Unit-II

Contents

- **Exception Handling:** Handling an exception, Exception Hierarchy, The Exception Model, Run Time Errors, try…….except…….else, try-finally-clause, Argument of an exception, Python standard exceptions and user defined exceptions, Handling IO Exceptions.
- **Multithreading:** Starting a new thread, the threading module, synchronizing threads, race condition, multithreaded priority queue.

Errors in Python

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

```
# initialize the amount variable
amount = 10000

# check that You are eligible to
#  purchase Dsa Self Paced or not
if(amount>2999)
    print("You are eligible to purchase Dsa Self Paced")
```

```
  File "/home/ac35380186f4ca7978956ff46697139b.py", line 4
    if(amount>2999)
                  ^
SyntaxError: invalid syntax
```

Exceptions

- **Exceptions:** Exceptions are raised when the program is syntactically correct but the code resulted in an error. This error does not stop the execution of the program, however, it changes the normal flow of the program.
- **Example:-**
- # initialize the amount variable
- marks = 10000

- # perform division with 0
- a = marks / 0
- print(a)

- Exceptions are errors that are detected during execution. Whenever there is an error in a program, exceptions are raised.
- If these exceptions are not handled, it drives the program into a halt state. Exception handling is required to prevent the program from terminating abruptly.
- In python, an exception is a class that represents error.

Why Exception Handling

- Exceptions are convenient in many ways for handling errors and special conditions in a program. When you think that you have a code which can produce an error then you can use exception handling.
- Exceptions can be unexpected or in some cases, a developer might expect some disruption in the code flow due to an exception that might come up in a specific scenario. Either way, it needs to be handled.

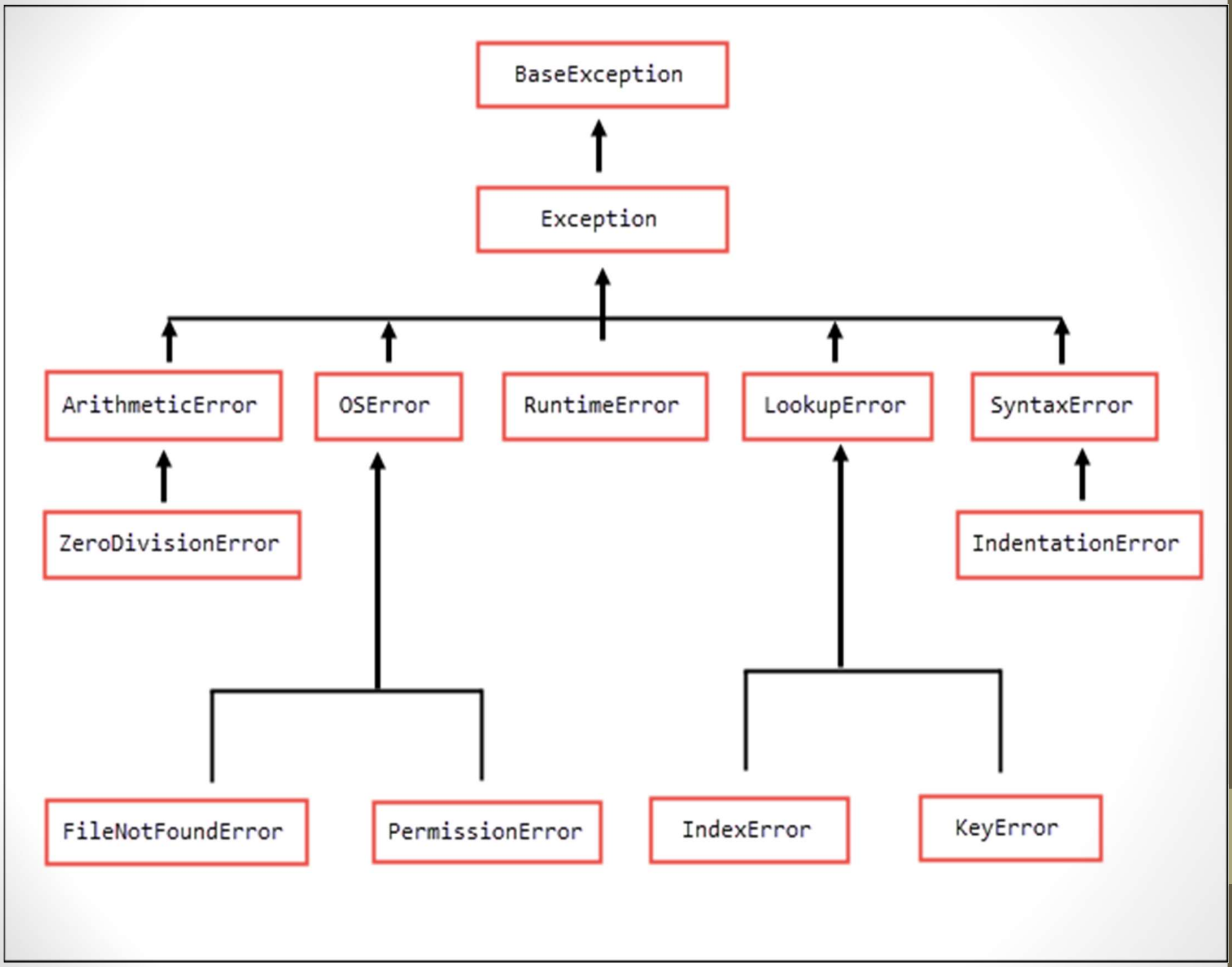
- The cause of an exception is often external to the program itself.
- For example, an incorrect input, a malfunctioning IO device etc.
- Because the program abruptly terminates on encountering an exception, it may cause damage to system resources, such as files.
- Hence, the exceptions should be properly handled so that an abrupt termination of the program is prevented.

Common Exceptions in Python

- we need to understand what some of the common exceptions that python throws are.
- All the inbuilt exceptions in python are inherited from the common ‘Exception’ class.

Common Examples of Exception:

•Division by Zero
•Accessing a file which does not exist.
•Addition of two incompatible types
•Trying to access a non-existent index of a sequence


BaseException
Exception
ArithmeticError
OSError
RuntimeError
LookupError
SyntaxError
ZeroDivisionError
IndentationError
FileNotFoundError
PermissionError
IndexError
KeyError

Exception Name	Description
Exception	All exceptions inherit this class as the base class for all exceptions.
ArithmeticError	Errors that occur during numeric calculation are inherited by it.
OverflowError	When a calculation exceeds the max limit for a specific numeric data type
ZeroDivisionError	Raised when division or modulo by zero takes place.

AttributeError	Raised in case of failure of attribute assignment
NameError	Raised when an identifier is not found in the local or non-local or global scope.
SyntaxError	Raised when there is an error in python syntax.
IndentationError	Raised when indentation is not proper
TypeError	Raised when a specific operation of a function is triggered for an invalid data type

ValueError	Raised when invalid values are provided to the arguments of some builtIn function for a data type that has a valid type of arguments.
RuntimeError	Raised when an error does not fall into any other category
NotImplementedError	Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.

What is Exception Handling in Python?

- Python just like any other programming language provides us with a provision to handle exceptions. And that is by **try & except block.**
- Try block allows us to write the code that is prone to exceptions or errors.
- If any exception occurs inside the try block, the except block is triggered, in which you can handle the exception as now the code will not go into a halt state but the control will flow into the except block where it can be manually handled.

- Any critical code block can be written inside a try clause.
- Once the expected or unexpected exception is raised inside this try, it can be handled inside the except clause (This ‘except’ clause is triggered when an exception is raised inside ‘try’), as this except block detects any exception raised in the try block.
- By default, except detects all types of exceptions, as all the built-in exceptions in python are inherited from common class Exception.
- **Syntax:-**
- **try:**
- **   # critical statement**
- **except:**
- **   # Executed if we get an error in the try block**
- **   # Handle exception here**

- Try and except go hand in hand i.e. the syntax is to write and use both of them. Writing just try or except will give an error.
- **Example:-**

```
def divideNos(a, b):
  return a/b

try:
  divideNos(10, 0)
except:
  print('some exception occurred')
```

- when a python interpreter raises an exception, it is in the form of an object that holds information about the exception type and message.
- Also, every exception type in python inherits from the base class Exception.

Example:-

```
def divideNos(a, b):
  return a/b
try:
  divideNos(10, 0)
# Any exception raised by the python interpreter, is inherited by the base class 'Exception', hence any exception raised in the try block will be detected and collected further in the except block for handling.
except Exception as e:
 print(e) # as e is an object of type Exception, Printing here to see what message it holds.
 print(e.__class__)
```

Keyword for Exception Handling

- Try
- Except
- Raise
- Else
- Finally

Catching Specific Exceptions in Python

- we caught the exception that was being raised in the try block, but the except blocks are going to catch all the exceptions that try might raise.
- Well, it’s considered a good practice to catch specific types of exceptions and handle them accordingly.
- And yes, try can have multiple except blocks. We can also use a tuple of values in an except to receive multiple specific exception types.

```
def divideNos(a, b):
  return a/b # An exception might raise here if b is 0
(ZeroDivisionError)
try:
   a = input('enter a:')
   b = input('enter b:')
   print('after division', divideNos(a, b))
   a = [1, 2, 3]
   print(a[3]) # An exception will raise here as size of array ‘a’ is
3 hence is accessible only up until 2nd index
# if IndexError exception is raised
except IndexError:
   print('index error')
# if ZeroDivisionError exception is raised
except ZeroDivisionError:
   print('zero division error')
```

```
def divideNos(a, b):
        return a/b
try:
    a = int(input('enter a:'))
     b = int(input('enter b:'))
     print('after division', divideNos(a,b))
     a = [1, 2, 3]
     print(a[3])
except (IndexError, ZeroDivisionError):
      print('index error OR zero division error')
```

- Only one of the except blocks is triggered when an exception is raised.
- Consider an example where we have one except as except IndexError and another as except(IndexError, ZeroDivisionError) then the one written first will trigger.

Raising Custom Exceptions

- Even though exceptions in python are automatically raised in runtime when some error occurs.
- Custom and predefined exceptions can also be thrown manually by raising it for specific conditions or a scenario using the raise keyword.
- A custom exception is an exception that a programmer might raise if they have a requirement to break the flow of code in some specific scenario or condition
- String messages supporting the reasons for raising an exception can also be provided to these custom exceptions.

Syntax to Raise an Exception

```
try:
   # on some specific condition or otherwise
   raise SomeError(OptionalMsg)
Except SomeError as e:
   # Executed if we get an error in the try block
   # Handle exception ‘e’ accordingly
```

```
def isStringEmpty(a):
   if(type(a)!=str):
      raise TypeError('a has to be string')
   if(not a):
      raise ValueError('a cannot be null')
   a.strip()
   if(a == ' '):
      return False
   return True

try:
 a = 123
 print('isStringEmpty:', isStringEmpty(a))
except ValueError as e:
   print('ValueError raised:', e)

except TypeError as e:
   print('TypeError raised:', e)
```

try except and ELSE!

- Sometimes you might have a use case where you want to run some specific code only when there are no exceptions.
- For such scenarios, the else keyword can be used with the try block.
- Note that this else keyword and its block are optional.
- **Syntax with Else:**
- try:
- `    # on some specific condition or otherwise`
- `    raise SomeError(OptionalMsg)`
- except SomeError as e:
- `    # Executed if we get an error in the try block`
- `    # Handle exception ‘e’ accordingly`
- else
- `    # Executed if no exceptions are raised`

When an exception is not raised, it flows into the optional else block.

Example:-

```
try:
      b  = 10
      c = 2
      a = b/c
      print(a)
except:
       print('Exception raised')
else:
     print('no exceptions raised')
```

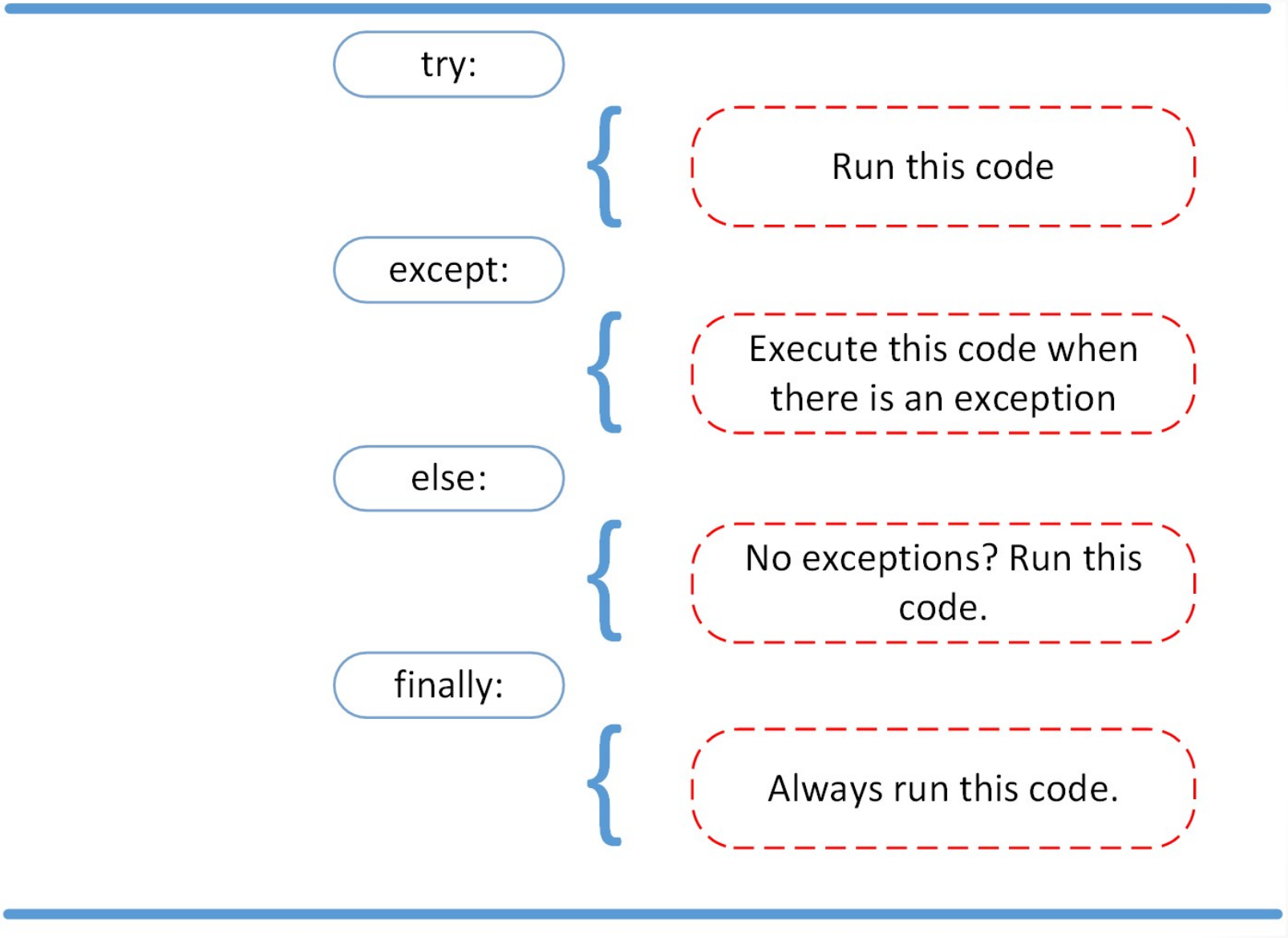
- In the above code, As both the inputs are greater than 0 which is not a risk to DivideByZeroException,
- hence try block won't raise any exception and hence 'except' block won't be triggered.
- And only when the control doesn't flow to the except block, it flows to the else block.
- Further handling can be done inside this else block if there is something you want to do.

Try Clause with Finally

- Finally is a keyword that is used along with try and except, when we have a piece of code that is to be run irrespective of if try raises an exception or not.
- Code inside finally will always run after try and except.

Example:

```
try:
   temp = [1, 2, 3]
    temp[4]
except Exception as e:
 print('in exception block: ', e)
else:
 print('in else block')
finally:
 print('in finally block')
```


try:
Run this code
except:
Execute this code when there is an exception
else:
No exceptions? Run this code.
finally:
Always run this code.

Demonstration-I

```
import sys
a=int(input("Enter first value:"))
b=int(input("Eneter second value"))
try:
    if(a<0):
        raise ValueError("Please enter positive number")
    elif(b<0):
        raise ValueError("Second number Enetred is Positive Number")
    else:
        print("Sum is:",a+b)
except ValueError as n:
    print(n)
```

Output

```
In [1]: runfile('E:/Documents/Python Programs/excp.py', wdir='E:/Documents/Python
Programs')

Enter first value: 10

Eneter second value 20
Sum is: 30

In [2]: runfile('E:/Documents/Python Programs/excp.py', wdir='E:/Documents/Python
Programs')

Enter first value: -2

Eneter second value 20
Please enter positive number

In [3]: runfile('E:/Documents/Python Programs/excp.py', wdir='E:/Documents/Python
Programs')

Enter first value: 20

Eneter second value -9
Second number Enetred is Positive Number
```

Demonstration-II

```
try:
    a=int(input("Enter first number: "))
    b=int(input("Enter second number: "))
    if b==0:
        raise ZeroDivisionError
    else:
        m=a/b
        print("The division is:",m)
except ZeroDivisionError:
    print("This is an exception. Please Dont divide number by zero!!!!")
```

Output

```
In [4]: runfile('E:/Documents/Python Programs/exception3.py', wdir='E:/Documents/Python
Programs')

Enter first number: 10

Enter second number: 5
The division is: 2.0

In [5]: runfile('E:/Documents/Python Programs/exception3.py', wdir='E:/Documents/Python
Programs')

Enter first number: 10

Enter second number: 0
This is an exception. Please Dont divide number by zero!!!!

In [6]:
```

Demonstration-III

```
1 mca=[23,4,67,9,102,45]
2 try:
3     m=int(input("Please specify position you want to access:"))
4     if m>(len(mca)):
5         raise IndexError
6     else:
7         print("The values is: " ,mca[m])
8 except IndexError:
9     print("Position specified is out of range")
```

Output

```
In [7]: runfile('E:/Documents/Python Programs/exception2.py', wdir='E:/Documents/Python
Programs')

Please specify position you want to access:4
The values is:  102

In [8]: runfile('E:/Documents/Python Programs/exception2.py', wdir='E:/Documents/Python
Programs')

Please specify position you want to access: 10
Position specified is out of range

In [9]:
```

Demonstration-IV

```
import sys

mca=[23,2,14,60,'c',12,0,34,'d',58]

for n in mca:
    try:
        print("The value is:",n)
        r=1/float(n)
        print(r)
        continue
    except:
        print("oops",sys.exc_info(),"exception occured")
        print("Please proceed with next entry")
```

Output

```
The value is: 23
0.043478260869565216
The value is: 2
0.5
The value is: 14
0.07142857142857142
The value is: 60
0.016666666666666666
The value is: c
oops (<class 'ValueError'>, ValueError("could not convert string to float: 'c'"),
<traceback object at 0x00000000097D6448>) exception occured
Please proceed with next entry
The value is: 12
0.08333333333333333
The value is: 0
oops (<class 'ZeroDivisionError'>, ZeroDivisionError('float division by zero'),
<traceback object at 0x00000000097D6508>) exception occured
Please proceed with next entry
The value is: 34
0.029411764705882353
The value is: d
oops (<class 'ValueError'>, ValueError("could not convert string to float: 'd'"),
<traceback object at 0x00000000097D63C8>) exception occured
Please proceed with next entry
The value is: 58
0.017241379310344827
```

Demonstration-V

```
mca=[23,4,67,9,102,45]
try:
    m=int(input("Please specify position you want to access:"))
    if m>(len(mca)):
        raise IndexError
    else:
        print("The values is: " ,mca[m])
except IndexError:
    print("Position specified is out of range")
else:
    print("Program has no exception so else block executed")
finally:
    print("Program has exception or not but finally will execute always")
```

Output

```
In [11]: runfile('E:/Documents/Python Programs/exception4.py', wdir='E:/Documents/Python
Programs')

Please specify position you want to access: 5
The values is:  45
Program has no exception so else block executed
Program has exception or not but finally will execute always

In [12]: runfile('E:/Documents/Python Programs/exception4.py', wdir='E:/Documents/Python
Programs')

Please specify position you want to access: 15
Position specified is out of range
Program has exception or not but finally will execute always

In [13]:
```

Customized Exception

- Python has numerous built-in exceptions that force your program to output an error when something in the program goes wrong.
- However, sometimes you may need to create your own custom exceptions that serve your purpose.
- In Python, users can define custom exceptions by creating a new class. This exception class has to be derived, either directly or indirectly, from the built-in Exception class. Most of the built-in exceptions are also derived from this class.

- When we are developing a large Python program, it is a good practice to place all the user-defined exceptions that our program raises in a separate file.
- Many standard modules do this. They define their exceptions separately as exceptions.py or errors.py (generally but not always).
- User-defined exception class can implement everything a normal class can do, but we generally make them simple and concise.
- Most implementations declare a custom base class and derive others exception classes from this base class.

```
class UnderAge(Exception):
    pass

def checkAge():
    try:
        age=23
        if age<18:
            raise UnderAge
    except UnderAge:
        print("Age less than 18 is not allowed")
    else:
        print("You are allowed to login")

checkAge()
```

```
class CheckforSalary(Exception):
    pass

def salary():
    try:
        s=int(input("Eneter salary of employee: "))
        if s<5000:
            raise CheckforSalary
        else:
            print("Employee registered succussfully")
    except:
        print("Minimum salary of any employee in this organization is 5000")

salary()
```

```
class CheckMobile(Exception):
    pass

def MobileNum():
    try:
        MN="9826265889"
        if len(MN)<10:
            raise CheckMobile
        elif len(MN)>10:
            raise CheckMobile
    except CheckMobile:
        print("Mobile number should be of 10 digits, Please eneter correctly")
    else:
        print("You have entered correct Mobile number")

MobileNum()
```

```
class CheckDataType(Exception):
    pass

def checkforInt():
    try:
        data=123
        t=type(data)
        if t==int:
            print("You have eneted correct data")
        else:
            raise CheckDataType
    except CheckDataType:
        print("Please enter Integer type of data only")

checkforInt()
```

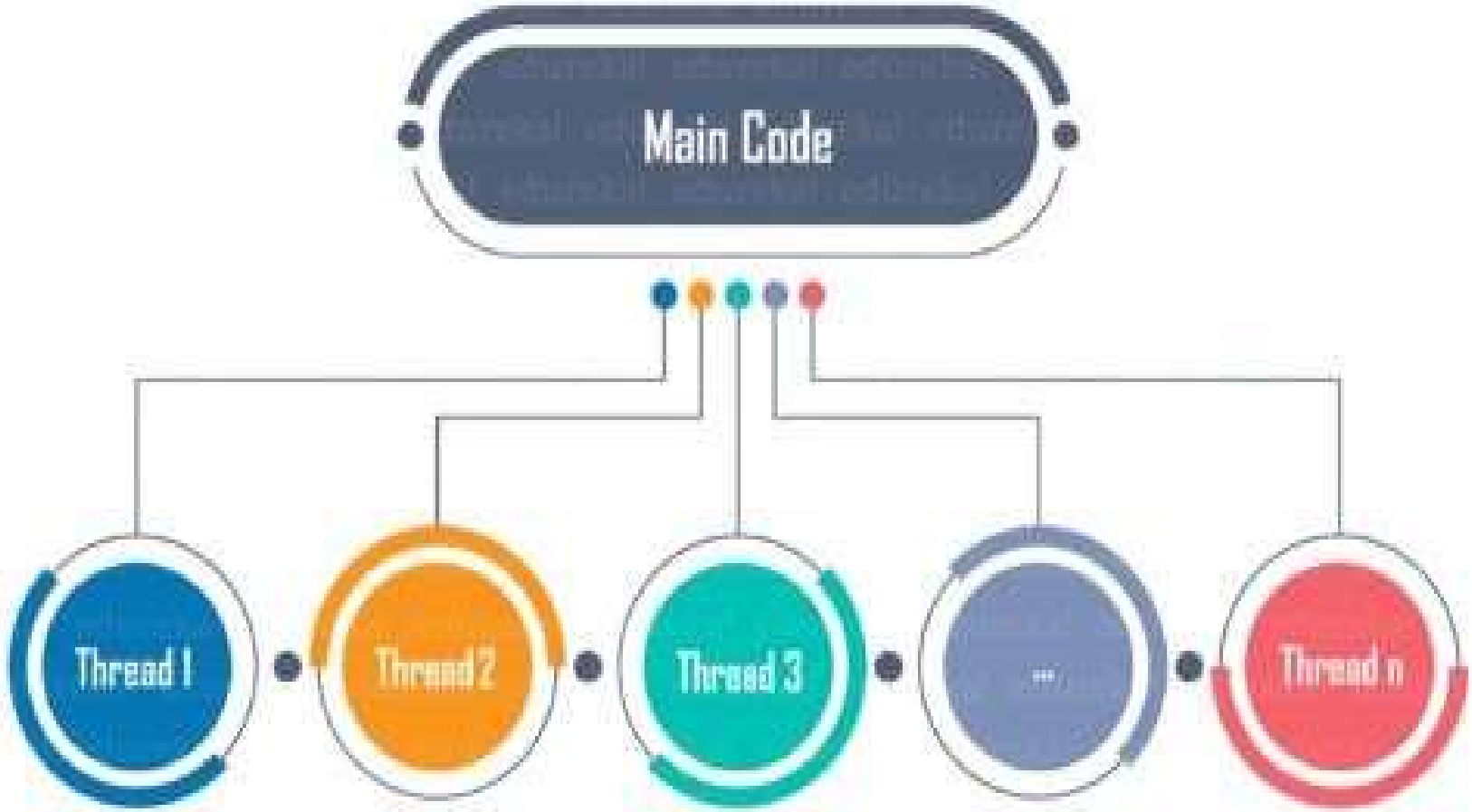
```
# define Python user-defined exceptions
class Error(Exception):
    """Base class for other exceptions"""
    pass


class ValueTooSmallError(Error):
    """Raised when the input value is too small"""
    pass


class ValueTooLargeError(Error):
    """Raised when the input value is too large"""
    pass


# you need to guess this number
number = 10
```

```
# user guesses a number until he/she gets it right
while True:
    try:
        i_num = int(input("Enter a number: "))
        if i_num < number:
            raise ValueTooSmallError
        elif i_num > number:
            raise ValueTooLargeError
        break
    except ValueTooSmallError:
        print("This value is too small, try again!")
        print()
    except ValueTooLargeError:
        print("This value is too large, try again!")
        print()

print("Congratulations! You guessed it correctly.")
```

Multithreading in Python

Introduction

- A thread is the smallest unit of execution with the independent set of instructions.
- It is a part of the process and operates in the same context sharing program's runnable resources like memory.
- 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
- *For Example,* when you are playing a game on your PC, the game as a whole is a single process, but it consists of several threads responsible for playing the music, taking input from the user, running the opponent synchronously, etc. All these are separate threads responsible for carrying out these different tasks in the same program.


Main Code
Thread 1
Thread 2
Thread 3
...
Thread n

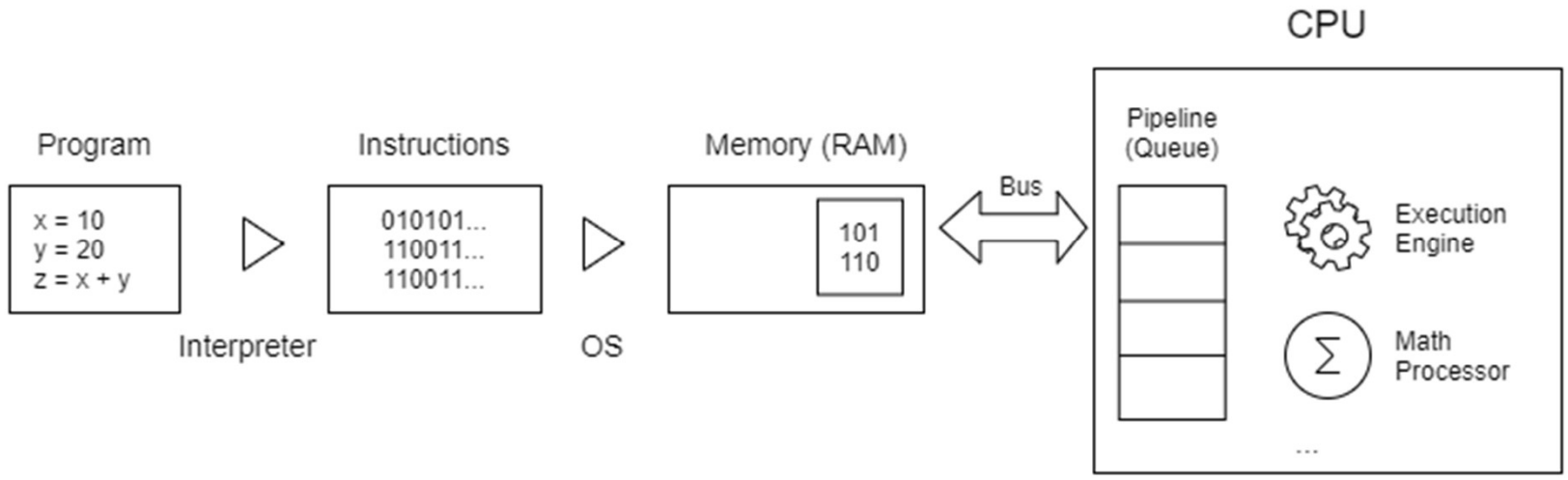
Introduction to processes and threads

- Suppose that you have a simple Python program:

```
x = 10
y = 20
z = x + y
```

- Computers don't understand Python. They only understand machine code, which is a set of instructions containing zero and one.
- Therefore, you need a Python interpreter to execute this Python program that translates the Python code to machine code.
- When you execute the python app.py command, Python interpreter converts the app.py into machine code.
- The operating system (OS) needs to load the program into the memory (RAM) to run the program.
- Once the OS loads the program to memory, it moves the instructions to the CPU for execution via bus.

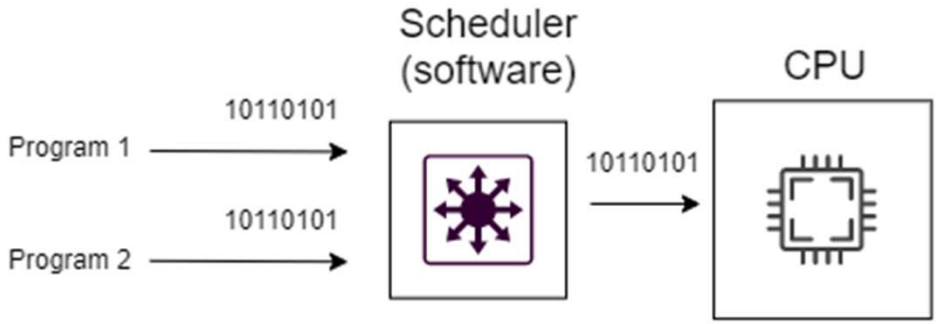
- In general, the OS moves the instructions to a queue, also known as a pipeline. Then, the CPU will execute the instructions from the pipeline.
- **By definition, a process is an instance of the program running on a computer. And a thread is a unit of execution within a process.**
- The following picture illustrates the flow of running a program in Python on a computer:



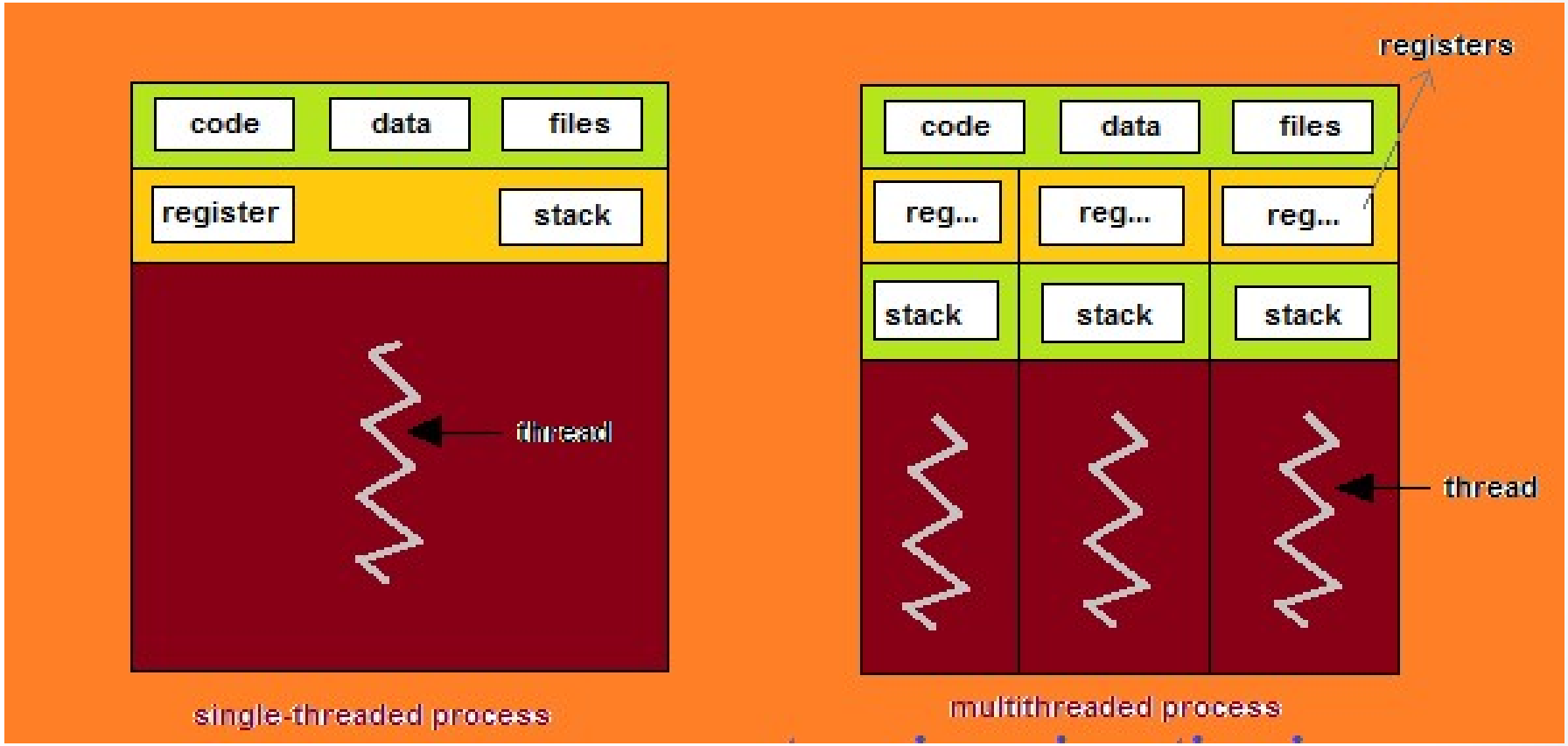
- So far, you've learned how to develop a program that has one process with one thread. Therefore, the terms process and thread are often used interchangeably sometimes.
- Typically, a program may have one or more processes. And a process can have one or more threads.

Single-core processors

- In the past, a CPU has only one core. In other words, it can run only a single thread at one time.
- To execute multiple threads “at the same time”, the OS uses a software component called **scheduler**:



- The scheduler is like a switch that handles process scheduling. The main task of the scheduler is to select the instructions and submit them for execution regularly.
- The scheduler switches between processes so fast (about 1ms) that you feel the computer can run multiple processes simultaneously.
- When a program has multiple threads, it's called a **multi-threaded application**. Otherwise, it's called a **single-threaded application**.
- **Multithreading** is the ability of a single-core CPU to provide multiple threads of execution concurrently supported by the scheduler of the OS.


code
data
files
register
stack
thread
single-threaded process
registers
code
data
files
reg...
reg...
reg...
stack
stack
stack
thread
multithreaded process

Multiple-core processors

- Today, the CPU often has multiple cores, e.g., two cores (dual-core) and four cores (quad-core).
- A dual-core CPU can execute exactly two processes, and a quad-core CPU can execute four processes simultaneously.
- Generally, the more cores the CPU has, the more processes it can truly execute simultaneously.
- **Multiprocessing** uses a multi-core CPU within a single computer, which indeed executes multiple processes in parallel.

Process

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 process is basically the program in execution. When you start an application in your computer (like a browser or text editor), the operating system creates a **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).

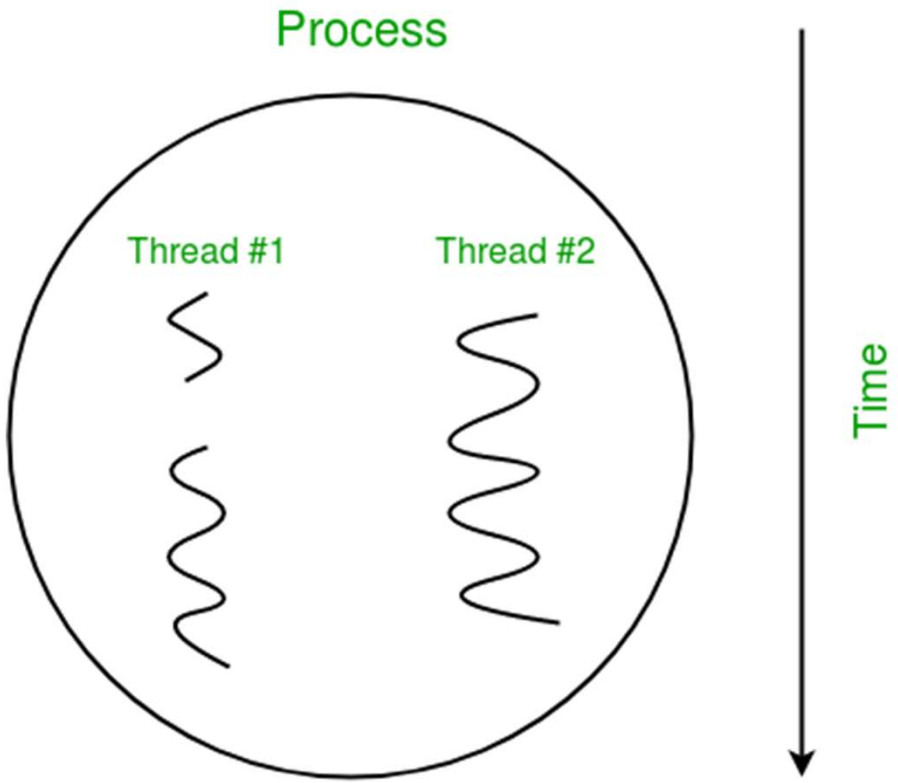
To understand processes and threads, consider this scenario: An .exe file on your computer is a program. When you open it, the OS loads it into memory, and the CPU executes it. The instance of the program which is now running is called the process.

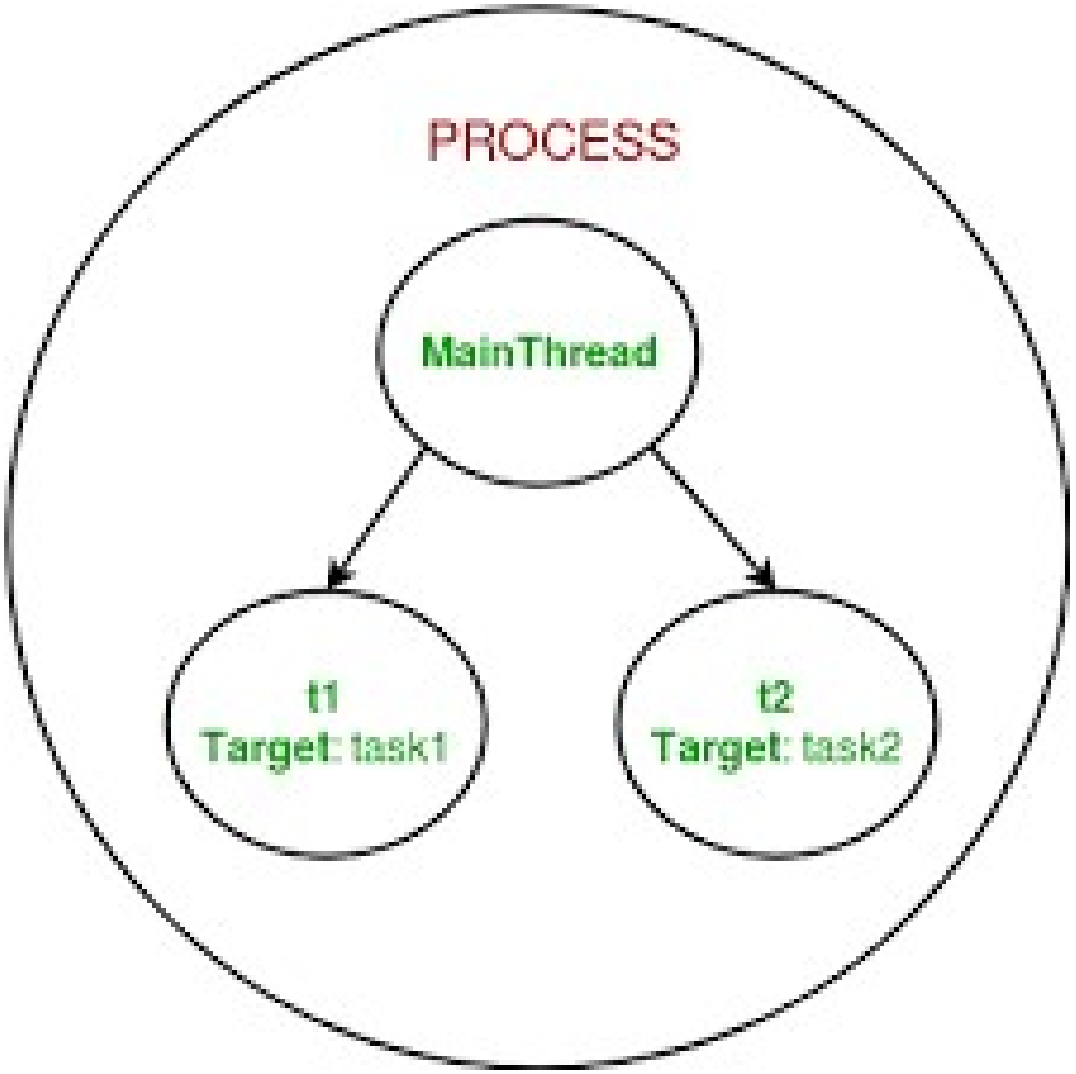
Every process will have 2 fundamental components:

- The Code
- The Data

Now, a process can contain one or more sub-parts called **threads.** This depends on the OS architecture. You can think about a thread as a section of the process which can be executed separately by the operating system.

In other words, it is a stream of instructions which can be run independently by the OS. Threads within a single process share the data of that process and are designed to work together for facilitating parallelism.


Process
Thread #1
Thread #2
Time


PROCESS
MainThread
t1
Target: task1
t2
Target: task2

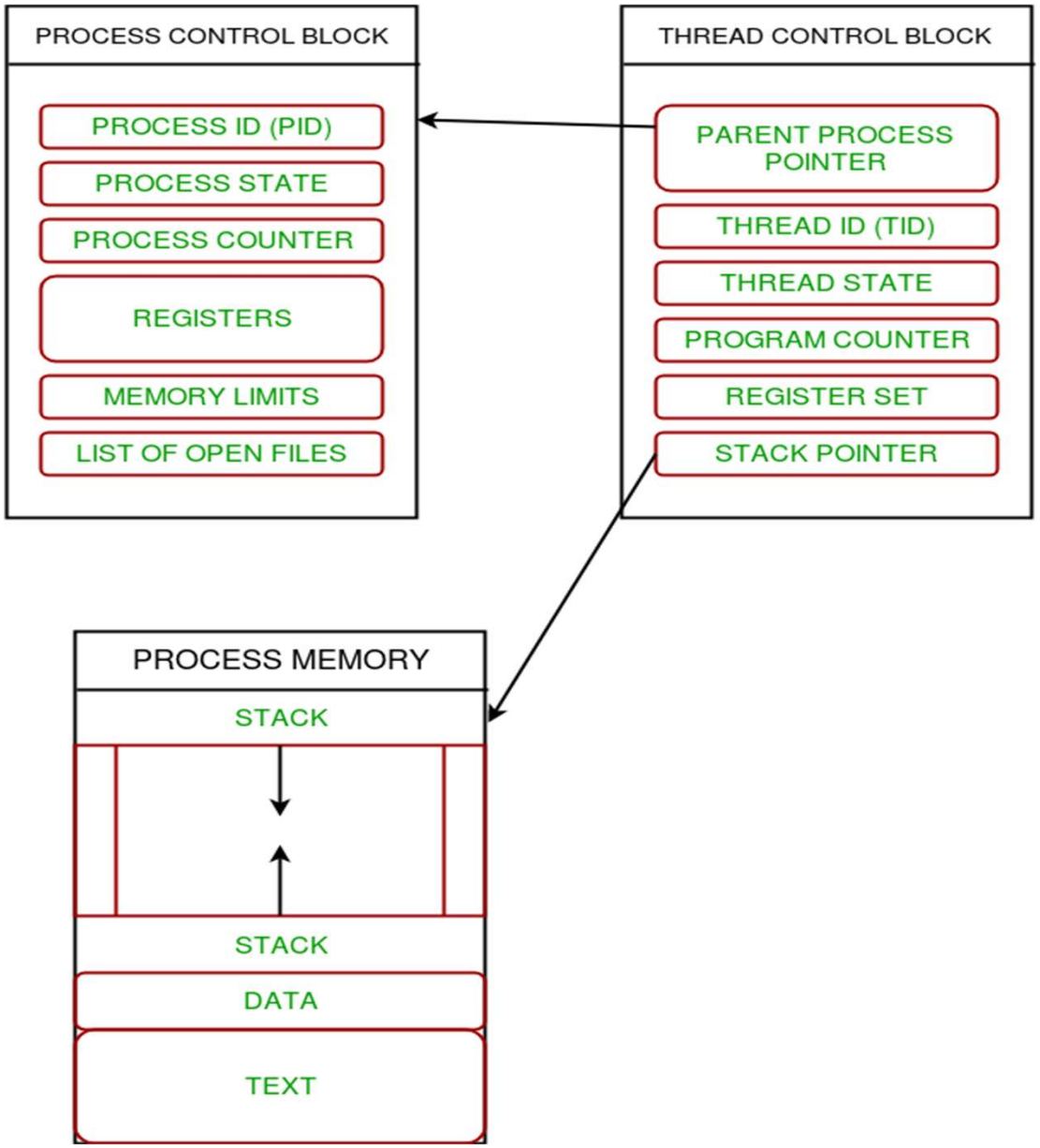
Thread

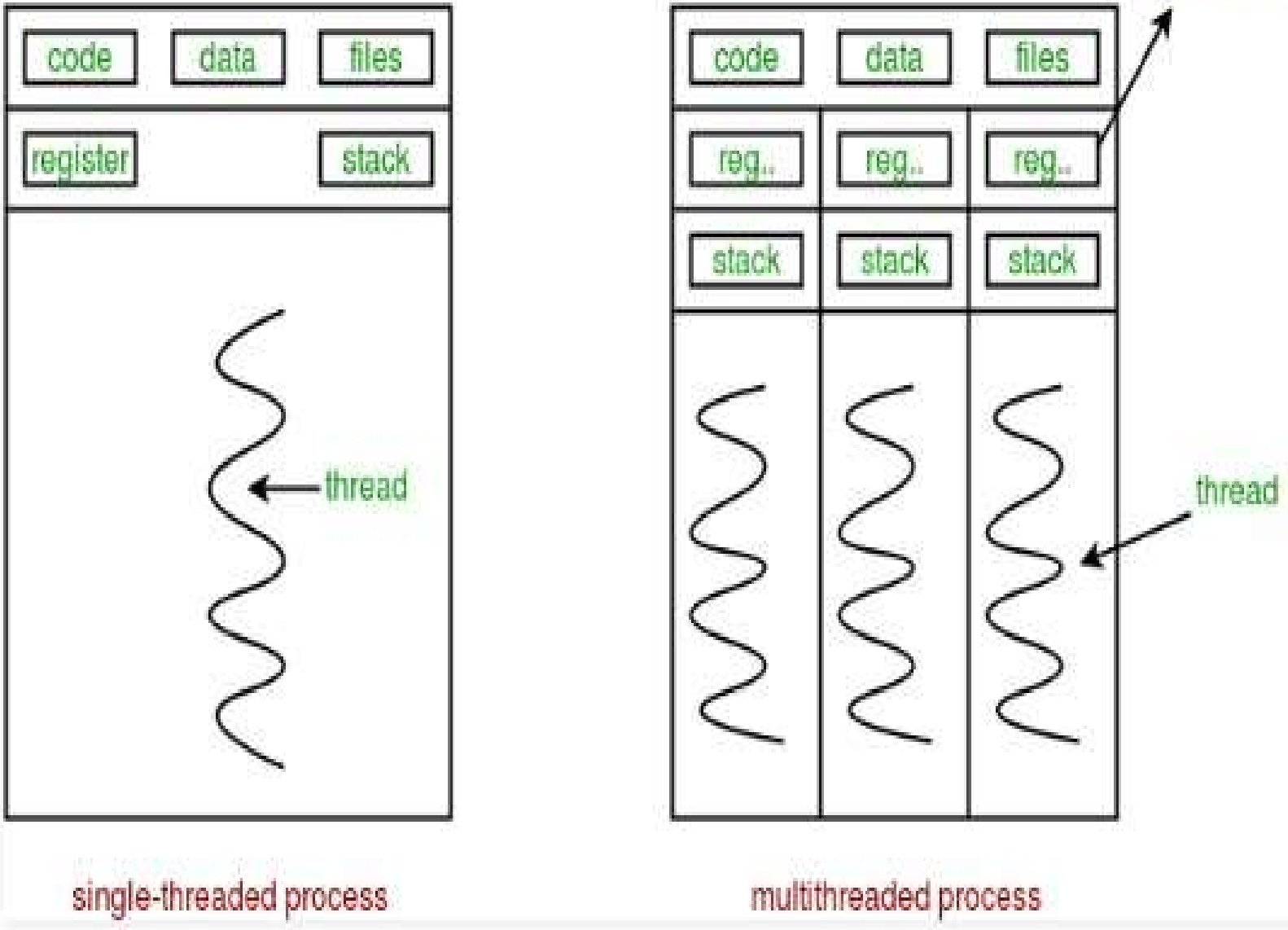
- In simple words, a **thread** is a sequence of such instructions within a program that can be executed independently of other code. For simplicity, you can assume that a thread is simply a subset of a process!
- A thread contains all this information in a **Thread Control Block (TCB)**:

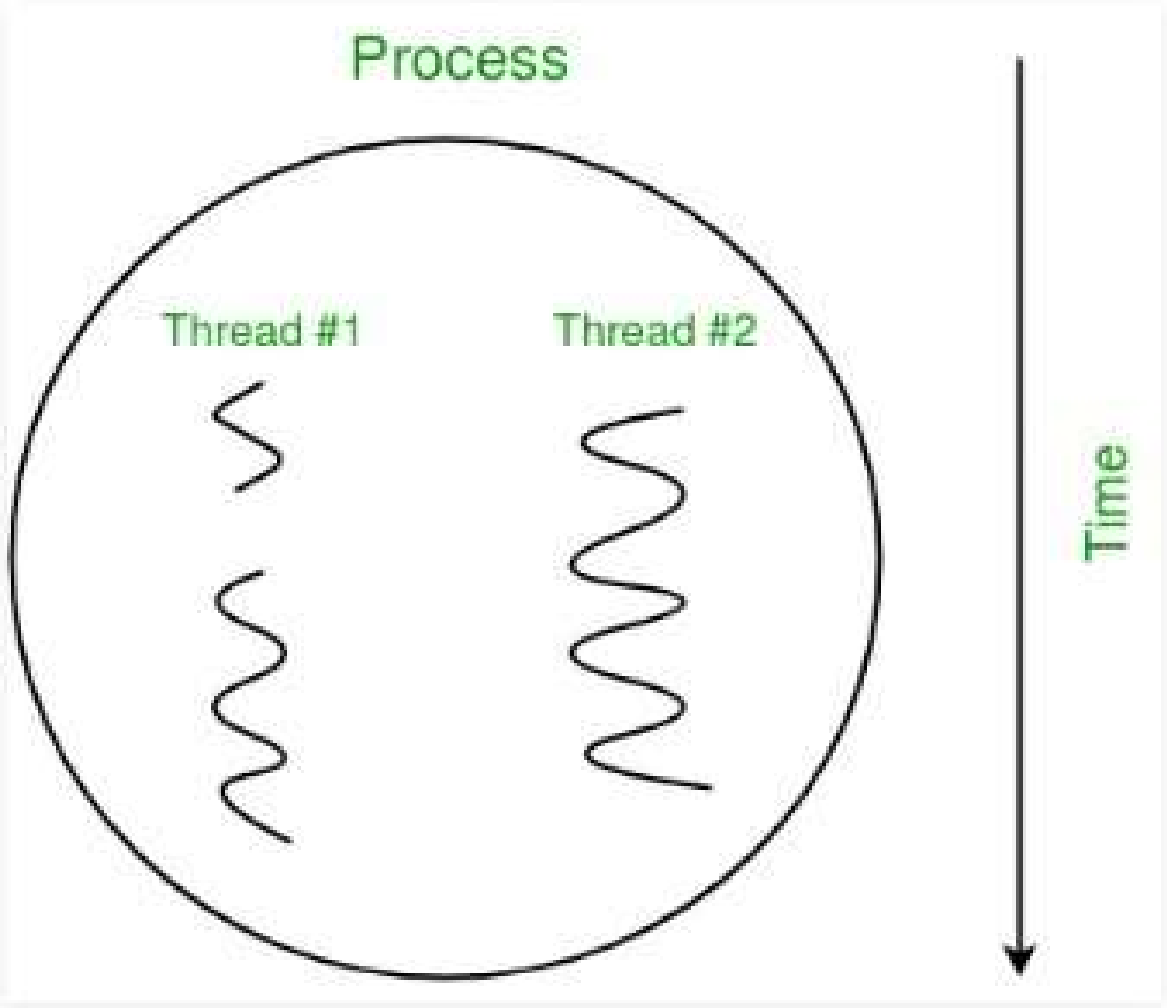
1. **Thread Identifier:** Unique id (TID) is assigned to every new thread
2. **Stack pointer:** Points to thread's stack in the process. Stack contains the local variables under thread's scope.
3. **Program counter:** a register which stores the address of the instruction currently being executed by thread.
4. **Thread state:** can be running, ready, waiting, start or done.
5. **Thread's register set:** registers assigned to thread for computations.
6. **Parent process Pointer:** A pointer to the Process control block (PCB) of the process that the thread lives on.

Multiple Threads

- Multiple threads can exist within one process where:
- Each thread contains its own **register set** and **local variables (stored in stack)**.
- All thread of a process share **global variables (stored in heap)** and the **program code**.
- Consider the diagram below to understand how multiple threads exist in memory:


PROCESS CONTROL BLOCK
PROCESS ID (PID)
PROCESS STATE
PROCESS COUNTER
REGISTERS
MEMORY LIMITS
LIST OF OPEN FILES
THREAD CONTROL BLOCK
PARENT PROCESS POINTER
THREAD ID (TID)
THREAD STATE
PROGRAM COUNTER
REGISTER SET
STACK POINTER
PROCESS MEMORY
STACK
STACK
DATA
TEXT


code
data
files
register
stack
thread
single-threaded process
registers
code
data
files
reg..
reg..
reg..
stack
stack
stack
thread
multithreaded process



Multithreading in Python

When to Use

- Multithreading is very useful for saving time and improving performance, but it cannot be applied everywhere. In the previous example, the music thread is independent of the thread that takes your input and the thread that takes your input is independent of the thread that runs your opponent. These threads run independently because they are not inter-dependent.
- Therefore, multithreading can be used only when the dependency between individual threads does not exist.

Benefits of Multithreading

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.

Applications

- **Web Browsers -** A web browser can download any number of files and web pages (multiple tabs) at the same time and still lets you continue browsing. If a particular web page cannot be downloaded, that is not going to stop the web browser from downloading other web pages.
- **Web Servers -** A threaded web server handles each request with a new thread. There is a thread pool and every time a new request comes in, it is assigned to a thread from the thread pool.
- **Computer Games -** You have various objects like cars, humans, birds which are implemented as separate threads. Also playing the background music at the same time as playing the game is an example of multithreading.

- **Text Editors -** When you are typing in an editor, spell-checking, formatting of text and saving the text are done concurrently by multiple threads. The same applies for Word processors also.
- **IDE -** IDEs like Android Studio run multiple threads at the same time. You can open multiple programs at the same time. It also gives suggestions on the completion of a command which is a separate thread.

What is Multithreading in Python?

- **Multithreading in Python** programming is a well-known technique in which multiple threads in a process share their data space with the main thread which makes information sharing and communication within threads easy and efficient.
- Threads are lighter than processes.
- Multi threads may execute individually while sharing their process resources.
- The purpose of multithreading is to run multiple tasks and function cells at the same time.

How to create thread in python

- Threads in Python can be created in three ways:

1. Without creating a class
2. By extending Thread class
3. Without extending Thread class

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

1.The thread module
2.The threading module

- Python 2.x used to have the <*thread*> module. But it got deprecated in Python 3.x and renamed to <*_thread*> module for backward compatibility.
- The principal difference between the two modules is that the module <*_thread*> implements a thread as a function.
- On the other hand, the module <*threading*> offers an object-oriented approach to enable thread creation.

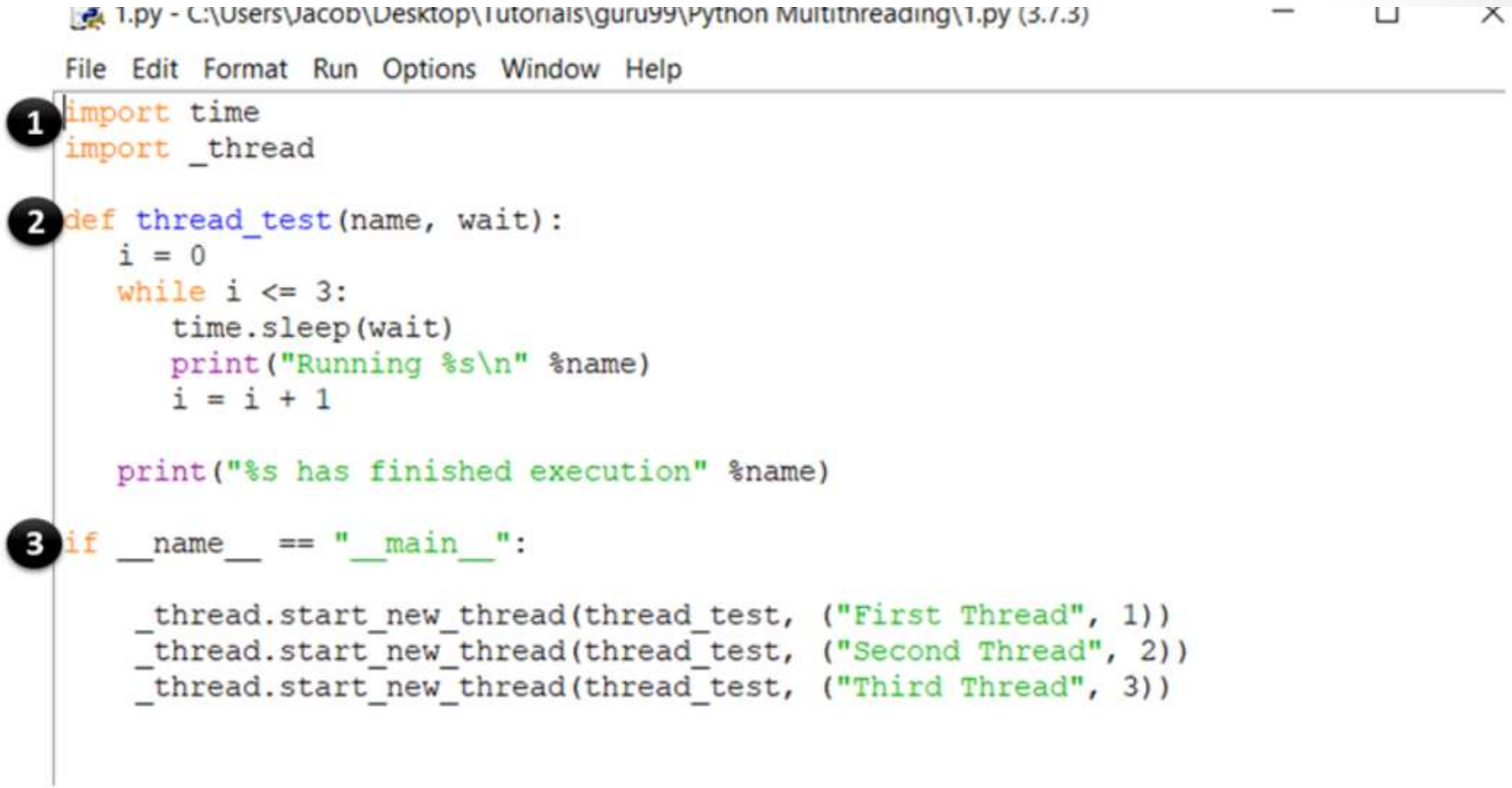
Thread Module

- If you decide the <*thread*> module to apply in your program, then use the following method to create threads.
- **#Syntax**

```
thread.start_new_thread(function, args[,kwargs] )
```

- This method is quite efficient and straightforward for creating threads.
- You can use it to run programs in both Linux and Windows.

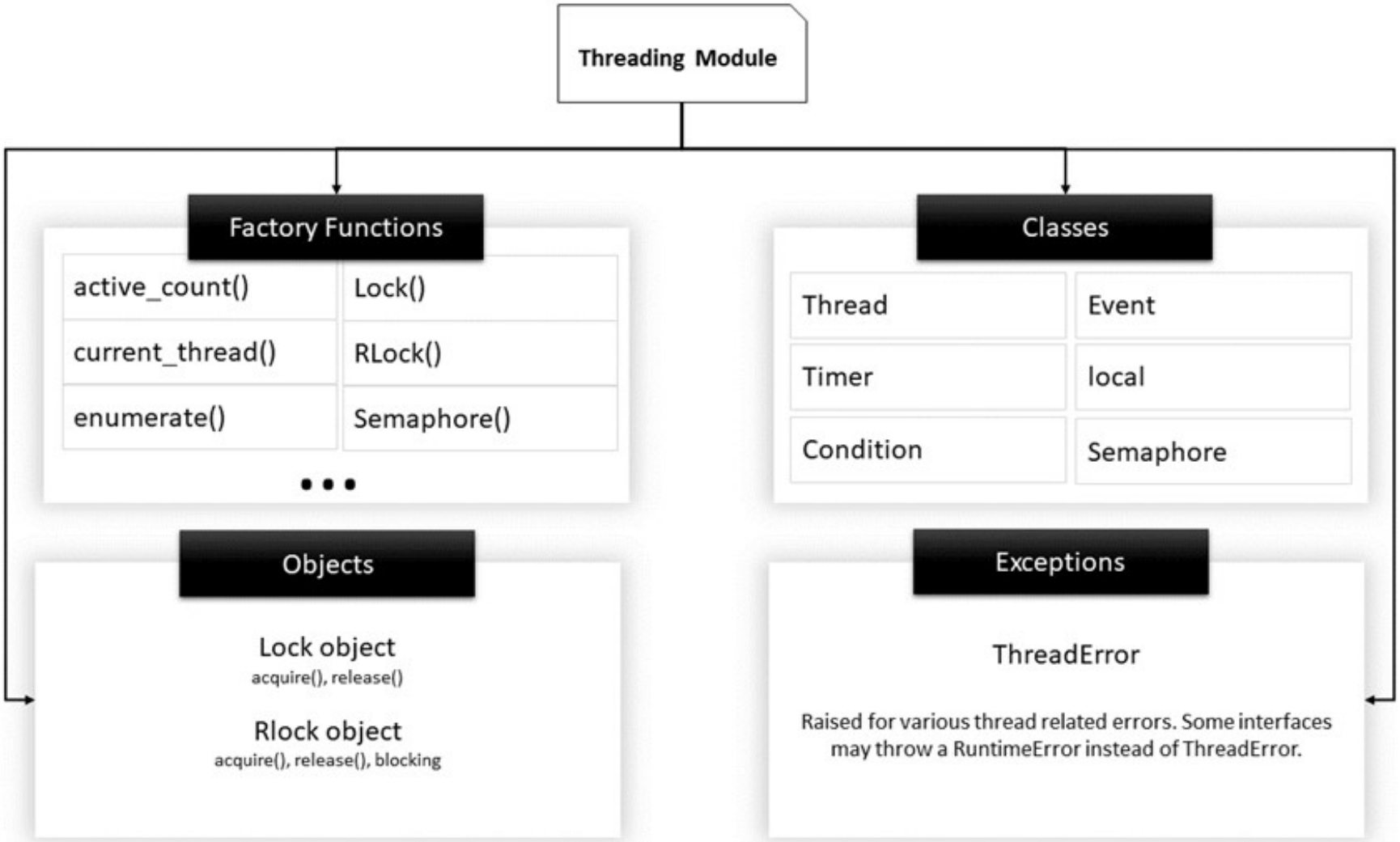
- This method starts a new thread and returns its identifier.
- It'll invoke the function specified as the "function" parameter with the passed list of arguments.
- When the <*function*> returns, the thread would silently exit.
- Here, *args* is a tuple of arguments; use an empty tuple to call <*function*> without any arguments.
- The optional <*kwargs*> argument specifies the dictionary of keyword arguments.


1.py - C:\Users\Jacob\Desktop\Tutorials\guru99\Python Multithreading\1.py (3.7.3)
File Edit Format Run Options Window Help
1
import time
import _thread
2
def thread_test(name, wait):
i = 0
while i <= 3:
time.sleep(wait)
print("Running %s\n" %name)
i = i + 1
print("%s has finished execution" %name)
3
if __name__ == "__main__":
_thread.start_new_thread(thread_test, ("First Thread", 1))
_thread.start_new_thread(thread_test, ("Second Thread", 2))
_thread.start_new_thread(thread_test, ("Third Thread", 3))

1.These statements import the time and thread module which are used to handle the execution and delaying of the Python threads.
2.Here, you have defined a function called **thread_test,** which will be called by the **start_new_thread** method. The function runs a while loop for four iterations and prints the name of the thread which called it. Once the iteration is complete, it prints a message saying that the thread has finished execution.
3.This is the main section of your program. Here, you simply call the **start_new_thread** method with the **thread_test** function as an argument.This will create a new thread for the function you pass as argument and start executing it. Note that you can replace this (thread_test) with any other function which you want to run as a thread.

Threading Module

- The latest <*threading*> module provides rich features and better support for threads than the legacy <*thread*> module
- The <*threading*> module is an excellent example of Python Multithreading.
- The <*threading*> module combines all the methods of the <*thread*> module and exposes few additional methods.


Threading Module
Factory Functions
active_count()
Lock()
current_thread()
RLock()
enumerate()
Semaphore()
• • •
Classes
Thread
Event
Timer
local
Condition
Semaphore
Objects
Lock object
acquire(), release()
Rlock object
acquire(), release(), blocking
Exceptions
ThreadError
Raised for various thread related errors. Some interfaces may throw a RuntimeError instead of ThreadError.

Function Name	Description
activeCount()	Returns the count of **Thread** objects which are still alive
currentThread()	Returns the current object of the Thread class.
enumerate()	Lists all active Thread objects.
isDaemon()	Returns true if the thread is a daemon.
isAlive()	Returns true if the thread is still alive.

Thread Class methods	
start()	Starts the activity of a thread. It must be called only once for each thread because it will throw a runtime error if called multiple times.
run()	This method denotes the activity of a thread and can be overridden by a class that extends the Thread class.
join()	It blocks the execution of other code until the thread on which the join() method was called gets terminated.

Steps to implement threads using the threading module

- You may follow the below steps to implement a new thread using the <*threading*> module.

•Construct a subclass from the <*Thread*> class.

•Override the <*__init__(self [,args])*> method to supply arguments as per requirements.

•Next, override the <*run(self [,args])*> method to code the business logic of the thread.

- Once you define the new <*Thread*> subclass, you have to instantiate it to start a new thread. Then, invoke the <*start()*> method to initiate it. It will eventually call the <*run()*> method to execute the business logic.

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:

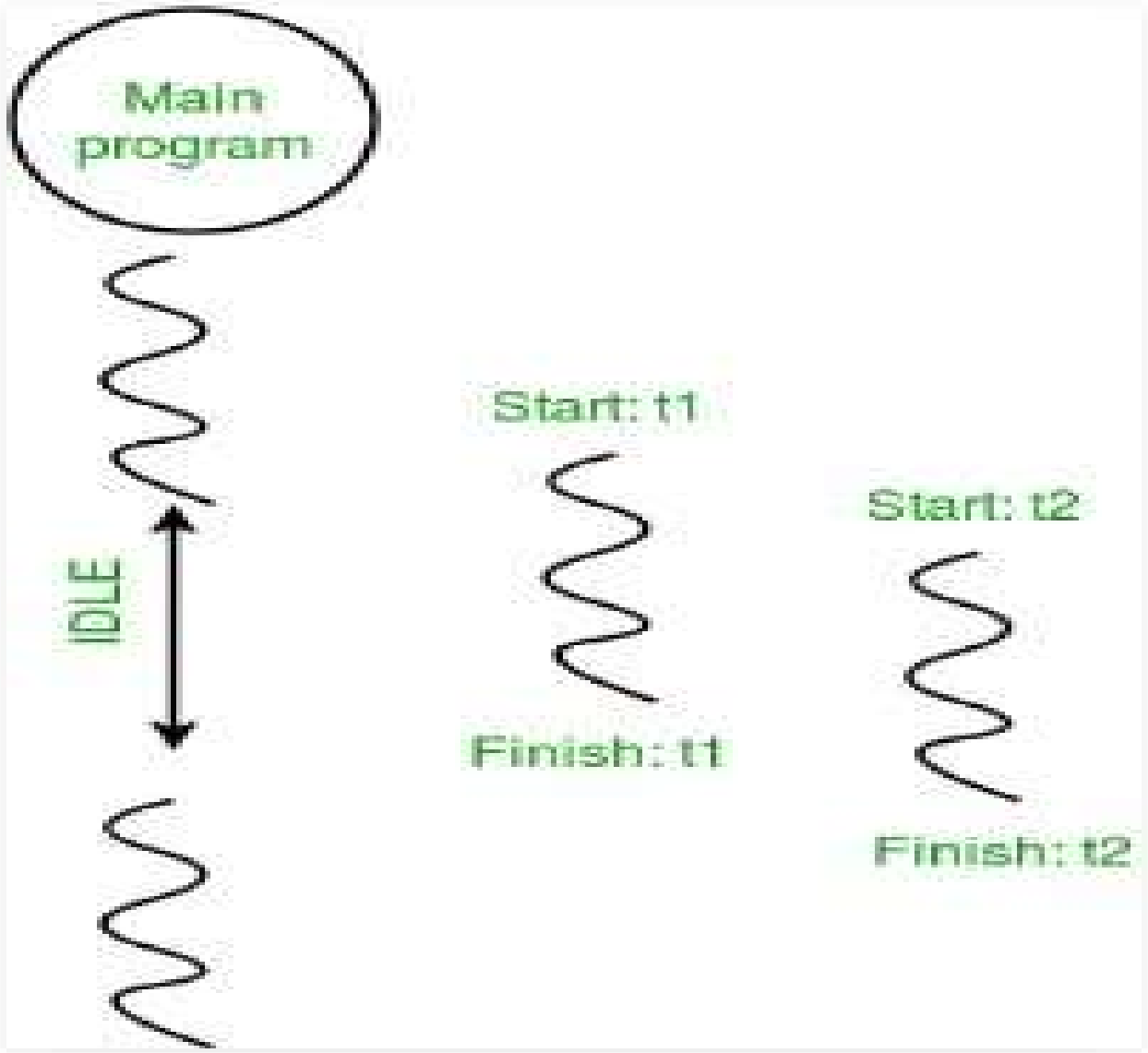
```
new_thread.start()
```

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 second thread to complete before it is terminated.

```
In [31]: import threading
    ...:
    ...: def showA():
    ...:     print("A Thread")
    ...:
    ...: def showB():
    ...:     print("B Thread")
    ...:
    ...:
    ...: t1=threading.Thread(target=showA)
    ...: t2=threading.Thread(target=showB)
    ...:
    ...: t1.start()
    ...: t2.start()
    ...:
    ...: t2.join()
    ...: print("Final Thread")
    ...: t1.join()
A Thread
B Thread
Final Thread
```


Main program
IDLE
Start: t1
Finish: t1
Start: t2
Finish: t2

```
import threading
import os

def showA():
    print("A Thread")
    print("Name of current thread is: ", threading.current_thread().name)
    print("ID of First thread is: ", os.getpid())
def showB():
    print("B Thread")
    print("Name of secong current thread: ",threading.current_thread().name)
    print("ID of second thread: ", os.getpid())
t1=threading.Thread(target=showA,name="Thread1")
t2=threading.Thread(target=showB,name="Thread2")

t1.start()
t2.start()

t1.join()
t2.join()

print("Final Thread")
```

```
A ThreadB Thread
Name of secong current thread:  Thread2
ID of second thread:  6468

Name of current thread is:  Thread1
ID of First thread is:  6468
Final Thread
```

Functions of Thread Class

Method	Description
activeCount()	Returns the count of **Thread** objects which are still alive
currentThread()	Returns the current object of the Thread class.
enumerate()	Lists all active Thread objects.
isAlive()	Returns true if the thread is still alive.
start()	Starts the activity of a thread. It must be called only once for each thread because it will throw a runtime error if called multiple times.

Functions of Thread Class

Method	Description
run()	This method denotes the activity of a thread and can be overridden by a class that extends the Thread class.
join()	It blocks the execution of other code until the thread on which the join() method was called gets terminated.
getName():	The getName() method retrieves the name of a thread.
setName():	The setName() method updates the name of a thread.

Without Creating a Class

- Multithreading in Python can be accomplished without creating a class as well

```
from threading import *
print(current_thread().getName())
def mt():
    print("Child Thread\n")
child=Thread(target=mt)
child.start()
print("Executing thread name :\n",current_thread().getName())
```

Output

```
MainThread
Child Thread

Executing thread name :
 MainThread
```

By Extending Thread Class

- When a child class is created by extending the Thread class, the child class represents that a new thread is executing some task. When extending the Thread class, the child class can override only two methods i.e. the __init_() method and the run() method. No other method can be overridden other than these two methods.

```
import threading
import time
class mythread(threading.Thread):
    def run(self):
        for x in range(7):
            print("Hi from child")
a = mythread()
a.start()
a.join()
print("Bye from",current_thread().getName())
```

Output

```
Hi from child
Hi from child
Hi from child
Hi from child
Hi from child
Hi from child
Hi from child
Bye from MainThread
```

Without Extending Thread class

- To create a thread without extending the Thread class

```
from threading import *
class ex:
    def myfunc(self):
        for x in range(7):
            print("Child")
myobj=ex()
thread1=Thread(target=myobj.myfunc)
thread1.start()
thread1.join()
print("done")
```

Output

```
Child
Child
Child
Child
Child
Child
Child
done
```

Demonstration-I

```
import threading
import time
class mythread(threading.Thread):
    def run(self):
        for x in range(7):
            print("ITM Universe")
a = mythread()
a.start()
a.join()
print("Bye from",current_thread().getName())
print(current_thread().getName())
current_thread().setName("ITM CSE")
print(current_thread().getName())
print(current_thread().isAlive())
print(current_thread().isDaemon())
```

Output

```
In [28]: runfile('E:/Documents/Python Programs/Multithreading/multi2.py', wdir='E:/
Documents/Python Programs/Multithreading')
ITM Universe
ITM Universe
ITM Universe
ITM Universe
ITM Universe
ITM Universe
ITM Universe
ITM Universe
Bye from My New Thread
My New Thread
ITM CSE
True
False
```

Demonstration-II

```
import time
import threading

class Demo (threading.Thread):
    def __init__(self, id, name, i):
        threading.Thread.__init__(self)
        self.id = id
        self.name = name
        self.i = i

    def run(self):
        test(self.name, self.i, 5)
        print ("%s has finished execution " %self.name)

def test(name, wait, i):

    while i:
        time.sleep(wait)
        print ("Running %s \n" %name)
        i = i - 1

if __name__=="__main__":
    t1 = Demo(1, "First Thread", 1)
    t2 = Demo(2, "Second Thread", 2)
    t3 = Demo(3, "Third Thread", 3)

    t1.start()
    t2.start()
    t3.start()

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

Output

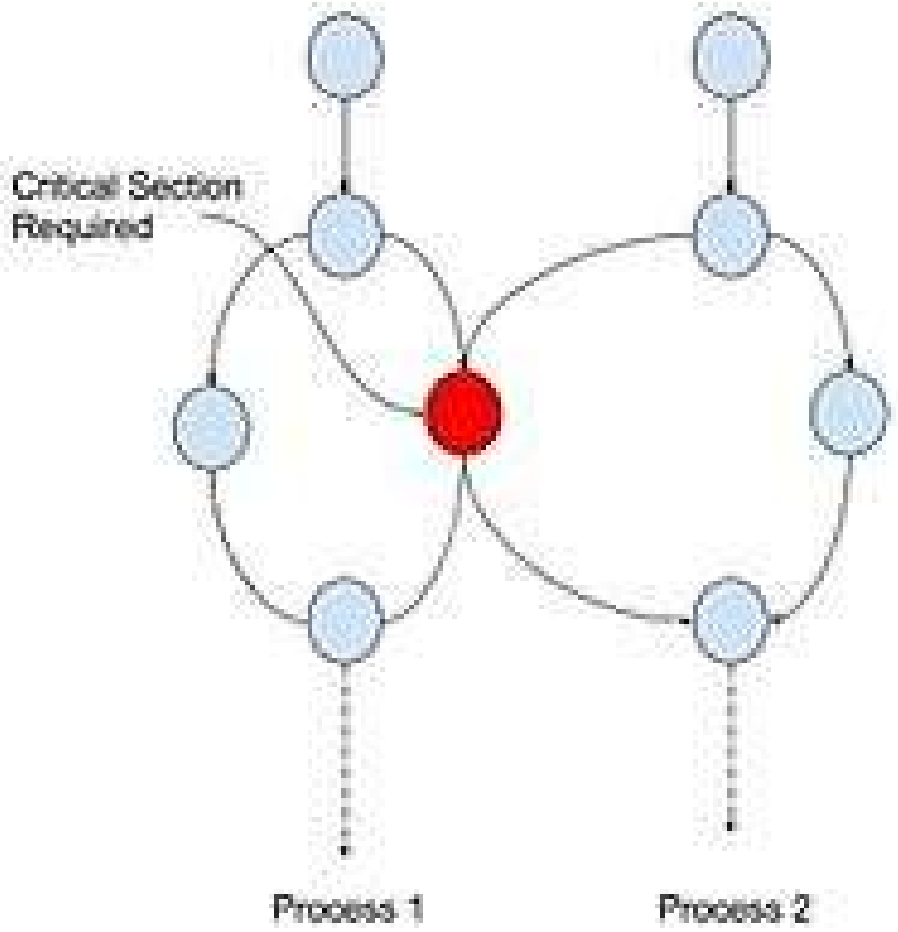
```
Running First Thread

Running Second Thread

Running First Thread

Running Third Thread

Running First Thread

Running Second Thread

Running First Thread

Running First Thread

First Thread has finished execution
Running Third Thread
Running Second Thread


Running Second Thread

Running Third Thread

Running Second Thread

Second Thread has finished execution
Running Third Thread

Running Third Thread

Third Thread has finished execution
```

Deadlocks and Race conditions

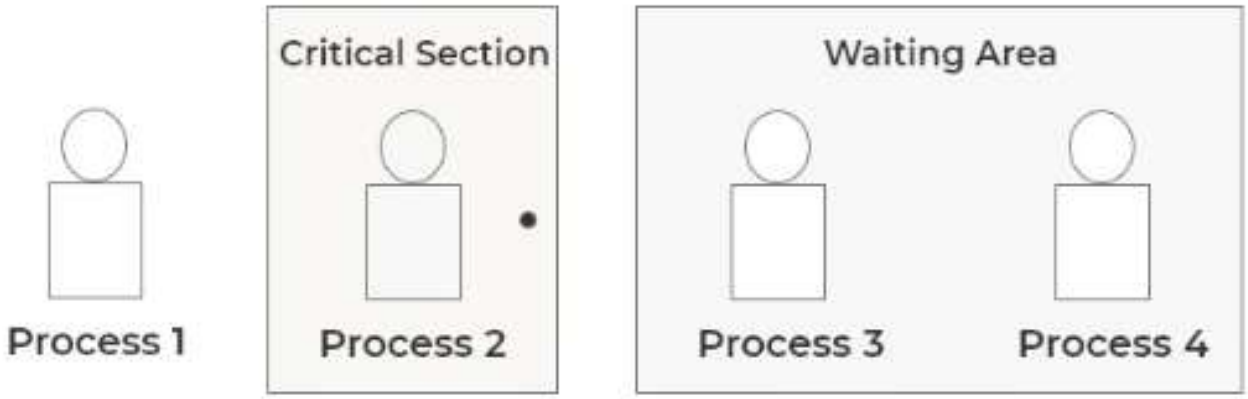
- Before learning about deadlocks and race conditions, it'll be helpful to understand a few basic definitions related to concurrent programming:

1. **Critical Section :** It is a fragment of code that accesses or modifies shared variables and must be performed as an atomic transaction.

- The critical section is a code segment where the shared variables can be accessed.
- An atomic action is required in a critical section i.e. only one process can execute in its critical section at a time.
- All the other processes have to wait to execute in their critical sections.


Critical Section
Required
Process 1
Process 2

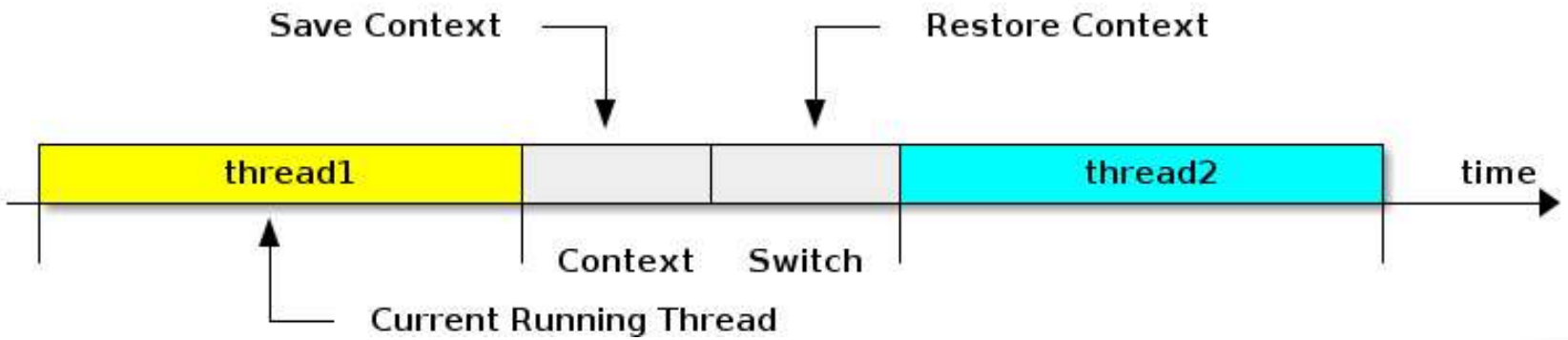
Critical Section in Operating System python



- Process 1 has operated thought critical section
- Process 2 is operating in critical section
- Process 3 is waiting for process 2 to finish from critical section
- Process 4 is waiting for process 3 to finish

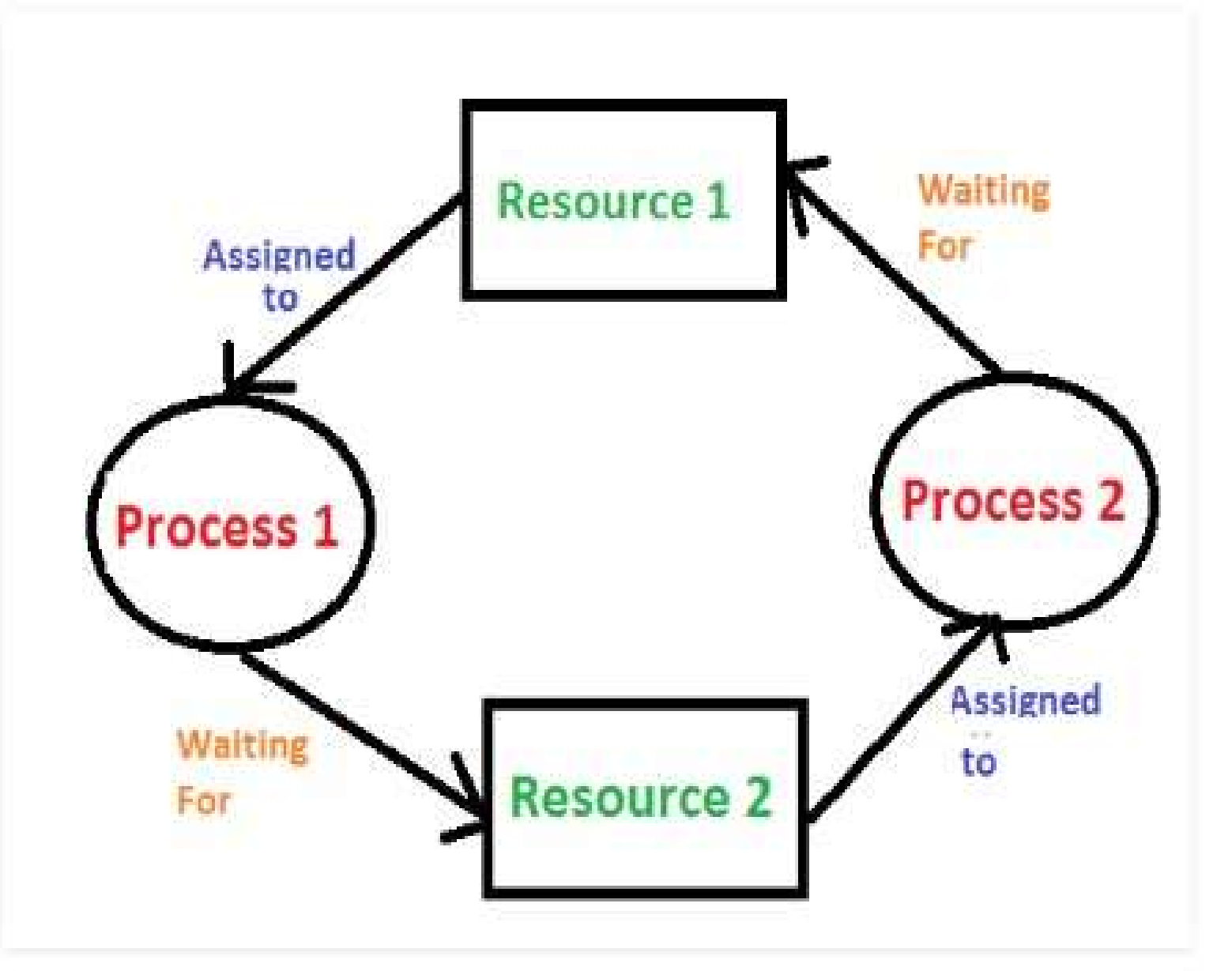
2. Context Switch : It is the process that a CPU follows to store the state of a thread before changing from one task to another so that it can be resumed from the same point later.

As the process is running, another process arrives in the ready queue, which has a high priority of completing its task using CPU. Here we used context switching that switches the current process with the new process requiring the CPU to finish its tasks.



Deadlock

- ***Deadlock*** is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.
- Consider an example when two trains are coming toward each other on the same track and there is only one track, none of the trains can move once they are in front of each other. A similar situation occurs in operating systems when there are two or more processes that hold some resources and wait for resources held by other(s).
- For example, in the below diagram, Process 1 is holding Resource 1 and waiting for resource 2 which is acquired by process 2, and process 2 is waiting for resource 1.


Resource 1
Assigned
to
Waiting
For
Process 1
Process 2
Waiting
For
Resource 2
Assigned
to

Deadlocks

- Deadlocks are the most feared issue that developers face when writing concurrent/multithreaded applications in python. The best way to understand deadlocks is by using the classic computer science example problem known as the **Dining Philosophers Problem.**
- The problem statement for dining philosophers is as follows:
- Five philosophers are seated on a round table with five plates of spaghetti (a type of pasta) and five forks, as shown in the diagram.

Dining Philosophers Problem

- At any given time, a philosopher must either be eating or thinking.
- Moreover, a philosopher must take the two forks adjacent to him (i.e., the left and right forks) before he can eat the spaghetti. The problem of deadlock occurs when all five philosophers pick up their right forks simultaneously.
- Since each of the philosophers has one fork, they will all wait for the others to put their fork down. As a result, none of them will be able to eat spaghetti.
- Similarly, in a concurrent system, a deadlock occurs when different threads or processes (philosophers) try to acquire the shared system resources (forks) at the same time. As a result, none of the processes get a chance to execute as they are waiting for another resource held by some other process.

Race Conditions

- A race condition is an unwanted state of a program which occurs when a system performs two or more operations simultaneously. For example, consider this simple for loop:

```
i=0; # a global variable
for x in range(100):
    print(i)
    i+=1;
```

- If you create **n** number of threads which run this code at once, you cannot determine the value of i (which is shared by the threads) when the program finishes execution.
- This is because in a real multithreading environment, the threads can overlap, and the value of i which was retrieved and modified by a thread can change in between when some other thread accesses it.
- A race condition occurs when two threads try to access a shared variable simultaneously.
- The first thread reads the value from the shared variable. The second thread also reads the value from the same shared variable.

- Then both threads try to change the value of the shared variable. And they race to see which thread writes a value to the variable last.
- The value from the thread that writes to the shared variable last is preserved because it overwrites the value that the previous thread wrote.

Synchronizing threads

- To deal with race conditions, deadlocks, and other thread-based issues, the threading module provides the **Lock** object.
- The idea is that when a thread wants access to a specific resource, it acquires a lock for that resource. Once a thread locks a particular resource, no other thread can access it until the lock is released.
- As a result, the changes to the resource will be atomic, and race conditions will be averted.
- A lock is a low-level synchronization primitive implemented by the**__thread** module.

- At any given time, a lock can be in one of 2 states: **locked** or **unlocked.** It supports two methods:

- **acquire():** When the lock-state is unlocked, calling the acquire() method will change the state to locked and return. However, If the state is locked, the call to acquire() is blocked until the release() method is called by some other thread.

- **release():** The release() method is used to set the state to unlocked, i.e., to release a lock. It can be called by any thread, not necessarily the one that acquired the lock.

```
import threading
lock = threading.Lock()

def first_function():
    for i in range(5):
        lock.acquire()
        print ('lock acquired')
        print ('Executing the first funcion')
        lock.release()

def second_function():
    for i in range(5):
        lock.acquire()
        print ('lock acquired')
        print ('Executing the second funcion')
        lock.release()


thread_one = threading.Thread(target=first_function)
thread_two = threading.Thread(target=second_function)

thread_one.start()
thread_two.start()

thread_one.join()
thread_two.join()
```

```
lock acquired
Executing the first funcion
lock acquired
Executing the first funcion
lock acquired
Executing the first funcion
lock acquired
Executing the first funcion
lock acquired
Executing the first funcion
lock acquired
Executing the second funcion
lock acquired
Executing the second funcion
lock acquired
Executing the second funcion
lock acquired
Executing the second funcion
lock acquired
Executing the second funcion
```

Multithreaded Priority Queue

- The Queue module is primarily used to manage to process large amounts of data on multiple threads. It supports the creation of a new queue object that can take a distinct number of items.
  The get() and put() methods are used to add or remove items from a queue respectively. Below is the list of operations that are used to manage Queue:

1. **get():** It is used to add an item to a queue.
2. **put():** It is used to remove an item from a queue.
3. **qsize():** It is used to find the number of items in a queue.
4. **empty():** It returns a boolean value depending upon whether the queue is empty or not.
5. **full():** It returns a boolean value depending upon whether the queue is full or not.

A Priority Queue is an extension of the queue with the following properties:

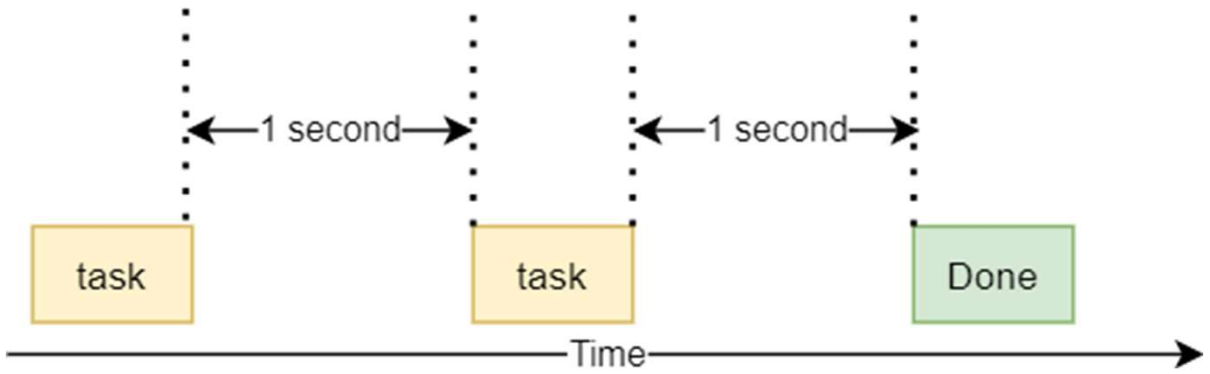
- An element with high priority is dequeued before an element with low priority.
- If two elements have the same priority, they are served according to their order in the queue

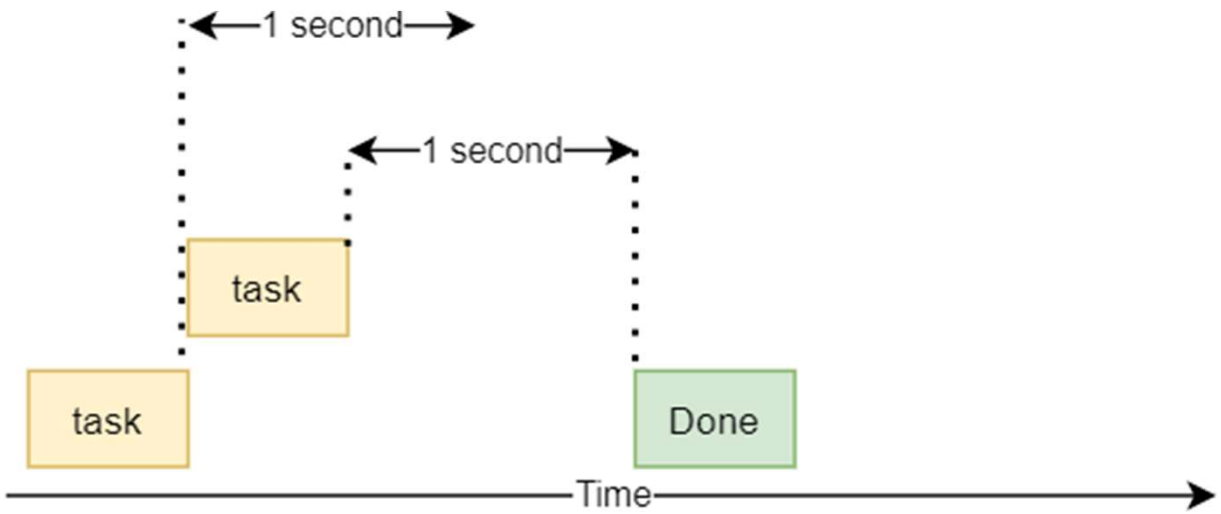
Advantages

1. Better utilization of resources
2. Simplifies the code
3. Allows concurrent and parallel occurrence of various tasks
4. Reduces the time consumption or response time, thereby, increasing the performance.

Disadvantages

1. On a single processor system, multithreading won't hit the speed of computation. The performance may downgrade due to the overhead of managing threads.
2. Synchronization is needed to prevent mutual exclusion while accessing shared resources. It directly leads to more memory and CPU utilization.
3. Multithreading increases the complexity of the program, thus also making it difficult to debug.
4. It raises the possibility of potential deadlocks.
5. It may cause starvation when a thread doesn't get regular access to shared resources. The application would then fail to resume its work.


1 second
1 second
task
task
Done
Time


1 second
1 second
task
task
Done
Time