

**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ
АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО
ОБРАЗОВАНИЯ «СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ
УНИВЕРСИТЕТ» ИНСТИТУТ ЦИФРОВОГО РАЗВИТИЯ**

Отчет по лабораторной работе № 2.23

«Управление потоками в Python»

По дисциплине «Основы программной инженерии»

Выполнила:

Образцова Мария Дмитриевна,
2 курс, группа ПИЖ-б-о-21-1,

Проверил:


Доцент кафедры инфокоммуникаций,
Воронкин Р.А.

Ставрополь, 2023 г.

Методика и порядок выполнения работы

1. Изучить теоретический материал работы.
2. Создать общедоступный репозиторий на GitHub, в котором будет использована лицензия MIT и язык программирования Python.

Owner *

 obrMaria ▾


Repository name *

OPI_2.23


✓ OPI_2.23 is available.

Great repository names are short and memorable. Need inspiration?

Description (optional)

☒  **Public**

Anyone on the internet can see this repository. You choose who can

☐  **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](#)

Add .gitignore

.gitignore template: Python ▾

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

Choose a license

License: MIT License ▾

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


This will set  main as the default branch. Change the default name

Рисунок 1 – создание репозитория

```
C:\Users\A\Desktop>git clone https://github.com/obrMaria/OPI_2.23.git
Cloning into 'OPI_2.23'...
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.
C:\Users\A\Desktop>cd OPI_2.23
```

Рисунок 2 – клонирование репозитория

```
$ 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? []
```

Рисунок 3 – создание ветки develop

Проработайте примеры лабораторной работы. Создайте для них отдельные модули языка Python. Зафиксируйте изменения в репозитории.

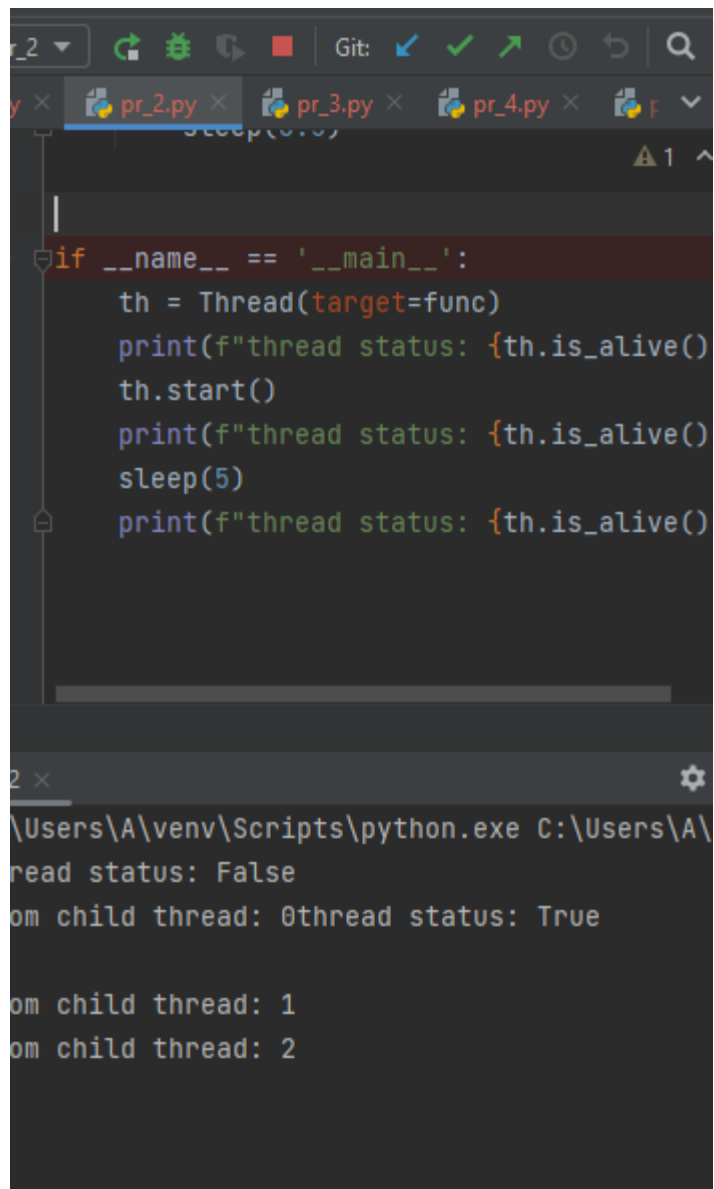
The image shows a code editor with four tabs: pr_1.py, pr_2.py, pr_3.py, and pr_4.py. The active tab is pr_1.py, which contains the following Python code:

```
1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4
5 from threading import Thread
6 from time import sleep
7
8
9 def func():
10     for i in range(5):
11         print(f"from child thread: {i}")
12         sleep(0.5)
13
14 if __name__ == '__main__':
15     t = Thread(target=func)
16     t.start()
17     t.join()
```

Below the code editor, the output of the program is displayed in a terminal window. The output shows the thread status changing from False to True, followed by five lines of output from the child thread (0 to 4), and finally the thread status returning to False. The process finishes with exit code 0.

```
↑ thread status: False
↓ from child thread: 0 thread status: True
↺
↻ from child thread: 1
from child thread: 2
from child thread: 3
from child thread: 4
thread status: False
Process finished with exit code 0
```

Рисунок – Создание и ожидание завершения работы потоков



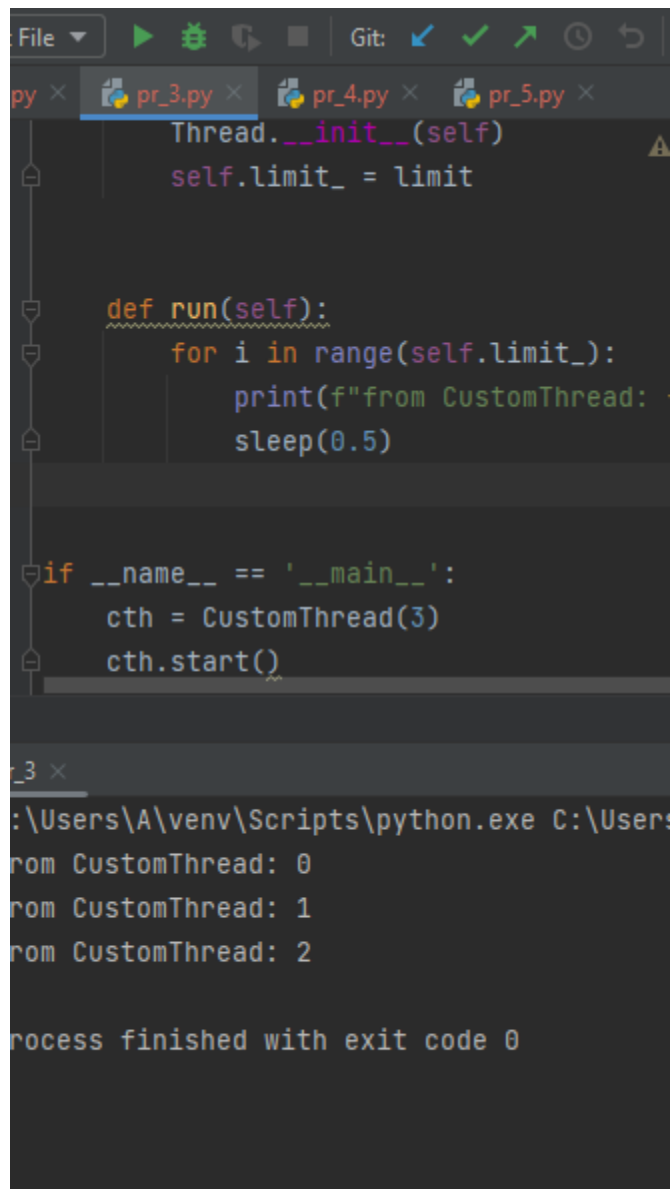
The image shows a code editor window with several tabs labeled pr_2.py, pr_3.py, pr_4.py, and pr_5.py. The active tab is pr_2.py, which contains the following Python code:

```
if __name__ == '__main__':  
    th = Thread(target=func)  
    print(f"thread status: {th.is_alive()}")  
    th.start()  
    print(f"thread status: {th.is_alive()}")  
    sleep(5)  
    print(f"thread status: {th.is_alive()}")
```

Below the code editor is a terminal window showing the output of the program. The output is as follows:

```
C:\Users\A\venv\Scripts\python.exe C:\Users\A\...  
thread status: False  
from child thread: 0thread status: True  
  
from child thread: 1  
from child thread: 2
```

Рисунок – Метод is_alive()



The screenshot shows a Python IDE with a dark theme. The top toolbar includes icons for File, Run, Debug, and Git. The editor has three tabs: 'pr_3.py', 'pr_4.py', and 'pr_5.py'. The 'pr_3.py' tab is active and displays the following code:

```
Thread.__init__(self)
self.limit_ = limit

def run(self):
    for i in range(self.limit_):
        print(f"from CustomThread: {i}")
        sleep(0.5)

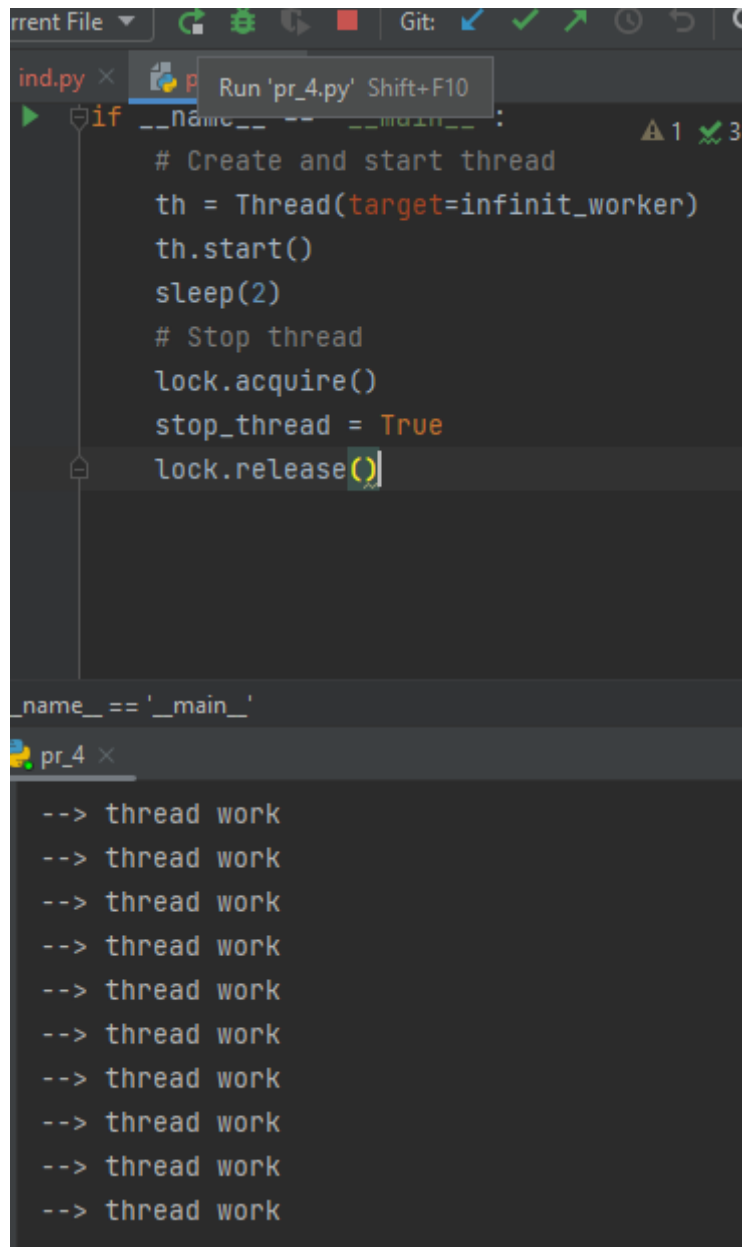
if __name__ == '__main__':
    cth = CustomThread(3)
    cth.start()
```

Below the editor, the output console shows the execution results:

```
C:\Users\A\venv\Scripts\python.exe C:\Users\A\venv\Scripts\python.exe
from CustomThread: 0
from CustomThread: 1
from CustomThread: 2

Process finished with exit code 0
```

Рисунок – Создание классов наследников от Thread

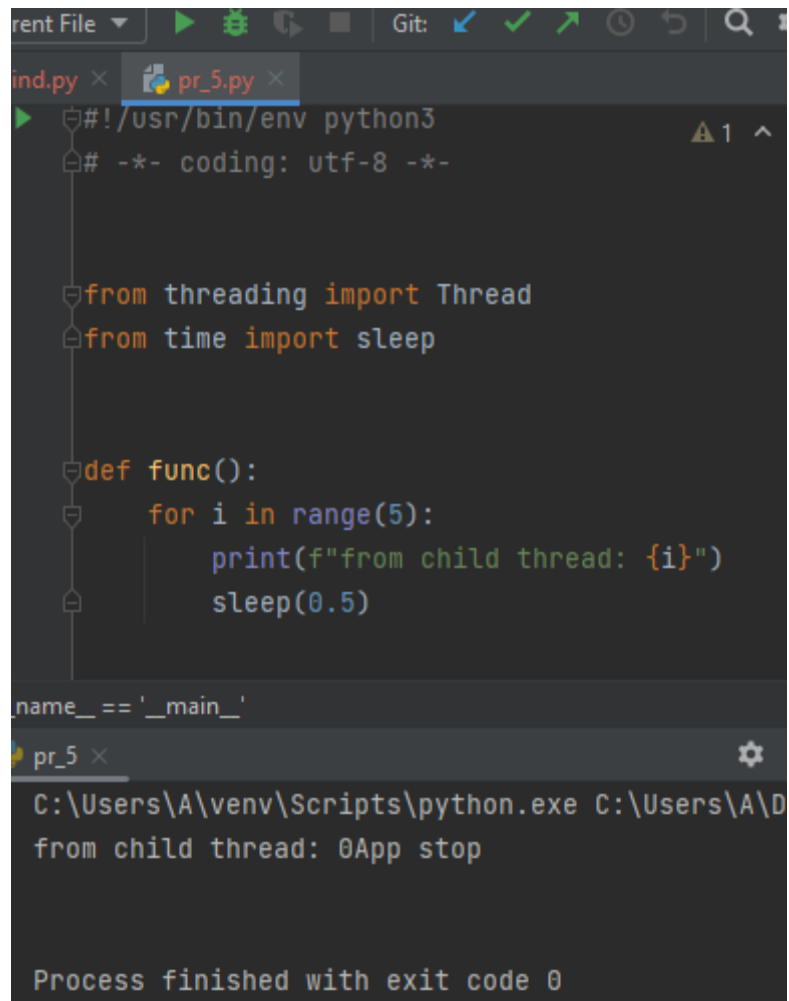


The screenshot shows a Python IDE with a file named `pr_4.py`. The code defines a function `infinite_worker` (partially visible) and a `main` function. In `main`, a thread `th` is created with `target=infinite_worker` and started. After a 2-second sleep, a lock is acquired, `stop_thread` is set to `True`, and the lock is released. The output window shows the thread printing `--> thread work` multiple times before terminating.

```
if __name__ == '__main__':  
    # Create and start thread  
    th = Thread(target=infinite_worker)  
    th.start()  
    sleep(2)  
    # Stop thread  
    lock.acquire()  
    stop_thread = True  
    lock.release()  
  
if __name__ == '__main__':  
    # ...  
    lock.release()  
  
if __name__ == '__main__':  
    # ...  
    lock.release()
```

```
--> thread work  
--> thread work  
--> thread work  
--> thread work  
--> thread work  
--> thread work  
--> thread work  
--> thread work  
--> thread work  
--> thread work  
--> thread work
```

Рисунок – Принудительное завершение работы потока



The screenshot shows a code editor with a file named `pr_5.py`. The code is a Python script that uses threading to create five child threads. Each thread prints a message and sleeps for 0.5 seconds. The terminal output shows the execution of the script, with the command `C:\Users\A\venv\Scripts\python.exe C:\Users\A\pr_5.py` and the output `from child thread: 0App stop`. The process finished with exit code 0.

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

from threading import Thread
from time import sleep

def func():
    for i in range(5):
        print(f"from child thread: {i}")
        sleep(0.5)

if __name__ == '__main__':
    t = Thread(target=func)
    t.start()
    t.join()

pr_5 ×
```

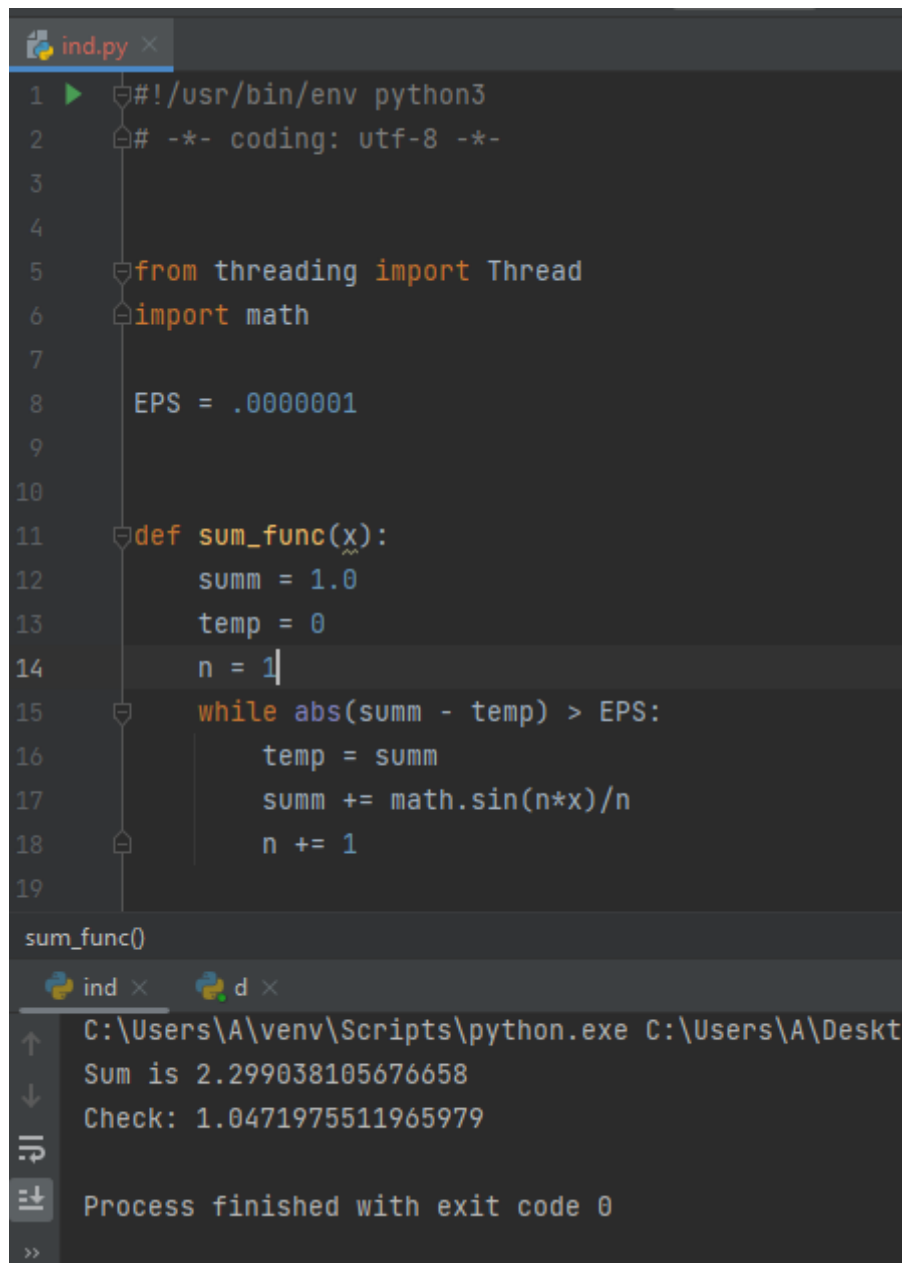
C:\Users\A\venv\Scripts\python.exe C:\Users\A\pr_5.py
from child thread: 0App stop
Process finished with exit code 0

Рисунок – Поток-демоны

Индивидуальное задание

С использованием многопоточности для заданного значения найти сумму ряда S с точностью члена ряда по абсолютному значению $\varepsilon = 10^{-7}$ и произвести сравнение полученной суммы с контрольным значением функции y для двух бесконечных рядов. Вариант 21

$$S = \sum_{n=1}^{\infty} \frac{\sin nx}{n} = \sin x + \frac{\sin 2x}{2} + \dots; \quad x = \frac{\pi}{3}; \quad y = \frac{\pi - x}{2}.$$



```
1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3
4
5  from threading import Thread
6  import math
7
8  EPS = .0000001
9
10
11 def sum_func(x):
12     summ = 1.0
13     temp = 0
14     n = 1
15     while abs(summ - temp) > EPS:
16         temp = summ
17         summ += math.sin(n*x)/n
18         n += 1
19
20 sum_func()
```

ind x d x

C:\Users\A\venv\Scripts\python.exe C:\Users\A\Desktop

Sum is 2.299038105676658

Check: 1.0471975511965979

Process finished with exit code 0

Рисунок – Индивидуальное задание

ВОПРОСЫ

1. Что такое синхронность и асинхронность?

Синхронное выполнение программы подразумевает последовательное выполнение операций. Асинхронное – предполагает возможность независимого выполнения задач.

2. Что такое параллелизм и конкурентность?

Конкурентность предполагает выполнение нескольких задач одним

исполнителем. Параллельность предполагает параллельное выполнение задач разными исполнителями.

3. Что такое GIL? Какое ограничение накладывает GIL?

GIL — это аббревиатура от Global Interpreter Lock – глобальная блокировка интерпретатора. Он является элементом эталонной реализации языка Python, которая носит название CPython. Суть GIL заключается в том, что выполнять байт код может только один поток. Это нужно для того, чтобы упростить работу с памятью (на уровне интерпретатора) и сделать комфортной разработку модулей на языке C. Это приводит к некоторым особенностям, о которых необходимо помнить. Условно, все задачи можно разделить на две большие группы: в первую входят те, что преимущественно используют процессор для своего выполнения, например, математические, их ещё называют CPU-bound, во вторую – задачи работающие с вводом выводом (диск, сеть и т.п.), такие задачи называют IO-bound. Если вы запустили в одном интерпретаторе несколько потоков, которые в основном используют процессор, то скорее всего получите общее замедление работы, а не прирост производительности. Пока выполняется одна задача, остальные простаивают (из-за GIL), переключение происходит через определенные промежутки времени. Таким образом, в каждый конкретный момент времени, будет выполняться только один поток несмотря на то, что у вас может быть многоядерный процессор (или многопроцессорный сервер), плюс ко всему, будет тратиться время на переключение между задачами. Если код в потоках в основном выполняет операции ввода-вывода, то в этом случае ситуация будет в вашу пользу. В CPython все стандартные библиотечные функций, которые выполняют блокирующий ввод-вывод, освобождают GIL, это дает возможность поработать другим потокам, пока ожидается ответ от ОС.

4. Каково назначение класса Thread?

За создание, управление и мониторинг потоков отвечает класс Thread

из модуля `threading`. Поток можно создать на базе функции, либо реализовать свой класс – наследник `Thread` и переопределить в нем метод `run()`.

5. Как реализовать в одном потоке ожидание завершения другого потока?

Если необходимо дождаться завершения работы потока(-ов) перед тем как начать выполнять какую-то другую работу, то воспользуйтесь методом `join()`. У `join()` есть параметр `timeout`, через который задается время ожидания завершения работы потоков.

6. Как проверить факт выполнения потоком некоторой работы?

Для того, чтобы определить выполняет ли поток какую-то работу или завершился используется метод `is_alive()`.

7. Как реализовать приостановку выполнения потока на некоторый промежуток времени?

С помощью метода `sleep()` из модуля `time`.

8. Как реализовать принудительное завершение потока?

В Python у объектов класса `Thread` нет методов для принудительного завершения работы потока. Один из вариантов решения этой задачи – это создать специальный флаг, через который потоку будет передаваться сигнал остановки. Доступ к такому флагу должен управляться объектом синхронизации.

```
lock.acquire()
```

```
if stop_thread is True:
```

```
break
```

```
lock.release()
```

9. Что такое потоки-демоны? Как создать поток-демон?

Для того, чтобы потоки не мешали остановке приложения (т.е. чтобы

они останавливались вместе с завершением работы программы) необходимо при создании объекта Thread аргументу daemon присвоить значение True, либо после создания потока, перед его запуском присвоить свойству daemon значение True.

```
th = Thread(target=func, daemon=True)
```