5. Python Multithreading and Exception Handling

5.1. Exception Handling

5.2. Avoiding code break using exception handling

5.3. Safe guarding file operation using exception handling

5.4. Handling and helping developer with error code

5.5. Programming using Exception handling

5.6. Multithreading

5.7. Understanding threads

5.8. Synchronizing the threads

5.9. Programming using multithreading

5.1. Exception Handling

5.2. Avoiding code break using exception handling

5.3. Safe guarding file operation using exception handling

5.4. Handling and helping developer with error code

5.5. Programming using Exception handling

Python Exception

An exception can be defined as **an unusual condition** in a program resulting in the interruption in the flow of the program.

Whenever an exception occurs, the program stops the execution, and thus the further code is not executed. Therefore, **an exception is the run-time errors that are unable to handle to Python script. An exception is a Python object that represents an error.**

Python provides a way to handle the exception so that the code can be executed without any interruption. **If we do not handle the exception, the interpreter doesn't execute all the code that exists after the exception.**

Python has many **built-in exceptions** that enable our program to run without interruption and give the output. These exceptions are given below:

Common Exceptions

Python provides the number of built-in exceptions, but here we are describing the common standard exceptions. A list of common exceptions that can be thrown from a standard Python program is given below.

1. **ZeroDivisionError:** Occurs when a number is divided by zero.
2. **NameError:** It occurs when a name is not found. It may be local or global.
3. **IndentationError:** If incorrect indentation is given.
4. **IOError:** It occurs when Input Output operation fails.
5. **EOFError:** It occurs when the end of the file is reached, and yet operations are being performed.

The problem without handling exceptions

As we have already discussed, the exception is an abnormal condition that halts the execution of the program.

Suppose we have two variables **a** and **b**, which take the input from the user and perform the division of these values. What if the user entered the zero as the denominator? It will interrupt the program execution and through a **ZeroDivision** exception. Let's see the following example.

Example:

```
a = int(input("Enter a: "))
b = int(input("Enter b: "))
c = a/b
print(f"a/b = {c}")

#other code:
print("Hi I am other part of the program")
```

Output:

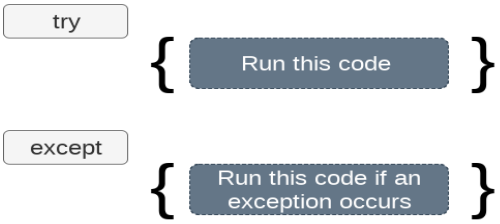
```
Enter a:10
Enter b:0
Traceback (most recent call last):
  File "exception-test.py", line 3, in <module>
    c = a/b;
ZeroDivisionError: division by zero
```

The above program is syntactically correct, but it throws the error because of unusual input. That kind of programming may not be suitable or recommended for the projects because these projects are required uninterrupted execution. That's why an exception-handling plays an essential role in handling these unexpected exceptions. We can handle these exceptions in the following way.

Exception handling in python

The try-except statement

If the Python program contains suspicious code that may throw the exception, we must place that code in the **try** block. The **try** block must be followed with the **except** statement, which contains a block of code that will be executed if there is some exception in the try block.



Syntax:

```
try:
   #block of code

except Exception1:
   #block of code

except Exception2:
   #block of code

#other code
```

Consider the following example.

Example:

```
try:
   a = int(input("Enter a: "))
   b = int(input("Enter b: "))
   c = a/b
   print(f"a/b = {c}")
except:
   print("Can't divide with zero")
```

Output:

Enter a:10

Enter b:0

Can't divide with zero

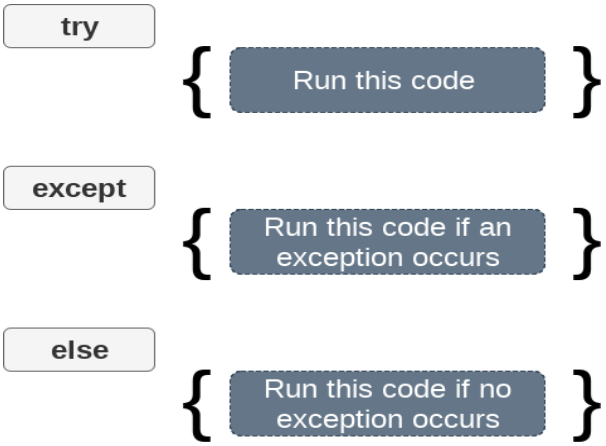
We can also use the else statement with the try-except statement in which, we can place the code which will be executed in the scenario if no exception occurs in the try block.

The syntax to use the else statement with the try-except statement is given below.

```
try:
    #block of code

except Exception1:
    #block of code

else:
    #this code executes if no except block is executed
```



Consider the following program.

Example:

```
try:
    a = int(input("Enter a: "))
    b = int(input("Enter b: "))
    c = a/b
    print(f"a/b = {c}")
```

```
# Using Exception with except statement. If we print(Exception) it will return exception class
except Exception:
    print("can't divide by zero")
    print(Exception)
else:
    print("Hi I am else block")
```

Output:

```
Enter a:10
Enter b:0
can't divide by zero
<class 'Exception'>
```

The except statement with no exception

Python provides the flexibility not to specify the name of exception with the exception statement. Consider the following example.

Example:

```
try:
    a = int(input("Enter a: "))
    b = int(input("Enter b: "))
    c = a/b;
    print(f"a/b = {c}")
except:
    print("can't divide by zero")
else:
    print("Hi I am in else block")
```

The except statement using with exception variable

We can use the exception variable with the **except** statement. It is used by using the **as** keyword. this object will return the cause of the exception. Consider the following example:

```
try:
```

```
    a = int(input("Enter a:"))
    b = int(input("Enter b:"))
    c = a/b
    print(f"a/b = {c}")
    # Using exception object with the except statement
except Exception as e:
    print("can't divide by zero")
    print(e)
else:
    print("Hi I am else block")
```

Output:

Enter a:10

Enter b: 0

can't divide by zero

division by zero

Points to remember

1. Python facilitates us not to specify the exception with the except statement.
2. We can declare multiple exceptions in the except statement since the try block may contain the statements which throw the different type of exceptions.
3. We can also specify an else block along with the try-except statement, which will be executed if no exception is raised in the try block.
4. The statements that don't throw the exception should be placed inside the else block.

Example

```
try:
    #this will throw an exception if the file doesn't exist.
    fileptr = open("file.txt","r")
except IOError:
    print("File not found")
else:
```

```
    print("The file opened successfully")
    fileptr.close()
```

Output:

File not found

Declaring Multiple Exceptions

The Python allows us to declare the multiple exceptions with the except clause. Declaring multiple exceptions is useful in the cases where **a try block throws multiple exceptions**. The syntax is given below.

Syntax

```
try:
    #block of code

except (<Exception 1>,<Exception 2>,<Exception 3>,...<Exception n>)
    #block of code

else:
    #block of code
```

Consider the following example.

```
try:
    a=10/0;
except(ArithmeticError, IOError):
    print("Arithmetic Exception")
else:
    print("Successfully Done")
```

Output:

Arithmetic Exception

Example: Multiple except Blocks

```
try:
    a = 5
    b = 0
    print (a/b)
except TypeError:
    print('Unsupported operation')
except ZeroDivisionError:
    print ('Division by zero not allowed')
print ('Out of try except blocks')
```

Output:

Division by zero not allowed

Out of try except blocks

The try...finally block

Python provides the optional **finally** statement, which is used with the **try** statement. It is executed no matter what exception occurs and used to release the external resource. The finally block provides a guarantee of the execution.

We can use the finally block with the try block in which we can pace the necessary code, which must be executed before the try statement throws an exception.

The syntax to use the finally block is given below.

Syntax

```
try:
    # block of code
    # this may throw an exception
finally:
    # block of code
    # this will always be executed
```

Example:

```
try:
    fileptr = open("file2.txt","r")
    try:
        fileptr.write("Hi I am good")
    finally:
        fileptr.close()
        print("file closed")
except:
    print("Error")
```

Output:

file closed

Error

else and finally

In Python, keywords **else** and **finally** can also be used along with the try and except clauses. While the except block is executed if the exception occurs inside the try block, the else block gets processed if the try block is found to be exception free.

Syntax:

```
try:
    #statements in try block
except:
    #executed when error in try block
else:
    #executed if try block is error-free
finally:
    #executed irrespective of exception occurred or not
```

The finally block consists of statements which should be processed regardless of an exception occurring in the try block or not. As a consequence, the error-free try block skips the except clause and enters the finally block before going on to execute the rest of the code. If, however,

there's an exception in the try block, the appropriate except block will be processed, and the statements in the finally block will be processed before proceeding to the rest of the code.

The example below accepts two numbers from the user and performs their division. It demonstrates the uses of else and finally blocks.

Example: try, except, else, finally blocks

```
try:
    print('try block')
    x=int(input('Enter a number: '))
    y=int(input('Enter another number: '))
    z=x/y
except ZeroDivisionError:
    print("except ZeroDivisionError block")
    print("Division by 0 not accepted")
else:
    print("else block")
    print("Division = ", z)
finally:
    print("finally block")
    x = 0
    y = 0
print ("Out of try, except, else and finally blocks." )
```

The first run is a normal case. The out of the else and finally blocks is displayed because the try block is error-free.

Output:

```
try block
Enter a number: 10
Enter another number: 2
```

else block
Division = 5.0
finally block
Out of try, except, else and finally blocks.

The second run is a case of division by zero, hence, the except block and the finally block are executed, but the else block is not executed.

Output:

try block
Enter a number: 10
Enter another number: 0
except ZeroDivisionError block
Division by 0 not accepted
finally block
Out of try, except, else and finally blocks.

In the third run case, an uncaught exception occurs. The finally block is still executed but the program terminates and does not execute the program after the finally block.

Output:

try block
Enter a number: 10
Enter another number: xyz
finally block
Traceback (most recent call last):
  File "C:\python36\codes\test.py", line 3, in <module>
    y=int(input('Enter another number: '))
ValueError: invalid literal for int() with base 10: 'xyz'

Typically the finally clause is the ideal place for cleaning up the operations in a process. For example closing a file irrespective of the errors in read/write operations.

Raising exceptions

An exception can be raised forcefully by using the **raise** clause in Python. It is useful in in that scenario where we need to raise an exception to stop the execution of the program.

For example, there is a program that requires 2GB memory for execution, and if the program tries to occupy 2GB of memory, then we can raise an exception to stop the execution of the program.

The syntax to use the raise statement is given below.

Syntax

raise Exception_class,<value>

Points to remember

1. To raise an exception, the raise statement is used. The exception class name follows it.
2. An exception can be provided with a value that can be given in the parenthesis.
3. To access the value "**as**" keyword is used. "**e**" is used as a reference variable which stores the value of the exception.
4. We can pass the value to an exception to specify the exception type.

Example:

```
try:
    age = int(input("Enter the age:"))
    if(age < 18):
        raise ValueError
    else:
        print("the age is valid")
except ValueError:
    print("The age is not valid")
```

Output:

Enter the age:17

The age is not valid

Example: Catch Specific Error Type

```
try:
    a=5
    b='0'
    print(a + b)
except TypeError:
    print('Unsupported operation')
print ("Out of try except blocks")
```

Output:

Unsupported operation

Out of try except blocks

Example: Raise the exception with message

```
try:
    num = int(input("Enter a positive integer: "))
    if(num <= 0):
# we can pass the message in the raise statement
        raise ValueError("That is  a negative number!")
except ValueError as e:
    print(e)
```

Output:

Enter a positive integer: -5

That is a negative number!

Example:

```
try:
    a = int(input("Enter a:"))
    b = int(input("Enter b:"))
    if b is 0:
```

```
        raise ArithmeticError
    else:
        print("a/b = {a/b}")
except ArithmeticError:
    print("The value of b can't be 0")
```

Output:

Enter a:10

Enter b:0

The value of b can't be 0

Example: Raise an Exception

```
try:

    x=int(input('Enter a number upto 100: '))
    if x > 100:
        raise ValueError(x)
except ValueError:
    print(f"{x} is out of allowed range")
else:
    print(f"{x} is within the allowed range")
```

Output:

Enter a number upto 100: 200

200 is out of allowed range

Enter a number upto 100: 50

50 is within the allowed range

Here, the raised exception is a `ValueError` type. **However, you can define your custom exception type to be raised.**

Custom Exception

The Python allows us to create our exceptions that can be raised from the program and caught using the except clause. However, we suggest you read this section after visiting the Python object and classes.

Consider the following example.

Example

```
class ErrorInCode(Exception):
    def __init__(self, data):
        self.data = data
    def __str__(self):
        return repr(self.data)

try:
    raise ErrorInCode(2000)
except ErrorInCode as ae:
    print("Received error:", ae.data)
```

Output:

```
Received error: 2000
```

5.6. Multithreading

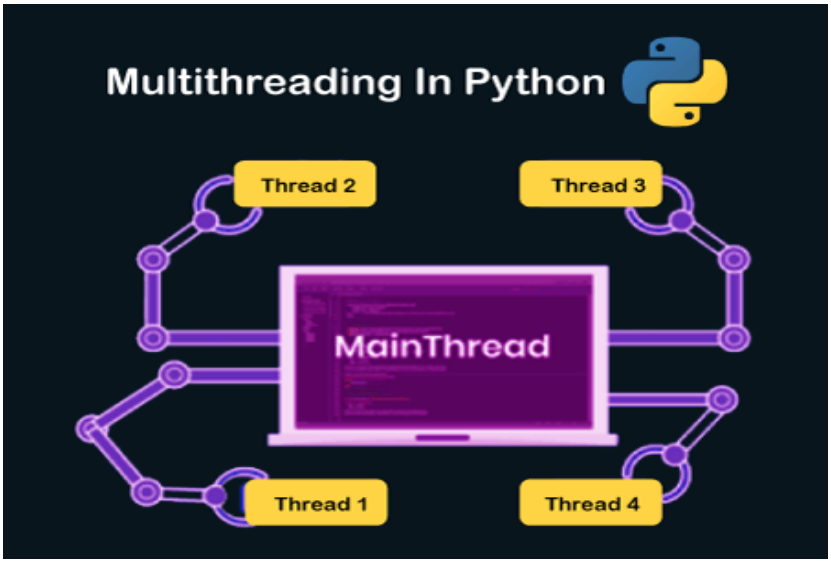
5.7. Understanding threads

5.8. Synchronizing the threads

5.9. Programming using multithreading

Multithreading in Python 3

A thread is the smallest unit of a program or process executed independently or scheduled by the Operating System. In the computer system, an Operating System achieves multitasking by dividing the process into threads. A thread is a lightweight process that ensures the execution of the process separately on the system. In Python 3, when multiple processors are running on a program, each processor runs simultaneously to execute its tasks separately.



Python Multithreading

Multithreading is a threading technique in Python programming to run multiple threads concurrently by rapidly switching between threads with a CPU help (called context switching). Besides, it allows sharing of its data space with the main threads inside a process that share information and communication with other threads easier than individual processes. Multithreading aims to perform multiple tasks simultaneously, which increases performance, speed and improves the rendering of the application.

> ***Note: The Python Global Interpreter Lock (GIL) allows running a single thread at a time, even the machine has multiple processors.***

Benefits of Multithreading in Python

Following are the benefits to create a multithreaded application in Python, as follows:

1. It ensures effective utilization of computer system resources.
2. Multithreaded applications are more responsive.
3. It shares resources and its state with sub-threads (child) which makes it more economical.
4. It makes the multiprocessor architecture more effective due to similarity.
5. It saves time by executing multiple threads at the same time.
6. The system does not require too much memory to store multiple threads.

When to use Multithreading in Python?

It is a very useful technique for time-saving and improving the performance of an application. Multithreading allows the programmer to divide application **tasks** into sub-tasks and simultaneously run them in a program. It allows threads to communicate and share resources such as files, data, and memory to the same processor. Furthermore, it increases the user's responsiveness to continue running a program even if a part of the application is the lengthy or blocked.

How to achieve multithreading in Python?

There are two main modules of multithreading used to handle threads in Python.

1. The thread module
2. The threading module

Thread modules

It is started with Python 3, designated as obsolete, and can only be accessed with **_thread** that supports backward compatibility.

Syntax:

_thread.start_new_thread (function_name, args[, kwargs])

To implement the thread module in Python, we need to import a **thread** module and then define a function that performs some action by setting the target with a variable.

The thread module has been “deprecated” for quite a long time. Users are encouraged to use the threading module instead. Hence, in Python 3, the module "thread" is not available anymore. However, it has been renamed to “_thread” for backwards compatibilities in Python3.

Thread.py

```
import thread # import the thread module  (not available in python 3)
import time # import time module

def cal_sqre(num): # define the cal_sqre function
    print(" Calculate the square of the given number")
    for n in num:
        time.sleep(0.3) # at each iteration it waits for 0.3 time
        print(' Square is : ', n * n)

def cal_cube(num): # define the cal_cube() function
    print(" Calculate the cube of  the given number")
    for n in num:
        time.sleep(0.3) # at each iteration it waits for 0.3 time
        print(" Cube is : ", n * n *n)

arr = [4, 5, 6, 7, 2] # given array

t1 = time.time() # get total time to execute the functions
cal_sqre(arr) # call cal_sqre() function
cal_cube(arr) # call cal_cube() function

print(" Total time taken by program threads is :", time.time() - t1) # print the total time
```

Output:

Calculate the square root of the given number

Square is: 16

Square is: 25

Square is: 36

Square is: 49

Square is: 4

Calculate the cube of the given number

Cube is: 64

Cube is: 125

Cube is: 216

Cube is: 343

Cube is: 8

Total time taken by Program threads is: 3.005793809890747

Threading Modules

The threading module is a high-level implementation of multithreading used to deploy an application in Python. To use multithreading, we need to import the threading module in Python Program.

Thread Class Methods

Methods	Description
start()	A **start()** method is used to initiate the activity of a thread. And it calls only once for each thread so that the execution of the thread can begin.
run()	A **run()** method is used to define a thread's activity and can be overridden by a class that extends the threads class.
join()	A **join()** method is used to block the execution of another code until the thread terminates.

Follow the given below steps to implement the threading module in Python Multithreading:

1. Import the threading module

Create a new thread by importing the **threading** module, as shown.

Syntax:

```
import threading
```

A **threading** module is made up of a **Thread** class, which is instantiated to create a Python thread.

2. Declaration of the thread parameters: It contains the target function, argument, and **kwargs** as the parameter in the **Thread()** class.

- **Target**: It defines the function name that is executed by the thread.
- **Args**: It defines the arguments that are passed to the target function name.

For example:

```
import threading
def print_hello(n):
print("Hello, how old are you ", n)
t1 = threading.Thread( target = print_hello, args =(18, ))
```

In the above code, we invoked the **print_hello()** function as the target parameter. The **print_hello()** contains one parameter **n**, which passed to the **args** parameter.

3. Start a new thread: To start a thread in Python multithreading, call the thread class's object. The start() method can be called once for each thread object; otherwise, it throws an exception error.

Syntax:

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

4. Join method: It is a join() method used in the thread class to halt the main thread's execution and waits till the complete execution of the thread object. When the thread object is completed, it starts the execution of the main thread in Python.

Joinmethod.py

```
import threading
def print_hello(n):
    print("Hello, how old are you? ", n)
T1 = threading.Thread( target = print_hello, args = (20, ))
T1.start()
T1.join()
print("Thank you")
```

Output:

Hello, how old are you? 20

Thank you

When the above program is executed, the join() method halts the execution of the main thread and waits until the thread t1 is completely executed. Once the t1 is successfully executed, the main thread starts its execution.

Note: If we do not use the join() method, the interpreter can execute any print statement inside the Python program. Generally, it executes the first print statement because the interpreter executes the lines of codes from the program's start.

5. Synchronizing Threads in Python

It is a thread synchronization mechanism that ensures no two threads can simultaneously execute a particular segment inside the program to access the shared resources. The situation may be termed as critical sections. We use a race condition to avoid the critical section condition, in which two threads do not access resources at the same time.

Let's write a program to use the threading module in Python Multithreading.

Threading.py

```
import time # import time module
import threading
from threading import *
def cal_sqre(num): # define a square calculating function
    print(" Calculate the square root of the given number")
    for n in num: # Use for loop
        time.sleep(0.3) # at each iteration it waits for 0.3 time
        print(' Square is : ', n * n)
```

```
def cal_cube(num): # define a cube calculating function
    print(" Calculate the cube of  the given number")
    for n in num: # for loop
        time.sleep(0.3) # at each iteration it waits for 0.3 time
        print(" Cube is : ", n * n *n)

ar = [4, 5, 6, 7, 2] # given array

t = time.time() # get total time to execute the functions
#cal_cube(ar)
#cal_sqre(ar)
th1 = threading.Thread(target=cal_sqre, args=(ar, ))
th2 = threading.Thread(target=cal_cube, args=(ar, ))
th1.start()
th2.start()
th1.join()
th2.join()
print(" Total time taking by threads is :", time.time() - t) # print the total time
print(" Again executing the main thread")
print(" Thread 1 and Thread 2 have finished their execution.")
```

Output:

```
Calculate the square root of the given number
 Calculate the cube of the given number
 Square is:  16
 Cube is:  64
 Square is:  25
 Cube is:  125
 Square is:  36
 Cube is:  216
 Square is:  49
 Cube is:  343
```

Square is: 4

Cube is: 8

Total time taken by threads is: 1.5140972137451172

Again executing the main thread

Thread 1 and Thread 2 have finished their execution.

Example:

```
# This is just a just a simple example for the purpose of demonstrating threading in python.
import threading

import requests


def connect_to_urls(urls):

    for url in urls:

        r = requests.get(url)

        print(f"{url} - Request Method: {r.request}")

        print(f"{url} - Status: {r.status_code}")

        print(f"{url} - Encoding method: {r.encoding}")


if __name__ == '__main__':

    url_list = ['https://google.com', 'https://youtube.com', 'https://wikipedia.com',
'https://microsoft.com']

    print(f"URL List: {url_list}")


    t1 = threading.Thread(target=connect_to_urls, args=(url_list[:2],)) # Create Thread 1

    t2 = threading.Thread(target=connect_to_urls, args=(url_list[2:],)) # Create Thread 2

    t1.start()  # Thread 1 starts here

    t2.start()  # Thread 2 starts here
```

Output:

```
URL List: ['https://google.com', 'https://youtube.com',
'https://wikipedia.com', 'https://microsoft.com']
https://google.com - Request Method: <PreparedRequest [GET]>
https://google.com - Status: 200
https://google.com - Encoding method: ISO-8859-1
https://wikipedia.com - Request Method: <PreparedRequest [GET]>
https://wikipedia.com - Status: 200
https://wikipedia.com - Encoding method: ISO-8859-1
https://microsoft.com - Request Method: <PreparedRequest [GET]>
https://microsoft.com - Status: 200
https://microsoft.com - Encoding method: utf-8
https://youtube.com - Request Method: <PreparedRequest [GET]>
https://youtube.com - Status: 200
https://youtube.com - Encoding method: utf-8
```

Example:

Let's consider 3 threads and a function which includes a 1 second sleep.

```
import threading
import time
import sys

def display_time(thread):
    """
    Display the time after a 1 second delay
    """
    print(f" Thread {thread} - Beginning to sleep at {time.time()}")
    time.sleep(1)
    print(f" Thread {thread} - Complete Sleep at {time.time()}")

if __name__ == '__main__':
```

```
t1 = threading.Thread(target = display_time, args=[1])
t2 = threading.Thread(target = display_time, args=[2])
t3 = threading.Thread(target = display_time, args=[3])

start = time.time()
t1.start()
t2.start()
t3.start()

t1.join()
t2.join()
t3.join()

print(f"We are at the end of the program.")
print(f"Total Time Taken by threads = {time.time() - start}")
```

Output:

```
Thread 1 - Beginning to sleep at 1628401793.0577068
Thread 2 - Beginning to sleep at 1628401793.0577068
Thread 3 - Beginning to sleep at 1628401793.0733325
Thread 1 - Complete Sleep at 1628401794.0733278
Thread 2 - Complete Sleep at 1628401794.0889518
Thread 3 - Complete Sleep at 1628401794.0889518
We are at the end of the program.
Total Time Taken by threads = 1.0312449932098389
```

 # Python program given below in which we print thread name and corresponding process for each task:

```
import threading
import os
```

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

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

if __name__ == "__main__":

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

    # print name of main thread
    print(f"Main thread name: {threading.current_thread().name}")

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

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

    # wait until all threads finish
    t1.join()
    t2.join()
```

Output:

```
ID of process running main program: 4524
Main thread name: MainThread
```

```
Task 1 assigned to thread: t1
ID of process running task 1: 4524
Task 2 assigned to thread: t2
ID of process running task 2: 4524
```