Python

Декораторы

Декораторы — это своеобразные «обёртки», которые дают возможность делать что-либо до и после того, что сделает декорируемая функция, не изменяя её.

```
def my_decorator(function_to_decorate):
    def wrapper():
        print("Before code")
        function_to_decorate()
        print("After code")
    return wrapper

def my_function():
    print("Stand alone function")

my_function()
print()

my_function = my_decorator(my_function)
my_function()
```

@декораторы

@декоратор - это конструкция, которая не влияет на выполнение программы, но делает использование языка более удобным (синтаксический сахар) и аналогична **func = decorator(func)**

```
def my_decorator(function_to_decorate):
    def wrapper():
        print("Before code")
        function_to_decorate()
        print("After code")
    return wrapper


@my_decorator
def my_function():
    print("Stand alone function")

my_function()
```

Практика

```
import time

def get_time(func):
    def wrapper(*args, **kwargs):
        t0 = time.time()
        result = func(*args, **kwargs)
        t1 = time.time()
        print("Time spent: {0} seconds".format((t1-t0)))
        return result
    return wrapper

def get_prime_list(max_num):
    prime_list = []
    for n in range(2, max_num + 1):
        for x in range(2, n):
            if n % x == 0:
                break
        else:
            prime_list.append(n)
    return prime_list

@get_time
def check_prime(num):
    prime_list = get_prime_list(num)
    if num in prime_list:
        return True
    else:
        return False

print(check_prime(22511))
print(check_prime(22511))
```

Итераторы

Итераторы - объекты, представляющие последовательный доступ к данным.

При этом память фактически не тратится, так как промежуточные данные выдаются по мере необходимости при запросе.

- range(1, 10)
- (char for char in ‘abcd’)
- map(str, [1, 2, 3, 4, 5])
- iter([1, 2, 3, 4, 5])

Следующее значение итератора можно получить с помощью метода **.__next__()**

При окончании последовательности выдается исключение **StopIteration**

```
my_iterator = iter([1, 2])

print(my_iterator)
print(my_iterator.__next__())  # 1
print(my_iterator.__next__())  # 2
my_iterator.__next__()  # StopIteration
```

Генераторы

Генератор-функция - это функция, которая создает итератор.

Особенность: **yield** вместо **return**.

```
def create_generator(n):
    my_list = range(1, n+1)
    for num in my_list :
        yield num*num

my_generator = create_generator(5)

for item in my_generator:
    print(item, end=" ")

# 1 4 9 16 25
```

Темы для самостоятельной подготовки

- Работа с модулями

https://docs.python.org/3/tutorial/modules.html

- Работа с пакетами модулей:

https://docs.python.org/3/tutorial/modules.html#packages

- Работа с файлами

https://docs.python.org/3/tutorial/inputoutput.html#reading-and-writing-files

- Вызов исключений

https://docs.python.org/3/tutorial/errors.html#raising-exceptions