Python Programming

Unit-III

Regular Expressions or RegEx

- **Introduction:**

The regular expressions can be defined as the sequence of characters which are used to search for a pattern in a string.

Example:

^a...s$

The above code defines a RegEx pattern. The pattern is: any five letter string starting with **a** and ending with **s**.

Expression	String	Matched?
^a...s$	abs	No match
	alias	Match
	abyss	Match
	Alias	No match
	An abacus	No match

Example:

[abc]

Here, [abc] will match if the string you are trying to match contains any of the a, b or c.

Expression	String	Matched?
[abc]	a	1 match
	ac	2 matches
	Hey Jude	No match
	abc de ca	5 matches

Note: Python has a built-in package called **re**, which can be used to work with Regular Expressions.

- **Special Symbols and Characters:**

A regular expression can be formed by using the mix of meta-characters and special sequences.

- **MetaCharacters**

Metacharacters are characters that are interpreted in a special way by a RegEx engine. Here's a list of metacharacters:

Metacharacter is a character with the specified meaning.

Metacharacter	Description	Example
[]	It represents the set of characters.	"[a-z]"
.	It represents any character except new line character.	"Ja.v."
^	It represents characters at the beginning of the string.	"^Java"
$	It represents characters at the ending of the string.	"python$"
*	It represents zero or more occurrences of a pattern in the string.	"hello*"
+	It represents one or more occurrences of a pattern in the string.	"hello+"
{}	It represents exactly the specified number of occurrences.	"java{2}"
|	It represents either this or that character is present.	"java|python"
()	Capture and group(i.e group sub-patterns)	(a|b)xy
\	It represents the special sequence.	"\r"

- **[] - Square brackets**

Square brackets specify a **set** of characters you wish to match.

Expression	String	Matched?
[abc]	a	1 match
	ac	2 matches
	Hey Jude	No match
	abc de ca	5 matches

Here, [abc] will match if the string you are trying to match contains any of the a, b or c.

You can also specify a range of characters using **-** inside square brackets.

[a-e] is the same as [abcde]
[1-4] is the same as [1234]
[0-39] is the same as [01239]

You can complement (invert) the character set by using caret ^ symbol at the start of a square-bracket.

[^abc] means any character except a or b or c.
[^0-9] means any non-digit character.

- **. - Period**

A period matches any **single** character (except newline '\n').

Expression	String	Matched?
a…n	abn	No match
	alian	Match
	abysn	Match
	Alian	No match
	An abacus	No match

- **^ - Caret**

The caret symbol **^** is used to check if a string **starts** with a certain character.

Expression	String	Matched?
^a	a	Match
	abc	Match
	bac	No match
^ab	abc	1 match
	acb	No match (starts with a but not followed by b)

- **$ - Dollar**

The dollar symbol **$** is used to check if a string **ends** with a certain character.

Expression	String	Matched?
a$	a	1 match
	formula	1 match
	cab	No match

- *** - Star**

The star symbol ***** matches **zero or more occurrences** of the pattern left to it.

Expression	String	Matched?
ma*n	mn	Match
	man	Match
	maaan	Match
	main	No match (a is not followed by n)
	woman	Match

- **+ - Plus**

The plus symbol **+** matches **one or more occurrences** of the pattern left to it.

Expression	String	Matched?
ma+n	mn	No match (no a character)
	man	Match
	maaan	Match
	main	No match (a is not followed by n)
	woman	Match

- **{} - Braces**

Consider this code: {n,m}. This means at least n, and at most m repetitions of the pattern left to it.

Expression	String	Matched?
a{2,3}	abc dat	No match
	abc daat	1 match (at daat)
	aabc daaat	2 matches (at aabc and daaat)
	aabc daaaat	2 matches (at aabc and daaaat)

- **| - Alternation**

Vertical bar | is used for alternation (or operator).

Expression	String	Matched?
a|b	cde	No match
	ade	1 match (match at ade)
	acdbea	3 matches (at acdbea)

Here, a|b match any string that contains either a or b

- **() - Group**

Parentheses () is used to group sub-patterns. For example, (a|b|c)xz match any string that matches either a or b or c followed by xz

Expression	String	Matched?
(a|b|c)xz	ab xz	No match
	abxz	1 match (match at abxz)
	axz cabxz	2 matches (at axzbc cabxz)

- **\ - Backslash**

Backlash \ is used to escape various characters including all metacharacters.

For example,

\$a match if a string contains $ followed by a. Here, $ is not interpreted by a RegEx engine in a special way.

If you are unsure if a character has special meaning or not, you can put \ in front of it. This makes sure the character is not treated in a special way.

➢ **Special Sequences:**

A special sequence is a \ followed by one of the characters in the list below, and has a special meaning:

Character	Description
\A	It returns a match if the specified characters are present at the beginning of the string.
\d	It returns a match if the string contains digits [0-9].
\D	It returns a match if the string doesn't contain the digits [0-9].
\s	It returns a match if the string contains any white space character.
\S	It returns a match if the string doesn't contain any white space character.
\w	It returns a match if the string contains any word characters.
\W	It returns a match if the string doesn't contain any word.
\Z	Returns a match if the specified characters are at the end of the string.

\A - Matches if the specified characters are at the start of a string.

Expression	String	Matched?
\Athe	the sun	Match
	In the sun	No match

\d - Matches any decimal digit. Equivalent to [0-9]

Expression	String	Matched?
\d	12abc3	3 matches (at 12abc3)
	Python	No match

\D - Matches any non-decimal digit. Equivalent to [^0-9]

Expression	String	Matched?
\D	1ab34"50	3 matches (at 1ab34"50)
	1345	No match

\s - Matches where a string contains any whitespace character. Equivalent to [\t\n\r\f\v].

Expression	String	Matched?
\s	Python RegEx	1 match
	PythonRegEx	No match

\S - Matches where a string contains any non-whitespace character. Equivalent to [^ \t\n\r\f\v].

Expression	String	Matched?
\S	a b	2 matches (at a b)
		No match

\w - Matches any alphanumeric character (digits and alphabets). Equivalent to [a-zA-Z0-9_]. underscore _ is also considered an alphanumeric character.

Expression	String	Matched?
\w	12&": ;c	3 matches (at 12&": ;c)
	%"> !	No match

\W - Matches any non-alphanumeric character. Equivalent to [^a-zA-Z0-9_]

Expression	String	Matched?
\W	1a2%c	1 match (at 1a2%c)
	Python	No match

\Z - Matches if the specified characters are at the end of a string.

Expression	String	Matched?
\ZPython	I like Python	1 match
	I like Python	No match
	Python is fun.	No match

➢ **RegEx Module(re module):**

Python has a built-in package called **re**, which can be used to work with Regular Expressions.

To use it, we need to import the module.

import re

The **re** module offers a set of methods that allows us to search a string for a match, those are

❖ **re.match(pattern, string):**

This method finds match if it occurs at start of the string. For example, calling match() on the string 'python programming' and looking for a pattern 'python' will match. However, if we look for only programming, the pattern will not match.

match() is the first **re** module method and RE object (regex object) method. The match() function attempts to match the pattern to the string, starting at the beginning. If the match is successful, a match object is returned, but on failure, None is returned.

Example:

```
import re
# matching python in the given sentence
result = re.match('python', 'python programming and python')
print (result)

result = re.match('programming', 'python programming and python')
print ('\n Result :', result)

result = re.match('python', 'python programming and python')
print ('\n Starting position of the match :',result.start())
print ('Ending position of the match :',result.end())
```

Output:

```
 <re.Match object; span=(0, 6), match='python'>
Matching string : python
Result : None
Starting position of the match : 0
Ending position of the match : 6
```

The group() method of a match object can be used to show the successful match. Here is an example of how to use match() [and group ()]:

Example:

```
import re
# matching python in the given sentence
m1 = re.match('python', 'python programming and python')
m2 = re.match('programming', 'python programming and python')
if m1:
        print (m1.group())
else:
        print("No Match")

if m2:
        print (m2.group())
else:
        print("No Match")
```

Output:

```
python
No Match
```

❖ **re.search(*pattern*, *string*):**

It is similar to match() but it doesn’t restrict us to find matches at the beginning of the string only. The search() function searches the string for a match, and returns a Match object if there is a match.
If there is more than one match, only the first occurrence of the match will be return. If no matches are found, the value None is returned.

Unlike previous method, here searching for pattern ‘programming’ will return a match.

Example:

```
import re
result = re.search('python', 'python programming and python')
print (result)
result = re.search('programming', 'python programming and python')
print (result)
```

Output:

```
<re.Match object; span=(0, 6), match='python'>
<re.Match object; span=(7, 18), match='programming'>
```

Example:

```
import re
m1 = re.search('python', 'python programming and python')
m2 = re.search('programming', 'python programming and python')
if m1:
        print (m1.group())
else:
        print("No Match")

if m2:
        print (m2.group())
else:
        print("No Match")
```

Output:

```
python
programming
```

Example:

```
import re
m1 = re.match('programming', 'python programming and python')
m2 = re.search('programming', 'python programming and python')
if m1:
        print (m1.group())
else:
        print("No Match")
if m2:
        print (m2.group())
else:
        print("No Match")
```

Output:

```
No Match
Programming
```

Here you can see that, search() method is able to find a pattern from any position of the string but it only returns the first occurrence of the search pattern.

❖ **re.findall (*pattern*, *string*):**

It helps to get a list of all matching patterns. It has no constraints of searching from start or end. If we will use method findall to search a given string it will return all occurrence of that string. While searching a string, I would recommend you to use **re.findall()** always, it can work like re.search() and re.match() both.

The re.findall() method returns a list of all matches of a pattern within the string. It returns the patterns in the order they are found. If there are no matches, then an empty list is returned.

Example:
```
import re
ms = re.findall('python', 'python programming and python scripting')
if ms:
        print(ms)
else:
        print("No Match")
```

Output:
```
['python', 'python']
```

Example:
```
import re
str = 'python 23 program 363 script 37'
ptrn = '\d+'
result = re.findall(ptrn, str)
print(result)
```

Output:
```
['23', '363', '37']
```

Example:
```
import re
Nameage = '''
Kiran is 22 and Tarun is 33
Ganesh is 44 and Jony is 21'''
ages = re.findall('\d+', Nameage)
names = re.findall('[A-Z][a-z]*',Nameage)
x = 0
for eachname in names:
   print(eachname,":",ages[x])
   x+=1
```
Output:
```
Kiran : 22
Tarun : 33
Ganesh : 44
Jony : 21
```

Example:
```
import re
#Matching words with patterns
Str = "Sat, hat, mat, pat"
allStr = re.findall("[shmp]at", Str)
for i in allStr:
    print(i)
```
Output:
```
hat
mat
pat
```

Example:
```
import re
#Matching series of range of characters
Str = "sat, hat, mat, pat"
someStr = re.findall("[h-m]at", Str)
for i in someStr:
    print(i)
```
Output:
```
hat
mat
```

Example:
```
import re
 #Matching series of range of characters using ^
Str = "sat, hat, mat, pat"
someStr = re.findall("[^h-m]at", Str)
for i in someStr:
    print(i)
```
Output:
```
sat
pat
```

- **re.split(pattern, string, [maxsplit=0]):**

The split() function returns a list where the string has been split at each match. This method helps to split string by the occurrences of given pattern.
The re.split method splits the string where there is a match and returns a list of strings where the splits have occurred.

Example:
```
import re
# \s matches white spaces
resultls = re.split('\s', 'python programming and python scripting')
print(resultls)
#split with maxsplit
resultls = re.split('\s', 'python programming and python scripting',2)
print(resultls)
```
Output:
```
['python', 'programming', 'and', 'python', 'scripting']
['python', 'programming', 'and python scripting']
```

Example:
```
import re
string = 'Madhu:521,Naveen:532,Ganesh:509,Ramesh:569'
pattern = ':\d{3}'
result = re.split(pattern, string)
print(result)
```
Output:
```
['Madhu', ',Naveen', ',Ganesh', ',Ramesh', '']
```

❖ **re.sub(pattern, replace, string):**

The sub() function replaces the matches with the specified characters.i.e It helps to search a pattern and replace with a new sub string. If the pattern is not found, string is returned unchanged.

The method returns a string where matched occurrences are replaced with the content of replace variable.

Example:
```
import re
str='python programming and python scripting'
resultls = re.sub('python','java',str)
print(resultls)

str='python programming and python scripting'
resultls = re.sub('\s','-',str)
print(resultls)
```
Output:
```
java programming and java scripting
python-programming-and-python-scripting
```

Example:
```
import re
string = ''' abc \t 123
de 45 \n  f 678'''
# matches all whitespace characters
pattern = '\s+'
# empty string
replace = ''
new_string = re.sub(pattern, replace, string)
print(new_string)
```
Output:
```
abc123de45f678
```

Multithreading (or) Multithreaded Programming

❖ **Introduction:**

First, we have to know about Multitasking.

➢ Multitasking is a process of executing multiple tasks simultaneously, we use multitasking to utilize the CPU.

➢ Multitasking can be achieved by two ways or classified into two types
- Process-Based Multitasking(Multiprocessing)
- Thread-Based Multitasking(Multithreading)

➢ **Process-Based Multitasking(Multiprocessing):**

Executing multiple tasks simultaneously, where each task is separate independent process (or) program is called as process based multitasking.

Example:

✓ Typing a python program in notepad
✓ Listening audio songs
✓ Download a file from internet

The above three tasks are performed simultaneously in a system, but there is no dependence between one task and another task.

Process based multitasking is best suitable at "Operating System" level not at programming level.

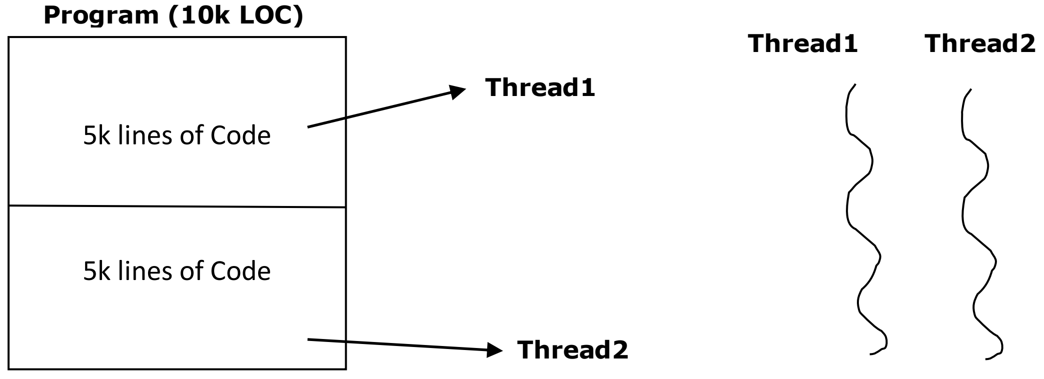
➢ **Thread-Based Multitasking(Multithreading):**

Executing multiple tasks simultaneously, where each task is separate independent part of process (or) program is called as thread based multitasking.

The each independent part is called as thread. The thread based multitasking is best suitable at programming level.

Example:

Let a program has 10k line of code, where last 5k lines of code doesn't depend on first 5k lines of code, then both are the execution simultaneously. So takes less time to complete the execution.



Note: Any type of multitasking is used to reduce response time of system and improves performance.

❖ Multithreading:

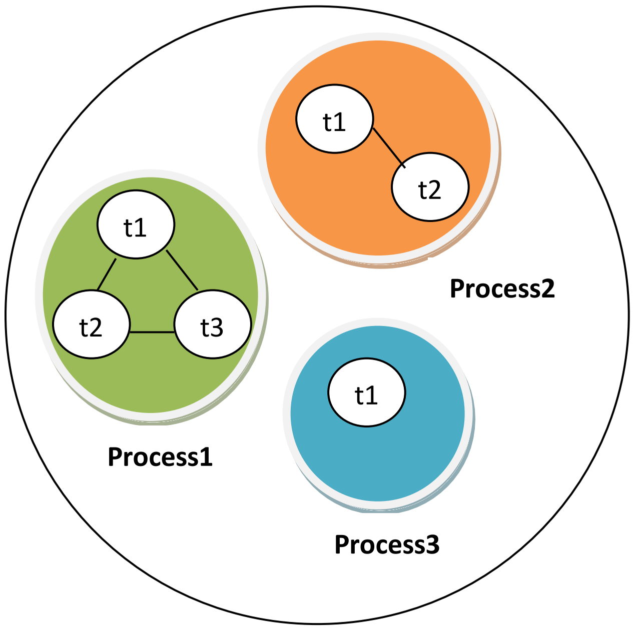
A thread is a lightweight process; also, it is the smallest unit of processing that can be performed in an OS (Operating System).

In simple words, a **thread** is a sequence of some instructions within a program that can be executed independently of other code.

- Threads are independent; if there is an exception in one thread it doesn't affect remaining threads.
- Threads shares common memory area.

In programming, 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.

A thread has a starting point, an execution sequence, and a result. It has an instruction pointer that holds the current state of the thread and controls what executes next in what order.



Operating System

As shown in the above figure, a thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS, and one process can have multiple threads.

Definition: Multithreading is a process of executing multiple threads simultaneously. Multithreading allows you to break down an application into multiple sub-tasks and run these tasks simultaneously.

In other words, the ability of a process to execute multiple threads parallelly is called multithreading. Ideally, multithreading can significantly improve the performance of any program.

Multiprocessing and Multithreading both are used to achieve multitasking, but we use multithreading than multiprocessing because threads shares a common memory area and context-switching between threads takes less time than process.

Advantages Of Multithreading

- Multithreading can significantly improve the speed of computation on multiprocessor or multi-core systems because each processor or core handles a separate thread concurrently.
- Multithreading allows a program to remain responsive while one thread waits for input, and another runs a GUI at the same time. This statement holds true for both multiprocessor and single processor systems.
- All the threads of a process have access to its global variables. If a global variable changes in one thread, it is visible to other threads as well. A thread can also have its own local variables.

Disadvantages Of Multithreading

- On a single processor system, multithreading won't hit the speed of computation. The performance may downgrade due to the overhead of managing threads.
- Synchronization is needed to prevent mutual exclusion while accessing shared resources. It directly leads to more memory and CPU utilization.
- It raises the possibility of potential deadlocks.
- It may cause starvation when a thread doesn't get regular access to shared resources. The application would then fail to resume its work.

The main application areas of multithreading are:

- Multimedia Movies
- Video Games
- Web Servers and Application Servers etc.

❖ Global Interpreter Lock (GIL):

Execution of Python code is controlled by the Python Virtual Machine. Python was designed in such a way that only one thread of control may be executing in this main loop, similar to how multiple processes in a system share a single CPU. Many programs may be in memory, but only one is live on the CPU at any given moment. Likewise, although multiple threads may be "running" within the Python interpreter, only one thread is being executed by the interpreter at any given time.

Access to the Python Virtual Machine is controlled by the global interpreter lock (GIL). This lock is what ensures that exactly one thread is running.

The Python Global Interpreter Lock or GIL, in simple words, is a mutex (or a lock) that allows only one thread to hold the control of the Python interpreter.

This means that only one thread can be in a state of execution at any point in time. The impact of the GIL isn't visible to developers who execute single-threaded programs, but it can be a performance bottleneck in CPU-bound and multi-threaded code.

Python uses reference counting for memory management. It means that objects created in Python have a reference count variable that keeps track of the number of references that point to the object. When this count reaches zero, the memory occupied by the object is released.

Let's take a look at a brief code example to demonstrate how reference counting works:

```
>>> import sys
>>> a = []
>>> b = a
>>> sys.getrefcount(a)
3
```

In the above example, the reference count for the empty list object [] was 3. The list object was referenced by a, b and the argument passed to sys.getrefcount ().

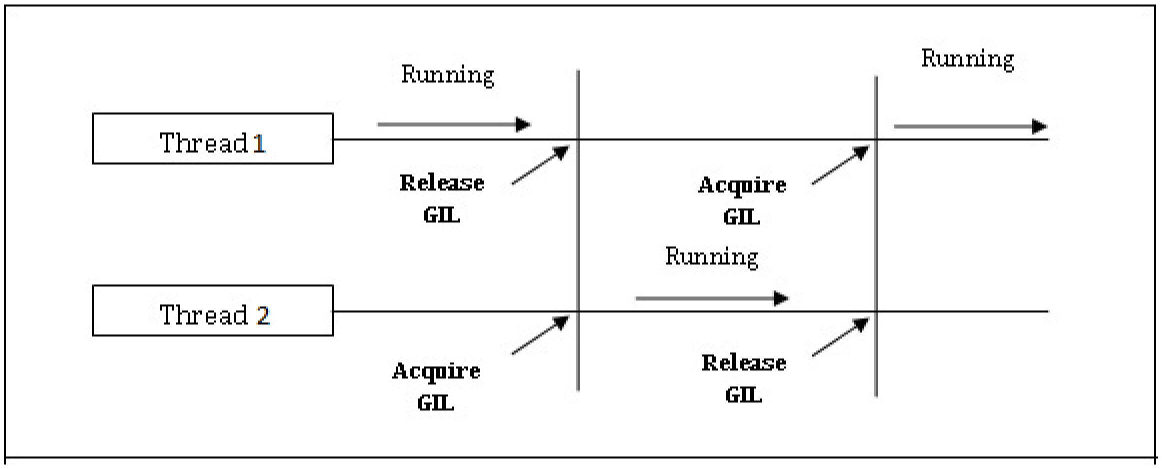
The problem was that this reference count variable needed protection from race conditions where two threads increase or decrease its value simultaneously. If this happens, it can cause either leaked memory that is never released or, even worse, incorrectly release the memory while a reference to that object still exists. This can cause crashes or other "weird" bugs in your Python programs.

This essentially means is a process can run only one thread at a time. When a thread starts running, it acquires GIL and when it waits for I/O, it releases the GIL, so that other threads of that process can run.

For e.g.,

Let us suppose process P1 has threads t1 and t2. Python threads are native threads that mean they are scheduled by the underlying operating system.

t1 running (acquire GIL) -> t1 waiting for I/O (releases GIL) -> t2 running (acquires GIL, by this time t1 is also ready but GIL is acquired by t2)



❖ **Multithreading Modules:**

Python offers two modules to implement threads in programs.

- thread module and
- threading module.

Note: For your information, 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(_thread):

This module provides low-level primitives for working with multiple threads (also called light-weight processes or tasks) — multiple threads of control sharing their global data space. For synchronization, simple locks (also called mutexes or binary semaphores) are provided.

Python 2.x used to have the <*thread*> module. But it got deprecated in Python 3.x and renamed to <*_thread*> module.

The <*_thread*> module supports one method to create thread. That is

- **thread.start_new_thread(function, args)**

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.

- **thread.get_ident()**

Return the 'thread identifier' of the current thread. This is a nonzero integer.

<u>Example:</u>

```
from _thread import start_new_thread,get_ident
from time import sleep

def num(n):
        for i in range(5):
                print(n)
        print(n,"thread identifier :",get_ident())
start_new_thread(num, ("hai", ))
start_new_thread(num, ("hello", ))
sleep(2)
print("threads are executed...")
```

<u>Output:</u>

```
>>>python multithr1.py
hai
hello
hai
hai
hai
hai
hai thread identifier : 9104
hello
hello
hello
hello
hello thread identifier : 6904
threads are executed...
```

❖ **Threading Module:**

The threading module provides more features and good support for threads than thread module. This module constructs higher-level threading interfaces on top of the lower level thread module.

This module combines all the methods of the <thread> module and provides few additional methods.

- **threading.activeCount():** It returns the total number of thread objects that are currently active.
- **threading.currentThread():** it returns current thread object, which is in the caller's thread control.
- **threading.enumerate():** It returns the complete list of thread objects that are currently active.

This Module also provides **Thread** class, and this thread class provide following methods

- **start()** – The start() method starts a thread by calling the run method.
- **join([time])** – The join() waits for threads to terminate.
- **isAlive()** – The isAlive() method checks whether a thread is still executing.
- **getName()** – The getName() method returns the name of a thread.
- **setName()** – The setName() method sets the name of a thread.

Creating Thread Using Threading Module

To implement a new thread using the threading module, use following code snippet

Syntax:

```
threading.Thread (group=None, target=None, name=None, args=())
```

This constructor has following arguments. Those are:

- **group** should be None; reserved for future extension when a ThreadGroup class is implemented.
- **target** is the callable function to be invoked by the run() method. Defaults to None, meaning nothing is called.
- **name** is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.
- **args** is the argument tuple for the function invocation. Defaults to ().

Example:

```
import threading
#function def
def display(msg):
        for i in range(5):
                print(msg)
# creating thread
t1 = threading.Thread(target=display, args=("Thread1",))
t2 = threading.Thread(target=display, args=("Thread2",))
# starting thread 1
t1.start()
```

```
# starting thread 2
t2.start()
# wait until thread 1 is completely executed
t1.join()
 # wait until thread 2 is completely executed
t2.join()
 # both threads completely executed
print("Done!")
```

Output:

```
>>>python multithr.py
Thread1
Thread1
Thread1
Thread1
Thread1
Thread2
Thread2
Thread2
Thread2
Thread2
Done!
```

Let us try to understand the above code:

- To import the threading module, we do:
  ```
  import threading
  ```

- To create a new thread, we create an object of Thread class. It takes following arguments:
  target: the function to be executed by thread
  args: the arguments to be passed to the target function

- In above example, we created 2 threads with different target functions:
  ```
  t1 = threading.Thread(target=display, args=("Thread1",))
  t2 = threading.Thread(target=display, args=("Thread2",))
  ```

- To start a thread, we use start method of Thread class.
  ```
  t1.start()
  t2.start()
  ```

- Once the threads start, the current program also keeps on executing. In order to stop execution of current program until a thread is complete, we use join method.
  ```
  t1.join()
  t2.join()
  ```

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

Example:

```
import threading

def sum(num1,num2):
	print("Thread : ",t1.name)
	print("Sum: {}".format(num1+num2))

def mul(num1,num2):
	print("Thread : ",t2.name)
	print("Mul : {}".format(num1 * num2))

t1 = threading.Thread(target=sum,name="Thread-1", args=(10,5))
t2 = threading.Thread(target=mul,name="Thread-2", args=(10,5))
t1.start()
t2.start()
t1.join()
t2.join()
print("Done!")
```

Output:

```
>>>python multithr.py
Thread :  Thread-1
Sum: 15
Thread :  Thread-2
Mul : 50
Done!
```