

Федеральное государственное автономное образовательное учреждение
высшего образования «Северо-Кавказский федеральный университет»
Институт цифрового развития

ОТЧЕТ ПО ДИСЦИПЛИНЕ
ОСНОВЫ ПРОГРАММНОЙ ИНЖЕНЕРИИ
Лабораторная работа №15

Выполнила:

Мирзаева Камилла Мирзаевна

2 курс, группа ПИЖ-б-о-20-1

Принял:

Воронкин Роман Александрович

Ставрополь, 2021 г.

Ход работы:

```
def hello_world():  
    print('Hello world!')
```

Рисунок 1 – Пример работы с областью видимости local

```
>>> def hello_world():  
...     print('Hello world!')  
...  
>>> type(hello_world)  
<class 'function'>  
>>> class Hello:  
...     pass  
...  
>>> type(Hello)  
<class 'type'>  
>>> type(10)  
<class 'int'>  
>>>
```

Рисунок 2 – Пример работы с областью видимости enclosing

```
>>> hello = hello_world  
>>> hello()
```

Рисунок 3 – Пример работы с функцией

```
>>> def wrapper_function():  
...     def hello_world():  
...         print('Hello world!')  
...     hello_world()  
...  
>>> wrapper_function()  
Hello world!
```

Рисунок 4 – Пример работы с функцией mul5

```
>>> def higher_order(func):  
...     print('Получена функция {} в качестве аргумента'.format(func))  
...     func()  
...     return func  
...  
>>> higher_order(hello_world)  
Получена функция <function hello_world at 0x000001DA1C2EA170> в качестве аргумента  
Hello world!  
<function hello_world at 0x000001DA1C2EA170>
```

Рисунок 5 – Пример замыкания

```
>>> def decorator_function(func):
...     def wrapper():
...         print('Функция-обёртка!')
...         print('Оборачиваемая функция: {}'.format(func))
...         print('Выполняем обёрнутую функцию...')
...         func()
...         print('Выходим из обёртки')
...     return wrapper
```

Рисунок 6 – Пример функции с использованием локальных и глобальных переменных

```
>>> @decorator_function
... def hello_world():
...     print('Hello world!')
...
>>> hello_world()
Функция-обёртка!
Оборачиваемая функция: <function hello_world at 0x000001DA1C2EA4D0>
Выполняем обёрнутую функцию...
Hello world!
Выходим из обёртки
```

Рисунок 7 – Пример работы с замыканием, как средством для построения иерархических данных

```
def benchmark(func):
    import time

    def wrapper():
        start = time.time()
        func()
        end = time.time()
        print('[*] Время выполнения: {} секунд.'.format(end-start))
    return wrapper

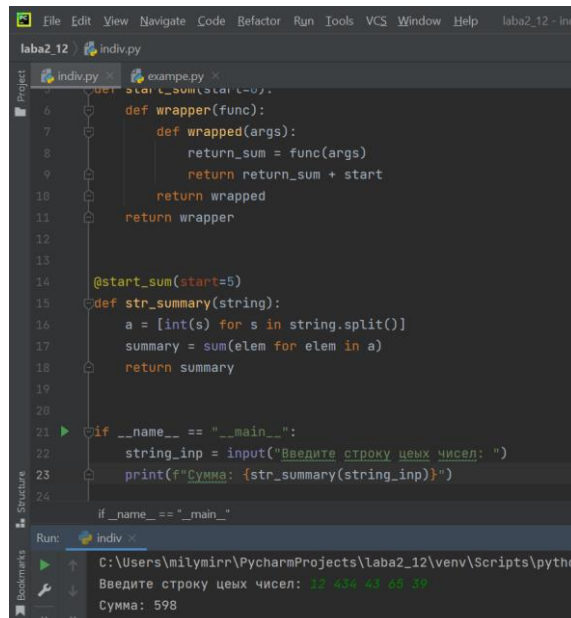
@benchmark
def fetch_webpage():
    import requests
    webpage = requests.get('https://google.com')
```

Рисунок 8 – Пример №1

Индивидуальное задание.

Вариант 15.

Вводится строка целых чисел через пробел. Напишите функцию, которая преобразовывает эту строку в список чисел и возвращает их сумму. Определите декоратор для этой функции, который имеет один параметр `start` – начальное значение суммы. Примените декоратор со значением `start=5` к функции и вызовите декорированную функцию. Результат отобразите на экране.



```
def start_sum(start=0):
    def wrapper(func):
        def wrapped(args):
            return_sum = func(args)
            return return_sum + start
        return wrapped
    return wrapper

@start_sum(start=5)
def str_summary(string):
    a = [int(s) for s in string.split()]
    summary = sum(elem for elem in a)
    return summary

if __name__ == "__main__":
    string_inp = input("Введите строку целых чисел: ")
    print(f"Сумма: {str_summary(string_inp)}")

if __name__ == "__main__":
    string_inp = input("Введите строку целых чисел: ")
    print(f"Сумма: {str_summary(string_inp)}")
```

Run: C:\Users\milymirr\PycharmProjects\laba2_12\venv\Scripts\python.exe
Введите строку целых чисел: 12 434 43 65 39
Сумма: 598

Рисунок 10 – Индивидуальное задание

Вопросы для защиты

1. Что такое декоратор?

Декоратор – это функция, которая позволяет обернуть другую функцию для расширения её функциональности без непосредственного изменения её кода.

2. Почему функции являются объектами первого класса?

Объектами первого класса в контексте конкретного языка программирования называется элементы, с которыми можно делать всё то же, что и с любым другим объектом: передавать, как параметр, возвращать из функции и присваивать переменной.

3. Каково назначение функций высших порядков?

Функции высших порядков – это такие функции, которые могут принимать в качестве аргументов и возвращать другие функции.

4. Как работают декораторы?

Декоратор – это функция, которая позволяет обернуть другую функцию для расширения её функциональности без непосредственного изменения её кода.

Внутри декораторы мы определяем другую функцию, обёртку, так сказать, которая обёртывает функцию-аргумент и затем изменяет её поведение. Мы создаём декоратор, измеряющий время выполнения функции. Далее мы используем его функции, которая делает GET-запрос к главной странице. Чтобы измерить скорость, мы сначала сохраняем время перед выполнением обёрнутой функции, выполняем её снова сохраняем текущее время и вычитаем из него начальное.

Выражение `@decorator_function` вызывает `decorator_function()` с `hello_world` в качестве аргумента и присваивает имени `hello_world` возвращаемую функцию.

```
def benchmark(func):
    import time

    def wrapper():
        start = time.time()
        func()
        end = time.time()
        print('[*] Время выполнения: {} секунд.'.format(end-start))
    return wrapper

@benchmark
def fetch_webpage():
    import requests
    webpage = requests.get('https://google.com')

fetch_webpage()
```

5. Какова структура декоратора функций?

```

def benchmark(func):
    import time

    def wrapper(*args, **kwargs):
        start = time.time()
        return_value = func(*args, **kwargs)
        end = time.time()
        print('[*] Время выполнения: {} секунд.'.format(end-start))
        return return_value
    return wrapper

@benchmark
def fetch_webpage(url):
    import requests
    webpage = requests.get(url)
    return webpage.text

webpage = fetch_webpage('https://google.com')
print(webpage)

```

6. Самостоятельно изучить как можно передать параметры декоратору, а не декорируемой функции?

```

import functools

def decoration(*args):
    def dec(func):
        @functools.wraps(func)
        def decor():
            func()
            print(*args)
        return decor
    return dec

@decoration('This is *args')
def func_ex():
    print('Look at that')

if __name__ == '__main__':
    func_ex()

```

```

Look at that
This is *args

Process finished with exit code 0

```