File Handling in Python

Python too 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, but like other concepts of Python, this concept here is also easy and short. 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.

Working of open() function

f = open(filename, mode)

Where the following mode is supported:

1. **r:** open an existing file for a read operation.
2. **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.
3. **a:** open an existing file for append operation. It won't override existing data.
4. **r+:** To read and write data into the file. The previous data in the file will be overridden.

5. **w+:** To write and read data. It will override existing data.
6. **a+:** To append and read data from the file. It won’t override existing data.

Working of read() mode

Creating a file using write() mode

Python Exception Handling

- **SyntaxError:** This exception is raised **when the interpreter encounters a syntax error** in the code, such as a misspelled keyword, a missing colon, or an unbalanced parenthesis.

- **TypeError**: This exception is raised when an **operation or function is applied to an object of the wrong type**, such as adding a string to an integer.

- **NameError**: This exception is **raised when a variable or function name is not found** in the current scope.

- **IndexError**: This exception is raised **when an index is out of range for a list**, tuple, or other sequence types.

- **KeyError**: This exception is **raised when a key is not found in a dictionary**.

- **ValueError**: This exception is raised **when a function or method is called with an invalid argument or input, such as trying to convert a string to an integer** when the string does not represent a valid integer.

- **AttributeError**: This exception is raised **when an attribute or method is not found on an object**, such as trying to access a non-existent attribute of a class instance.

- **IOError**: This exception is raised when **an I/O operation, such as reading or writing a file, fails due to an input/output error**.

- **ZeroDivisionError**: This exception is raised when an attempt is made to divide a number by zero.

- **ImportError**: This exception is raised **when an import statement fails to find or load a module**

Difference between Syntax Error and Exceptions
Syntax Error: As the name suggests this error is caused by the wrong syntax in the code. **It leads to the termination of the program.**

Exceptions: Exceptions are raised **when the program is syntactically correct,** but the code results in an error. This error does not stop the execution of the program, however, it changes the normal flow of the program.

Python Try Except

Error in Python can be of two types i.e. Syntax errors and Exceptions. Errors are the problems in a program due to which the program will stop the execution. On the other hand, exceptions are raised when some internal events occur which changes the normal flow of the program

- **IOError:** if the file can't be opened
- **KeyboardInterrupt:** when an unrequired key is pressed by the user
- **ValueError:** when the built-in function receives a wrong argument
- **EOFError:** if End-Of-File is hit without reading any data

- **ImportError:** if it is unable to find the module

Try Except in Python

Try and Except statement is used to handle these errors within our code in Python. The **try block is used to check some code for errors** i.e the **code inside the try block will execute when there is no error in the program.** Whereas the **code inside the except block will execute whenever the program encounters some error in the preceding try block**

Multithreading in Python

Multithreading is a way of achieving multitasking. In multithreading, the concept of **threads** is used.

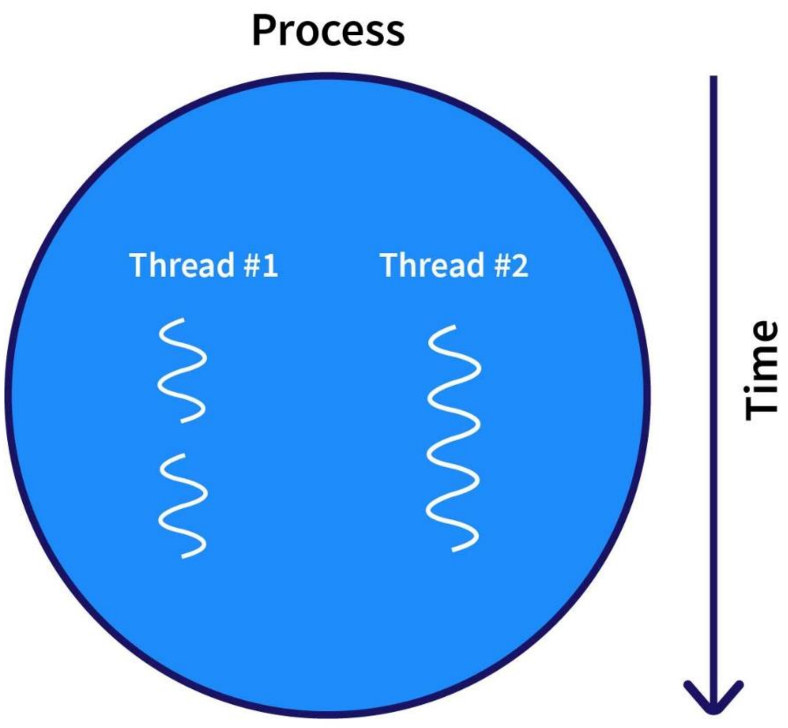
In computing, a **process** is an **instance of a computer program that is being executed**. Any process has 3 basic components:

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

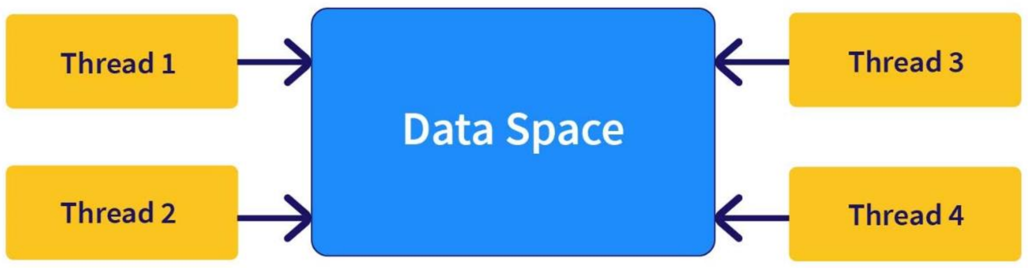
A **thread** is an **entity within a process** that **can be scheduled for execution**. Also, it is the smallest unit of processing that can be performed in an OS (Operating System). In simple words, a **thread** is a **sequence of such instructions within a program that can be executed independently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

Multithreading is the ability of a program or an operating system to enable more than one user at a time without requiring multiple copies of the program running on the computer.

Multithreading can also handle various requests from the same user. Each user request for a program or system service is tracked **as a thread with a separate identity.** As programs **work on behalf of the initial thread request** and are **interrupted by other demands,** the work status of the initial request is tracked until the work is completed



SCALER Topics



SCALER Topics

Multiprocessing in Python

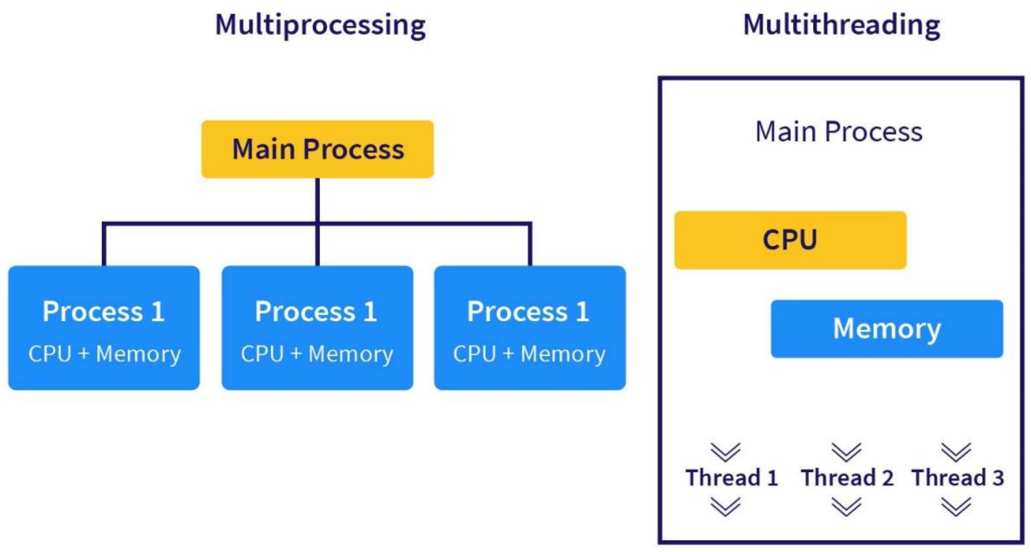
Multiprocessing refers to the ability of a system to support more than one processor at the same time. Applications in a multiprocessing system **are broken to smaller routines that run independently.** The operating system allocates these threads to the processors improving performance of the system.

The more tasks a single processor is burdened with, the more difficult it becomes for the processor to keep track of them**. It evidently gives rise to the need for multiprocessing.** Multiprocessing tries to ensure that every processor gets its own processor/processor core and that execution is hassle-free.

Python Multithreading vs Multiprocessing

The main differences to note between Multithreading and Multiprocessing are as follows –

Multithreading	**Multiprocessing**
It is a technique **where a process spawns multiple threads simultaneously**.	It is the technique where **multiple processes run across multiple processors**/processor cores simultaneously.
Python multithreading implements concurrency.	Python multiprocessing implements parallelism in its truest form.
It gives the illusion that they are running parallelly, but they work in a concurrent manner.	It is parallel in the sense that the multiprocessing module facilitates the running of independent processes parallelly by using subprocesses.
In multithreading, the GIL or Global Interpreter Lock prevents the threads from running simultaneously.	In multiprocessing, each process has its own Python Interpreter performing the execution.



SCALER Topics

Logging in Python

Logging is a means of tracking events that happen when some software runs. Logging is important for software developing, debugging, and running. If you don’t have any logging record and your program crashes, there are very few chances that you detect the cause of the problem. And if you detect the cause, it will consume a lot of time.

Why Printing is not a good option?

Some developers use the concept of printing the statements to validate if the statements are executed correctly or some error has occurred. But printing is not a good idea. It may solve your issues for simple scripts but for complex scripts, the printing approach will fail. Python has a built-in module **logging** which allows writing status messages to a file or any other output streams. The file can contain the information on which part of the code is executed and what problems have been arisen.

Levels of Log Message

- **Debug :** These are used to give Detailed information, typically of interest only when diagnosing problems.
- **Info :** These are used to confirm that things are working as expected
- **Warning :** These are used an indication that something unexpected happened, or is indicative of some problem in the near future
- **Error :** This tells that due to a more serious problem, the software has not been able to perform some function
- **Critical :** This tells serious error, indicating that the program itself may be unable to continue running

Dunder or Magic Methods in Python

Dunder or magic methods in Python are the methods having **two prefix and suffix underscores** in the method name. Dunder here means **“Double Under (Underscores)”**

These are commonly used for **operator overloading**. (**They provide extended meaning beyond the predefined meaning to an operator**). Few examples for magic methods are: __init__, __add__, __len__, __repr__ etc.

Python dunder methods can be easily understood by visualizing a contract between your implementation and the Python interpreter.

One of the **main terms of the contract involves Python performing some actions behind the scenes** under some given circumstances.

```
# declare our string class
class String:

   # magic method to initiate object
   def __init__(self, string):
      self.string = string

# Driver Code
if __name__ == '__main__':

   # object creation
   string1 = String('Hello Everyone')

   # print object location
   print(string1)
```

The code you provided **defines a String class** with an __init__() method. The __init__() method is a magic method **used to initialize an object of the class**. In this case, it takes a string argument and assigns it to the string attribute of the object.

In the **driver code section** (if __name__ == '__main__':), an instance of the String class is created with the string 'Hello Everyone' as the argument. The object is assigned to the variable string1

Single-threaded applications

Single-threaded applications

Let's start with a simple program:

```
from time import sleep, perf_counter

def task():
    print('Starting a task...')
    sleep(1)
    print('done')


start_time = perf_counter()

task()
task()

end_time = perf_counter()

print(f'It took {end_time- start_time: 0.2f} second(s) to
complete.')
```

How it works.

First, import the sleep() and perf_counter() functions from the time module:

```
from time import sleep, perf_counter
```

Second, define a function that takes one second to complete:

```
def task():
    print('Starting a task...')
    sleep(1)
    print('done')
```

Third, **get the value of the performance counter** by calling the perf_counter() function:

```
start_time = perf_counter()
```

Fourth, call the task() function twice:

```
task()
task()
```

Fifth, **get the value of the performance counter** by calling the perf_counter() function:

```
end_time = perf_counter()
```

Finally, output the time that takes to complete running the task() function twice:

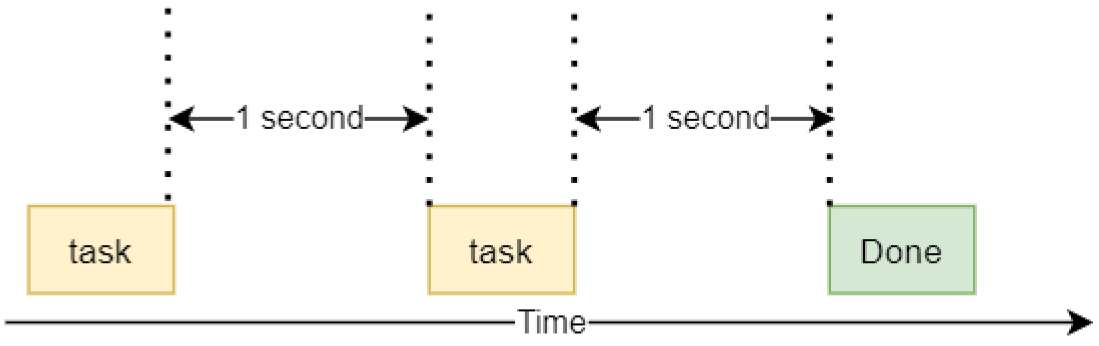
```
print(f'It took {end_time- start_time: 0.2f} second(s) to complete.')
```

Here is the output:

```
Starting a task...
done
Starting a task...
done
It took  2.00 second(s) to complete.
```

As you may expect, the program takes about two seconds to complete. If you call the task() function 10 times, it would take about 10 seconds to complete.

The following diagram illustrates how the program works:



First, **the task() function executes and sleeps for one second.** Then it executes the second time and also sleeps for another second. Finally, the program completes.

When the task() function calls the sleep() function, the CPU is idle. In other words, the CPU doesn't do anything, **which is not efficient in terms of resource utilization.**

This program has one process with a single thread, which is called the **main thread**. Because the program has only one thread, it's called a single-threaded program

Using Python threading to develop a multi-threaded program example

To create a multi-threaded program, you need to use the Python threading module.

First, import the Thread class from the threading module:

```
from threading import Thread
```

Second, **create a new thread by instantiating an instance of the Thread class**:

```
new_thread = Thread(target=fn,args=args_tuple)
```

The Thread() accepts many parameters. The main ones are:

- **target: specifies a function (fn)** to run in the new thread.
- **args: specifies the arguments of the function (fn).** The args argument is a tuple.

Third, start the thread by calling the start() method of the Thread instance:

```
new_thread.start()
```

If you want to wait for the thread to complete in the main thread, you can call the join() method:

```
new_thread.join()
```

By calling the join() method, **the main thread will wait for the child thread to complete before it is terminated.**

```
from time import sleep, perf_counter
from threading import Thread


def task():
    print('Starting a task...')
    sleep(1)
    print('done')


start_time = perf_counter()

# create two new threads
t1 = Thread(target=task)
t2 = Thread(target=task)

# start the threads
t1.start()
t2.start()

# wait for the threads to complete
t1.join()
t2.join()

end_time = perf_counter()
```

```
print(f'It took {end_time- start_time: 0.2f} second(s) to complete.')
```

How it works. (And we'll focus on the threading part only)

First, create two new threads:

```
t1 = Thread(target=task)
t2 = Thread(target=task)
```

Second, start both threads by calling the start() method:

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

Third, wait for both threads to complete:

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

Finally, show the executing time:

```
print(f'It took {end_time- start_time: 0.2f} second(s) to complete.')
```

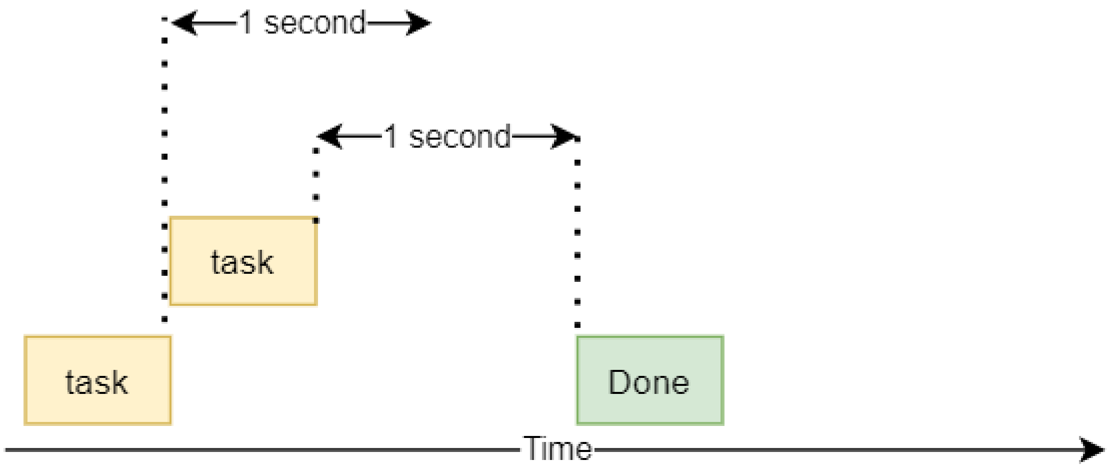
Output:

```
Starting a task...
Starting a task...
done
done
It took  1.00 second(s) to complete.
```

When the program executes, it’ll have three threads: the main thread and two other child threads.

As shown clearly from the output, the program took one second instead of two to complete.

The following diagram shows how threads execute:



Passing arguments to threads

The following program shows how to pass arguments to the function assigned to a thread:

```
from time import sleep, perf_counter
from threading import Thread


def task(id):
    print(f'Starting the task {id}...')
    sleep(1)
    print(f'The task {id} completed')
```

```
start_time = perf_counter()

# create and start 10 threads
threads = []
for n in range(1, 11):
    t = Thread(target=task, args=(n,))
    threads.append(t)
    t.start()

# wait for the threads to complete
for t in threads:
    t.join()

end_time = perf_counter()

print(f'It took {end_time- start_time: 0.2f} second(s) to complete.')
```

How it works.

First, define a task() function that **accepts an argument**:

```
def task(id):
    print(f'Starting the task {id}...')
    sleep(1)
    print(f'The task {id} completed')
```

Second, **create 10 new threads and pass an id to each**. The threads list is used to keep track of all newly created threads:

```
threads = []
for n in range(1, 11):
    t = Thread(target=task, args=(n,))
    threads.append(t)
    t.start()
```

Notice **that if you call the join() method inside the loop, the program will wait for the first thread to complete** before starting the next one.

Third, wait for all threads to complete by calling the join() method:

```
for t in threads:
    t.join()
```

The following shows the output of the program:

```
Starting the task 1...
Starting the task 2...
Starting the task 3...
Starting the task 4...
Starting the task 5...
Starting the task 6...
Starting the task 7...
Starting the task 8...
Starting the task 9...
Starting the task 10...
The task 10 completed
The task 8 completed
The task 1 completed
The task 6 completed
The task 7 completed
The task 9 completed
The task 3 completed
The task 4 completed
The task 2 completed
The task 5 completed
It took  1.02 second(s) to complete.
```

It just took 1.05 seconds to complete.

Notice that the program doesn’t execute the thread in the order from 1 to 10.

Python Multiprocessing Example

Python multiprocessing Process class

```
from multiprocessing import Process


def print_func(continent='Asia'):
    print('The name of continent is : ', continent)

if __name__ == "__main__":  # confirms that the code is under main
function

    names = ['America', 'Europe', 'Africa']
    procs = []
    proc = Process(target=print_func) # instantiating without any
argument

    procs.append(proc)
    proc.start()

    # instantiating process with arguments
    for name in names:
        # print(name)
        proc = Process(target=print_func, args=(name,))
        procs.append(proc)
        proc.start()

    # complete the processes
    for proc in procs:
        proc.join()
```

Python multiprocessing Queue class

```
from multiprocessing import Queue

colors = ['red', 'green', 'blue', 'black']
cnt = 1
# instantiating a queue object
queue = Queue()
print('pushing items to queue:')
for color in colors:
    print('item no: ', cnt, ' ', color)
    queue.put(color)
    cnt += 1

print('\npopping items from queue:')
cnt = 0
while not queue.empty():
    print('item no: ', cnt, ' ', queue.get())
    cnt += 1
```

Python multiprocessing Lock Class

```
from multiprocessing import Lock, Process, Queue, current_process
import time
import queue # imported for using queue.Empty exception


def do_job(tasks_to_accomplish, tasks_that_are_done):
    while True:
        try:
            '''
                try to get task from the queue. get_nowait() function will
                raise queue.Empty exception if the queue is empty.
                queue(False) function would do the same task also.
            '''
            task = tasks_to_accomplish.get_nowait()
        except queue.Empty:

            break
        else:
            '''
                if no exception has been raised, add the task completion
                message to task_that_are_done queue
            '''
            print(task)
            tasks_that_are_done.put(task + ' is done by ' +
current_process().name)
            time.sleep(.5)
    return True


def main():
    number_of_task = 10
    number_of_processes = 4
    tasks_to_accomplish = Queue()
    tasks_that_are_done = Queue()
    processes = []

    for i in range(number_of_task):
        tasks_to_accomplish.put("Task no " + str(i))
    # creating processes
    for w in range(number_of_processes):
        p = Process(target=do_job, args=(tasks_to_accomplish,
tasks_that_are_done))
        processes.append(p)
        p.start()

    # completing process
    for p in processes:
        p.join()

    # print the output
    while not tasks_that_are_done.empty():
        print(tasks_that_are_done.get())

    return True


if __name__ == '__main__':
    main()
```

Python multiprocessing Pool

```
from multiprocessing import Pool

import time

work = (["A", 5], ["B", 2], ["C", 1], ["D", 3])


def work_log(work_data):
    print(" Process %s waiting %s seconds" % (work_data[0],
work_data[1]))
    time.sleep(int(work_data[1]))
    print(" Process %s Finished." % work_data[0])


def pool_handler():
    p = Pool(2)
    p.map(work_log, work)


if __name__ == '__main__':
    pool_handler()
```

In Python's Multiprocessing module, there are several important components for parallel programming, including **Process, Queue, and Lock, along with the Pool class.** Here's an explanation of each:

1. **Process: The Process class represents an individual process that can be executed concurrently**. You can create **multiple instances of Process and run them in parallel to perform independent tasks**. Each Process object has its own memory space and resources.

2. **Queue: The Queue class provides a thread-safe way to share data between processes. It allows multiple processes to enqueue and dequeue items from a shared queue.** This is useful for exchanging data or messages among different processes in a synchronized manner.

3. **Lock: The Lock class is used to provide mutual exclusion or synchronization between processes**. It allows you **to protect shared resources and ensure that only one process can access the critical section at a tim**e. By acquiring and releasing a lock, **processes can coordinate access to shared resources** and avoid conflicts.

4. **Pool: The Pool class provides a convenient way to manage a pool of worker processes.** It allows you to create a fixed number of worker processes and distribute tasks among them automatically. The Pool class provides methods like **map()** and **apply_async()** to execute functions or methods in parallel across the worker processes. It **abstracts away the details of process creation and communication**, making parallel programming easier.