UNIT – III

Regular Expressions:

Agenda:

- **Regular Expression (RE):** Introduction,
- Special Symbols and Characters,
- REs and Python.

Examupdt.in

Regular Expression (RE):

- A regular expression is a series of characters used to search or find a pattern in a string.
- In other words, a regular expression is a special sequence of characters that form apattern.
- The regular expressions are used to perform a variety of operations like searching a substring in a string, replacing a string with another, splitting a string, etc.
- The Python programming language provides a built-in module re to work with regular expressions.
- There is a built-in module that gives us a variety of built-in methods to work with regular expressions. In Python, the regular expression is known as RegEx in shortform.

Special Symbols and Characters

Creating Regular Expression:

The regular expressions are created using the following.

- Metacharacters
- Special Sequences
- Sets

Metacharacters

- Metacharacters are the characters with special meaning in a regular expression. Thefollowing table provides a list of metacharacters with their meaning.

Examupdt.in

Metacharacters	Meaning
[ ]	Square brackets specifies a set of characters you wish to match.
\	Backlash \ is used to escape various characters including allmetacharacters.
.	A period matches any single character (except newline '\n')
^	The caret symbol ^ is used to check if a string starts with a certain character.
$	The dollar symbol $ is used to check if a string ends with a certain character.
*	The star symbol * matches zero or more occurrences of the pattern left to it.
+	The plus symbol + matches one or more occurrences of the pattern left to it.

{ }	Consider this code: {n,m}. This means at least n, and at most m repetitions of the pattern left to it.
|	Vertical bar | is used for alternation (or operator).
()	Parentheses () is used to group sub-patterns.
?	The question mark symbol ? matches zero or one occurrence of the pattern left to it.

Examupdt.in

[] - Square brackets

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

. - Period

Expression	String	Matched?
[abcd]	Cse	1 match
	cse ds	2 matches
	iot	No match
	cse ds aiml	3 matches

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

Expression	String	Matched?
..	C	No match
	Ds	1 match
	Cse	1 match
	Cseds	2 matches
	cse ds	3 matches(including space)

^ - Caret

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

Expression	String	Matched?
^c	C	1 match
	Cse	1 match
	Ds	No match
^ds	cse ds	1 match
	Cse	No match

Examupdt.in

$ - Dollar

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

Expression	String	Matched?
c$	C	1 match
	aimldsc	1 match
	Ds	No match

*** - Star**

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

Expression	String	Matched?
abc*	ab	1 match
	abc	1 match
	abcabc	2 match
	cse	No match

	Dsabc	1 match

Examupdt.in

+ - Plus

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

Expression	String	Matched?
ab+c	Ac	No match (no a character)
	Abc	1 match
	Abbbc	1 match
	Cse	No match (a is not followed by n)
	Dsabc	1 match

? - Question Mark

- **The question mark symbol ? matches zero or one occurrence of the pattern left to it.**

Expression	String	Matched?
ab?c	ac	1 match
	abc	1 match
	abbbc	No match
	abrc	No match
	cseabc	1 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)

() - 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**

Examupdt.in

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.**

```
>>>
print(re.findall(r'$a','dscse$
a'))[]
>>>
print(re.findall(r'\$a','dscse$
a'))['$a']
```

Special Sequences

- A special sequence is a character prefixed with \, and it has a special meaning. The following table gives a list of special sequences in Python with their meaning.

Special Sequences	Meaning
\A	Matches if the specified characters are at the start of a string.
\b	Matches if the specified characters are at the beginning or end of a word.
\B	Opposite of \b. Matches if the specified characters are not at the beginning or end of a word.
\d	Matches any decimal digit. Equivalent to [0-9]
\D	Matches any non-decimal digit. Equivalent to [^0-9]
\s	Matches where a string contains any whitespace character. Equivalent to [\t\n\r\f\v].
\S	Matches where a string contains any non-whitespace character. Equivalent to [^ \t\n\r\f\v].
\w	Matches any alphanumeric character (digits and alphabets). Equivalent to [a-zA-Z0-9_]. By the way, underscore _ is also considered an

Examupdt.in

	alphanumeric character.
\W	Matches any non-alphanumeric character. Equivalent to [^a-zA-Z0-9_]
\Z	Matches if the specified characters are at the end of a string.

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

Expression	String	Matched?
\Acse	cse ds	Match
	ds cse	No match

Examupdt.in

➧ **\b - Matches if the specified characters are at the beginning or end of a word.**

Expression	String	Matched?
\bcse	Cseaimlds	Match
	ds cse aimlds	Match
	Dscseaimlds	No match
ds\b	cse ds	Match
	cse ds aimlds	Match
	cse dsaimlds	No match

➧ **\B - Opposite of \b. Matches if the specified characters are not at the beginningor end of a word.**

Expression	String	Matched?
\Bcse	Cseaimlds	No match
	ds cseaimlds	No Match
	Dscseaimlds	Match
ds\B	cse ds	No match
	cse ds aimlds	No match
	cse dsaimlds	Match

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

Expression	String	Matched?
\d	12aimlds3	3 matches (at 12aimlds3)
	aimlds	No match

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

Expression	String	Matched?
\D	cseds123	5 matches (at cseds123)
	1345	No match

Examupdt.in

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

Expression	String	Matched?
\s	cse\tds\aimlds	2 match
	Cseaimlds	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_]. By the way, 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	1cse@ds	1 match (at 1cse@ds)
	aimlds	No match

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

Expression	String	Matched?
ds\Z	cse ds	1 match
	cse ds aiml	No match
	ds cse.	No match

Sets

- A set is a set character enclosed in [ ], and it has a special meaning. The followingtable gives a list of sets with their meaning.

Set	Meaning
[aeiou]	Matches with one of the specified characters are present
[d-s]	Matches with any lower case character from d to s
[^aeiou]	Matches with any character except the specified
[1234]	Matches with any of the specified digit
[3-8]	Matches with any digit from 3 to 8
[a-zA-Z]	Matches with any alphabet, lower or UPPER

Examupdt.in

- **[aeiou] Matches with one of the specified characters are present**

```
>>> print(re.findall(r'[aeiou]','cse and ds dept'))['e', 'a', 'e']
```

- **[d-s] Matches with any lower case character from d to s**

```
>>> print(re.findall(r'[a-h]','cse and ds dept'))['c', 'e', 'a', 'd', 'd', 'd', 'e']
```

- **[^aeiou] Matches with any character except the specified**

```
>>> print(re.findall(r'[^aeiou]','cse and ds dept'))['c', 's', ' ', 'n', 'd', ' ', 'd', 's', ' ', 'd', 'p', 't']
```

- **[1234] Matches with any of the specified digit**

```
>       e-2 cse-3 cse-4 ds'))['1', '2', '3', '4']
>
>

print(re.findall(r'[12345]','cse-1

cs
```

Examupdt.in

- **[a-zA-Z] Matches with any alphabet, lower or UPPER**

```
>>> print(re.findall(r'[a-zA-Z]','This is Cse and Ds Dept'))
['T', 'h', 'i', 's', 'i', 's', 'C', 's', 'e', 'a', 'n', 'd', 'D', 's', 'D', 'e', 'p', 't']
```

REs and Python

Built-in methods of re module

The re module provides the following methods to work with regular expressions.

1. match()
2. search()
3. findall()
4. finditer()
5. sub()
6. split()
7. compile()

1. match() in Python:

- We can use match function to check the given pattern at beginning of target string. If the match is available then we will get Match object, otherwise we will get None.
- The re.match() method will start matching a regex pattern from the very first character of the text, and if the match found, it will return a re.Match object. Later we can use the re.Match object to extract the matching string.

Syntax of re.match(): re.match(pattern, string, flags=0)

- The regular expression pattern and target string are the mandatory arguments, and flags are optional.
- **pattern:** The regular expression pattern we want to match at the beginning of the target string. Since we are not defining and compiling this pattern beforehand (like the compile method). The practice is to write the
- actual pattern using a raw string. **string:** The second argument is the variable pointing to the target string (In which we want to look for occurrences of the pattern).
- **flags:** Finally, the third argument is optional and it refers to regex flags by default no flags are applied.

Eg:

Examupdt.in

```
import
re
pattern
=
'\Aiitk'
test_stri
ng =
'iitkcse'
result =
re.matc
h(patter
n,
test_stri
ng)
if
result:

print("S
earch
success
ful.")
else:
print("S
earch
unsucc
essful.")
```

output:
Search success ful.

This re.Matc h object contains the followin g items.

Examupdt.in

- A span attribute that shows the locations at which the match starts and ends. i.e., isthe tuple object contains the start and end index of a successful match.Save this tuple and use it whenever you want to retrieve a matching string from the target string
- Second, A match attribute contains an actual match value that we can retrieve usinga group() method.
- The Match object has several methods and attributes to get the information about the matching string. Let's see those.

Method	Description
group()	Return the string matched by the regex
start()	Return the starting position of the match
end()	Return the ending position of the match
span()	Return a tuple containing the (start, end) positions of the match.

```
>>> res=re.match(r'\w{4}',str)
>>> res
<re.Match object; span=(0, 4), match='This'>
>>>
res.group()
'This'
>>>
res.start()0
>>>
res.end()4
>>>
res.span()
(0, 4)
```

Examupdt.in

search( ) in Python:

- Python regex re.search() method looks for occurrences of the regex pattern inside the entire target string and returns the corresponding Match Object instance where the match found.
- **Syntax:re.search(pattern, string, flags=0)**

program:

```
import re
string = 'iitk 01 iitd 2 iitm 03'
match=re.search('iitk',string)
if match:
    print('iitk is present')
```

```
else:
    print('iitk is not present')
```

output: **iitk is present**

Examupdt.in

findall( ) in Python:

- The RE module's re.findall() method scans the regex pattern through the entire target string and returns all the matches that were found in the form of a Python list.
- **Syntax:re.findall(pattern, string, flags=0)**
- **program**

```
import re
string = 'we are aiml 66 ds 67 students belongs to cse 05'
pattern = '\d+'
result = re.findall(pattern, string)
print(result)
```

output: ['66', '67', '05']

finditer() in python:

- The re.finditer() works exactly the same as the re.findall() method except it returns an iterator yielding match objects matching the regex pattern in a string instead of a list. It scans the string from left-to-right, and matches are returned in the iterator form. Later, we can use this iterator object to extract all matches.
- In simple words, finditer() returns an iterator over MatchObject objects.

```
import re
res=re.finditer(r'\b\w{3}\b','cse
ds raj')for match in res:
  print(match.group())
print(re.findall(r'\b\w{3}\b','cse
ds raj'))
```

outPut:

```
cs
e
ra
j
['cse', 'raj']
```

Program:

```
import re
itr=re.finditer("[a-z]","a7b9c5k8z")
for m in itr:
   print(m.start(),"...",m.end(),"...",m.group())
```

outPut:

Examupdt.in

6 ... 7 ... k
0 ... 1 ... a 8 ... 9 ... z
2 ... 3 ... b
4 ... 5 ... c

sub( ) in Python:

- The sub( ) method of re object replaces the match pattern with specified text in a string.
- The syntax of sub( ) method is **sub(pattern, text, string).**
- The sub( ) method does not modify the actual string instead, it returns the modified string as a new string.

Program:

```
import re
 # multiline string
string = 'iitk 01 iitd 2 iitm 03'
# matches all whitespace characters
pattern = '\s+'
 # empty string
replace =' '
new_string = re.sub(pattern, replace, string)
print(new_string)
```

Output:

iitk 01 iitd 2 iitm 03

Examupdt.in

Examupdt.in

split( ) in Python

- The Pythons re module's re.split() method split the string by the

occurrences of theregex pattern, returning a list containing the resulting substrings.

Examupdt.in

- **Syntax:re.split(pattern, string, maxsplit=0, flags=0)**
- The regular expression pattern and target string are the mandatory arguments. Themaxsplit, and flags are optional.
- **pattern:** the regular expression pattern used for splitting the target string.
- **string:** The variable pointing to the target string (i.e., the string we want
- to split). **maxsplit:** The number of splits you wanted to perform. If maxsplit is nonzero, atmost maxsplit splits occur, and the remainder of the string is returned as the finalelement of the list.
- **flags:** By default, no flags are applied.

```
print(re.split('\.','http://www.google.com/'))
['http://www', 'google', 'com/']

str=" cse and ds depts
in aiml'
print(re.split(r'\s+',str))
['cse', 'and', 'ds', 'depts', 'in', 'aiml']
```

```
import re
string = 'aiml:66 ds:67 cse:05.'
pattern = '\d+'
result = re.split(pattern, string, 1)
print(result)
```

Output:

```
['aiml:', ' ds:67 cse:05.']
```

compile() in python:

- Python's re.compile() method is used to compile a regular expression patternprovided as a string into a regex pattern object (re.Pattern).
- Later we can use this pattern object to search for a match inside different targetstrings using regex methods such as a re.match() or re.search().
- In simple terms, We can compile a regular expression into a regex object to look foroccurrences of the same pattern inside various target strings without rewriting it.

Syntax of re.compile()

- re.compile(pattern, flags=0)
- **pattern:** regex pattern in string format, which you are trying to match inside thetarget string.
- **flags:** The expression's behavior can be modified by specifying regex flag values.This is an optional parameter

```
>>> pattern=re.compile(r'\b\w{4}\b')
>>> result=patten.findall('abcd raaj')
```

Examupdt.in

```
>>> result=pattern.findall('abcd raaj')
>>> result
['abcd', 'raaj']
>>> print(re.findall(pattern,'abcd
raaj'))['abcd', 'raaj']
>>>
```

match.re and match.string

The re attribute of a matched object returns a regular expression object.
Similarly, string attribute returns the passed string.

```
>>> match.re
re.compile('(\\d{3}) (\\d{2})')

>>> match.string
'39801 356, 2102 1111'
```

Using r prefix before RegEx

When r or R prefix is used before a regular expression, it means raw string. For example, '\n' is a new line whereas r'\n' means two characters: a backslash \ followed by n.
Backlash \ is used to escape various characters including all metacharacters. However, using r prefix makes \ treat as a normal character.

Program:

```
import re

string = '\n and \r are escape sequences.'

result = re.findall(r'[\n\r]', string)
print(result)
```

Output:

```
['\n', '\r']
```

re.subn()

The re.subn() is similar to re.sub() except it returns a tuple of 2 items containing the new string and the number of substitutions made.

Examupdt.in

```
import re
 # multiline string
string = 'iitk 01 iitd 2 iitm 03'
# matches all whitespace characters
pattern = '\s+'
 # empty string
replace =' '
new_string = re.subn(pattern, replace, string)
print(new_string)
```

output:

('iitk 01 iitd 2 iitm 03', 5)

Examupdt.in

Examples

Example 1: Write a regular expression to search digit inside a string

```
import re
str="The total no of students are120"
res=re.findall(r'\d',str)
print(res)
```

output:

```
['1', '2', '0']
```

Example2: match 3-letter word anywhere in the string

```
str=' civil cse eee ece ds iot ai&ml cs mech' 
print(re.findall(r'\w{3}',str))
```

```
[ 'civ', 'cse', 'eee', 'ece', 'iot', 'mec']
```

Example 3 : Extract all characters from the paragraph using Python Regular Expression.

```
import re
str="The total no of students are120"
print(re.findall(r'.',str))
```

```
['T', 'h', 'e', ' ', 't', 'o', 't', 'a', 'l', ' ', 'n', 'o', ' ', 'o', 'f', ' ', 's', 't', 'u', 'd', 'e', 'n', 't', 's', '', 'a', 'r', 'e', '1', '2', '0']
```

Example 4: Extract all of the words and numbers

```
import re
str="The total no of students are120"
print(re.findall(r'\w+',str))
```

```
['The', 'total', 'no', 'of', 'students', 'are120']
```

Example 5: Extract only numbers

```
import re
str="The total no of students are120"
print(re.findall(r'\d+',str))
```

```
['120']
```

Examupdt.in

Example 6: Extract the beginning word

```
import re
```

Examupdt.in

```
str="The total no of students
are120"
print(re.findall(r'^\w+',str))
```

['The']

Example 7: Extract first two characters from each word (not

```
the numbers)import re
str="The total no of students
are120"
print(re.findall(r'\b[a-zA-
Z].',str))
```

['Th', 'to', 'no', 'of', 'st', 'ar']

Example 8: Find out all of the words, which start with

```
a vowel.import re
str="The total no of students
are120"
print(re.findall(r'\b[aeiou]\w
+',str))
```

['of', 'are120']

Example 9:Extract date from the string

```
import re
str='Today date is June 09,
2021.'
pattern=r'(\w+)(\s)(\d+)([,]\s)
(\d+)'
print(re.findall(pattern,str))
```

[('June', ' ', '09', ', ', '2021')]

```
import re
str='Today date is 06-09-
2021'
pattern=r'(\d+)(.)(\d+)(.)(\d+
)'
```

Examupdt.in

```
print(re.findall(pattern,str))
```

```
[('06', '-', '09', '-', '2021')]
```

```
import re
str='Today date is 06-09-2021'
match=re.search(r'\d{2}-\d{2}-
\d{4}',str)
```

Examupdt.in

```
print(match.group())
```

06-09-2021

Example 10:Extract date from the string

```
import re
str = "Please contactus at aimlds@gmail.com for
furtherinformation."
email = re.findall(r"[a-z0-9\.\-+_]+@[a-z0-9\.\-
+_]+\.[a-z]+", str)print(email)
```

[aimlds @gmail.com']

Example 11:Write a Python program that matches a string that has an a followed by zero or more b's.

```
import re
def match(str):
    pattern = 'ab*?'
    if
        re.search(pattern,s
        tr): return 'Found
        a match!'
    else:
        return('Not matched!')

print(match("ac"))
print(match("abc"
))
print(match("abbc
"))
print(match("abbb
c"))
print(match("$12"
))
```

```
Found a
match!
Found a
match!
Found a
match!
Found a
```

Examupdt.in

```
match!  Not
matched!
```

Example 12:Replace maximum 2 occurrences of space, comma, or dot with a colon

```
import re
text = 'CSE DS, AIML.'
print(re.sub("[ ,.]", ":", text, 2))
```

Examupdt.in

CSE:DS: AIML.

Example 13:Develop a Python program to match a string that contains only upper and lowercase letters, numbers, and underscores.

```
import re
str = 'aiml_1254'
pattern='^[a-zA-Z0-
9_]*$'
print(re.findall(patter
n,str))
```

['aiml_1254']

Example 14: write python program by using sub() method

```
import re
 # multiline string
string = 'iitk 01 iitd 2 iitm 03'
# matches all whitespace characters
pattern = '\s+'
 # empty string
replace =' '
new_string = re.sub(pattern, replace, string)
print(new_string)
```

Examupdt.in

output:
iitk 01 iitd 2 iitm 03

Example 15: write python program by using group() method

```
import re
string ='39801 356, 2102 1111'
 # Three digit number followed by space followed by two digit number
pattern = '(\d{3}) (\d{2})'
# match variable contains a Match object.
match = re.search(pattern, string)
if match:
    print(match.group())
```

```
else:
    print("pattern not found")
```

Output:

801 35

Examupdt.in

Multithreaded Programming
Introduction
Threads and Processes
Python, Threads, and the Global Interpreter Lock
Thread Module
Threading Module
Related Modules

Thread

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

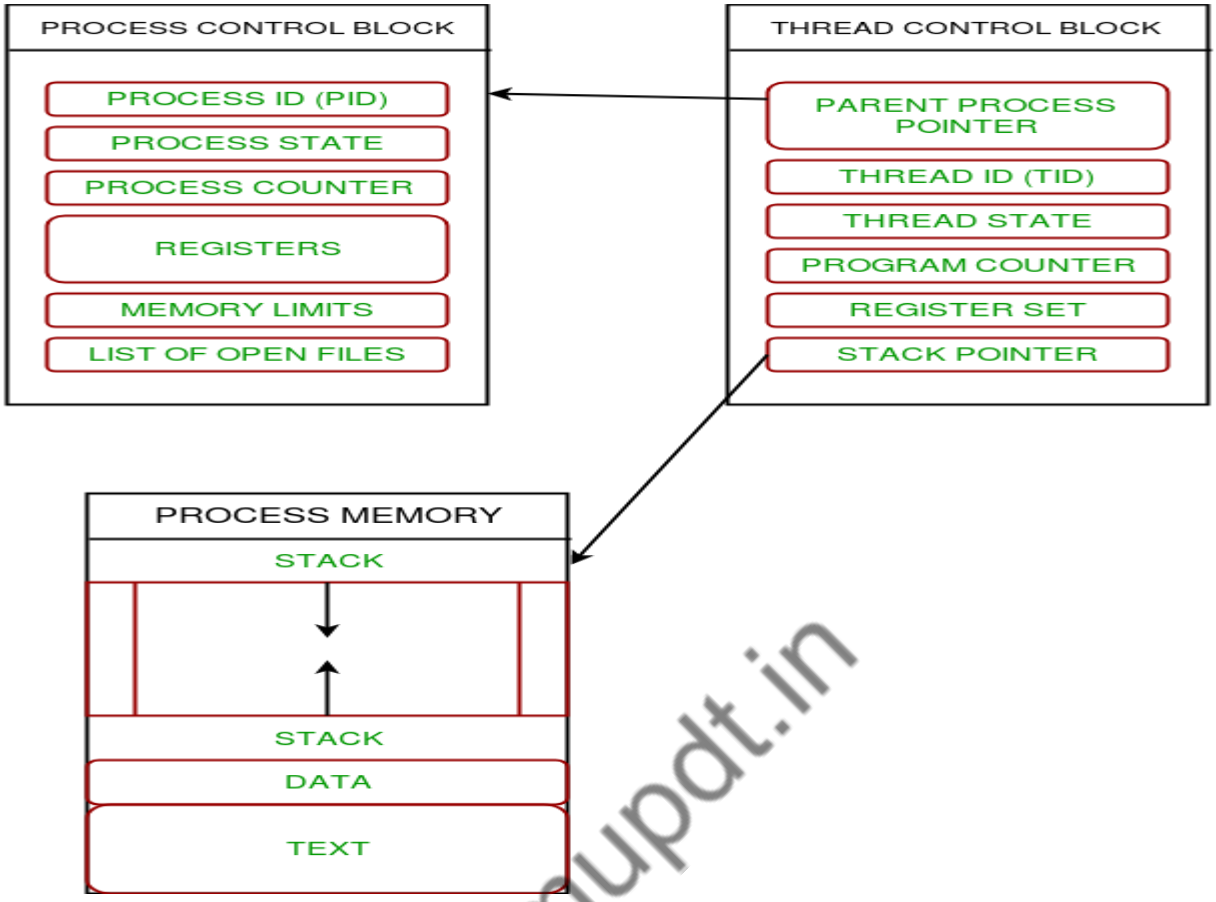
- An executable program.
- The associated data needed by the program (variables, work space, buffers, etc.)
- The execution context of the program (State of process)

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

A thread contains all this information in a **Thread Control Block (TCB)**:

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

Examupdf.in

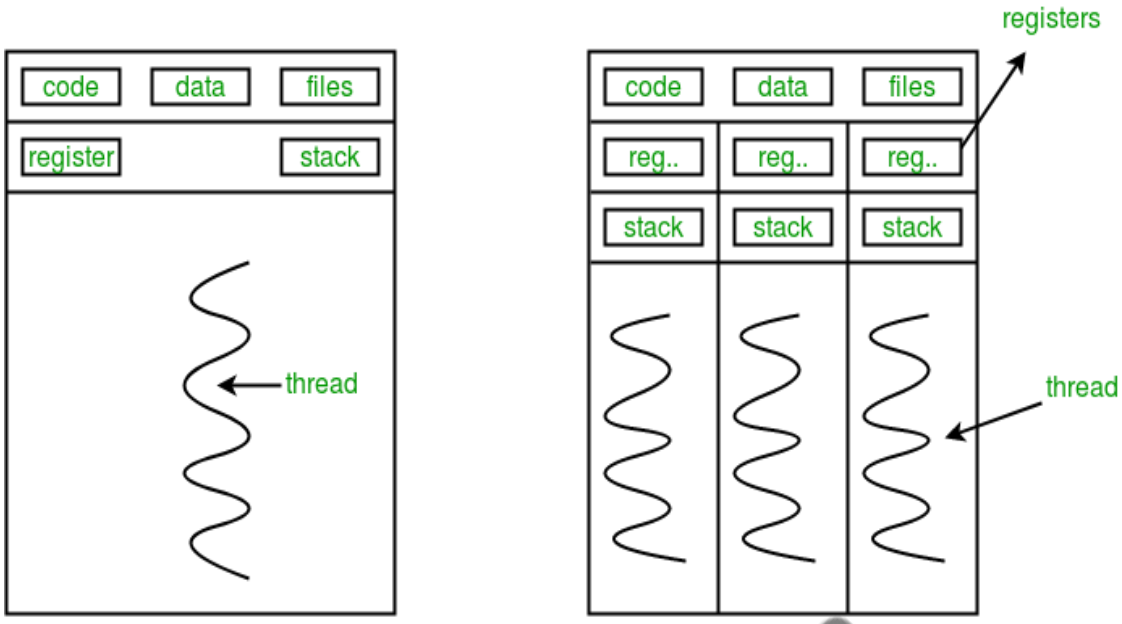


Examupdt.in

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:



single-threaded process multithreaded process

Examupdt.in

Threads and Processes

A *process* is a program in execution. Each process has its own address space, memory, a data stack, and other auxiliary data to keep track of execution. The operating system manages the execution of all processes on the system, dividing the time fairly between all processes.

Global Interpreter Lock (GIL) in python is a process lock or a mutex used while dealing with the processes. It makes sure that one thread can access a particular resource at a time and it also prevents the use of objects and bytecodes at once. This benefits the single-threaded programs in a performance increase. GIL in python is very simple and easy to implement.
A lock can be used to make sure that only one thread has access to a particular resource at a given time.

One of the features of Python is that it uses a global lock on each interpreter process, which means that every process treats the python interpreter itself as a resource.

For example, suppose you have written a python program which uses two threads to perform both CPU and 'I/O' operations. **When you execute this program, this is what happens:**

1. **The python interpreter creates a new process and spawns the threads**
2. **When thread-1 starts running, it will first acquire the GIL and lock it.**

3. **If thread-2 wants to execute now, it will have to wait for the GIL to be released even if another processor is free.**
4. **Now, suppose thread-1 is waiting for an I/O operation. At this time, it will release the GIL, and thread-2 will acquire it.**
5. **After completing the I/O ops, if thread-1 wants to execute now, it will again have to wait for the GIL to be released by thread-2.**

Python Multithreading

Multithreading is defined as the ability of a processor to execute multiple threads concurrently.

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.

Examupdt.in

Using an MT program with a shared data structure such as a Queue (a multithreaded queue data
structure discussed later in this chapter), this programming task can be organized with a few threads
that have specific functions to perform:

● **UserRequestThread**: Responsible for reading client input, perhaps from an I/O channel. A number of threads would be created by the program, one for each current client, with requests being entered into the queue.
● **RequestProcessor**: A thread that is responsible for retrieving requests from the queue and processing them, providing output for yet a third thread.
● **ReplyThread**: Responsible for taking output destined for the user and either sending it back, if in a networked application, or writing data to the local file system or database.

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.

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

.

1. The thread module
2. The threading module

Thread modules

The **thread module** also provides a basic synchronization data structure called a *lock object* (aka primitive lock, simple lock, mutual exclusion lock, mutex, binary semaphore).

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.

Examupdt.in

thread Module and Lock Objects

Function/Method	**Description**
tHRead ***Module Functions***	
start_new_thread(function, args, kwargs=None)	Spawns a new thread and execute function with the given args and optional kwargs
allocate_lock()	Allocates **LockType** lock object
exit()	Instructs a thread to exit
LockType Lock Object Methods	
acquire(wait=**None**)	**Attempts to acquire lock object**
locked()	**Returns True if lock acquired, False otherwise**
release()	**Releases lock**

The key function of the thread module is start_new_thread(). Its syntax is exactly that of the apply() built-in function, taking a function along with arguments and optional keyword arguments.

program:

```
import threading

def coder(number):
    print('Coders:  %s'%number)
    return

threads = []
for k in range(5):
    t = threading.Thread(target=coder, args=(k,))
    threads.append(t)
    t.start()
```

Output:

```
Coders:  0Coders:  1Coders:  2Coders:  3Coders:  4
```

Examupdt.in

Program: write a python program by using thread module

```
import thread # import the thread module
import time # import time module

def cal_sqre(num): # define the cal_sqre function
    print(" Calculate the square root 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 threads is :", time.time() - t1) # print the total time
```

Output:

```
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 threads is : 3.7105774879455566
```

Examupdt.in

To create a thread using the threading module, you must do the following:

1. Create a class which extends the **Thread** class.
2. Override its constructor (__init__).
3. Override its **run()** method.
4. Create an object of this class.

A thread can be executed by calling the **start()** method.

The **join()** method can be used to block other threads until this thread (the one on which join was called) finishes execution.

A race condition occurs when multiple threads access or modify a shared resource at the same time.

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.

threading Module Objects

threading Module Objects	Description
Thread	Object that represents a single thread of execution
Lock	Primitive lock object (same lock object as in the **thread** module)
RLock	Re-entrant lock object provides ability for a single thread to (re)acquire an already-held lock (recursive locking)
Condition	Condition variable object causes one thread to wait until a certain "condition" has been satisfied by another thread, such as changing of state or of some data value EventGeneral version of condition variables whereby any number of threads are waiting for some event to occur and all will awaken when the event happens
Semaphore	Provides a "waiting area"-like structure for threads waiting on a lock
BoundedSemaphore	Similar to a **Semaphore** but ensures it never exceeds its initial value
Timer	Similar to **Thread** except that it waits for an allotted period of time before running

Examupdt.in

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.

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.

Examupdt.in

Program: Write a python program by using threading module

```
from threading import *
print(current_thread().getName())
current_thread().setName("aimlds")
print(current_thread().getName())
print(current_thread().name)
```

Output:

```
aimlds
aimlds
```

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:

1. t1.start()
2. t2.start()

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.

Program: Write a python program by using join method

```
import threading
def aimlds(n):
    print('we are aimlds students',n)
T1=threading.Thread(target=aimlds,args=(117,))
T1.start()
T1.join()
print('we are practicing python programs')
```

Output:

```
we are aimlds students 117
we are practicing python programs
```

Examupdt.in

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

Program: Write a python program by using threading module

```
from threading import *
import time
def display():
    for i in range(10):
        print("aimlds Thread")
time.sleep(2)

t=Thread(target=display)
t.start()
t.join()#This Line executed by Main Thread
for i in range(10):
    print("se cs iot cse Thread")
```

Output:

```
aimlds Thread
aimlds Thread
```

aimlds Thread
aimlds Thread
aimlds Thread
aimlds Thread
aimlds Thread
aimlds Thread
aimlds Thread
aimlds Thread
se cs iot cse Thread
se cs iot cse Thread
se cs iot cse Thread
se cs iot cse Thread
se cs iot cse Thread
se cs iot cse Thread
se cs iot cse Thread
se cs iot cse Thread
se cs iot cse Thread
se cs iot cse Thread

Setting and Getting Name of a Thread:

Every thread in python has name. It may be default name generated by Python or Customized Name provided by programmer.

We can get and set name of thread by using the following

Thread class methods. t.getName() □ Returns Name of Thread
t.setName(newName) □ To set our own name

Examupdt.in

Note: Every Thread has implicit variable "name" to represent name of Thread.

Eg:

```
1) from threading import *
2) print(current_thread().getName())
3) current_thread().setName("Pawan Kalyan")
4) print(current_thread().getName())
5) print(current_thread().name)
```

Output:
MainThread
Pawan Kalyan
Pawan Kalyan

Thread Identification Number (ident):

For every thread internally a unique identification number is available. We can access this id by using implicit variable "ident"

```
1) from threading import *
2) def test():
3)   print("Child Thread")
4) t=Thread(target=test)
                t.start()
6) print("Main Thread Identification Number:",current_thread().ident)
7)  print("Child Thread Identification Number:",t.ident)
```

Output:

```
Child Thread
Main Thread Identification
Number: 2492 Child Thread
Identification Number: 2768
```

active_count():

This function returns the number of active threads currently running.

Eg:

```
1) from threading import *
2) import time
3) def display():
4)    print(current_thread().getName(),"...started")
5)    time.sleep(3)
6)    print(current_thread().getName(),"...ended")
7) print("The Number of active Threads:",active_count())
8) t1=Thread(target=display,name="ChildThread1")
9) t2=Thread(target=display,name="ChildThread2")
10) t3=Thread(target=display,name="ChildThread3")
11) t1.start()
12) t2.start()
13) t3.start()
14) print("The Number of active Threads:",active_count())
15) time.sleep(5)
16) print("The Number of active Threads:",active_count())
```

Output:

```
D:\python_classes>py
test.py The Number of
active Threads: 1
ChildThread1 ...started
ChildThread2 ...started
ChildThread3 ...started
The Number of active
```

Examupdt.in

```
Threads: 4 ChildThread1
...ended ChildThread2
...ended ChildThread3
...ended
The Number of active Threads: 1
```

enumerate() function:

This function returns a list of all active threads currently running.

Eg:

```
) from threading import *
2) import time
3) def display():
4)     print(current_thread().getName(),"...started")
5)     time.sleep(3)
6)     print(current_thread().getName(),"...ended")
7)   t1=Thread(target=display,name="ChildThread1")
8) t2=Thread(target=display,name="ChildThread2")
9)   t3=Thread(target=display,name="ChildThread3")
10) t1.start()
11) t2.start()
12) t3.start()
13) l=enumerate()
14) for t in l:
15)   print("Thread Name:",t.name)
16) time.sleep(5)
17) l=enumerate()
18) for t in l:
19)   print("Thread Name:",t.name)
```

Examupdate.in

Output:

```
D:\python_classes>py
test.py ChildThread1
...started ChildThread2
...started ChildThread3
...started Thread Name:
MainThread Thread
Name: ChildThread1
Thread Name:
ChildThread2 Thread
Name: ChildThread3
ChildThread1 ...ended
ChildThread2 ...ended
ChildThread3  ...ended
Thread Name: MainThread
```

isAlive():

isAlive() method checks whether a thread is still executing or not.

Eg:

```
1) from threading import *
2) import time
3) def display():
4)   print(current_thread().getName(),"...started")
5)   time.sleep(3)
6)   print(current_thread().getName(),"...ended")
7) t1=Thread(target=display,name="ChildThread1")
8) t2=Thread(target=display,name="ChildThread2")

9) t1.start()
10) t2.start()
11)
12) print(t1.name,"is Alive :",t1.isAlive())
13) print(t2.name,"is Alive :",t2.isAlive())
14) time.sleep(5)
15) print(t1.name,"is Alive :",t1.isAlive())
16) print(t2.name,"is Alive :",t2.isAlive())
```

Output:

```
D:\python_classes>py
test.py ChildThread1
...started ChildThread2
...started ChildThread1 is
Alive : True ChildThread2
is Alive : True
ChildThread1 ...ended
ChildThread2 ...ended
ChildThread1 is Alive :
False ChildThread2 is
Alive : False
```

Examupdt.in

Daemon threads

The threads which are always going to run in the background that provides supports to main or non-daemon threads, those background executing threads are considered as **Daemon Threads.**

The **Daemon Thread** does not block the main thread from exiting and continues to run in the background.

The threads which are running in the background are called Daemon Threads.

The main objective of Daemon Threads is to provide support for Non Daemon Threads(like main thread)

Eg: Garbage Collector

Whenever Main Thread runs with low memory, immediately PVM runs Garbage Collector to destroy useless objects and to provide free memory,so that Main Thread can continue its execution without having any memory problems.

We can check whether thread is Daemon or not by using t.isDaemon() method of Thread class or by using daemon property.

Evaluupdt.in

Eg:

```
1) from threading import *
2) print(current_thread().isDaemon()) #False
3) print(current_thread().daemon) #False
```

We can change Daemon nature by using setDaemon() method of Thread class. t.setDaemon(True)
But we can use this method before starting of Thread.i.e once thread started,we cannot change its Daemon nature,otherwise we will get RuntimeException:cannot set daemon status of active thread

Eg:

```
1) from threading import *
        print(current_thread().isDaemon())
1)          current_thread().setDaemon(True)
```

RuntimeError: cannot set daemon status of active thread

Default Nature:

By default Main Thread is always non-daemon.But for the remaining threads Daemon nature will be inherited from parent to child.i.e if the Parent Thread is Daemon then child thread is also Daemon and if the Parent Thread is Non Daemon then ChildThread is also Non Daemon.

Eg:

```
1)  from threading import *
2) def job():
3)    print("Child Thread")
4) t=Thread(target=job)
5)   print(t.isDaemon())#False
6) t.setDaemon(True)
7)   print(t.isDaemon()) #True
```

Note: Main Thread is always Non-Daemon and we cannot change its Daemon Nature b'z it is already started at the beginning only.

Whenever the last Non-Daemon Thread terminates automatically all Daemon Threads will be terminated.

Eg:

```
1)  from threading import *
2)  import time
3)  def job():
4)    for i in range(10):
5)      print("Lazy Thread")
6)      time.sleep(2)
7)
8)  t=Thread(target=job)
9)  #t.setDaemon(True)===>Line-1
10) t.start()
11) time.sleep(5)
12) print("End Of Main Thread")
```

Examupdt.in

In the above program if we comment Line-1 then both Main Thread and Child Threads are Non Daemon and hence both will be executed until their completion.
In this case output is:

Lazy Thread

Lazy Thread

Lazy Thread

End Of Main Thread Lazy Thread
Lazy Thread
Lazy Thread
Lazy Thread

Lazy Thread
Lazy Thread
Lazy Thread

If we are not commenting Line-1 then Main Thread is Non-Daemon and Child Thread is Daemon. Hence whenever MainThread terminates automatically child thread will be terminated. In this case output is

Lazy Thread
Lazy Thread
Lazy Thread
End of Main Thread

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.

Examupdate.in

It can be avoided by Synchronizing threads.

Python supports 6 ways to synchronize threads:

1. Locks
2. RLocks
3. Semaphores
4. Conditions
5. Events, and
6. Barriers

- Locks allow only a particular thread which has acquired the lock to enter the critical section.
- A Lock has 2 primary methods:

1. **acquire()**: It sets the lock state to **locked.** If called on a locked object, it blocks until the resource is free.
2. **release()**: It sets the lock state to **unlocked** and returns. If called on an unlocked object, it returns false.

- The global interpreter lock is a mechanism through which only 1 CPython interpreter process can execute at a time.

- It was used to facilitate the reference counting functionality of CPythons's garbage collector.
- To make Python apps with heavy CPU-bound operations, you should use the multiprocessing module.

Using Locks

- **threading** module provides a **Lock** class to deal with the race conditions. Lock is implemented using a **Semaphore** object provided by the Operating System. **Lock** class provides following methods:

1. **acquire([blocking]) :** To acquire a lock. A lock can be blocking or non-blocking.
   - When invoked with the blocking argument set to **True** (the default), thread execution is blocked until the lock is unlocked, then lock is set to locked and return **True**.
   - When invoked with the blocking argument set to **False**, thread execution is not blocked. If lock is unlocked, then set it to locked and return **True** else return **False** immediately.

2. **release() :** To release a lock.
   - When the lock is locked, reset it to unlocked, and return. If any other threads are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed.
   - If lock is already unlocked, a **ThreadError** is raised.

Examupdt.in

Advantages:

- It doesn't block the user. This is because threads are independent of each other.
- Better use of system resources is possible since threads execute tasks parallely.
- Enhanced performance on multi-processor machines.
- Multi-threaded servers and interactive GUIs use multithreading exclusively.

Program: Write a python program by using lock()

```
#!/usr/bin/env python
```

```
import thread
from time import sleep, ctime

loops = [4,2]

def loop(nloop, nsec, lock):
    print('start loop', nloop, 'at:', ctime())
    sleep(nsec)
    print('loop', nloop, 'done at:', ctime())
    lock.release()

def main():
    print('starting at:', ctime())
    locks = []
    nloops = range(len(loops))

    for i in nloops:
        lock = thread.allocate_lock()
        lock.acquire()
        locks.append(lock)

    for i in nloops:
        thread.start_new_thread(loop,(i, loops[i], locks[i]))

    for i in nloops:
        while locks[i].locked(): pass
    print('all DONE at:', ctime())

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

Examupdt.in

Output:

```
Thu Feb 10 16:48:16 2022

'all DONE  Thu Feb 10 16:48:16 2022
```

To overcome this problem, we should go for RLock(Reentrant Lock). Reentrant means the thread can acquire the same lock again and again.If the lock is held by other threads then only the thread will be blocked. Reentrant facility is available only for owner thread but not for other threads.

Eg:

```
1) from threading import *
2) l=RLock()
3) print("Main Thread trying to acquire Lock")
4) l.acquire()
5) print("Main Thread trying to acquire Lock Again")
6) l.acquire()
```

In this case Main Thread won't be Locked b'z thread can acquire the lock any number of times.

This RLock keeps track of recursion level and hence for every acquire() call compulsory release() call should be available. i.e the number of acquire() calls and release() calls should be matched then only lock will be released.

Eg:

```
l=RLock()
l.acquire()
l.acquire()
l.release()
l.release()
```

After 2 release() calls only the Lock will be released.

Examupdt.in

Note:

1. **Only owner thread can acquire the lock multiple times**
2. **The number of acquire() calls and release() calls should be matched.**

Demo Program for synchronization by using RLock:

```
from threading import *
 import time
       l=RLock()
def factorial(n):
       l.acquire()
       if n==0:
         result=1
       else:
         result=n*factorial(n-1)
       l.release()
         return result


def results(n):
       print("The Factorial of",n,"is:",factorial(n))
```

```
t1=Thread(target=results,args=(5,))
t2=Thread(target=results,args=(9,))
t1.start()
t2.start()
```

Output:
The Factorial of 5 is: 120
The Factorial of 9 is: 362880

In the above program instead of RLock if we use normal Lock then the thread will be blocked.

Difference between Lock and RLock:

table

Lock:

1. **Lock object can be acquired by only one thread at a time.Even owner thread also cannot acquire multiple times.**
2. **Not suitable to execute recursive functions and nested access calls**
3. **In this case Lock object will takes care only Locked or unlocked and it never takes care about owner thread and recursion level.**

RLock:

1. **RLock object can be acquired by only one thread at a time, but owner thread can acquire same lock object multiple times.**
2. **Best suitable to execute recursive functions and nested access calls**
3. **In this case RLock object will takes care whether Locked or unlocked and owner thread information, recursiion level.**

Exambdt.in

Synchronization by using Semaphore:

In the case of Lock and RLock,at a time only one thread is allowed to execute.

Sometimes our requirement is at a time a particular number of threads are allowed to access(like at a time 10 memebers are allowed to access database server,4 members are allowed to access Network connection etc).To handle this requirement we cannot use Lock and RLock concepts and we should go for Semaphore concept.

Semaphore can be used to limit the access to the shared resources with limited capacity.

We can create Semaphore object as follows.

s=Semaphore(counter)

Here counter represents the maximum number of threads are allowed to access simultaneously. The default value of counter is 1.

Whenever thread executes acquire() method,then the counter value will be decremented by 1 and if thread executes release() method then the counter value will be incremented by 1.

i.e for every acquire() call counter value will be decremented and for every release() call counter value will be incremented.

Case-1: s=Semaphore()
In this case counter value is 1 and at a time only one thread is allowed to access. It is exactly same as Lock concept.

Case-2: s=Semaphore(3)
In this case Semaphore object can be accessed by 3 threads at a time.The remaining threads have to wait until releasing the semaphore.

Eg:

```
1)  from threading
2)  import time

4)  def wish(name):

6)     for i in range(10):
7)        print("Good
8)        time.sleep
9)        print(na
10)    s.release()
11)
12) t1=Thread(target=wish,args=("Dhoni",))
13)
14) t3=Thread(target=wish,args=("Kohli",))
raj",))
16) t5=Thread(target=wish,args=("Pandya",))
15)
18) t2.start()
it",))

20) t4.start()
```

Examupdt.in

In the above program at a time 2 threads are allowed to access semaphore and hence 2 threads are allowed to execute wish() function.

BoundedSemaphore:

Normal Semaphore is an unlimited semaphore which allows us to call release() method any number of times to increment counter.The number of release() calls can exceed the number of acquire() calls also.

Eg:

```
1) from threading import *
2) s=Semaphore(2)
3)   s.acquire()
4) s.acquire()
5)   s.release()
6) s.release()
7)   s.release()
8) s.release()
9)  print("End")
```

It is valid because in normal semaphore we can call release() any number of times.

BounedSemaphore is exactly same as Semaphore except that the number of release() calls should not exceed the number of acquire() calls,otherwise we will get

ValueError: Semaphore released too many times

Eg:

```
1) from threading import *
2) s=BoundedSemaphore(2)
3)   s.acquire()
4) s.acquire()
5)   s.release()
6) s.release()
7)   s.release()
8) s.release()
9)  print("End")
```

ValueError: Semaphore released too many times
It is invalid b'z the number of release() calls should not exceed the number of acquire() calls in BoundedSemaphore.

Examupdt.in

Note: **To prevent simple programming mistakes, it is recommended to use BoundedSemaphore over normal Semaphore.**

Inter Thread Communication:

Some times as the part of programming requirement, threads are required to communicate with each other. This concept is nothing but interthread communication.

Eg: **After producing items Producer thread has to communicate with Consumer thread to notify about new item.Then consumer thread can consume that new item.**

In Python, we can implement interthread communication by using the following ways

1. **Event**
2. **Condition**
3. **Queue etc**

ExamUpdt.in

Interthread communication by using Event Objects:

Event object is the simplest communication mechanism between the threads. One thread signals an event and other thereds wait for it.

We can create Event object as follows...

```
event = threading.Event()
```

**Event manages an internal flag that can set()
or clear() Threads can wait until event set.**

Methods of Event class:

1. **set()** **internal flag value will become True and it represents GREEN signal for all waiting threads.**

2. **clear()** **inernal flag value will become False and it represents RED signal for all waiting threads.**

3. **isSet()** **This method can be used whether the event is set or not**

Interthread communication by using Queue:

Queues Concept is the most enhanced Mechanism for interthread communication and to share data between threads.

Queue internally has Condition and that Condition has Lock.Hence whenever we are using Queue we are not required to worry about Synchronization.

If we want to use Queues first we should import

queue module. import queue

We can create Queue object as follows

```
q =
```

Examupdt.in

Important Methods of Queue:

1. **put():** Put an item into the queue.
2. **get():** Remove and return an item from the queue.

Producer Thread uses put() method to insert data in the queue. Internally this method has logic to acquire the lock before inserting data into queue. After inserting data lock will be released automatically.

put() method also checks whether the queue is full or not and if queue is full then the Producer thread will entered in to waiting state by calling wait() method internally.

Consumer Thread uses get() method to remove and get data from the queue. Internally this method has logic to acquire the lock before removing data from the queue.Once removal completed then the lock will be released automatically.

If the queue is empty then consumer thread will entered into waiting state by calling wait() method internally.Once queue updated with data then the

thread will be notified automatically.

Note:
The queue module takes care of locking for us which is a great advantage.

Eg:

```
1)  from threading import *
2) import time
3)   import random
4) import queue
5)   def produce(q):
```

```
6)    while True:
7)       item=random.randint(1,100)
8)       print("Producer Producing Item:",item)
9)       q.put(item)
10)     print("Producer giving Notification")
11)     time.sleep(5)
12) def consume(q):
13)  while True:
14)     print("Consumer waiting for updation")
15)     print("Consumer consumed the item:",q.get())
16)     time.sleep(5)
17)
18) q=queue.Queue()
19) t1=Thread(target=consume,args=(q,))
20) t2=Thread(target=produce,args=(q,))
21) t1.start()
22) t2.start()
```

Examupdt.in

Output:
Consumer waiting for
updation Producer Producing
Item: 58 Producer giving
Notification Consumer
consumed the item: 58

Types of Queues:

Python Supports 3 Types of Queues.

1. FIFO Queue:

q = queue.Queue()

This is Default Behaviour. In which order we put items in the queue, in the same order the items will come out (FIFO-First In First Out).

Eg:

```
1) import queue
2) q=queue.Queue()
3) q.put(10)
4) q.put(5)
5) q.put(20)
6) q.put(15)
7) while not q.empty():
8)   print(q.get(),end=' ')
```

Output: 10 5 20 15

---

2. LIFO Queue:

The removal will be happend in the reverse order of insertion(Last In First Out)

Eg:

```
1) import queue
2) q=queue.LifoQueue()
3) q.put(10)
4) q.put(5)
5) q.put(20)
6) q.put(15)
7) while not q.empty():
8)   print(q.get(),end=' ')
```

Output: 15 20 5 10

3. Priority Queue:

The elements will be inserted based on some priority order.

```
1) import queue
2) q=queue.PriorityQueue()
3) q.put(10)
4) q.put(5)
5) q.put(20)
6) q.put(15)
7) while not q.empty():
```

Examupdt.in

```
8)    print(q.get(),end='  ')
```

Output: 5 10 15 20

Eg 2: If the data is non-numeric, then we have to provide our data in the form of tuple.

(x,y)

x is priority
y is our element

```
1)  import queue
2)  q=queue.PriorityQueue()
3)  q.put((1,"AAA"))
4)  q.put((3,"CCC"))
5)  q.put((2,"BBB"))
6)  q.put((4,"DDD"))
7)  while not q.empty():
8)    print(q.get()[1],end='  ')
```

Output: AAA BBB CCC DDD

Examupdt.in