File Handling

- **File handling in Python** is a powerful and versatile tool that can be used to perform a wide range of operations. However, it is important to carefully consider the advantages and disadvantages of file handling when writing Python programs, to ensure that the code is secure, reliable, and performs well.

- In this topic, we will explore
- ***Python File Handling,***
- ***Advantages, Disadvantages and How open,***
- ***write and append functions works in python file.***

Python File Handling

- Python supports file handling and allows users to handle files i.e.,

to read and write files, along with many other file handling options, to operate on files.

- The concept of file handling has stretched over various other languages, but the implementation is either complicated or lengthy, like other concepts of Python, this concept here is also easy and short.

Python File Handling

- Python treats files differently as text or binary and this is important.
- Each line of code includes a sequence of characters, and they form a text file.
- Each line of a file is terminated with a special character, called the **EOL** or **End of Line** characters like **comma {,}** or **newline character.**
- It ends the current line and tells the interpreter a new one has begun. Let's start with the reading and writing files.

Advantages of File Handling

- **Versatility**:
- File handling in Python allows you to perform a wide range of operations, such as creating, reading, writing, appending, renaming, and deleting files.

- **Flexibility**:
- File handling in Python is highly flexible, as it allows you to work with different file types (e.g. text files, binary files, CSV files, etc.), and to perform different operations on files (e.g. read, write, append, etc.).

Advantages of File Handling

- **User–friendly**:
- Python provides a user-friendly interface for file handling, making it easy to create, read, and manipulate files.

- **Cross-platform**:
- Python file-handling functions work across different platforms (e.g. Windows, Mac, Linux), allowing for seamless integration and compatibility.

Disadvantages of File Handling

- **Error-prone:**
- File handling operations in Python can be prone to errors, especially if the code is not carefully written or if there are issues with the file system (e.g. file permissions, file locks, etc.).

- **Security risks**:
- File handling in Python can also pose security risks, especially if the program accepts user input that can be used to access or modify sensitive files on the system.

Disadvantages of File Handling

- **Complexity**:
- File handling in Python can be complex, especially when working with more advanced file formats or operations. Careful attention must be paid to the code to ensure that files are handled properly and securely.

- **Performance**:
- File handling operations in Python can be slower than other programming languages, especially when dealing with large files or performing complex operations.

Python File Open

Before performing any operation on the file like reading or writing, first, we have to open that file. For this, we should use Python's inbuilt function open() but at the time of opening, we have to specify the mode, which represents the purpose of the opening file.

Let's use a training.txt file which contains some random data.

f = open(filename, mode)

Python File Open

- Where the following mode is supported:

- **r:** open an existing file for a read operation.
- **w:** open an existing file for a write operation. If the file already contains some data, then it will be overridden but if the file is not present then it creates the file as well.
- **a:** open an existing file for append operation. It won't override existing data.
- **r+:** To read and write data into the file. The previous data in the file will be overridden.
- **w+:** To write and read data. It will override existing data.
- **a+:** To append and read data from the file. It won't override existing data.

Working in Read Mode

- There is more than one way to How to read from a file in Python. Let us see how we can read the content of a file in read mode.

- **Example 1:** The open command will open the Python file in the read mode and the for loop will print each line present in the file.

```
1  # a file named "training", will be opened with the reading mode.
2  file = open('training.txt', 'r')
3
4  # This will print every line one by one in the file
5  for each in file:
6      print (each)
7
```

Working in Read Mode

Example 2: In this example, we will extract a string that contains all characters in the Python file then we can use **file.read()**.

```
1  # Python code to illustrate read() mode
2  file = open("training.txt", "r")
3  print (file.read())
4
```

Working in Read Mode

Example 3: In this example, we will see how we can read a file using the **with** statement in Python.

```
1  # Python code to illustrate with()
2  with open("training.txt") as file:
3      data = file.read()
4
5  print(data)
6
```

Working in Read Mode

Example 4: Another way to read a file is to call a certain number of characters like in the following code the interpreter will read the first five characters of stored data and return it as a string:

```
1  # Python code to illustrate read() mode character wise
2  file = open("training.txt", "r")
3  print (file.read(5))
4
```

Working in Read Mode

Example 5: We can also split lines while reading files in Python. The split() function splits the variable when space is encountered.

You can also split using any characters as you wish.

```
# Python code to illustrate split() function
with open("training.txt", "r") as file:
    data = file.readlines()
    for line in data:
        word = line.split()
        print (word)
```

Creating a File using write() function

Just like reading a file in Python, there are a number of ways to Writing to file in Python.

Let us see how we can write the content of a file using the write() function in Python.

Working in Write Mode

- Let’s see how to create a file and how the write mode works.
- **Example 1:** In this example, we will see how the write mode and the write() function is used to write in a file. The close() command terminates all the resources in use and frees the system of this particular program.

```
1  # Python code to create a file
2  file = open('training.txt','w')
3  file.write("This is the write command")
4  file.write("It allows us to write in a particular file")
5  file.close()
6
```

Working in Write Mode

Example 2: We can also use the written statement along with the with() function.

```
1  # Python code to illustrate with() alongwith write()
2  with open("file.txt", "w") as f:
3      f.write("Hello World!!!")
4
```

Working of Append Mode

- Let us see how the append mode works.
- **Example:** For this example, we will use the Python file created in the previous example.

```
1  # Python code to illustrate append() mode
2  file = open('training.txt', 'a')
3  file.write("This will add this line")
4  file.close()
```

Working in Append Mode

There are also various other commands in Python file handling that are used to handle various tasks:

rstrip(): This function strips each line of a file off spaces from the right-hand side.

lstrip(): This function strips each line of a file off spaces from the left-hand side.

It is designed to provide much cleaner syntax and exception handling when you are working with code. That explains why it’s good practice to use them with a statement where applicable. This is helpful because using this method any files opened will be closed automatically after one is done, so auto-cleanup.

Implementing all the functions in File Handling

In this example, we will cover all the concepts that we have seen above. Other than those, we will also see how we can delete a file using t

```python
import os

def create_file(filename):
    try:
        with open(filename, 'w') as f:
            f.write('Hello, world!\n')
        print("File " + filename + " created successfully.")
    except IOError:
        print("Error: could not create file " + filename)

def read_file(filename):
    try:
        with open(filename, 'r') as f:
            contents = f.read()
            print(contents)
    except IOError:
```

Implementing all the functions in File Handling

```
19 def append_file(filename, text):
20     try:
21         with open(filename, 'a') as f:
22             f.write(text)
23         print("Text appended to file " + filename + " successfully.")
24     except IOError:
25         print("Error: could not append to file " + filename)
26 
27 def rename_file(filename, new_filename):
28     try:
29         os.rename(filename, new_filename)
30         print("File " + filename + " renamed to " + new_filename + "
successfully.")
31     except IOError:
32         print("Error: could not rename file " + filename)
33 
34 def delete_file(filename):
35     try:
36         os.remove(filename)
37         print("File " + filename + " deleted successfully.")
38     except IOError:
39         print("Error: could not delete file " + filename)
```

Implementing all the functions in File Handling

```
40
41
42  if __name__ == '__main__':
43      filename = "example.txt"
44      new_filename = "new_example.txt"
45
46      create_file(filename)
47      read_file(filename)
48      append_file(filename, "This is some additional text.\n")
49      read_file(filename)
50      rename_file(filename, new_filename)
51      read_file(new_filename)
52      delete_file(new_filename)
53
```

Implementing all the functions in File Handling

Output:

```
File example.txt created successfully.
Hello, world!
Text appended to file example.txt successfully.
Hello, world!
This is some additional text.
File example.txt renamed to new_example.txt successfully.
Hello, world!
This is some additional text.
File new_example.txt deleted successfully.
```

Module 11 –
Multithreading in Python

Multithreading in Python

- This module covers the basics of multithreading in Python programming language.
- Just like multiprocessing, multithreading is a way of achieving multitasking. In multithreading, the concept of **threads** is used.
- Let us first understand the concept of **thread** in computer architecture.

What is a Process in Python ?

- In computing, a **process** is an instance of a computer program that is being executed. Any process has 3 basic components:

- An executable program.
- The associated data needed by the program (variables, workspace, buffers, etc.)
- The execution context of the program (State of the process)

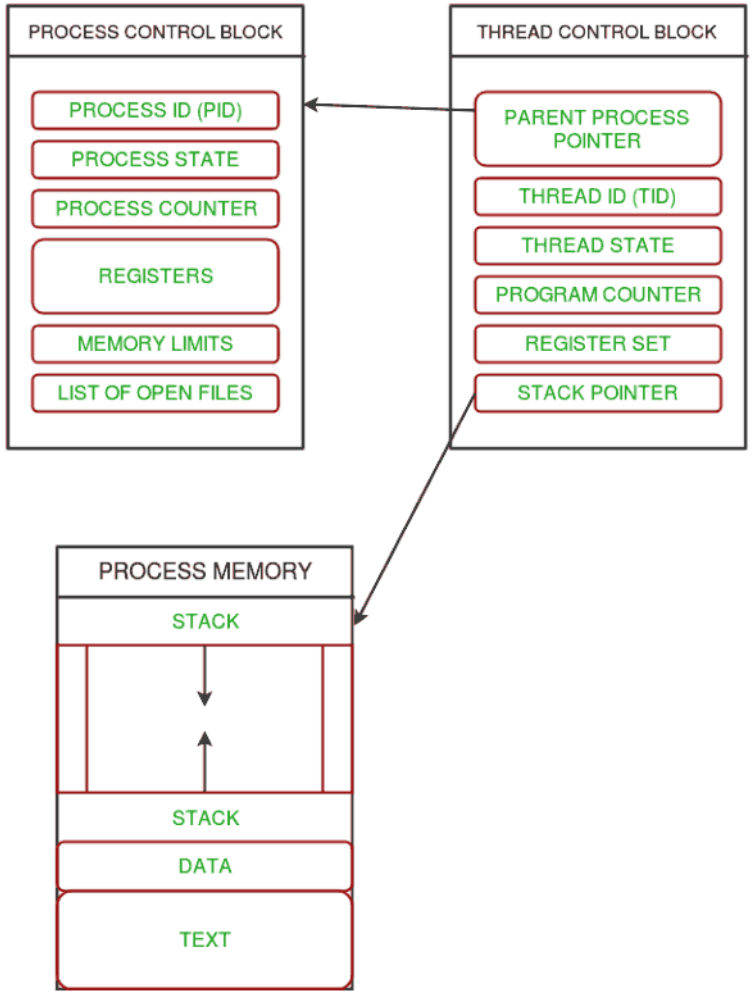
Intro to Python Threading

- A **thread** is an entity within a process that can be scheduled for execution.
- Also, it is the smallest unit of processing that can be performed in an OS (Operating System).
- In simple words, a thread is a sequence of such instructions within a program that can be executed independently of other code.
- For simplicity, you can assume that a thread is simply a subset of a process!

Intro to Python Threading

- A thread contains all this information in a **Thread Control Block (TCB)**:
- **Thread Identifier:** Unique id (TID) is assigned to every new thread
- **Stack pointer:** Points to the thread's stack in the process. The stack contains the local variables under the thread's scope.
- **Program counter:** a register that stores the address of the instruction currently being executed by a thread.
- **Thread state:** can be running, ready, waiting, starting, or done.
- **Thread's register set:** registers assigned to thread for computations.
- **Parent process Pointer:** A pointer to the Process control block (PCB) of the process that the thread lives on.

Python Threading - Diagram



Python Threading

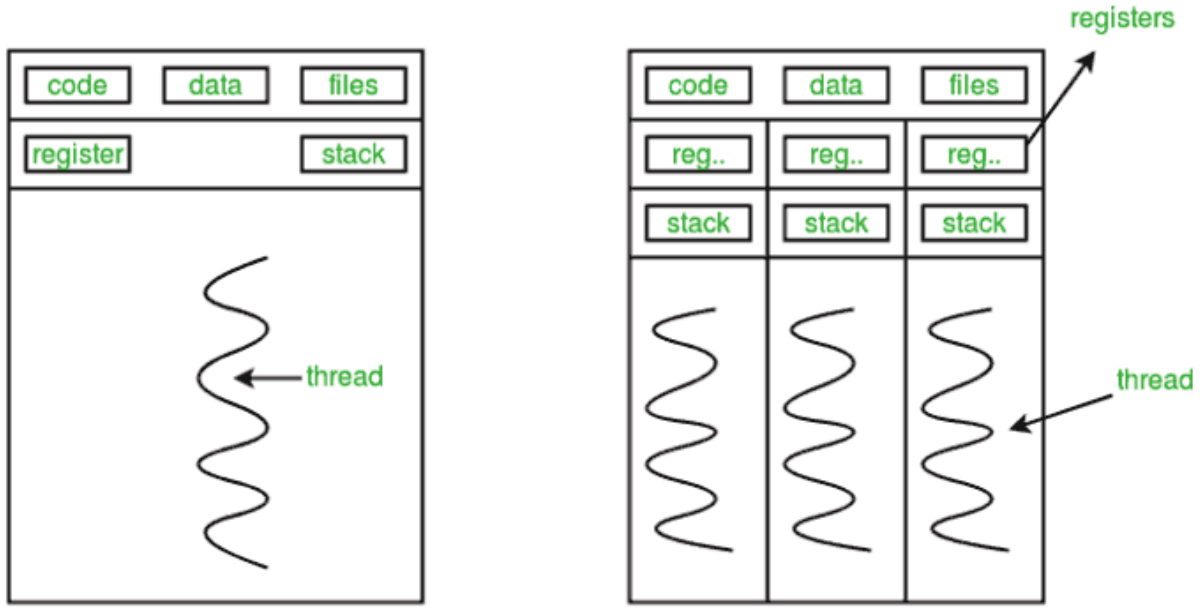
- Multiple threads can exist within one process where:

➢ Each thread contains its own **register set** and **local variables (stored in the stack)**.

➢ All threads of a process share **global variables (stored in heap)** and the **program code**.

Python Threading

Consider the diagram below to understand how multiple threads exist in memory:



single-threaded process

multithreaded process

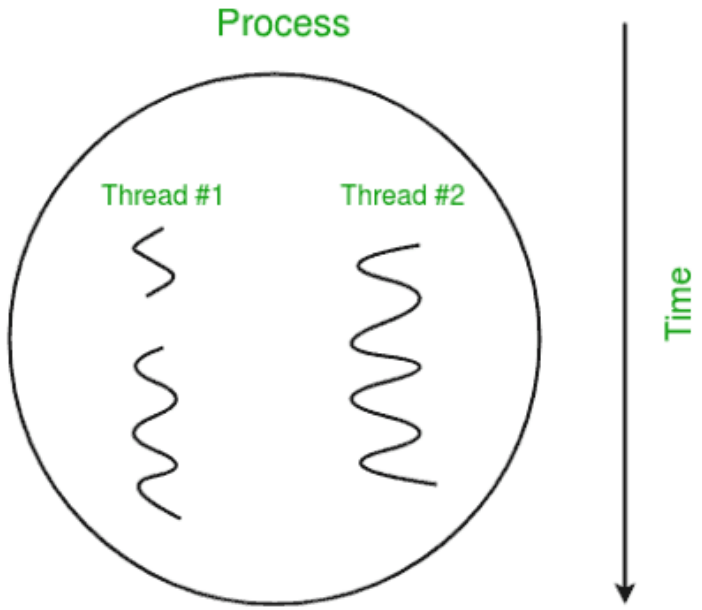
Multithreading in Python

Multithreading is defined as the ability of a processor to execute multiple threads concurrently. In a simple, single-core CPU, it is achieved using frequent switching between threads. This is termed **context switching**.

In context switching, the state of a thread is saved and the state of another thread is loaded whenever any interrupt (due to I/O or manually set) takes place. Context switching takes place so frequently that all the threads appear to be running parallelly (this is termed **multitasking**).

Multithreading in Python

Consider the diagram below in which a process contains two active threads:



Multithreading in Python

- In Python, the **threading** module provides a very simple and intuitive API for spawning multiple threads in a program. Let us try to understand multithreading code step-by-step.

- **Step 1:** Import Module
-
```
import threading
```

Multithreading in Python

- **Step 2:** Create a Thread
- To create a new thread, we create an object of the **Thread** class. It takes the 'target' and 'args' as the parameters. The **target** is the function to be executed by the thread whereas the **args is** the arguments to be passed to the target function

```
t1 = threading.Thread(target, args)
t2 = threading.Thread(target, args)
```

- **Step 3:** Start a Thread
- To start a thread, we use the **start()** method of the Thread class.

Multithreading in Python

```
t1.start()
t2.start()
```

- **Step 4:** End the thread Execution
- Once the threads start, the current program (you can think of it like a main thread) also keeps on executing. In order to stop the execution of the current program until a thread is complete, we use the

```
t1.join()
t2.join()
```

Multithreading in Python

As a result, the current program will first wait for the completion of **t1** and then **t2**. Once, they are finished, the remaining statements of the current program are executed.

Example:

Let us consider a simple example using a threading module.

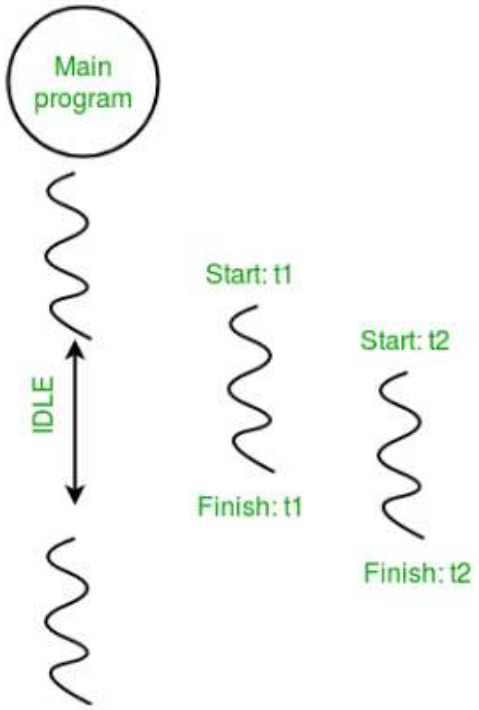
This code demonstrates how to use Python's threading module to calculate the square and cube of a number concurrently. Two threads, **t1** and **t2**, are created to perform these calculations. They are started, and their results are printed in parallel before the program prints "Done!" when both threads have finished. Threading is used to achieve parallelism and improve program performance when dealing with computationally intensive tasks.

Multithreading in Python

```
import threading


def print_cube(num):
    print("Cube: {}" .format(num * num * num))


def print_square(num):
    print("Square: {}" .format(num * num))


if __name__ =="__main__":
    t1 = threading.Thread(target=print_square, args=(10,))
    t2 = threading.Thread(target=print_cube, args=(10,))

    t1.start()
    t2.start()

    t1.join()
    t2.join()

    print("Done!")
```

Multithreading in Python

Consider the diagram below for a better understanding of how the above program



Multithreading in Python

- **Example:**

- In this example, we use **os.getpid()** function to get the ID of the current process.
- We use **threading.main_thread()** function to get the main thread object.
- In normal conditions, the main thread is the thread from which the Python interpreter was started. **name** attribute of the thread object is used to get the name of the thread.
- Then we use the **threading.current_thread()** function to get the current thread object.

Multithreading in Python

Consider the Python program given below in which we print the thread name and corresponding process for each task.

This code demonstrates how to use Python’s threading module to run two tasks concurrently. The main program initiates two threads, **t1** and **t2**, each responsible for executing a specific task. The threads run in parallel, and the code provides information about the process ID and thread names. The os module is used to access the process ID, and the **‘threading'** module is used to manage threads and their execution.

•

Multithreading in Python

```
import threading
import os

def task1():
    print("Task 1 assigned to thread: {}".format(threading.current_thread().name))
    print("ID of process running task 1: {}".format(os.getpid()))

def task2():
    print("Task 2 assigned to thread: {}".format(threading.current_thread().name))
    print("ID of process running task 2: {}".format(os.getpid()))

if __name__ == "__main__":

    print("ID of process running main program: {}".format(os.getpid()))

    print("Main thread name: {}".format(threading.current_thread().name))

    t1 = threading.Thread(target=task1, name='t1')
    t2 = threading.Thread(target=task2, name='t2')

    t1.start()
    t2.start()

    t1.join()
    t2.join()
```

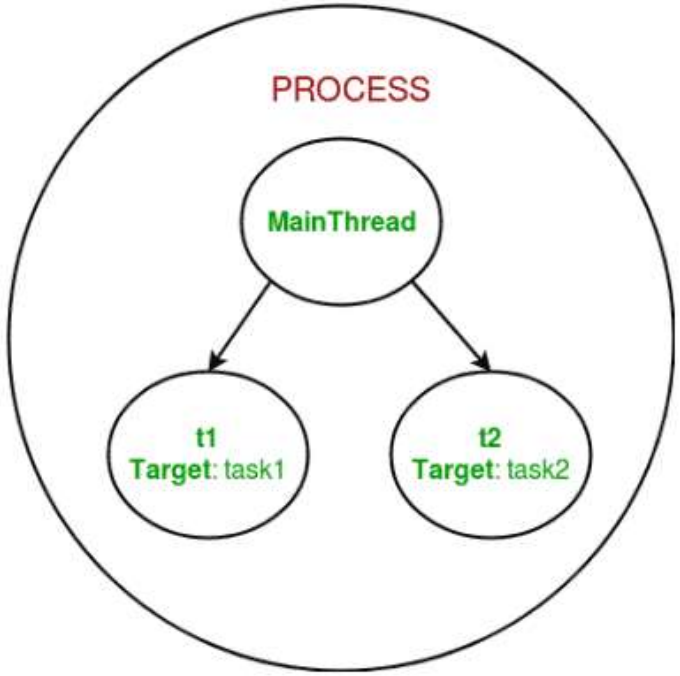
Multithreading in Python

Output:

```
ID of process running main program: 1141
Main thread name: MainThread
Task 1 assigned to thread: t1
ID of process running task 1: 1141
Task 2 assigned to thread: t2
ID of process running task 2: 1141
```

Multithreading in Python

The diagram given below clears the above concept:



Python Thread Pool

- A thread pool is a collection of threads that are created in advance and can be reused to execute multiple tasks. The concurrent.futures module in Python provides a ThreadPoolExecutor class that makes it easy to create and manage a thread pool.

- In this example, we define a function worker that will run in a thread. We create a ThreadPoolExecutor with a maximum of 2 worker threads. We then submit two tasks to the pool using the submit method. The pool manages the execution of the tasks in its worker threads. We use the shutdown method to wait for all tasks to complete before the main thread continues.

Python Thread Pool

Multithreading can help you make your programs more efficient and responsive. However, it’s important to be careful when working with threads to avoid issues such as race conditions and deadlocks.

This code uses a thread pool created with concurrent.futures.ThreadPoolExecutor to run two worker tasks concurrently. The main thread waits for the worker threads to finish using pool.shutdown(wait=True). This allows for efficient parallel processing of tasks in a multi-threaded environment.

Python Thread Pool

```python
import concurrent.futures

def worker():
    print("Worker thread running")

pool = concurrent.futures.ThreadPoolExecutor(max_workers=2)

pool.submit(worker)
pool.submit(worker)

pool.shutdown(wait=True)

print("Main thread continuing to run")
```

Threading Module

The **threading** module in Python provides a high-level interface for working with threads. It allows you to create, start, pause, resume, and terminate threads easily.

Here's a simple example demonstrating how to create and start threads using the **threading** module:

Threading Module

```python
import threading
import time

# Define a function to be executed by the thread
def print_numbers():
    for i in range(5):
        print(i)
        time.sleep(1)

# Create a thread
thread = threading.Thread(target=print_numbers)

# Start the thread
thread.start()
```

```python
# Start the thread
thread.start()

# Main thread continues execution while the new thread runs concurrently
for letter in 'ABCDE':
    print(letter)
    time.sleep(1)

# Wait for the thread to finish
thread.join()

print("Thread execution completed.")
```

Threading Module

- In this example:

➢ We define a function **print_numbers()** that simply prints numbers from 0 to 4 with a one-second delay between each print.

➢ We create a thread using **threading.Thread()** constructor and passing our function **print_numbers** as the **target**.

➢ We start the thread using the **start()** method.

➢ While the thread is running concurrently, the main thread continues executing its own logic.

➢ We use **thread.join()** to wait for the thread to finish execution before the program exits.

-