

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 ЛАБОРАТОРНОЙ РАБОТЕ №2.25
дисциплины «Анализ данных»

Выполнила:

Мурашко Анастасия Юрьевна
2 курс, группа ИВТ-б-о-22-1,
11.03.02 «Информатика и
вычислительная техника», очная
форма обучения

(подпись)

Руководитель практики:

Воронкин Р.А., канд. тех. наук, доцент,
доцент кафедры инфокоммуникаций

(подпись)

Отчет защищен с оценкой _____ Дата защиты _____

Ставрополь, 2024 г.

Тема: Управление потоками в Python.

Цель работы: приобретение навыков написания многопоточных приложений на языке программирования Python версии 3.x.

Порядок выполнения работы:

Задание 1.

Изучила теоретический материал работы, создала общедоступный репозиторий на GitHub, в котором использована лицензий MIT и язык программирования Python, также добавила файл .gitignore с необходимыми правилами.

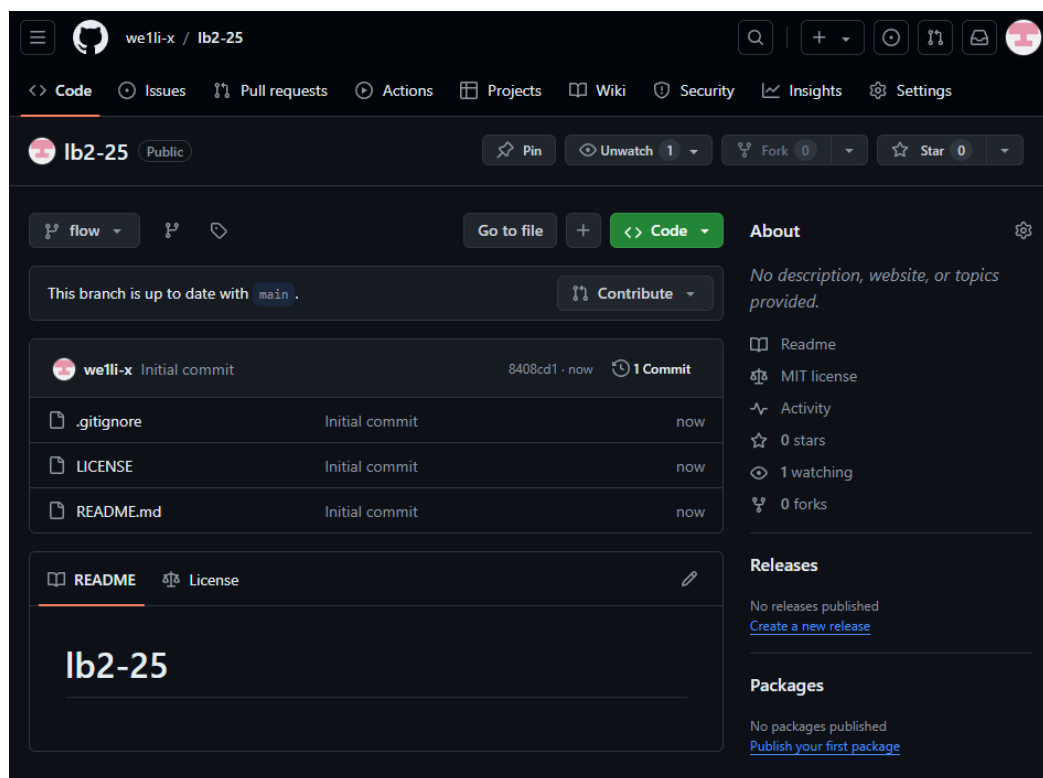


Рисунок 1. Новый репозиторий

Задание 2.

Проклонировала свой репозиторий на свой компьютер.

Организовала свой репозиторий в соответствии с моделью ветвления git-flow, появилась новая ветка develop.

Реализовывала примеры и индивидуальные задания на основе ветки develop, без создания дополнительной ветки feature/(название ветки) по указанию преподавателя.

Задание 3.

Создала виртуальное окружение (ВО) Miniconda и активировал его, также установила необходимые пакеты isort, black, flake8.

```
(base) C:\Users\nasty>cd C:\Users\nasty\PycharmProjects\pythonProject

(base) C:\Users\nasty\PycharmProjects\pythonProject>conda create -n 2.25 python=3.11
Channels:
 - defaults
Platform: win-64
Collecting package metadata (repodata.json): done
Solving environment: done

## Package Plan ##

  environment location: C:\Users\nasty\anaconda3\envs\2.25

added / updated specs:
 - python=3.11
```

Рисунок 3. Создание ВО

```
(base) C:\Users\nasty\PycharmProjects\pythonProject>conda activate 2.25

(2.25) C:\Users\nasty\PycharmProjects\pythonProject>conda install -c conda-forge black
Channels:
 - conda-forge
 - defaults
Platform: win-64
Collecting package metadata (repodata.json): done
Solving environment: done
```

Рисунок 4. Установка пакета black

```
(2.25) C:\Users\nasty\PycharmProjects\pythonProject>conda install -c conda-forge flake8
Channels:
 - conda-forge
 - defaults
Platform: win-64
Collecting package metadata (repodata.json): done
Solving environment: done
```

Рисунок 5. Установка пакета flake8

```
(2.25) C:\Users\nasty\PycharmProjects\pythonProject>conda install -c conda-forge isort
Channels:
- conda-forge
- defaults
Platform: win-64
Collecting package metadata (repodata.json): done
Solving environment: done

## Package Plan ##

environment location: C:\Users\nasty\anaconda3\envs\2.25
```

Рисунок 6. Установка пакета isort

Пакет isort (isrot) является инструментом для автоматической сортировки импортов в Python-кодах. Он используется для удобства чтения и поддержания порядка в коде.

Пакет black представляет инструмент автоматического форматирования кода для языка Python. Он помогает обеспечить единообразие стиля кодирования в проекте и улучшает читаемость кода.

Пакет flake8 отвечает за статический анализ и проверку Python-кода. Он проводит проверку на соответствие стилю кодирования PEP 8, а также наличие потенциальных ошибок и проблемных паттернов в коде.

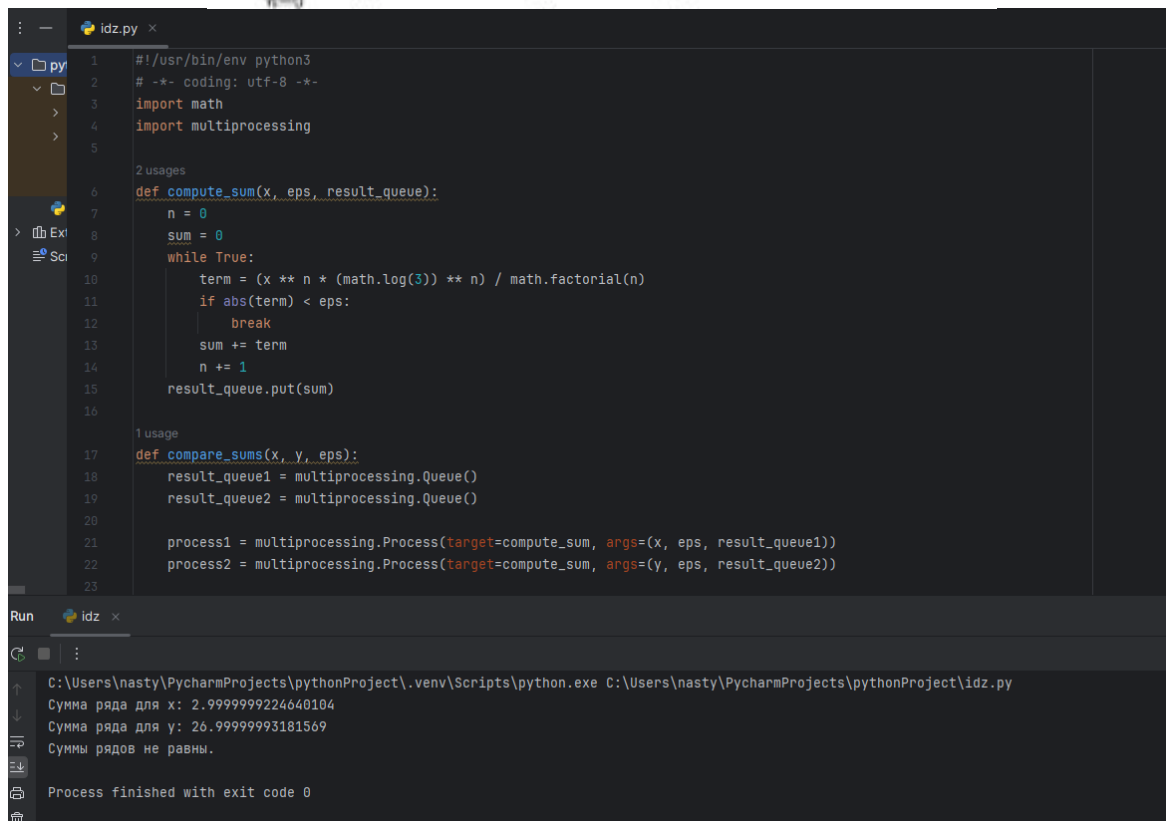
Задание 3.

Выполнение индивидуального задания.

Для своего индивидуального задания лабораторной работы 2.23 необходимо реализовать вычисление значений в двух функций в отдельных процессах.

Условие задания 2.23: с использованием многопоточности для заданного значения x найти сумму ряда S с точностью члена ряда по абсолютному значению $\epsilon = 10^{-7}$ и произвести сравнение полученной суммы с контрольным значением функции для двух бесконечных рядов.

$$S = \sum_{n=0}^{\infty} \frac{x^n \ln^n 3}{n!} = 1 + \frac{x \ln 3}{1!} + \frac{x^2 \ln^2 3}{2!} + \dots; \quad x = 1; \quad y = 3^x.$$



```
1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  import math
4  import multiprocessing
5
6  2 usages
7  def compute_sum(x, eps, result_queue):
8      n = 0
9      sum = 0
10     while True:
11         term = (x ** n * (math.log(3)) ** n) / math.factorial(n)
12         if abs(term) < eps:
13             break
14         sum += term
15         n += 1
16     result_queue.put(sum)
17
18  1 usage
19  def compare_sums(x, y, eps):
20     result_queue1 = multiprocessing.Queue()
21     result_queue2 = multiprocessing.Queue()
22
23     process1 = multiprocessing.Process(target=compute_sum, args=(x, eps, result_queue1))
24     process2 = multiprocessing.Process(target=compute_sum, args=(y, eps, result_queue2))
25
26     process1.start()
27     process2.start()
28
29     process1.join()
30     process2.join()
31
32     sum1 = result_queue1.get()
33     sum2 = result_queue2.get()
34
35     return sum1, sum2
```

Run idz x

C:\Users\nasty\PycharmProjects\pythonProject\.venv\Scripts\python.exe C:\Users\nasty\PycharmProjects\pythonProject\idz.py

Сумма ряда для x: 2.9999999224640104

Сумма ряда для y: 26.99999993181569

Суммы рядов не равны.

Process finished with exit code 0

Рисунок 7. Результат индивидуального задания

Код начинается с импорта необходимых модулей - `math` для математических операций и `multiprocessing` для работы с многопоточностью.

Для реализации вычисления значений двух функций в отдельных процессах, мы будем использовать модуль `multiprocessing` вместо `threading`. Многопроцессорность более подходит для задач, требующих интенсивных вычислений, так как каждый процесс выполняется в отдельном адресном пространстве и использует отдельный набор ядер процессора.

Следующим определяется класс `SumThread`, который наследуется от `threading.Thread`. Этот класс переопределяет метод `__init__`, чтобы принимать два аргумента: `x` и `eps`. В методе `__init__` также инициализируется переменная `self.sum`, которая будет хранить вычисленную сумму.

Метод `run` вычисляет члены ряда и добавляет их к `self.sum`, пока абсолютное значение члена ряда больше `eps`.

В этом коде мы создаем процессы вместо потоков. Каждый процесс вычисляет сумму ряда для заданного значения `x` и `y` с помощью функции `compute_sum`. Результаты вычислений передаются через очереди `multiprocessing.Queue()`, чтобы можно было получить их в основном процессе для сравнения.

Запуск функции `compare_sums` происходит внутри блока `if __name__ == "__main__":`, что гарантирует корректный запуск процессов при импорте этого скрипта в другие модули или при запуске его как основного скрипта.

Следующим определяется функция `compare_sums`, которая создает два экземпляра класса `SumThread`, каждый из которых вычисляет сумму ряда для

своего аргумента. Затем она запускает оба потока с помощью метода `'start'`, а затем ожидает их завершения с помощью метода `'join'`.

После завершения работы потоков, функция `'compare_sums'` извлекает вычисленные суммы из потоков и выводит их на экран. Затем она сравнивает эти суммы и выводит сообщение о том, равны ли они.

Наконец, вызывается функция `'compare_sums'` с аргументами `'1'`, `'3*1'` (то же самое, что и `'3'`) и `'10**-7'`.

Ответы на контрольные вопросы:

1. Как создаются и завершаются процессы в Python?

Создание процессов:

Модуль `os`:

- Используйте функцию `os.fork()`, которая создает копию текущего процесса.
- Возвращает 0 в дочернем процессе и PID (идентификатор процесса) дочернего процесса в родительском процессе.

Модуль `subprocess`:

- Предоставляет функции более высокого уровня для работы с процессами, такие как `Popen()`, `check_output()` и `call()`.
- Упрощает запуск процессов, захват их вывода и управление ими.

Модуль `multiprocessing`:

- Предназначен для более сложного управления несколькими процессами.
- Предоставляет классы `Process` и `Queue` для создания и взаимодействия между процессами.

Завершение процессов:

Метод `join()`:

- Блокирует родительский процесс до завершения дочернего.

Метод `terminate()`:

- Отправляет процессу сигнал `SIGTERM`, который **обычно** приводит к его **корректной остановке**.

Метод `kill()`:

- Отправляет процессу сигнал `SIGKILL`, который **немедленно**

завершает его работу без очистки ресурсов.

2. Особенности создания классов-наследников от Process: Возможность создать собственные классы-наследники от Process, чтобы лучше структурировать код и данные, связанные с процессом. Основное отличие заключается в том, что вы можете переопределить метод run, который будет выполняться при запуске процесса. Это позволяет более гибко управлять поведением процесса. Совместимость с threading.Thread: Поддерживает сигнатуры методов и конструктора, упрощая переход от многопоточного к многопроцессному приложению.

3. Принудительное завершение процесса: Принудительное завершение процесса осуществляется методом terminate. После вызова этого метода процесс завершится немедленно. Стоит отметить, что terminate может оставить ресурсы в некорректном состоянии, поэтому его следует использовать с осторожностью.

4. Процессы-демоны: Процессы-демоны (daemon processes) работают в фоновом режиме и автоматически завершаются, когда завершится основной процесс. Запуск процесса-демона: Чтобы запустить процесс в режиме демона, установите его атрибут daemon в True перед вызовом start.

Вывод: в ходе выполнения лабораторной работы были приобретены навыки написания многопоточных приложений на языке программирования Python версии 3.x.