Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт цифрового развития
Кафедра инфокоммуникаций

ОТЧЕТ

По практической работе №2.12

Дисциплины «Программирование на Python»

Выполнил:

Пустяков Андрей Сергеевич

2 курс, группа ИВТ-б-о-22-1,

09.03.01 «Информатика и вычислительная техника (профиль) «Программное обеспечение средств вычислительной техники и автоматизированных систем», очная форма обучения

_____

(подпись)

Руководитель практики:

Воронкин Р. А. кандидат технических наук, доцент, доцент кафедры инфокоммуникаций

_____

(подпись)

Ставрополь, 2023 г.

Тема: Декораторы функций в языке Python.

Цель: приобрести навыки по работе с декораторами функций при написании программ с помощью языка программирования Python версии 3.x.

Ход работы:

Создание общедоступного репозитория на «GitHub», клонирование репозитория, редактирование файла «.gitignore», организация репозитория согласно модели ветвления «git-flow» (рис. 1).

```
C:\Program Files\Git>cd C:\Users\Andrey\Desktop\пайтон\15 лаба

C:\Users\Andrey\Desktop\пайтон\15 лаба>git clone https://github.com/AndreyPust/Python_laboratory_work_2.12.git
Cloning into 'Python_laboratory_work_2.12'...
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 5 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (5/5), done.

C:\Users\Andrey\Desktop\пайтон\15 лаба>cd C:\Users\Andrey\Desktop\пайтон\15 лаба\Python_laboratory_work_2.12

C:\Users\Andrey\Desktop\пайтон\15 лаба\Python_laboratory_work_2.12>git flow init

Which branch should be used for bringing forth production releases?
   - main
Branch name for production releases: [main]
Branch name for "next release" development: [develop]

How to name your supporting branch prefixes?
Feature branches? [feature/]
Bugfix branches? [bugfix/]
Release branches? [release/]
Hotfix branches? [hotfix/]
Support branches? [support/]
Version tag prefix? []
Hooks and filters directory? [C:/Users/Andrey/Desktop/пайтон/15 лаба/Python_laboratory_work_2.12/.git/hooks]
```

Рисунок 1 – Организация модели ветвления «git-flow».

Проработка примеров лабораторной работы:

Код программы примеров лабораторной работы (проработка функций и использование созданных декораторов) и результаты работы программы (рис. 2, 3).

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-


def hello_world():
    """
    Функция выводит надпись "Hello world!", пример простой функции.
    """
    print('Hello world!')
```

```
class Hello:
    """
    Определение нового класса по примеру.
    """
    pass


def wrapper_function():
    """
    Функция содержит внутри себя другую функцию и вызывает ее,
    которая выводит надпись 'Hello world!'.
    Пример функции, которая содержит в себе другую.
    """
    def hello_world_two():
        print('Hello world 2!')
    hello_world_two()


def higher_order(func):
    """
    Функция выводит информацию о функции, которую ей передали в качестве
аргумента.
    """
    print('Получена функция {} в качестве аргумента'.format(func))
    func()
    return func


def decorator_function(func):
    """
    Пример функции декоратора, которая изменяет поведение функции при
передаче этой функции
    в качестве аргумента другой функции.
    """
    def wrapper():
        print('Функция-обёртка!')
        print('Оборачиваемая функция: {}'.format(func))
        print('Выполняем обёрнутую функцию...')
        func()
        print('Выходим из обёртки')
    return wrapper


@decorator_function
def hello_world_thee():
    """
    Функция выводит надпись 'Hello world!'.
    Пример использования декоратора decorator_function.
    """
    print('Hello world 3!')


def benchmark(func):
    """
    Пример декоратора, который получает время выполнения функции.
    """
    import time

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

```
    return wrapper


@benchmark
def hello_world_four():
    """
    Функция выводит надпись 'Hello world!'.
    Пример использования декоратора benchmark для измерения времени.
    """
    print('Hello world 4!')


def benchmark_mod(func):
    """
    Модифицированный пример декоратора, который получает время выполнения
функции.
    """
    import time

    def wrapper_two(*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_two


@benchmark_mod
def hello_world_five():
    """
    Функция выводит надпись 'Hello world!'.
    Пример использования декоратора benchmark_mod.
    """
    print('Hello world 5!')


if __name__ == "__main__":
    print(type(hello_world))    # hello_world принадлежит типу <class
'function'>

    print(type(Hello))          # Класс Hello принадлежит классу type

    print(type(10))             # Пример класса 'int'

    hello = hello_world         # Пример хранения функции в переменной
    hello()

    wrapper_function()          # Пример функции, которая содержит в себе
другую функцию

    print(higher_order(hello_world))  # Пример передачи функции в качестве
аргумента

    hello_world_thee()          # Пример использования функции декоратора

    hello_world_four()          # Пример использования декоратора benchmark

    print(hello_world_five)     # Пример использования декоратора
benchmark_mod
```

Рисунок 2 – Код программы примеров лабораторной работы.

```
C:\Users\Andrey\AppData\Local\Programs\Python\Python39\python.exe "C:\Users\Andrey\Desktop\найт
<class 'function'>
<class 'type'>
<class 'int'>
Hello world!
Hello world 2!
Получена функция <function hello_world at 0x0000010B4400F310> в качестве аргумента
Hello world!
<function hello_world at 0x0000010B4400F310>
Функция-обёртка!
Оборачиваемая функция: <function hello_world_thee at 0x0000010B4400F820>
Выполняем обёрнутую функцию...
Hello world 3!
Выходим из обёртки
Hello world 4!
[*] Время выполнения: 0.0 секунд.
<function benchmark_mod.<locals>.wrapper_two at 0x0000010B4400FC10>
```

Рисунок 3 – Результаты работы программы примеров.

Выполнение индивидуального задания:

Необходимо объявить функцию, которая переводит переданную ей строку в нижний регистр. Необходимо определить декоратор для этой функции, который имеет один параметр «tag», определяющий строку с названием тега. Начальное значение тега это «h1». Этот декоратор должен заключать возвращенную функцией строку в соответствующий тег и возвращать результат. Пример заключения строки «Python» в тег «h1»: «<h1>python</h1>». Необходимо в качестве примера вызвать декорированную функцию со значением тега tag = div и вывести результат на экран (Вариант 26 (6)).

Код программы индивидуального задания и результаты работы программы с различными значениями тега «tag» и различными строками (рис. 4, 5).

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-


def tag_decorator(func):
    """
    Функция декоратор, которая обращает строку, которую возвращает функция
string_lover
    в соответствующий тег.
    """

    def wrapper(text):
```

```
        result = func(text)
        print(f"<{tag_html}>{result}</{tag_html}>")
    return wrapper


@tag_decorator
def string_lover(text):
    """
    Функция переводит переданную ей строку в нижний регистр.
    """
    return str(text).lower()


if __name__ == "__main__":

    # Необходимо объявить функцию, которая переводит переданную ей строку в
нижний регистр.
    # Необходимо определить декоратор для этой функции, который имеет один
параметр «tag»,
    # определяющий строку с названием тега. Начальное значение тега это «h1».
Этот декоратор
    # должен заключать возвращенную функцией строку в соответствующий тег и
возвращать результат.
    # Пример заключения строки «Python» в тег «h1»: «<h1>python</h1>».
Необходимо в качестве
    # примера вызвать декорированную функцию со значением тега tag = div и
вывести результат на
    # экран (Вариант 26 (6)).

    unmarked_text = str(input("Введите строку, которую необходимо перевести в
нижний регистр:"))
    tag_html = str(input("Введите тег, в который необходимо заключить
строку:"))

    string_lover(unmarked_text)
```

Рисунок 4 – Код программы индивидуального задания.

```
C:\Users\Andrey\AppData\Local\Programs\Python\Python39\python.exe "C:\Users\Andrey\Des
Введите строку, которую необходимо перевести в нижний регистр:Python
Введите тег, в который необходимо заключить строку:h1
<h1>python</h1>
```

```
C:\Users\Andrey\AppData\Local\Programs\Python\Python39\python.exe "C:\Users\Andrey\Desktop\пайтон\15 лаба
Введите строку, которую необходимо перевести в нижний регистр:Пример разметки текста для HTML-документа.
Введите тег, в который необходимо заключить строку:div
<div>пример разметки текста для html-документа.</div>

Process finished with exit code 0
```

Рисунок 5 – Результаты работы программы индивидуального задания.

Ответы на контрольные вопросы:

1. Декоратор – функция, которая позволяет обернуть другую функцию для расширения её функциональности без непосредственного изменения её кода. Декораторы можно рассматривают как практику метапрограммирования, когда программы могут работать с другими программами как со своими данными.

2. В Python всё является объектами, благодаря этому можно сохранять функции в переменные, передавать их в качестве аргументов и возвращать из других функций. Можно даже определить одну функцию внутри другой. Иными словами, функции – это объекты первого класса (Объектами первого класса называются элементы, с которыми можно делать всё то же, что и с любым другим объектом: передавать как параметр, возвращать из функции и присваивать переменной). Чтобы функцию можно было передавать или возвращать ее сделали объектом первого класса.

3. Функции высших порядков принимают в качестве аргументов (и возвращать) другие функции. Аналогично в математике – операции дифференцирования или интегрирования.

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

5. Структура декоратора представляет собой функцию с вложенной в нее другой функцией. В качестве аргументов декоратору передается необходимая для декорирования функция. Декоратор возвращает результаты вложенной функции. Вложенной функции могут передаваться те же параметры, что и декорируемой функции. Для применения декоратора необходимо перед функцией написать имя декоратора «@example_decorator».

6. Для передачи параметров декоратору, а не декорируемой функции, можно создать функцию-обёртку для декоратора, принимающую необходимые параметры и возвращающую сам декоратор.

Вывод: в ходе выполнения лабораторной работы были более подробно рассмотрены функции декораторы языка программирования Python, которые

в значительно расширяют работу функций без изменения их кода. Были рассмотрены способы создания декораторов, способы применения декораторов на декорируемых функциях, а также структура декораторов. Также были более подробно рассмотрены функции в Python как объекты первого класса. Были рассмотрены способы написания собственных декораторов для функций помимо встроенных.