

**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ
ФЕДЕРАЦИИ**

**Федеральное государственное автономное образовательное учреждение
высшего образования**

«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт цифрового развития

Кафедра информационных систем и технологий

Отчет по лабораторной работе №15.

Дисциплина: «Основы программной инженерии»

Выполнил:

Студент группы ПИЖ-б-о-22-1,

направление подготовки: 09.03.04

«Программная инженерия»

ФИО: Джараян Арег Александрович

Проверил:

Воронкин Р. А.

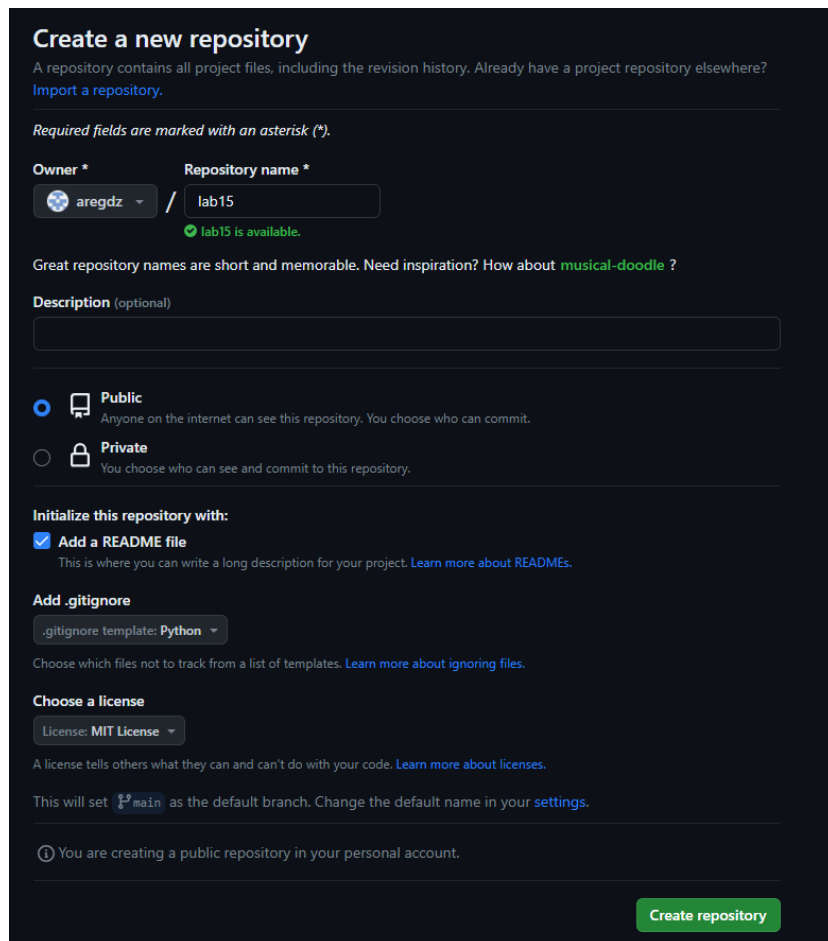
Ставрополь 2023

Тема: Лабораторная работа 2.12 Декораторы функций в языке Python.

Цель работы: приобретение навыков по работе с декораторами функций при написании программ с помощью языка программирования Python версии 3.x.

Выполнение работы:

1. Изучил теоретический материал работы.
2. Создал репозиторий на git.hub.



Create a new repository
A repository contains all project files, including the revision history. Already have a project repository elsewhere?
[Import a repository.](#)

Required fields are marked with an asterisk ().*

Owner * aregdz / **Repository name *** lab15
lab15 is available.

Great repository names are short and memorable. Need inspiration? How about [musical-doodle](#) ?

Description (optional)

☐ **Public**
Anyone on the internet can see this repository. You choose who can commit.

☐ **Private**
You choose who can see and commit to this repository.

Initialize this repository with:

☒ **Add a README file**
This is where you can write a long description for your project. [Learn more about READMEs.](#)

Add .gitignore
.gitignore template: [Python](#)

Choose which files not to track from a list of templates. [Learn more about ignoring files.](#)

Choose a license
License: [MIT License](#)

A license tells others what they can and can't do with your code. [Learn more about licenses.](#)

This will set `main` as the default branch. Change the default name in your [settings](#).

ⓘ You are creating a public repository in your personal account.

[Create repository](#)

Рисунок 1 – создание репозитория

3. Клонировал репозиторий.

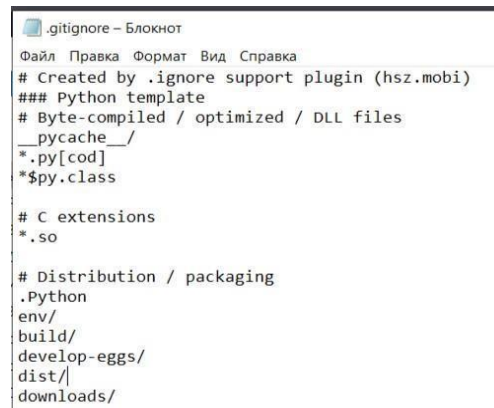
```
aregd@DESKTOP-5KV9QA9 MINGW64 /d/Рабочий стол/git 14/lab14 (main)
$ cd "D:\Рабочий стол\git 15"

aregd@DESKTOP-5KV9QA9 MINGW64 /d/Рабочий стол/git 15
$ git clone https://github.com/aregdz/lab15.git
Cloning into 'lab15'...
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 5 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (5/5), done.

aregd@DESKTOP-5KV9QA9 MINGW64 /d/Рабочий стол/git 15
```

Рисунок 2 – клонирование репозитория 4.

Дополнить файл gitignore необходимыми правилами.



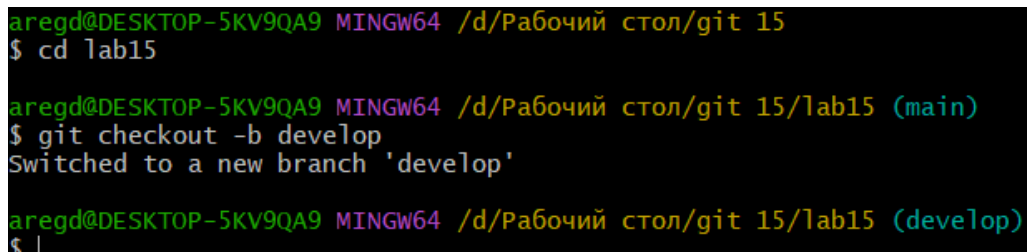
```
.gitignore – Блокнот
Файл  Правка  Формат  Вид  Справка
# Created by .ignore support plugin (hsz.mobi)
### Python template
# Byte-compiled / optimized / DLL files
__pycache__/
*.py[cod]
*$py.class

# C extensions
*.so

# Distribution / packaging
.Python
env/
build/
develop-eggs/
dist/
downloads/
```

Рисунок 3 – .gitignore для IDE PyCharm

4. Организовать свой репозиторий в соответствии с моделью ветвления git-flow.



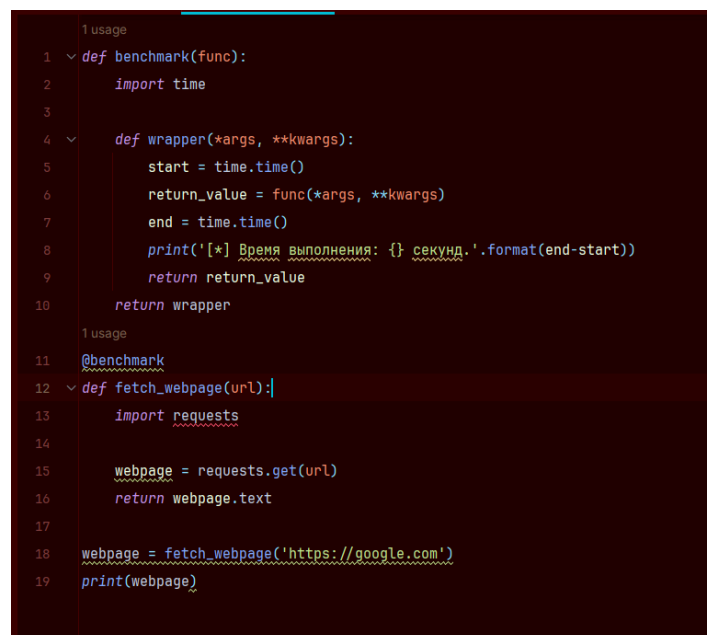
```
aregd@DESKTOP-5KV9QA9 MINGW64 /d/Рабочий стол/git 15
$ cd lab15

aregd@DESKTOP-5KV9QA9 MINGW64 /d/Рабочий стол/git 15/lab15 (main)
$ git checkout -b develop
Switched to a new branch 'develop'

aregd@DESKTOP-5KV9QA9 MINGW64 /d/Рабочий стол/git 15/lab15 (develop)
$ |
```

Рисунок 4 – создание ветки develop

5. Проработал примеры из методички.



```
1 usage
2
3
4 1 def benchmark(func):
5     import time
6
7     def wrapper(*args, **kwargs):
8         start = time.time()
9         return_value = func(*args, **kwargs)
10        end = time.time()
11        print('[*] Время выполнения: {} секунд.'.format(end-start))
12        return return_value
13
14    return wrapper
15
16 usage
17
18 @benchmark
19 def fetch_webpage(url):
20     import requests
21
22     webpage = requests.get(url)
23     return webpage.text
24
25 webpage = fetch_webpage('https://google.com')
26 print(webpage)
```

Рисунок 5 – пример 1

6. 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.

```
1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3
4  def sum_decorator(start):
5      def decorator(func):
6          def wrapper(input_string):
7              numbers = [int(num) for num in input_string.split()]
8              result = func(numbers)
9              return result + start
10         return wrapper
11     return decorator
12
13 @sum_decorator(start=5)
14 def sum_of_numbers(numbers):
15     return sum(numbers)
16
17 if __name__ == "__main__":
18     # Ввод строки целых чисел через пробел
19     input_string = input("Введите строку целых чисел через пробел: ")
20
21     # Вызов декорированной функции и вывод результата
22     result = sum_of_numbers(input_string)
23     print(f"Сумма чисел с учетом начального значения: {result}")
```

Рисунок 6 – индивидуальное задание

```
C:\Users\aregd\AppData\Local\Programs\Python\Python311\python.exe
Введите строку целых чисел через пробел: 1 2 3 4 5 6 7 8 9 10
Сумма чисел с учетом начального значения: 60
Alt+4
Process finished with exit code 0
```

Рисунок 7 – индивидуальное задание

7. Зафиксировал все изменения в github в ветке develop.

```

aregd@DESKTOP-5KV9QA9 MINGW64 /d/Рабочий стол/git 15/lab15 (develop)
$ git add .

aregd@DESKTOP-5KV9QA9 MINGW64 /d/Рабочий стол/git 15/lab15 (develop)
$ git commit -m"задания"
[develop 6f4a8ad] задания
 2 files changed, 42 insertions(+)
 create mode 100644 PyCharm/individual.py
 create mode 100644 PyCharm/primer 1.py

aregd@DESKTOP-5KV9QA9 MINGW64 /d/Рабочий стол/git 15/lab15 (develop)
$ git push origin develop
Enumerating objects: 6, done.
Counting objects: 100% (6/6), done.
Delta compression using up to 8 threads
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 1.13 KiB | 1.13 MiB/s, done.
Total 5 (delta 0), reused 0 (delta 0), pack-reused 0
remote:
remote: Create a pull request for 'develop' on GitHub by visiting:
remote:   https://github.com/aregdz/lab15/pull/new/develop
remote:
To https://github.com/aregdz/lab15.git
 * [new branch]      develop -> develop

aregd@DESKTOP-5KV9QA9 MINGW64 /d/Рабочий стол/git 15/lab15 (develop)

```

Рисунок 8 – фиксация изменений в ветку develop

8. Слил ветки.

```

aregd@DESKTOP-5KV9QA9 MINGW64 /d/Рабочий стол/git 15/lab15 (develop)
$ git checkout main
Switched to branch 'main'
Your branch is up to date with 'origin/main'.

aregd@DESKTOP-5KV9QA9 MINGW64 /d/Рабочий стол/git 15/lab15 (main)
$ git merge develop
Updating 6a87881..6f4a8ad
Fast-forward
 PyCharm/individual.py | 23 ++++++
 PyCharm/primer 1.py   | 19 ++++++
 2 files changed, 42 insertions(+)
 create mode 100644 PyCharm/individual.py
 create mode 100644 PyCharm/primer 1.py

aregd@DESKTOP-5KV9QA9 MINGW64 /d/Рабочий стол/git 15/lab15 (main)
$ |

```

Рисунок 9 – сливание ветки develop в ветку main

Контрольные вопросы:

1. Что такое декоратор?

Декораторы — один из самых полезных инструментов в Python, однако новичкам они могут показаться непонятными. 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 называются элементы, с которыми можно делать всё то же, что и с любым другим объектом: передавать как параметр, возвращать из функции

и присваивать переменной. Именно поэтому функции являются объектами первого класса.

3. Каково назначение функций высших порядков?

Он принимает на входе функцию и возвращает другую функцию, производную от исходной. Функции высших порядков в программировании работают точно так же — они либо принимают функцию(и) на входе и/или возвращают функцию(и).

4. Как работают декораторы?

Декораторы в Python представляют собой способ изменить поведение функции или метода, обернув его в другую функцию. Это мощный механизм, который позволяет добавлять или изменять функциональность функций без изменения их кода. Декораторы часто используются для внесения дополнительной логики, проверок или изменений в функции.

Декораторы позволяют модифицировать поведение функций или методов, делая код более модульным и легким для понимания. Они часто используются, например, для логирования, обработки ошибок, кеширования, аутентификации и других аспектов функциональности программы.

5. Какова структура декоратора функций?

```
def decorator_function(original_function):  
    def wrapper_function(*args, **kwargs):  
        # Дополнительный код, выполняемый перед вызовом оригинальной  
функции  
        result = original_function(*args, **kwargs)  
        # Дополнительный код, выполняемый после вызова оригинальной  
функции  
        return result  
    return wrapper_function
```

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

Использование функций-фабрик декораторов:

```
def decorator_factory(param):
    def decorator_function(original_function):
        def wrapper_function(*args, **kwargs):
            print(f"Дополнительный код с параметром {param} перед вызовом функции")
            result = original_function(*args, **kwargs)
            print(f"Дополнительный код с параметром {param} после вызова функции")
            return result
        return wrapper_function
    return decorator_function

# Использование декоратора с параметром
@decorator_factory(param="some_parameter")
def example_function():
    print("Оригинальная функция")

# Вызов функции, обернутой в декоратор
example_function()
```

2. Использование частичного применения (functools.partial):

```
from functools import partial

def decorator_function(param, original_function, *args, **kwargs):
    print(f"Дополнительный код с параметром {param} перед вызовом функции")
    result = original_function(*args, **kwargs)
    print(f"Дополнительный код с параметром {param} после вызова функции")
    return result

# Создание частичной функции с фиксированным параметром
decorator_with_param = partial(decorator_function, param="some_parameter")

# Использование декоратора с параметром
@decorator_with_param
def example_function():
    print("Оригинальная функция")

# Вызов функции, обернутой в декоратор
example_function()
```