Python — модулем pickle

Термины лекции

- **Замыкание (англ. closure) в программировании** — функция первого класса, в теле которой присутствуют ссылки на переменные, объявленные вне тела этой функции в окружающем коде и не являющиеся её параметрами.
- **Декоратор** — структурный шаблон проектирования, предназначенный для динамического подключения дополнительного поведения к объекту.

Подробный текст лекции

1. Что такое замыкания

Прежде чем погрузиться в декораторы поговорим о замыканиях в программировании вообще и в Python в частности. Плюс стоит вспомнить об областях видимости в Python.

Области видимости

```
def func(a):
    x = 42
    res = x + a
    return res


x = 73
print(func(64))
```

В этом примере глобальная переменная x равна 73, но при сложении внутри функции к значению a прибавляется 42 — значение локальной переменной x.

Функция как объект высшего порядка

Рассмотрим простой пример функции:

```
def add_str(a: str, b: str) -> str:
    return a + ' ' + b


print(add_str('Hello', 'world!'))
```

На вход передаём две строки и возвращаем новую из двух старый и пробела посередине. Но функцию можно переписать иначе. Вспомним, что в Python все функции высшего порядка. А это значит, что их можно передавать как объекты в другие функции:

```
from typing import Callable


def add_one_str(a: str) -> Callable[[str], str]:
    def add_two_str(b: str) -> str:
        return a + ' ' + b

    return add_two_str


print(add_one_str('Hello')('world!'))
```

Результат получили такой же, но код работает иначе.

- Функция add_one_str принимает на вход один параметр в качестве начала строки и возвращает локальную функцию add_two_str. Обратите внимание на отсутствие круглых скобок. Функцию передаём , а не вызываем.
- Функция add_two_str принимает вторую часть строки, соединяет её с первой и возвращает ответ.
- При вызове функций значения указывается в отдельных круглых скобках. Первое попадает в параметр a. Далее часть строки: add_one_str('Hello') возвращает функцию add_two_str и уже она вызывается и принимает аргумент во вторых скобках.

Благодаря передаче одной функции другой мы можем создавать замыкания.

Замыкаем функцию с параметрами

Внесём небольшие правки в пример кода:

```
from typing import Callable


def add_one_str(a: str) -> Callable[[str], str]:
    def add_two_str(b: str) -> str:
        return a + ' ' + b

    return add_two_str


hello = add_one_str('Hello')
bye = add_one_str('Good bye')

print(hello('world!'))
print(hello('friend!'))
print(bye('wonderful world!'))

print(f'{type(add_one_str) = }, {add_one_str.__name__ = }, {id(add_one_str) = }')
print(f'{type(hello) = }, {hello.__name__ = }, {id(hello) = }')
print(f'{type(bye) = }, {bye.__name__ = }, {id(bye) = }')
```

Во-первых мы не изменяли исходную функцию. Но мы создали две переменные hello и bye и поместили в них результат работы функции add_one_str с разными аргументами. Теперь мы можем вызывать новые функции и получать объединённые

строки передавая только окончание. Первая часть строки оказалась замкнута в локальной области видимости. И у каждой из двух новых функций область своя и начало строки своё.
А теперь посмотрите на результат работы трёх нижних строк кода. Все три переменные являются функциями, что очевидно. Но если функция add_one_str является самой собой, то функции hello и bye на самом деле являются двумя разными экземплярами функции add_two_str. id, т.е. адреса в оперативной памяти разные, а названия указывают на оригинал.

Замыкаем изменяемые и неизменяемые объекты

В очередной раз внесём правки в пример кода:

```
from typing import Callable


def add_one_str(a: str) -> Callable[[str], str]:
    names = []

    def add_two_str(b: str) -> str:
        names.append(b)
        return a + ', ' + ', '.join(names)

    return add_two_str


hello = add_one_str('Hello')
bye = add_one_str('Good bye')

print(hello('Alex'))
print(hello('Karina'))
print(bye('Alina'))
print(hello('John'))
print(bye('Neo'))
```

Во внешнюю функцию добавлен список names. При каждом вызове внутренней функции в список добавляется новое значение из параметра b и возвращается полное содержимое списка в виде строки. У каждой из двух функций hello и bye оказывается свой список names. Они не связаны между собой, но каждый хранит

список имён до конца работы программы. Обратите внимание, что list является изменяемым типом данных. Что будет, если мы перепишем код и заменим list на неизменяемый str?

```
from typing import Callable


def add_one_str(a: str) -> Callable[[str], str]:
    text = ''

    def add_two_str(b: str) -> str:
        nonlocal text
        text += ', ' + b
        return a + text

    return add_two_str


hello = add_one_str('Hello')
bye = add_one_str('Good bye')

print(hello('Alex'))
print(hello('Karina'))
print(bye('Alina'))
print(hello('John'))
print(bye('Neo'))
```

Изменения способа получения строки с join для списка на конкатенацию для строки не принципиально. Но стоит помнить, что сложение строк более дорогая операция по времени и по памяти, особенно если она находится внутри цикла.
Что более важно - неизменяемый тип данных у строки text. Без добавления строчки кода nonlocal text была бы получена ошибка UnboundLocalError: local variable 'text' referenced before assignment. Мы явно указали, что хотим обращаться к неизменяемому объекту для изменения его значения.
Как можно изменить неизменяемое? Мы создаём новый объект и присваиваем ему старое имя. Старый объект будет удалён сборщиком мусора. А команда nonlocal сообщает Python, что изменения ссылки на объект должны затронуть область видимости за пределами функции add_two_str.

Подведём промежуточный итог. Благодаря тому что в Python всё объект, а функции являются функциями высшего порядка, мы можем вкладывать во внешнюю функцию различные переменные и внутренние функции. Далее возвращая из внешней функции внутреннюю создаём замыкания.

Задание

Перед вами несколько строк кода. Напишите что выведет программа, не запуская код. У вас 3 минуты.

```
from typing import Callable


def main(x: int) -> Callable[[int], dict[int, int]]:
    d = {}

    def loc(y: int) -> dict[int, int]:
        for i in range(y):
            d[i] = x ** i
        return d

    return loc


small = main(42)
big = main(73)
print(small(7))
print(big(7))
print(small(3))
```

2. Простой декоратор без параметров

Передача функции в качестве аргумента

До этого момента наш код возвращал функции, но не принимал их. Исправим ситуацию на примере самописной функции нахождения факториала. Напомним, что факториал числа - произведение чисел от единицы до заданного числа.

```
import time
```

```
from typing import Callable


def main(func: Callable):
    def wrapper(*args, **kwargs):
        print(f'Запуск функции {func.__name__} в {time.time()}')
        result = func(*args, **kwargs)
        print(f'Результат функции {func.__name__}: {result}')
                print(f'Завершение функции {func.__name__} в
{time.time()}')
        return result

    return wrapper


def factorial(n: int) -> int:
    f = 1
    for i in range(2, n + 1):
        f *= i
    return f


print(f'{factorial(1000) = }')
control = main(factorial)
print(f'{control.__name__ = }')
print(f'{control(1000) = }')
```

- Функция main принимает на вход другую функцию. Внутри функции определена функция wrapper, которая возвращается функцией main.
- Функция wrapper принимает пару параметров *args и **kwargs. С ними вы уже знакомы. Подобная запись позволяет принять любое число позиционных аргументов и сохранить их в кортеже args, а также любое число ключевых аргументов с сохранением в словаре kwargs.
  Обязательной строкой внутри wrapper является result = func(*args, **kwargs). Переданная в качестве аргумента функция func вызывается со всеми аргументами, которые были переданы. Дополнительно выводим информацию о времени запуска, результатах и времени завершения работы функции. Не забываем вернуть результат работы func из wrapper.
- Функция factorial вычисляет факториал для заданного числа.
- В нижней части кода запускаем поиск факториала, проверяем работоспособность. Далее мы создаём функцию control в которую помещается wrapper с замкнутой внутри функций func — нашей функцией factorial. При вызове контрольной функции помимо результата поиска факториала получаем вывод прописанный внутри wrapper.

Замыкание переданной в качестве аргумента функции внутри другой функции называется декорированием функции. В нашем примере main — декоратор, которым мы декорировали функцию factorial.

Синтаксический сахар Python, @

В языке Python есть более элегантная возможность создания декораторов — синтаксический сахар. Для этого используется символ “@” слитно с именем декоратора. Строка кода пишется непосредственно над определением функции или метода.

```
import time
from typing import Callable


def main(func: Callable):
    def wrapper(*args, **kwargs):
        print(f'Запуск функции {func.__name__} в {time.time()}')
        result = func(*args, **kwargs)
        print(f'Результат функции {func.__name__}: {result}')
                print(f'Завершение функции {func.__name__} в
{time.time()}')
        return result

    return wrapper


@main
def factorial(n: int) -> int:
    f = 1
    for i in range(2, n + 1):
        f *= i
    return f


print(f'{factorial(1000) = }')
```

Добавили декоратор @main к функции factorial. Необходимость в присваивании значения новой переменной отпала. Несколько нижних строк кода из старого примера удалили за ненадобностью. Кроме того мы сохранили старое имя функции.

🔥 **Важно!** Функция декоратор должна быть определена в коде раньше, чем использована. В противном случае получим ошибку NameError

Синтаксический сахар упрощает написание кода, но не является обязательным к применению. Однако в случае с передачей функции в замыкание использование символа @ считается нормой. Связано это с тем, что присваивание переменной нового значения происходит очень часто в коде. И понять создаём мы замыкание функции или присваиваем что-то другое сложно. Когда же речь идёт о присваивании через @, сразу ясно что используется декоратор

Множественное декорирование

Python позволяет использовать несколько декораторов на одной функции. Рассмотрим на простом примере.

```
from typing import Callable


def deco_a(func: Callable):
    def wrapper_a(*args, **kwargs):
        print('Старт декоратора A')
        print(f'Запускаю {func.__name__}')
        res = func(*args, **kwargs)
        print(f'Завершение декоратора A')
        return res

    print('Возвращаем декоратор A')
    return wrapper_a


def deco_b(func: Callable):
    def wrapper_b(*args, **kwargs):
        print('Старт декоратора B')
        print(f'Запускаю {func.__name__}')
        res = func(*args, **kwargs)
        print(f'Завершение декоратора B')
        return res

    print('Возвращаем декоратор B')
    return wrapper_b
```

```
def deco_c(func: Callable):
    def wrapper_c(*args, **kwargs):
        print('Старт декоратора C')
        print(f'Запускаю {func.__name__}')
        res = func(*args, **kwargs)
        print(f'Завершение декоратора C')
        return res

    print('Возвращаем декоратор C')
    return wrapper_c


@deco_c
@deco_b
@deco_a
def main():
    print('Старт основной функции')


main()
```

Мы создали три одинаковых декоратора, которые сообщают о начале и завершении работы и о моменте декорирования: A, B, C.
Обратите внимание на порядок декораторов у функции main. Ближайший к функции декоратор A. Декоратор C находится первым в списке, т.е. он максимально удалён от основной функции.
При запуске кода процесс декорирования начинает снизу вверх, с A, далее B и лишь потом C.
Прежде чем выполнить код основной функции запускается код верхнего декоратора C, далее B, в конце нижний A и только потом код функции main. После того как декорированная функция завершила работу и вернула результат декораторы завершают работу в обратном старту порядке, снизу вверх. В зависимости от решаемых задач порядок декорирования может привести к разным результатам.

Дополнительные переменные в декораторе

Мы уже замыкали внутри функции список для хранения переданных имён. Декораторы открывают большие возможности по модификации основной функции. Рассмотрим пример простого кэширующего декоратора.

```
from typing import Callable


def cache(func: Callable):
    _cache_dict = {}

    def wrapper(*args):
        if args not in _cache_dict:
            _cache_dict[args] = func(*args)
        return _cache_dict[args]

    return wrapper


@cache
def factorial(n: int) -> int:
    print(f'Вычисляю факториал для числа {n}')
    f = 1
    for i in range(2, n + 1):
        f *= i
    return f


print(f'{factorial(10) = }')
print(f'{factorial(15) = }')
print(f'{factorial(10) = }')
print(f'{factorial(20) = }')
print(f'{factorial(10) = }')
print(f'{factorial(20) = }')
```

Внутри декоратора cache создали пустой словарь _cache_dict. При каждом вызове функции factorial внутри обёртки wrapper происходит проверка. Если переданное для нахождения факториало число не является ключём словаря, создаём соответствующий ключ и в качестве значения присваиваем ему результат вычисления факториала. Когда в словаре есть ключ, декорируемая функция не вызывается, а ответ сразу возвращается из словаря.

🔥 **Важно!** Мы специально исключили параметр **kwargs из функции wrapper, т.к. это словарь ключевых аргументов. Попытка использования в

качестве ключа словаря _cache_dict другого словаря kwargs приведёт к ошибке. Ключом может выступать только неизменяемые объекты.

Задание

Перед вами несколько строк кода. Напишите что выведет программа, не запуская код. У вас 3 минуты.

```
import random
from typing import Callable


def cache(func: Callable):
    _cache_dict = {}

    def wrapper(*args):
        if args not in _cache_dict:
            _cache_dict[args] = func(*args)
        return _cache_dict[args]

    return wrapper


@cache
def rnd(a: int, b: int) -> int:
    return random.randint(a, b)


print(f'{rnd(1, 10) = }')
print(f'{rnd(1, 10) = }')
print(f'{rnd(1, 10) = }')
```

3. Декоратор с параметрами

До этого мы вкладывали одну функцию в другую для создания замыкания. Если мы хотим передавать в декоратор дополнительные параметры, понадобится третий

уровень вложенности. Рассмотрим пример кода.

```
import time
from typing import Callable


def count(num: int = 1):
    def deco(func: Callable):
        def wrapper(*args, **kwargs):
            time_for_count = []
            result = None
            for _ in range(num):
                start = time.perf_counter()
                result = func(*args, **kwargs)
                stop = time.perf_counter()
                time_for_count.append(stop - start)
            print(f'Результаты замеров {time_for_count}')
            return result

        return wrapper

    return deco


@count(10)
def factorial(n: int) -> int:
    f = 1
    for i in range(2, n + 1):
        f *= i
    return f


print(f'{factorial(1000) = }')
print(f'{factorial(1000) = }')
```

- Внешняя функция count принимает на вход целое число num. Данный параметр будет использован для цикла for.
- Функция deco как и в прошлых примерах принимает декларируемую функцию.
- Внутренняя функция wrapper создаёт список time_for_count для хранения результатов замеров быстродействия.
    - Запускаем цикл for столько раз, сколько мы передали в декоратор: `@count(10)`

    - Внутри цикла for замеряем текущее время. Далее выполняем функцию и сохраняем результат в переменную. Замеряем время после окончания работы функции и сохраняем разницу в список.
    - После завершения цикла сообщаем результаты из списка time_for_count и возвращаем результат работы декларируемой функции.
- Используя обёртку для факториала делаем 10 замеров и смотрим время на вычисления.

🔥 **Важно!** Последняя строка дублируется не случайно. Каждый из двух запусков делает по 10 замеров. Если бы список time_for_count был создан на уровень выше, в функции deco, произошло бы его замыкание. В результате каждый новый вызов функции factorial дополнял бы уже существующий список, а не создавал бы новые 10 значений.

Декоратор с параметром может принимать любые значения в зависимости от предназначения.

🔥 **Важно!** Для оценки быстродействия кода рекомендуется использовать модуль timeit из “батареек Python”, а не созданный выше декоратор.

Задание

Перед вами несколько строк кода. Напишите что выведет программа, не запуская код. У вас 3 минуты.

```
import random
from typing import Callable


def count(num: int = 1):
    def deco(func: Callable):
        counter = []

        def wrapper(*args, **kwargs):
            for _ in range(num):
                result = func(*args, **kwargs)
                counter.append(result)
            return counter
```

```
        return wrapper

    return deco


@count(10)
def rnd(a: int, b: int) -> int:
    return random.randint(a, b)


print(f'{rnd(1, 10) = }')
print(f'{rnd(1, 100) = }')
print(f'{rnd(1, 1000) = }')
```

4. Декораторы functools

Дополнительные возможности декорирования предоставляет модуль functools декоратор.

Декоратор wraps

Рассмотрим код из прошлой главы, но добавим строку документации в функцию factorial.

```
...
@count(10)
def factorial(n: int) -> int:
    """Returns the factorial of the number n."""
    f = 1
    for i in range(2, n + 1):
        f *= i
    return f


print(f'{factorial(1000) = }')
print(f'{factorial.__name__ = }')
```

```
help(factorial)
```

Вместо ожидаемого вывода документации о функции и её названия получаем информацию об обёртке wrapper:

```
factorial.__name__ = 'wrapper'
Help on function wrapper in module __main__:

wrapper(*args, **kwargs)
```

Чтобы исправить ситуацию, воспользуемся декоратором wraps из functools.

```
import time
from typing import Callable
from functools import wraps


def count(num: int = 1):
    def deco(func: Callable):
        @wraps(func)
        def wrapper(*args, **kwargs):
            time_for_count = []
            ...
```

Декоратор wraps добавляется к функции wrapper, т.е. к самой глубоко вложенной функции. В качестве аргумента wraps должен получить параметр декларируемой функции. Теперь factorial возвращает свои название и строку документации.

Декоратор cache

Рассматривая возможности по замыканию переменных мы создали кэширующий декоратор. В модуле functools есть декоратор cache с подобным функционалом. При необходимости кэширования данных рекомендуется использовать именно его, как более оптимальное решение из коробки.

```
from functools import cache


@cache
def factorial(n: int) -> int:
```

```
    print(f'Вычисляю факториал для числа {n}')
    f = 1
    for i in range(2, n + 1):
        f *= i
    return f


print(f'{factorial(10) = }')
print(f'{factorial(15) = }')
print(f'{factorial(10) = }')
print(f'{factorial(20) = }')
print(f'{factorial(10) = }')
print(f'{factorial(20) = }')
```

Как вы видите только первые вызовы запускают функцию. Повторный вызов с уже передававшимся аргументом обрабатывается декоратором cache.

Вывод

На этой лекции мы:

1. Разобрали замыкания в программировании
2. Изучили возможности Python по созданию декораторов
3. Узнали как создавать декораторы с параметрами
4. Разобрали работу некоторых декораторов из модуля functools

Краткий анонс следующей лекции

1. Разберёмся с объектно-ориентированным программированием в Python.
2. Изучим особенности инкапсуляции в языке
3. Узнаем о наследовании и механизме разрешения множественного наследования.
4. Разберёмся с полиморфизмом объектов.

Домашнее задание

1. Примените рассматриваемые на лекции декораторы к функциям, созданным на прошлых уроках.
2. Попробуйте создать свои декораторы. Например вы можете написать декоратор, который считает количество вызовов функции.