InnoDom

Виды функций

Виды функций

- Именованная функция
- Lambda-функция
- Функция-генератор
- Вложенная функция
- Декоратор

Lambda-функции

Lambda-функция - это **простая анонимная** функция, которая нужна для **одноразового**, в плане программы, использования и занимает одну строчку. Если функцию можно представить как лямбду и использовать только в одном месте программы, то lambda-функция будет куда эффективнее в плане читабельности и понимания.

В большинстве случаев такие функции используется как аргументы встроенных, мощных функций таких, как *map* или *filter*, также их можно использовать параметром в *sorted*.

Lambda-функции

Для создания lambda-функции используется ключевое слово `lambda`, далее идут **передаваемые аргументы**, после двоеточия пишется **возвращаемое значение**.

```
lambda1 = lambda a, b: a + b
print(lambda1(1, 2))
list1 = [1, 2, 3, 4, 5, 6]
print(list(map(lambda x: x + 10, list1)))
```

Вложенные функции

это функции, которые находятся **друг в друге**. Такие функции могут быть использованы только **внутри родительской**. Благодаря этому можно добиться взаимодействия между функциями вместе с их общими аргументами.

```
def print_hello_world(val):
    def print_hello():
        print('hello')

    def print_world():
        print('world')

    if val == 1:
        print_hello()
    else:
        print_world()

print_hello_world(1)
print_hello_world(2)
```

Декораторы

это функции, позволяющие **обернуть другие функции**, для расширения функциональности функции, которую изменяет декоратор. Иными словами, **декоратор** - это функция, которая изменяет либо расширяет функционал другой функции.

Декораторы

Декораторы работают на основе *вложенных* функций: принимают в себя объект функции, которую в последующем можно вызвать внутри декоратора. Ведь функции, как и переменные, хранят только код, а не значения.

```
beef sandwich(func):
    def wrapper():
        print('\\\\\хлеб////')
        print("~~~салат~~~")
        func()
        print('...кетчуп...')
        print('\\\\хлеб////')

    return wrapper
```

Декораторы

Декораторы применяются на функции, которые требуется декорировать.

Такой декоратор можно применять на любую функцию, подходящую по смыслу. При вызове такой функции, декоратор будет отрабатывать вместе с ней.

```
@sandwich
def svinina():
    print('свинина')

@sandwich
def kurica():
    print('курица')

@sandwich
def govydina():
    print('говядина')

svinina()
govydina()
kurica()
```

Декораторы

Изменим код декоратора:

```
beef sandwich(func):
    def wrapper(*args, **kwargs):
        print('\\\\\хлеб////')
        print("~~~салат~~~")
        func(*args, **kwargs)
        print('...кетчуп...')
        print('\\\\хлеб////')

    return wrapper
```

И теперь декоратор работает с аргументами декорируемой функции:

```
@sandwich
def get_meat(meat):
    print(meat)

get_meat('курица')
```

Декораторы

С помощью декораторов можно легко создавать функции, считающие время их выполнения.

```python
import time


def check_time(func):
    def wraper(*args, **kwargs):
        start_time = time.time()
        original_val = func()
        end_time = time.time()
        print(f"Время выполнения: {end_time - start_time}")
        return original_val

    return wraper


@check_time
def freez():
    time.sleep(2)
    return True


print(freez())
```

Библиотека time – это стандартная библиотека Python для работы со временем, в основном в *unix*-формате. UNIX-формат – это формат времени, представленный в секундах, прошедших с полуночи 1 января 1970г. Мы использовали функцию `time()`, которая возвращает текущее время и функцию `sleep()`, которая замораживает время выполнения на кол-во передаваемых секунд.

Декораторы

Также декораторы можно писать с аргументами:

```python
import time


def str_print(str1):
    def check_time(func):
        def wrapper(*args, **kwargs):
            start_time = time.time()
            original_val = func()
            end_time = time.time()
            print(f"Время выполнения: {end_time - start_time}")
            return original_val + " " + str1

        return wrapper

    return check_time


@str_print("world!")
def freez():
    time.sleep(2)
    return "Hello"


print(freez())
```

Генераторы и итераторы

Итеративный объект – это объект, по которому можно совершить итерацию.
Итераторы – это функции либо конструкции языка, которые позволяют проходиться по итеративным объектам.

Вы уже знакомы со способом прохода через цикл. Однако, если в итеративном объекте хранится много значений, тогда на помощь приходит функция `iter()` и `next()`, которые позволяют не хранить целый объект в памяти, а извлекают его и хранят только **текущий** элемент.

Генераторы и итераторы

! Когда элементы заканчиваются, вызывается исключение **StopIteration**.

```
list1 = [1, 2, 3, 4, 5]
iter1 = iter(list1)
print(next(iter1))
print(next(iter1))
print(next(iter1))
print(next(iter1))
print(next(iter1))
print(next(iter1))
print(next(iter1))
```

Генераторы и итераторы

Генератор – это функция, которая позволяет делать вам **свои итераторы**.

Создадим итератор, который увеличивает значения в 2 раза:

```
def double_val_generator(val):
    while val < 1000000000:
        yield val
        val *= 2


mygenerator = double_val_generator(1234)
print(next(mygenerator))
print(next(mygenerator))
print(next(mygenerator))
```

Итеративные и генеративные выражения

Итеративные и **генеративные** выражения – это списковые включения для упрощения читабельности и сокращения кода, такие включения можно делать для списков, словарей, множеств, генераторов.

Шаблон спискового включения:

```
[возвращаемое_значение цикл условие_не_обязательно]
```

Итеративные и генеративные выражения

Генеративное выражение – это списковое включение, которое создаёт новую последовательность, создание происходит в виде **кортежа**, т.е в круглых скобках.

```
mygenerator = (i * 2 for i in range(0, 10))
print(next(mygenerator))
print(next(mygenerator))
print(next(mygenerator))
```

Итеративные и генеративные выражения

Итеративное выражение – это списковое включение, которое работает уже с существующим итерируемым объектом.

```
nums = [1, 2, 3, 4, 5]
myiterator = [1 for i in nums if i % 2 == 0]
print(myiterator)
```