Unit-III

Regular Expressions

Topics

1. Introduction
2. Special Characters and Symbols
3. Regular Expressions and Python

1. Introduction

- **Regular expression** is a sequence of characters used for describing a search pattern.

Example: ^c......r$

The pattern is any five letters starting with c and ending with r.

The "^" symbol and the "$" symbol specifically refer to the start and the end of a string, respectively.

- Regular Expressions (Res) provide infrastructure for advanced text pattern matching, extraction and /or search and replace functionality.

- Regular expressions are simply strings that use special symbols and characters so that they can match a set of strings with similar characteristics described by the pattern.

“[A-Z a-z]\w+”

It means

The first character should be alphabetic, i.e., either A-Z or a-z, followed by at least one (+) alphanumeric character (\w).

- Python supports REs through the standard library re module.
- Searching and matching are the two main ways to accomplish pattern-matching :
- **Searching** - looking for a pattern match in any part of a string, it is done by **search()** method or function

Example :

```
re.search('(?<=abc)def', 'abcdef')
 >>> m.group(0)
‘def’
```

- **Matching** – attempting to match a pattern to an entire string (starting from the beginning), it is done by **match()** method or function

Example:

```
pattern = '^a…e$'
test = 'apple‘
 re.match(pattern, test)
```

Why regular expressions?

- To identify real and fake email addresses.
- To validate 10 digit phone number.
- To find date and time in log file.
- To collect data from websites

Regular Expression is essential in

- PHP
- JavaScript
- Python
- Perl
- Java
- grep (Unix) etc

First Regular Expression

- The alphabet used for regular expression is the set of all uppercase and lower case letters plus numeric digits. Specialized alphabets are also possible: {0,1}

Example : RE pattern : abc

String(s) Matched : abc

The power of regular expression comes in when special characters are used to define:

character sets, subgroup matching and pattern repetition

2. Special Symbols and Characters

Common Regular Expression Symbols and Special Characters

Notation	Description	Example RE
Symbols		
literal	Match literal string value literal	abc
re1|re2	Match regular expressions re1 and re2	abc|def
.	Match any single character(except) NEWLINE	sh.rt[shirt/short]
^	Matches start of string	^the
$	Matches end of string	India$
*	Matches 0 or more occurrences of preceding RE	[A-Za-z0-9]*
+	Matches 1 or more occurrences of preceding RE	[a-z]+\.com
?	Match 0 or 1 occurrences of preceding RE	chair?
{N}	Match N occurrences of preceding RE	[0-9](3)
[M,N]	Match from M to N occurrences of preceding RE	[0-9](5,9)
[. . .]	Match any single character from character class	[aeiou]
[. . x-y. .]	Match any single character in the range from x to y	[0-9],[A-Za-z]
[^. . .]	Do not match any character from character class, including any ranges, if present	[^aeiou], [^A-Za-z0-9_]
(*|+|?|{})?	Apply non greedy versions of above occurrence/ repetition symbols(*,+,?,())	.*?[a-z]
(. . .)	Match enclosed RE and save as subgroup	([0-9]{3})?,f(oo|u)bar

Special Characters

Notation	Description	Example RE
Symbols		
\d	Match any decimal digit, same as [0-9](\D is inverse of \d: do not match any numeric digit)	data\d+.txt
\w	Match any alphanumeric character, same as [A-Za-z0-9_] (\W is inverse of\w)	[A-Za-z_]\w+
\s	Match any whitespace character, same as [\n\t\r\v\f] (\S is inverse of \s)	of\s the
\b	Match any word boundary(\B is inverse of \b)	\bThe\b
\c	Match any special character c verbatim (i.e., without its special meaning, literal)	\., \\, *
\A (\Z)	Match start(end) of string (also see ^ and $ above)	\ADear

2.1. Matching more than one RE pattern with Alternation(|)

- The pipe(|) symbol, a vertical bar indicates an alternation operation, meaning that is used to choose from one of the different regular expressions separated by pipe symbol

Example:

RE pattern	Strings matched
at|home	at, home
sit|sat|set	sit, sat, set

It is also called as union or logical OR

2.2 Matching any single character(.)

- The dot(.) symbol matches any single character except for NEWLINE.
- The dot matches letter, number, whitespace not including "\n" , printable, nonprintable, or a symbol, the dot can match all of them.

Example:

RE pattern	Strings matched
bo.k	Any character between "bo" and "k"
. .	Any pair of characters
. end	Any character before the string end

2.3 Matching from the beginning or end of strings or word boundaries (^/$ /\b /\B)

- To match a pattern starting from the beginning, use carat symbol(^) or the special character (\A) (backslash-capital "A").

Example:

RE pattern	Strings matched
^although	Any string that starts with although
/bin/python$	Any string that ends with /bin/python
^Subject: application$	Any string that solely contains string Subject: application

- To match characters verbatim, use an escaping backslash(\$).

Example: match string ending with a dollar sign($)

“.*\$$”

- To match \b and \B special characters pertaining to word boundary matches.
- \b - pattern must be at the beginning of a word.
- \B – matches a pattern only if it appears starting in the middle of a word(that is not at a word boundary)

Example:

RE Pattern	Strings Matched
the	Any string containing the
\bthe	Any word starting with the
\bthe\b	Matches only the word the
\Bthe	Any string that contains but does not begin with the

2.4 Creating character classes([ ])

- The regular expression matches any of the enclosed characters.

Examples:

RE Pattern	Strings Matched
b[aeiu]t	bat, bet, bit, but
[cr] [23] [dp] [o2] Is similar to [c|r] [2|3] [d|p] [o|2]	A string of 4 characters : First is “c” or “r” , then “2” or “3”, followed by “d” or “p” and finally either “o” or “2” e.g., r3do, c3po, c2d2, etc.

Note: For single character Res, the pipe and brackets are equivalent

2.5 Denoting ranges(-) and negation(^)

- A **hyphen** between a pair of symbols enclosed in brackets is used to indicate a range of characters

Example: A-Z, a-z, or 0-9 for uppercase, lowercase and numeric digits respectively.

- A **caret** character immediately inside the open left bracket symbolizes a directive but not a match.

Examples:

RE Pattern	**Strings Matched**
b.[0-9]	"b followed by any character and then followed by a single digit
[m-o] [sw-y] [ab]	"m" "n" or "o" followed by " s" "w" or "y" followed by "a" "b"
[^aeiou]	A non vowel character
[^\t\n]	Not a TAB or NEWLINE
["-a]	In an ASCII system, all characters that fall between " " " and "a" i.e., between ordinals 34 and 97

2.6 Multiple occurrence/repetation using closure operators(*,+,?,{ })

- The **asterisk or star operator(*)** match zero or more occurrences of the RE immediately to its left(In Formal Languages and Automata Theory, this is called Kleene Closure)
- The **plus operator(+)** match one or more occurrences of an RE (known as Positive Closure)

- The **brace operator ({ })** with either a single value or a comma-seperated pair of values. These indicate a match of exactly N occurrences for ({N}) or a range of occurrences.
- These symbols may be escaped with backslash

i.e., “*” matches the asterisk,etc.

- The **question mark(?)** used for more than once meaning either matching 0 or 1 occurrences, or its other meaning : if it follows any matching using the close operators.
- It directs the regular expression engine to match as few repetitions as possible.

- When pattern matching is employed using the grouping operators, the regular expression engine will try to “absorb” as many characters as possible which match the pattern. This is known as being greedy.

- The closure operators

RE Pattern	Strings Matched
[dn]ot?	“d” or “n” followed by an “o” and at most one “t” after that, i.e., do, no, dot, not
0?[1-9][0-9]{15,16}	Any numeric digit, possibly prepended with a “0”, e.g., the set of numeric representations of the months January to September, whether single-or double digits Fifteen or sixteen digits.
</?[^>]+>	Strings that match all valid (and invalid) HTML tags.

2.7 Special characters representing character sets

- There are special characters that represents character sets.

1. **"0-9"** can be replaced with **"\d"** -indicates a match of any decimal digit.
2. **"A-Za-z0-9_"** and **"\s"(whitespace)** can be replaced with **"\w**"
3. Uppercase versions of these strings represents non-matches

- Example : **"\D"** matches any non decimal digit(same as **"[^0-9]"**)etc.,

- A few examples:

RE Pattern	Strings Matched
\w+-\d+	Alphanumeric string and number separated by a hyphen
[A-Za-z]\w*	Alphabetic first character, additional characters(if present) can be alphanumeric (almost equivalent to the set of valid Python identifiers)
\d{3}-\d{3}-\d{4}	(American) telephone numbers with an area code prefix, as in 800-555-1212
\w+@\w+\.com	Simple email addresses of the form xxx@yyy.com

2.8 Designating groups with parenthesis(())

- A pair of parentheses (()) accomplish either(or both) of the following when used with regular expressions:

1. Grouping regular expressions
2. Matching sub groups

Reasons for using groups

i. To compare a string
ii. To use for repetition

Grouping -it is a way to treat multiple characters as a single group.

Example: (cat) is a group containing letters “c” ”a” and “t”

Matching subgroups-it is a way to extract the patterns

Example: “(\w+)-(\d+)”

A few examples:

RE Pattern	Strings Matched
\d+(\.\d*)?	Strings representing simple floating point number, that is, any number of digits followed optionally by a single decimal point and zero or more numeric digits, as in “0.009,” “4,” “65,” etc.
(Mr?s?\.) ? [A-Z] [a-z] * [A-Za-z-]+	First name and last name, with a restricted first name(must start with uppercase; lowercase only for remaining letters, if any), the full name prepended by an optional title of “Mr.,” “Mrs.,” “Ms.,” or “M.,” and a flexible last name, allowing for multiple words, dashes and uppercase letters.

3.REs and Python

Python supports regular expressions through **re** module.

3.1 re Module: Core functions and Methods

- The most popular functions and methods from the **re** module are discussed here.
- Many of these functions are also available as methods of compiled regular expression objects “regex objects” and RE “match objects”.
- There are three main functions:

1. search()
2. match() and
3. compile()

Table: Common regular Expression Functions and Methods

Function/Method	Description
re Module Function Only	
compile(pattern,flags=0)	Compile RE pattern with any optional flags and return a regex object

Common regular expression functions and methods

Function/Method	Description
re Module Functions and regex Object Methods	
match(pattern, flags=0)	Attempt to match RE pattern to string with optional flags, return match object on success, None on failure.
search(pattern, string, flags=0)	Search for first occurrences of RE pattern within string with optional flags; return match object on success, None on failure
findall(pattern, string[,flags])	Look for all occurrences of pattern in string, return a list of matches
finditer(pattern, string[, flags])	Same as findall() except returns an iterator instead of a list; for each match, the iterator returns a match object.

re Module Functions and regex Object Methods	
split(pattern, string, max=0)	Split string into a list according to RE pattern delimiter and return list of successful matches, splitting at most max times(split all occurrences is the default)
sub(pattern,rep1,string,max=0)	Replace all occurrences of the RE pattern in string with rep1, substituting all occurrences unless max provided
match Object Methods	
group(num=0)	Return entire match(or specific subgroup num)
groups()	Return all matching subgroups in a tuple

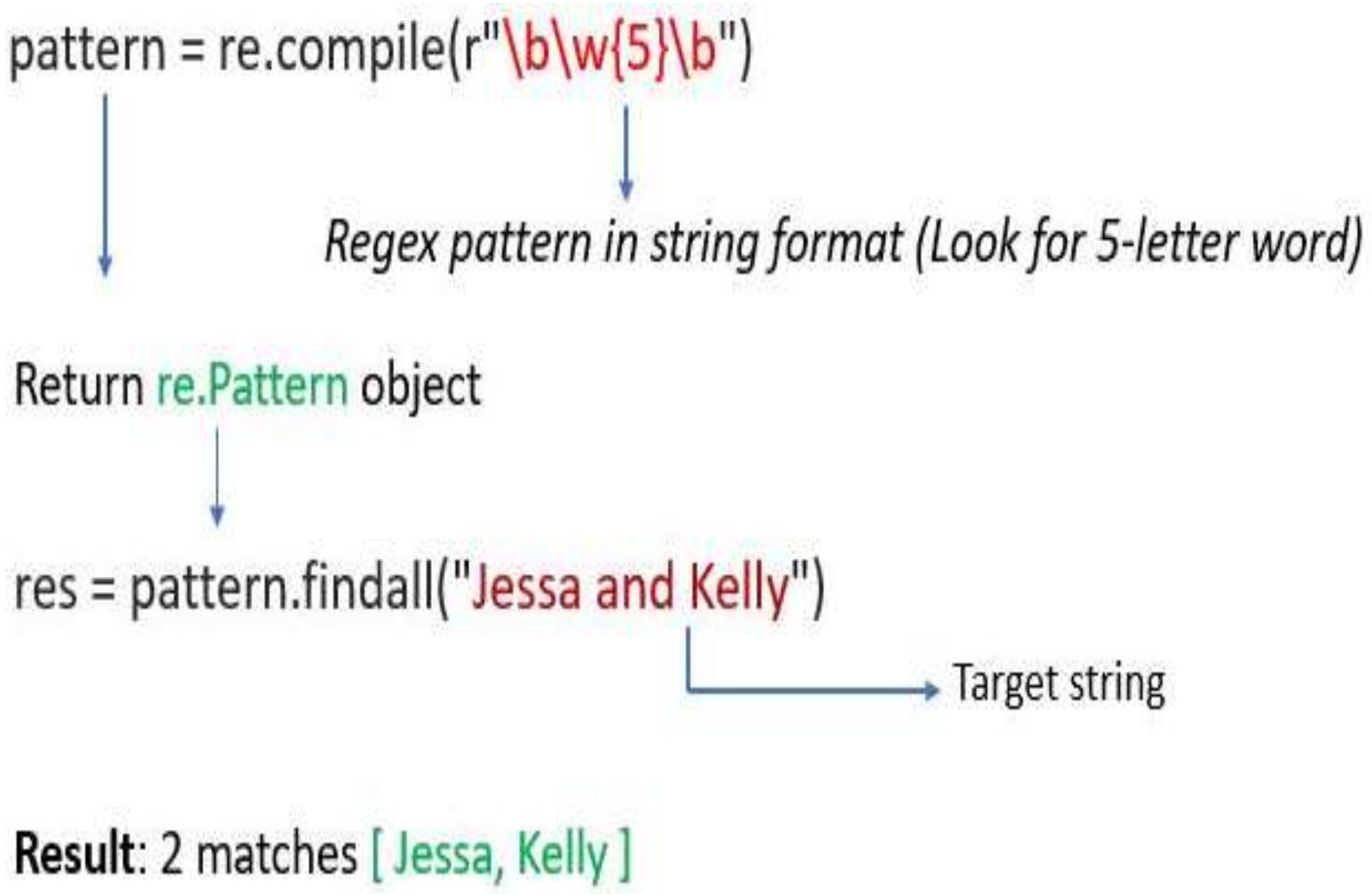
3.2 Compiling REs with compile()

- **re.compile()** - method is used to compile a regular expression pattern provided as a string into a **regex** pattern object(**re.Pattern**).
- **re.Pattern-** this pattern object is used to search for a match in different target strings using regex methods such as

➢ re.match()

➢ re.search()

Example:



```
pattern = re.compile(r"\b\w{5}\b")
res = pattern.findall("Jessa and Kelly")
```

Regex pattern in string format (Look for 5-letter word)
Return re.Pattern object
Target string
Result: 2 matches [Jessa, Kelly]

3.3 Match objects and the group() and groups() methods

Match objects have two primary methods

1. group() and
2. groups()

group()- it returns either the entire match, or a specific subgroup

Example:

```
txt = “its raining today"
x = re.search(r"\br\w+", txt)
x.group()
output: raining
```

- groups()- it returns a tuple consisting of only /all the subgroups.

Example:

```
m = re.match("(\d+)\.(\d+)", “123.15")
print(m.groups())
output: (‘123', ‘15')
```

3.4 Matching strings with match()

- match()-it is the first re module function and RE object(regex object) method.
- The match() function attempts to match the pattern to the string, starting at the beginning.

Successful match-match object returned

Unsuccessful match-none is returned.

Examples:

```
m = re.match(‘abc’,’abc’)#pattern matches string
if  m is not  None:  # show match if successful
    m.group()
Output: ‘abc’
```

```
re.match(‘foo’,’food on the table’).group()
Output:‘foo’
```

3.5 Looking for a pattern within a string with search() (searching vs matching)

Python supports two different operations based on regular expressions:

1. **search()**
2. **match()**

search() checks for a match anywhere in the string. It searches from left to right.

match() checks for a match only at the beginning of the string.

Examples:

```
m= re.match(‘foo’, ‘seafood’)
if m is not None:
m.group()
Output: []
```

```
m=re.search(‘foo’, ‘seafood’)
if m is not None:
m.group()
Output: ‘foo’
```

3.6 Matching more than one string(|)

- Pipe is used in the RE for mathcing more than one string

Example:

```
st=‘sat|set|sit’
m= re.match(st,’sat’)
if m is not None:
m.group()
Output: ‘sat’

m=re.match(st,’slt’)
if m is not None:
m.group()
Output: [
```

Example:

```
m=re.match(st, ‘sheis sitting there’)
if m is not None:
m.group()
Output: []

m=re.search(st,’she settled it”)
if m is not None:
m.group()
Output: ‘set’
```

3.7 Matching any single character(.)

A dot (.) cannot match a NEWLINE or a non-character i.e., the empty string.

Example:

```
m= re.match(“.end”,’bend’)
o/p: ‘bend’
m=re.match(‘.end’,’end’)
o/p: None
m=re.match(‘.end’,’\nend’)
o/p:None
m=re.search(‘.end’,’the end.’)
o/p:’ end’
```

- Example:

```
f1= ‘3.14’
f2=‘3\.14’

m=re.match(f2,’3.14’)
if m is not None: m.group()
o/p: ‘3.14’

m=re.match(f1,’3014’)
if m is not None: m.group()
o/p: ‘3014’

m=re.match(f1,’3.14’)
if m is not None: m.group()
o/p: ‘3.14’
```

3.8 Creating character classes([ ])

Examples:

```
m=re.match('[bsv][lp][o][gt]','vlog')
if m is not None: m.group()
o/p: ‘vlog’
```

3.9 Repetition, special characters, and grouping

- The common aspects of REs involve the use of special characters, multiple occurrences of RE patterns and using parentheses to group and extract submatch patterns.

Example:

Simple e-mail addresses:

(“\w+@\w\.com)

- To match more email addresses and to allow an additional hostname in front of the domain modify the existing RE.

Example:

“www.xxx.com”

- To indicate host name is optional, a pattern is created that matches the host name, use the ? operator indicating zero or one copy of this pattern, and insert the optional RE into previous regular expression as follows:

“\w+@(\w+\.)?\w+\.com”

Example:

```
p1 = ‘\w+@(\w+\.)?\w+\.com
re.match(p1, ‘hello@xxx.com’).group()
 Output: ‘hello@xxx.com’

 re.match(p1,’hello@www.xxx.com’).group()
 Output: ‘hello@www.xxx.com
```

It can be extended to allow any number of intermediate subdomain names with the pattern below. Slight change from ? To *.

```
“\w+@(\w+\.)*\w+\.com”
```

Example:

```
p1=‘\w+@(\w+\.)*\w+\.com’
re.match(p1,’hello@www.xxx.yyy.zzz.com’).group()
o/p:’hello@www.xxx.yyy.zzz.com’
```

```
m=re.match(‘\w\w\w-\d\d\d’,’abc-123’)
if m is not None: m.group()   #o/p: ‘abc-123’

m=re.match(‘(\w\w\w)-(\d\d\d)’).m.group()   #o/p:’abc-123’

m.group(1)   #o/p:‘abc’

m.group(2)   #o/p:’123’

m.groups()  #o/p:(‘abc’,’123’)
```

Example:

```
m=re.match('(a(b))','abc')
m.group()
o/p: ab

m.group(1)
o/p: ab

m.group(2)
o/p:b
m.groups()
o/p: ('ab', 'b')
```

3.11 Finding every occurrence with findall()

findall(): it finds all non-overlapping occurrences of an RE pattern in a string.

- It always returns a list.
- The list is empty if no occurrences are found.
- The list will consist of all matches if the occurrences are found.

Example:

```
re.findall('car','car')
o\p:['car']
re.findall('car','scary')
o\p:['car']
re.findall('car','carry the  baby
carrier near to car')
o/p:['car','car','car']
```

3.12 Searching and Replacing with sub() and [subn()]

- There are two functions/methods for search and replace functionality.

1.sub() and
2.subn()

- Both are almost identical and replace all matched occurrences of the RE pattern in a string with some sort of replacement.

- subn()- it returns the total number of substitutions made i.e., newly substituted string and the substitution count are returned as a 2-tuple.

Example :

```
re.sub(‘X’,’Mr. Smith’, ‘attn: X\n\nDear X,\n)
o/p: attn: Mr. Smith

        Dear Mr. Smith

re.subn('X','Mr. Smith', 'attn: X\n\nDear X,\n‘)
o/p: ('attn: Mr. Smith\n\nDear Mr. Smith,\n', 2)

re.subn('[ae]','X', 'abcdef')
o/p: (‘XbcdXf’,2)
```

3.13 Splitting(on Delimiting Pattern) with split()

- The **re** module and RE object method split() splits the string based on RE pattern.
- You can specify the maximum number of splits by setting a value(other than zero).

Example:

```
re.split('\s',"every day is special for you")
o/p: ['every', 'day', 'is', 'special', 'for', 'you']

re.split('[:\s]',"apple:20 banana:25")
o/p: ['apple', '20', 'banana', '25']
```

Unit-III
Multithreaded Programming

Topics

1. Introduction
2. Threads and Processes
3. Threads and Python
4. thread module
5. threading module
6. Related modules

1.Introduction

- Prior to multithreaded programming(MT), running of a program consist of a single sequence of steps to be executed in synchronous order by the central processing unit (CPU).
- A simple program performing a single task needs a single sequential order of steps for execution by CPU.
- If a program consisting of multiple subtasks, having no relationship with each other (that is results of subtasks do not affect other subtask outcomes) and suppose all these subtasks are to be executed at the same time, then such parallel processing improves the performance of the overall task by involving multithreaded (MT) programming.

- MT programming is perfect for programming tasks that are asynchronous, require multiple concurrent activities and where the processing of each activity is nondeterministic i.e., random and unpredictable.

Example:

Execution of such programming task can be accomplished by an MT program with a queue data structure having a few threads doing specific functions.

- The functions are:

1. UserRequestThread:

➢ Responsible for reading client input from I/O channel.

➢ A number of threads would be created by the program, one for each current client, with request being entered into the queue.

2. RequestProcessor:

➢ A thread that is responsible for retrieving requests from the queue and processing them, providing output for a third party.

3. 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.

- It reduces the complexity of the program.
- Implementation is clean, efficient and well organized.

2. Threads and Processes

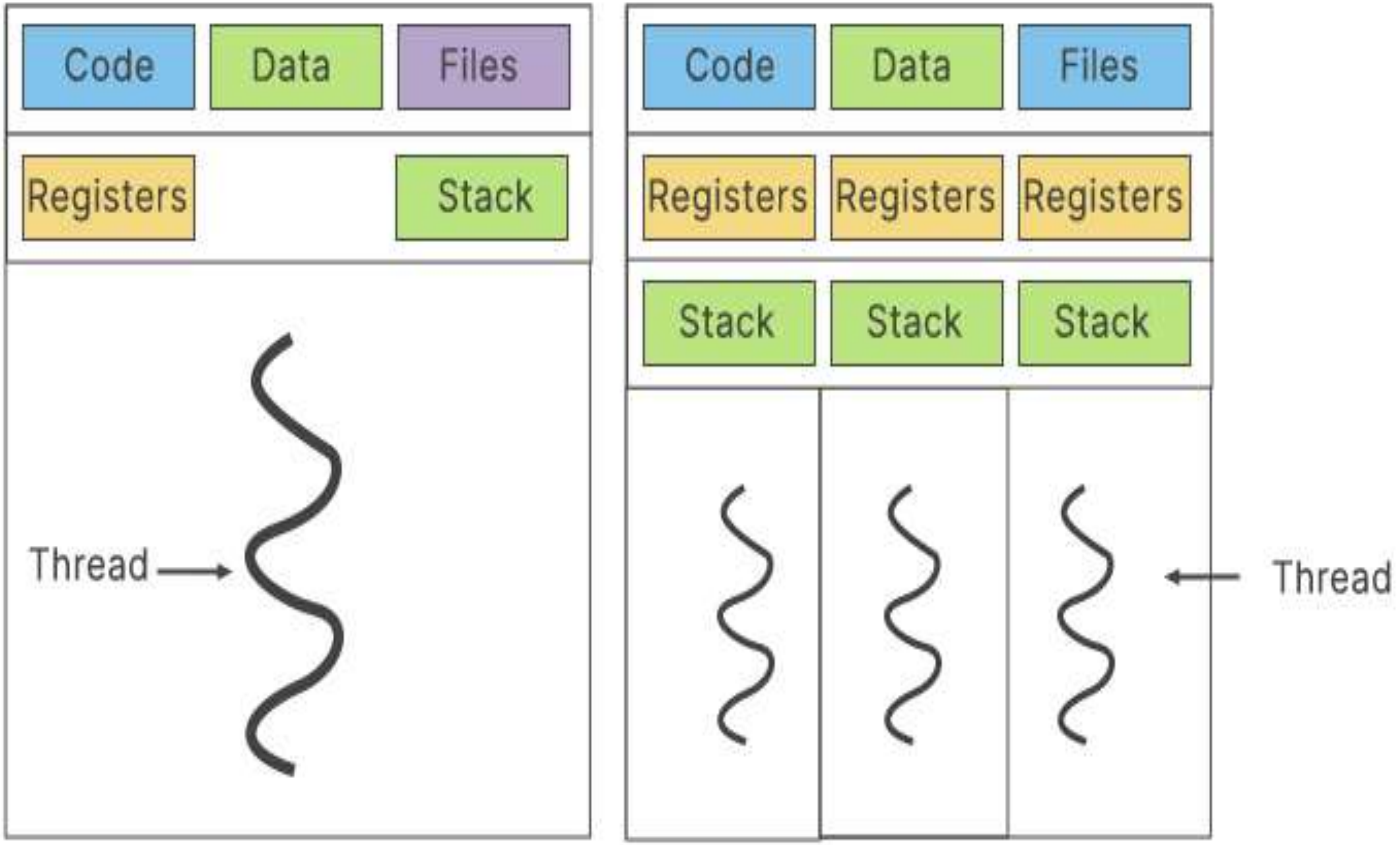
2.1 What are Processes?

- **Program:** It is a an executable code resides on disk. It takes life only after loaded into memory and invoked by the operating system.
- **Process:** It is a program under execution (sometimes called as a heavyweight process).
- 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 of the system by dividing the time between all processes.
- Processes can also fork or spawn(create) new processes to perform other tasks, but each new process has its own memory, data stack etc.
- Processes do not share data unless interprocess communication (IPC) is employed.

2.2 What are Threads?

- **Thread:** Threads are similar to processes except that they all execute within the same process and share the same context. It is also called as lightweight process.
- Threads are mini-processes running in parallel within main process or “main thread”.
- Each and every thread has a beginning, an execution sequence and a conclusion.
- It has an instruction pointer to keep track of where within its context it is currently running.



Single-Threaded Process

Multithreaded Process

- **Yielding :** A thread is preempted(interrupted) and put on hold(also called sleeping) while other threads are running.
- Multiple threads within a process share the same data space with the main thread and therefore can share information or communicate with one another more easily.
- Threads are generally executed in concurrent fashion.
- Throughout the execution of the entire process, each thread performs its own, separate tasks and communicates the results with other threads if necessary.

Race condition : It occurs when two or more threads access the same piece of data at the same time.

Example:

Thread1	Thread2	Integer Value
Read		10
Add by 2		10
Write		12
	Read	12
	Add by 2	12
	Write	14

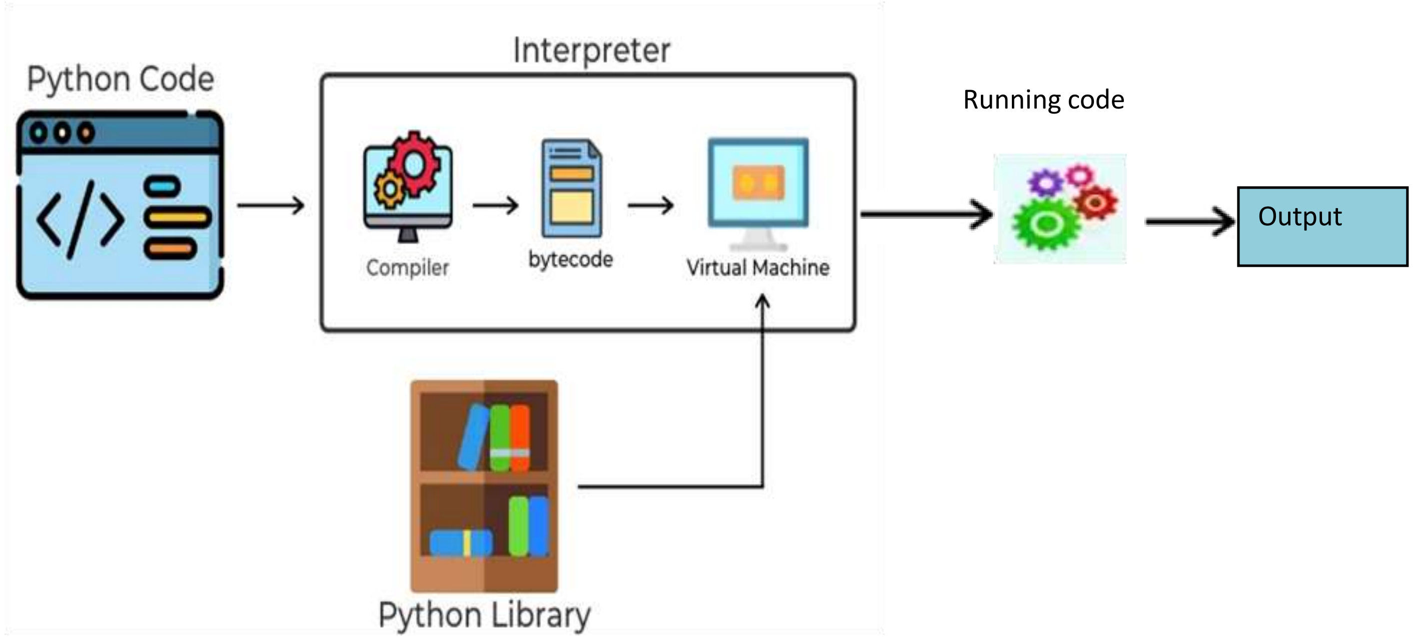
Thread1	Thread2	Integer Value
Read		10
	Read	10
Add by 2		10
	Add by 2	10
Write		12
	Write	12

Inconsistent results occurring due to order of access.

3. Python, Threads and the Global Interpreter Lock

3.1 Global Interpreter Lock(GIL)

- Python code execution is controlled by Python Virtual Machine(the interpreter main loop).



➢ Python Interpreter and CPU

Python Interpreter	CPU
Many threads may be existing but only one thread is being executed by the CPU at any given moment.	Many programs in memory. But only one program is alive on the CPU at any given moment.
All threads share the single interpreter.	All programs share the single CPU.

➢ Global Interpreter Lock(GIL) controls the access to the Python Virtual Machine(PVM) and ensures that only one thread is running on the interpreter.

- In Multithreading(MT) environment, the Python Virtual Machine(PVM) executes in the following manner.

1. ***Set the GIL***
2. ***Switch in a thread to run***
3. ***Execute either (a) or (b)***

   a. For a specified number of bytecode instructions, or

   b. If the thread voluntarily yields control(can be done by time.sleep())

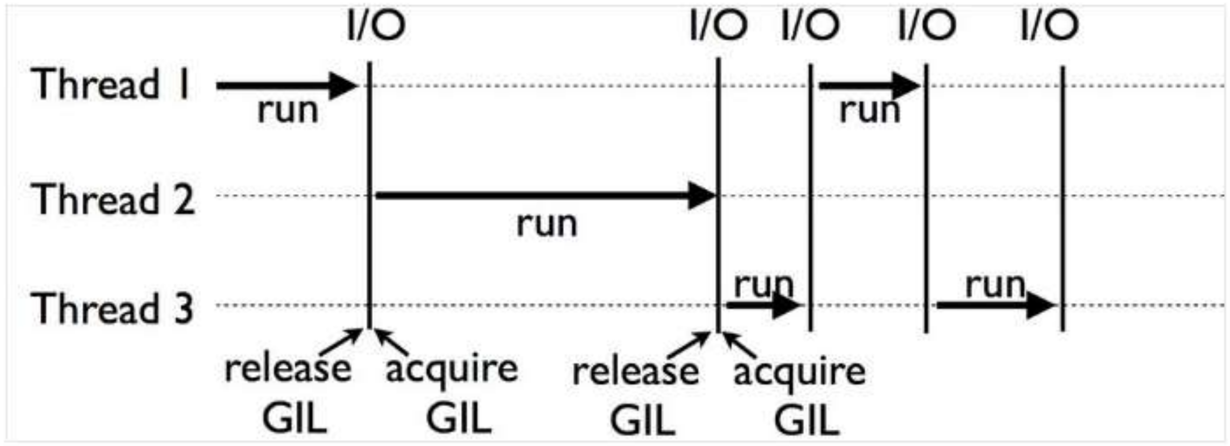
4. Put the thread back to sleep(switch out thread)

5. Unlock the GIL, and …

6. Do it all over again(lather, rinse, repeat).

- The GIL will be locked , when a call to external code such as built-in functions of C/C++ is made. Extension programmers do have the ability to unlock the GIL.

Example: Python I/O oriented routines.



The GIL is released before the I/O call is made, allowing other threads to run while the I/O is being performed.

3.2 Exiting Threads

A thread exits, when a thread completes execution of the function for which it is created.

Threads may also quit

a. By calling an exit function such as thread.exit()
b. A standard way of exiting i.e., sys.exit() or
c. Raising the exception such as SystemExit

- There are two modules related to threads

1. thread module: it is not mostly used because when the main thread exits, all the remaining threads die without cleanup.
2. threading module: it ensures that the whole process stays alive until all threads have exited.

Main thread should always be a good manager, it should know

a. what needs to be executed by individual threads
b. what data or arguments each of the spawned threads requires
c. when they complete execution, and
d. what results they provide.

The main thread can gather the individual results into final, meaningful conclusion.

3.3 Accessing threads from Python

- Python supports multithreaded programming, depending on the operating system on which it is running.
- It is supported on most unix-based platforms such as Linux, Solaris, Macos X, and win32 systems.
- Python uses POSIX(Portable Operating System Interface)-compliant threads, or pthreads.
- By default, the threads are enabled when python is installed.

- Check the availability of threads for interpreter by using the following statement:

```
>>>import thread
...
```

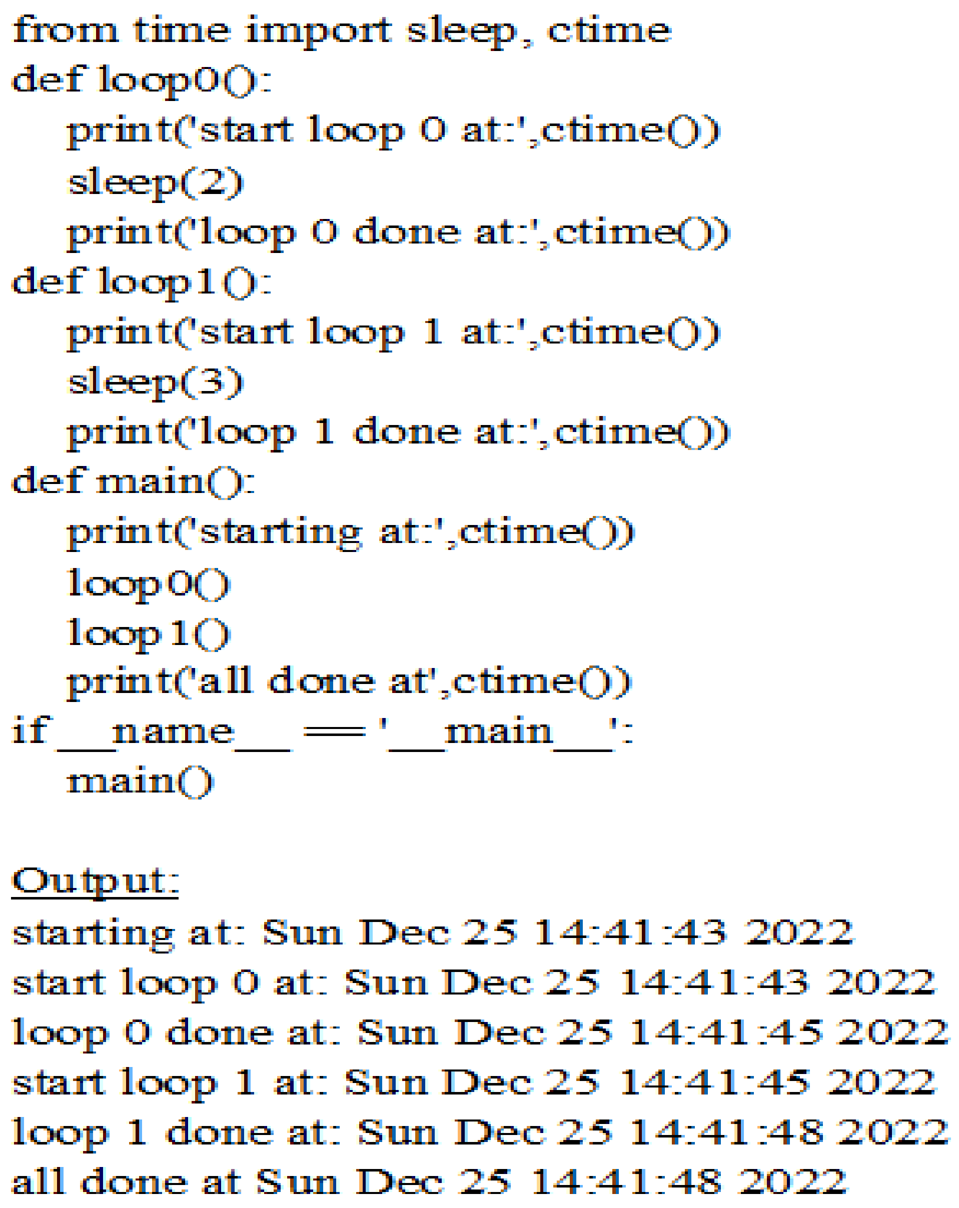
- If the python interpreter was not compiled with threads enabled, the module import fails:

```
>>>import thread
Traceback (most recent call last):
  File "main.py", line 1, in <module>
    import thread
ModuleNotFoundError: No module named 'thread'
```

3.4 Life without threads:

- Consider an example using time.sleep() function to show how threads work.
- time.sleep() takes a number argument and sleeps for the given number of seconds, means execution is halted for the specified amount of time.
- Take two “time loops”, which are executed in sequential order in one process or single threaded program having total time of execution in seconds.

- <u>Program:loops executed by a single thread</u>



```
from time import sleep, ctime
def loop0():
    print('start loop 0 at:',ctime())
    sleep(2)
    print('loop 0 done at:',ctime())
def loop1():
    print('start loop 1 at:',ctime())
    sleep(3)
    print('loop 1 done at:',ctime())
def main():
    print('starting at:',ctime())
    loop0()
    loop1()
    print('all done at',ctime())
if __name__ == '__main__':
    main()
```

Output:

```
starting at: Sun Dec 25 14:41:43 2022
start loop 0 at: Sun Dec 25 14:41:43 2022
loop 0 done at: Sun Dec 25 14:41:45 2022
start loop 1 at: Sun Dec 25 14:41:45 2022
loop 1 done at: Sun Dec 25 14:41:48 2022
all done at Sun Dec 25 14:41:48 2022
```

3.5 Python Threading Modules:

Python provides several modules to support MT programming: thread, threading and queue

The thread and threading modules allow programmer to create and manage threads.

a. thread module provides basic thread and locking support.

b. threading module provides higher-level, fully featured thread management.

c. queue module allows user to create a queue data structure that can be shared across multiple threads.

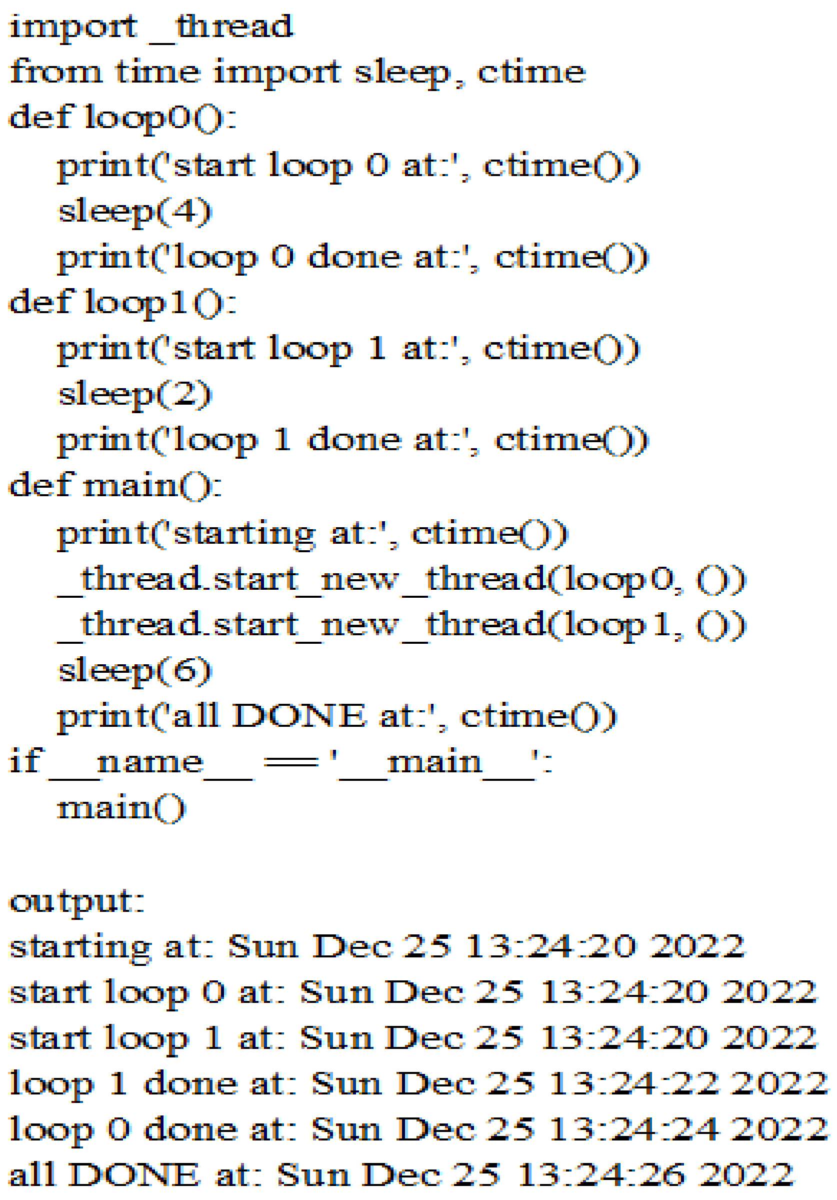
4. thread module

- thread module is able to spawn threads
- thread module also provides a basic synchronization data structure called a lock object (primitive lock, simple lock, mutual exclusion lock, mutex, and binary semaphore).

- **thread Module and Lock Objects**

Function/Method	Description
thread Module Functions	
start_new_thread(function, args, kwargs=None)	Spawns a new thread and executes 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

Program: using thread module



```
import _thread
from time import sleep, ctime
def loop0():
    print('start loop 0 at:', ctime())
    sleep(4)
    print('loop 0 done at:', ctime())
def loop1():
    print('start loop 1 at:', ctime())
    sleep(2)
    print('loop 1 done at:', ctime())
def main():
    print('starting at:', ctime())
    _thread.start_new_thread(loop0, ())
    _thread.start_new_thread(loop1, ())
    sleep(6)
    print('all DONE at:', ctime())
if __name__ == '__main__':
    main()
```

output:

```
starting at: Sun Dec 25 13:24:20 2022
start loop 0 at: Sun Dec 25 13:24:20 2022
start loop 1 at: Sun Dec 25 13:24:20 2022
loop 1 done at: Sun Dec 25 13:24:22 2022
loop 0 done at: Sun Dec 25 13:24:24 2022
all DONE at: Sun Dec 25 13:24:26 2022
```

Program: using thread and locks

```
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()
```

output:

```
starting at: Sun Dec 25 16:59:31 2022
start loop 0 at: Sun Dec 25 16:59:31 2022
start loop 1 at: Sun Dec 25 16:59:31 2022
loop 1 done at: Sun Dec 25 16:59:33 2022
loop 0 done at: Sun Dec 25 16:59:35 2022
all DONE at: Sun Dec 25 16:59:35 2022
```

5. threading Module

- threading module not only provides the Thread class but also provides a wide variety of synchronization mechanisms to use.
- Thread class is used to implement threading

- Threading Module Objects

Object	Description
Thread	Object that represents a single thread of execution
Lock	Primitive lock object (same lock as in 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
Event	General 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 “counter” of finite resources shared between threads; block when none are available
BoundedSemaphore	Similar to a Semaphore but ensures that it never exceeds its initial value
Timer	Similar to Thread, except that it waits for an allotted period of time before running

5.1 Thread class:

- The Thread class of the Threading is the main object.
- There are a variety of ways in which threads can be created using Thread class.
- Three of them are:

1. Create Thread instance, passing in function.
2. Create Thread instance, passing in callable class instance.
3. Subclass Thread and create subclass instance.

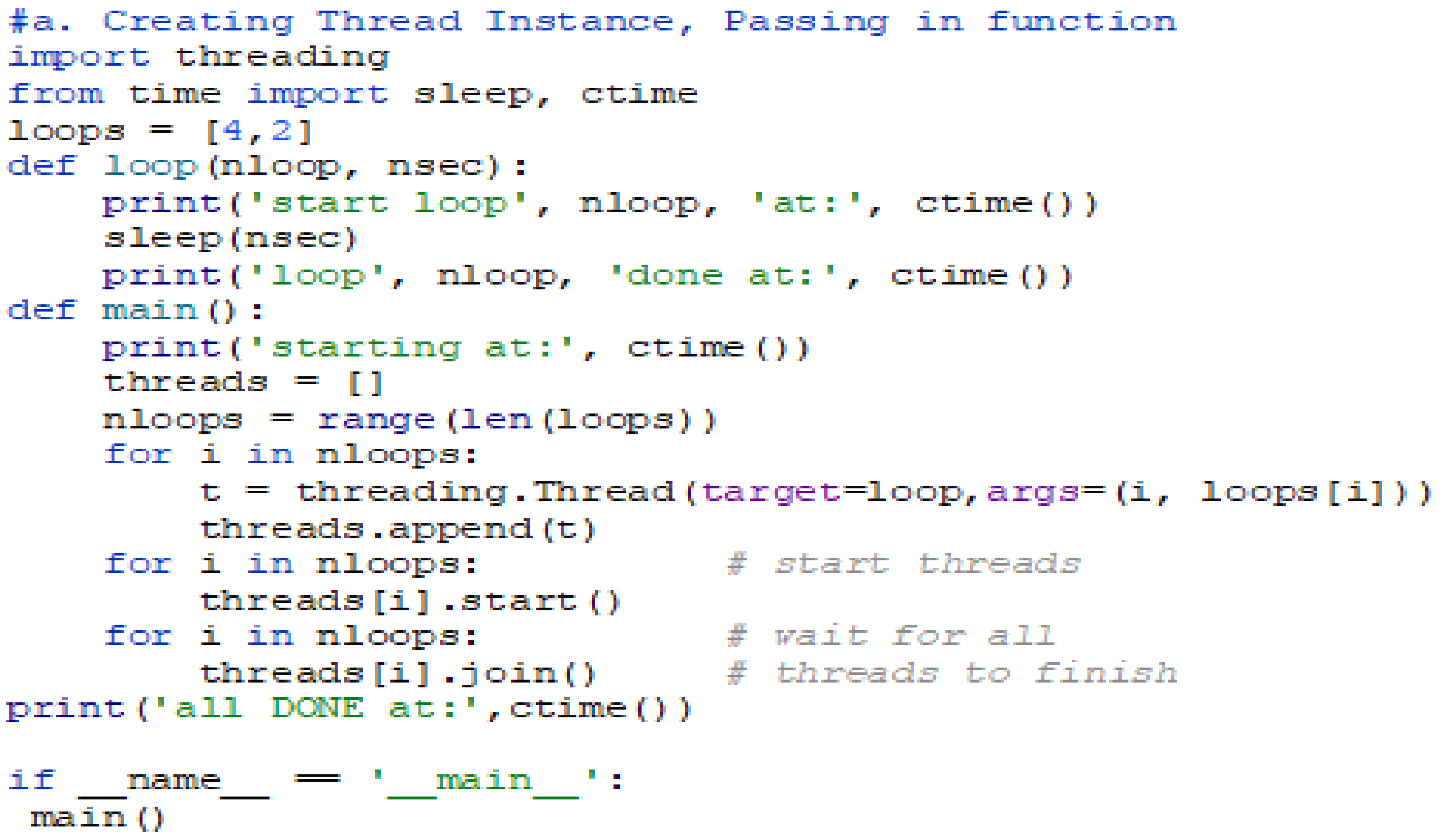
- Thread object methods:

Method	Description
start()	Begin thread execution.
run()	Method defining thread functionality (usually overridden by application writer in a subclass).
join(*timeout*=None)	Suspend until the started thread terminates; blocks unless *timeout* (in seconds) is given.
getName()	Return name of thread.
setName(*name*)	Set name of thread.
isAlive/is_alive()	Boolean flag indicating whether thread is still running.
isDaemon()	Return True if thread daemonic, False otherwise.
setDaemon(*daemonic*)	Set the daemon flag to the given Boolean *daemonic* value (must be called before thread start()).

1. Create Thread instance, passing in function.

- Instantiate Thread , pass in function.
- Create a set of Thread objects, pass in the function (target) and arguments (args) and receive thread instance in return.
- Instead of managing locks(allocating, acquiring, releasing, etc.) a join() method is called for each thread.
- join() method will wait until a thread terminates.

- Program: using the threading module



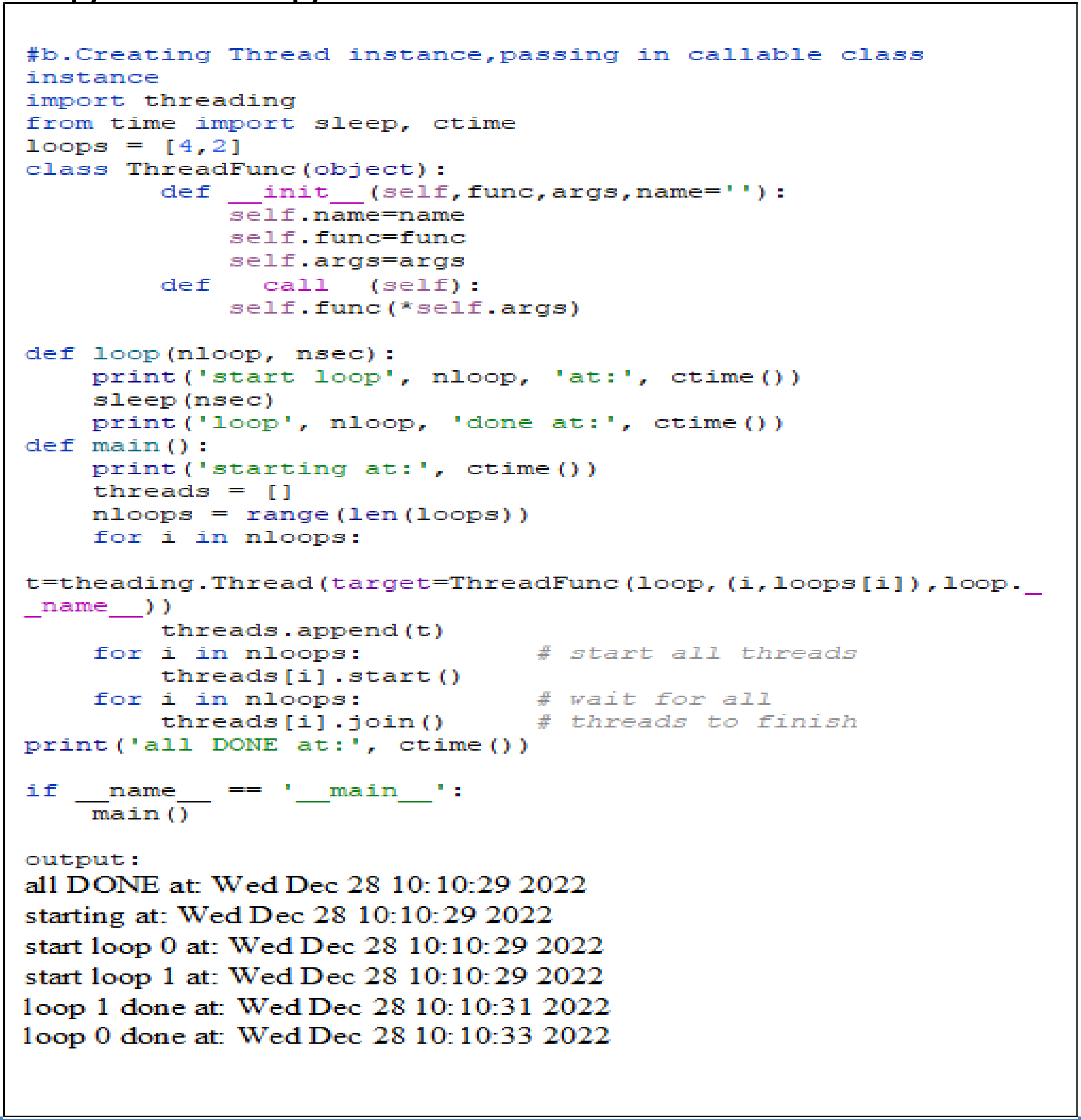
```
#a. Creating Thread Instance, Passing in function
import threading
from time import sleep, ctime
loops = [4,2]
def loop(nloop, nsec):
    print('start loop', nloop, 'at:', ctime())
    sleep(nsec)
    print('loop', nloop, 'done at:', ctime())
def main():
    print('starting at:', ctime())
    threads = []
    nloops = range(len(loops))
    for i in nloops:
        t = threading.Thread(target=loop,args=(i, loops[i]))
        threads.append(t)
    for i in nloops:                # start threads
        threads[i].start()
    for i in nloops:                # wait for all
        threads[i].join()           # threads to finish
print('all DONE at:',ctime())

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

```
all DONE at: Wed Dec 28 09:57:08 2022
starting at: Wed Dec 28 09:57:08 2022
start loop 0 at: Wed Dec 28 09:57:08 2022
start loop 1 at: Wed Dec 28 09:57:08 2022
loop 1 done at: Wed Dec 28 09:57:10 2022
loop 0 done at: Wed Dec 28 09:57:12 2022

Process finished with exit code 0
```

2. Create Thread instance, Passing in callable class instance

- Add a class ThreadFunc to the code.
- It has two methods __init__() and __call__().

Program using callable classes:



```
#b.Creating Thread instance,passing in callable class
instance
import threading
from time import sleep, ctime
loops = [4,2]
class ThreadFunc(object):
        def __init__(self,func,args,name=''):
            self.name=name
            self.func=func
            self.args=args
        def   call   (self):
            self.func(*self.args)

def loop(nloop, nsec):
    print('start loop', nloop, 'at:', ctime())
    sleep(nsec)
    print('loop', nloop, 'done at:', ctime())
def main():
    print('starting at:', ctime())
    threads = []
    nloops = range(len(loops))
    for i in nloops:

t=theading.Thread(target=ThreadFunc(loop,(i,loops[i]),loop._
_name__))
        threads.append(t)
    for i in nloops:              # start all threads
        threads[i].start()
    for i in nloops:              # wait for all
        threads[i].join()         # threads to finish
print('all DONE at:', ctime())

if __name__ == '__main__':
    main()

output:
all DONE at: Wed Dec 28 10:10:29 2022
starting at: Wed Dec 28 10:10:29 2022
start loop 0 at: Wed Dec 28 10:10:29 2022
start loop 1 at: Wed Dec 28 10:10:29 2022
loop 1 done at: Wed Dec 28 10:10:31 2022
loop 0 done at: Wed Dec 28 10:10:33 2022
```

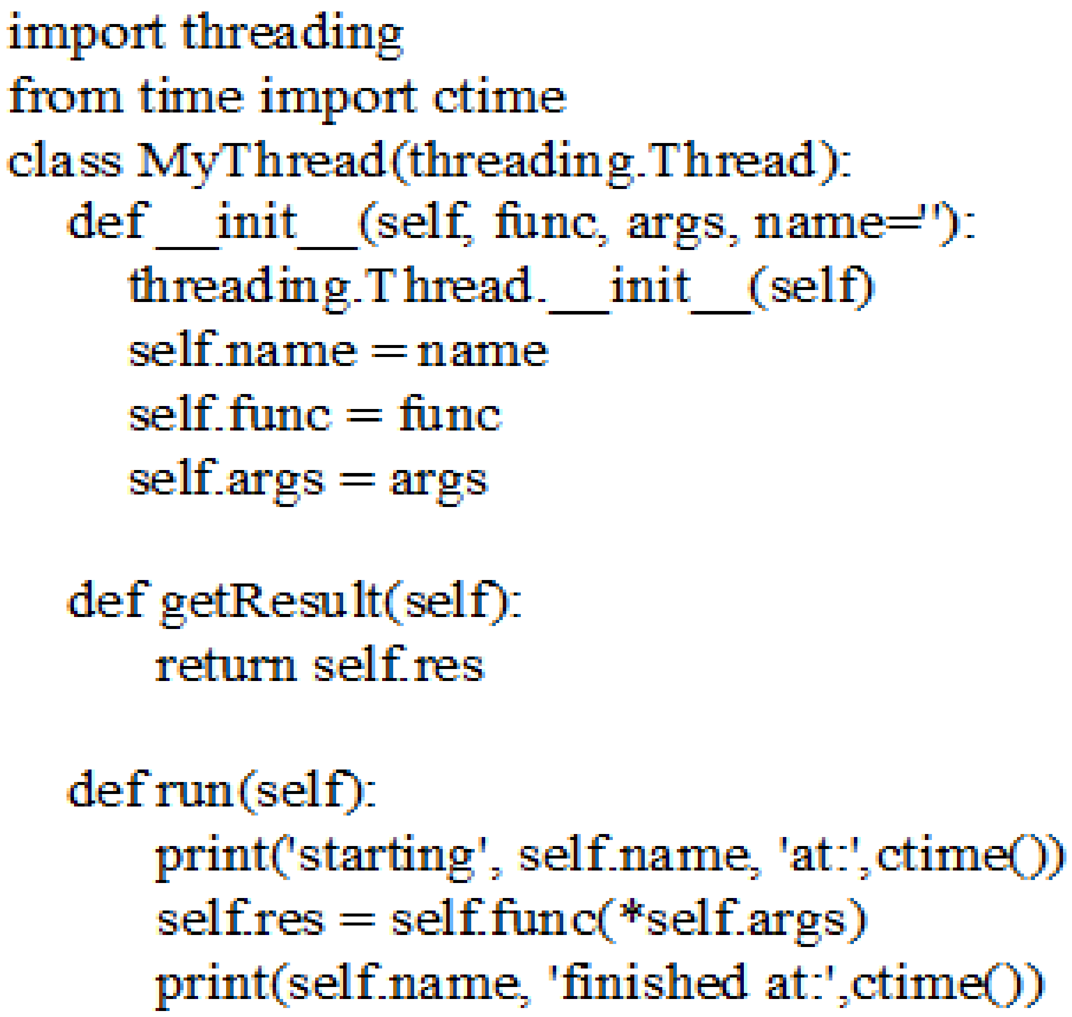
3. Subclass Thread and create subclass instance.

a. MyThread subclass constructor first invoke the base class constructor and
b. The former __call__() method called run in subclass.

Program using Subclassing Thread:

```
#c. Subclass Thread and Create Subclass Instance
import threading
from time import sleep, ctime
loops = (4, 2)
class MyThread(threading.Thread):
    def __init__(self, func, args, name=''):
        threading.Thread.__init__(self)
        self.name = name
        self.func = func
        self.args = args
    def run(self):
        self.func(*self.args)
def loop(nloop, nsec):
    print('start loop', nloop, 'at:', ctime())
    sleep(nsec)
    print('loop', nloop, 'done at:', ctime())

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

    for i in nloops:
        t = MyThread(loop, (i, loops[i]),loop.__name__)
        threads.append(t)

    for i in nloops:
        threads[i].start()

    for i in nloops:
        threads[i].join()

    print('all DONE at:', ctime())

if __name__ == '__main__':
    main()
output:
starting at: Wed Dec 28 10:23:53 2022
start loop 0 at: Wed Dec 28 10:23:53 2022
start loop 1 at: Wed Dec 28 10:23:53 2022
loop 1 done at: Wed Dec 28 10:23:55 2022
loop 0 done at: Wed Dec 28 10:23:57 2022
all DONE at: Wed Dec 28 10:23:57 2022
```

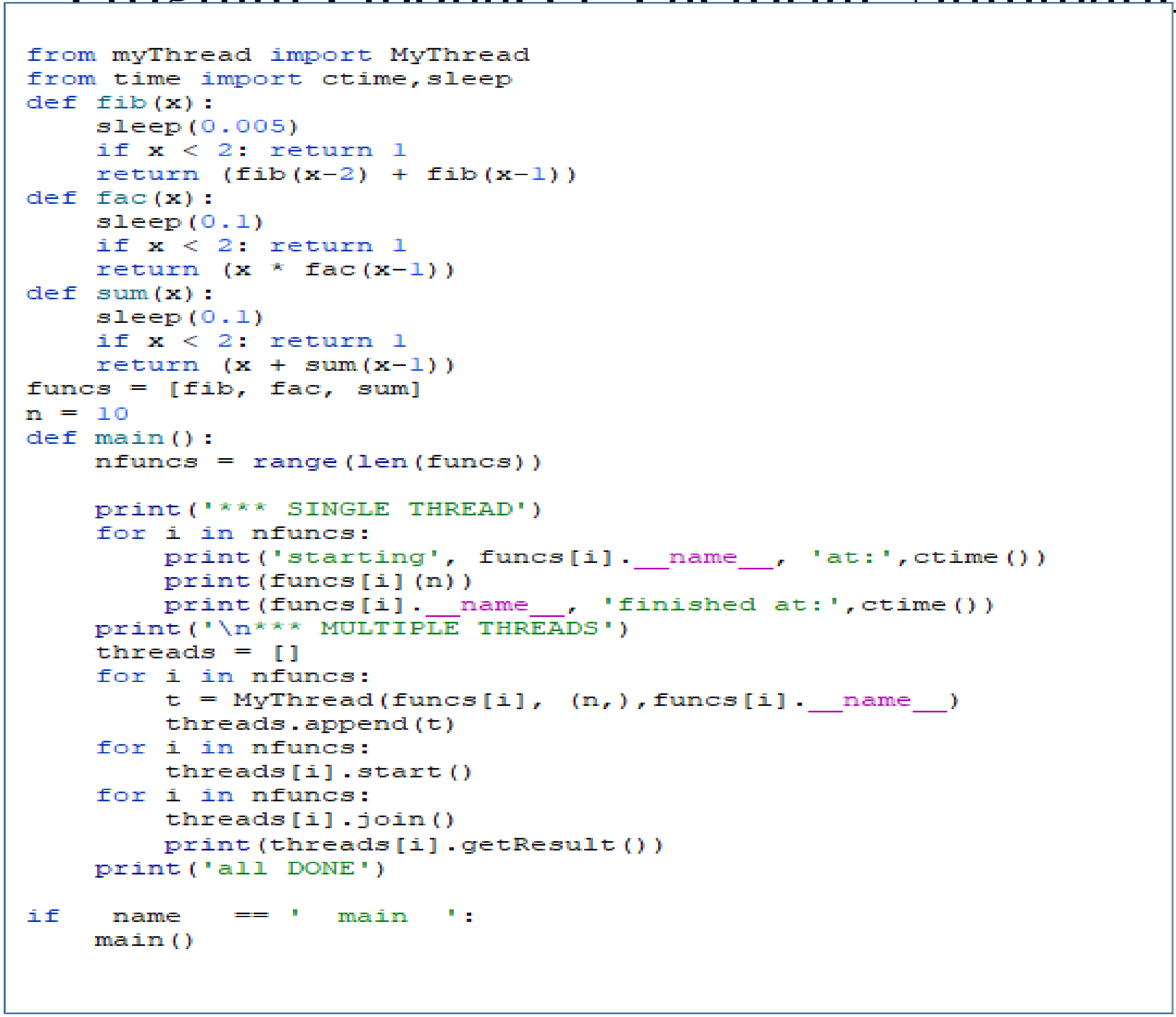
Program MyThread Subclass of Thread



```
import threading
from time import ctime
class MyThread(threading.Thread):
    def __init__(self, func, args, name=''):
        threading.Thread.__init__(self)
        self.name = name
        self.func = func
        self.args = args

    def getResult(self):
          return self.res

    def run(self):
          print('starting', self.name, 'at:',ctime())
          self.res = self.func(*self.args)
          print(self.name, 'finished at:',ctime())
```

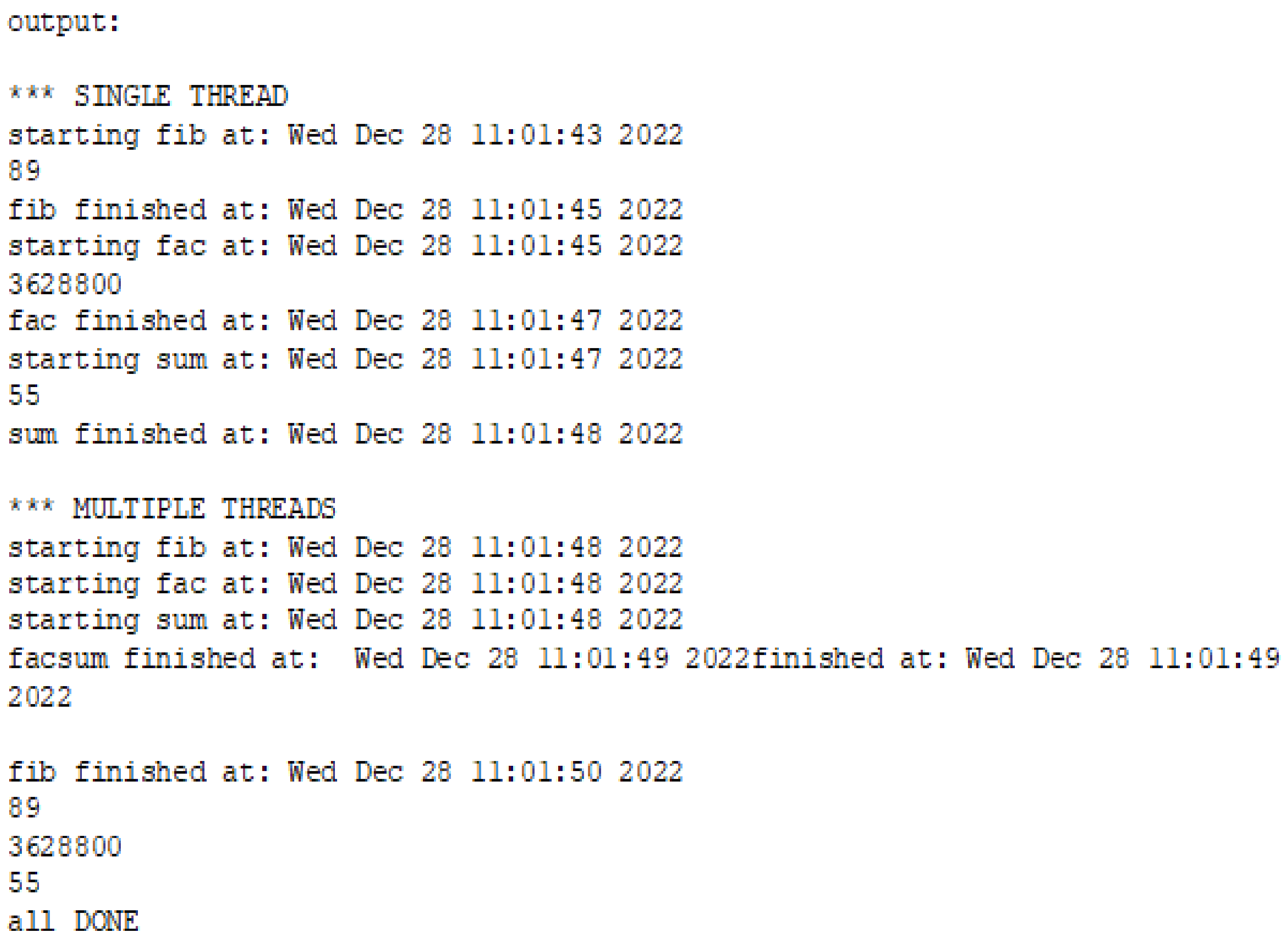
- Fibonacci and Factorial…Take Two, Plus Summation
- The program runs three functions in a single threaded manner, then performs the same task using threads.

• Program Fibonacci Factorial Summation



```
from myThread import MyThread
from time import ctime,sleep
def fib(x):
    sleep(0.005)
    if x < 2: return 1
    return (fib(x-2) + fib(x-1))
def fac(x):
    sleep(0.1)
    if x < 2: return 1
    return (x * fac(x-1))
def sum(x):
    sleep(0.1)
    if x < 2: return 1
    return (x + sum(x-1))
funcs = [fib, fac, sum]
n = 10
def main():
    nfuncs = range(len(funcs))

    print('*** SINGLE THREAD')
    for i in nfuncs:
        print('starting', funcs[i].__name__, 'at:',ctime())
        print(funcs[i](n))
        print(funcs[i].__name__, 'finished at:',ctime())
    print('\n*** MULTIPLE THREADS')
    threads = []
    for i in nfuncs:
        t = MyThread(funcs[i], (n,),funcs[i].__name__)
        threads.append(t)
    for i in nfuncs:
        threads[i].start()
    for i in nfuncs:
        threads[i].join()
        print(threads[i].getResult())
    print('all DONE')

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

- Output of fib, fac and sum



```
output:

*** SINGLE THREAD
starting fib at: Wed Dec 28 11:01:43 2022
89
fib finished at: Wed Dec 28 11:01:45 2022
starting fac at: Wed Dec 28 11:01:45 2022
3628800
fac finished at: Wed Dec 28 11:01:47 2022
starting sum at: Wed Dec 28 11:01:47 2022
55
sum finished at: Wed Dec 28 11:01:48 2022

*** MULTIPLE THREADS
starting fib at: Wed Dec 28 11:01:48 2022
starting fac at: Wed Dec 28 11:01:48 2022
starting sum at: Wed Dec 28 11:01:48 2022
facsum finished at:  Wed Dec 28 11:01:49 2022finished at: Wed Dec 28 11:01:49
2022

fib finished at: Wed Dec 28 11:01:50 2022
89
3628800
55
all DONE
```

5.3 Other threading module function

- The threading module has some supporting functions.

Function	Description
activeCount/ active_count()	Number of currently active Thread objects
currentThread()/ current_thread	Returns the current Thread object
enumerate()	Returns list of all currently active Threads
settrace(*func*)	Sets a trace *function* for all threads
setprofile(*func*)	Sets a profile *function* for all threads

5.4 Producer-Consumer Problem and the Queue module

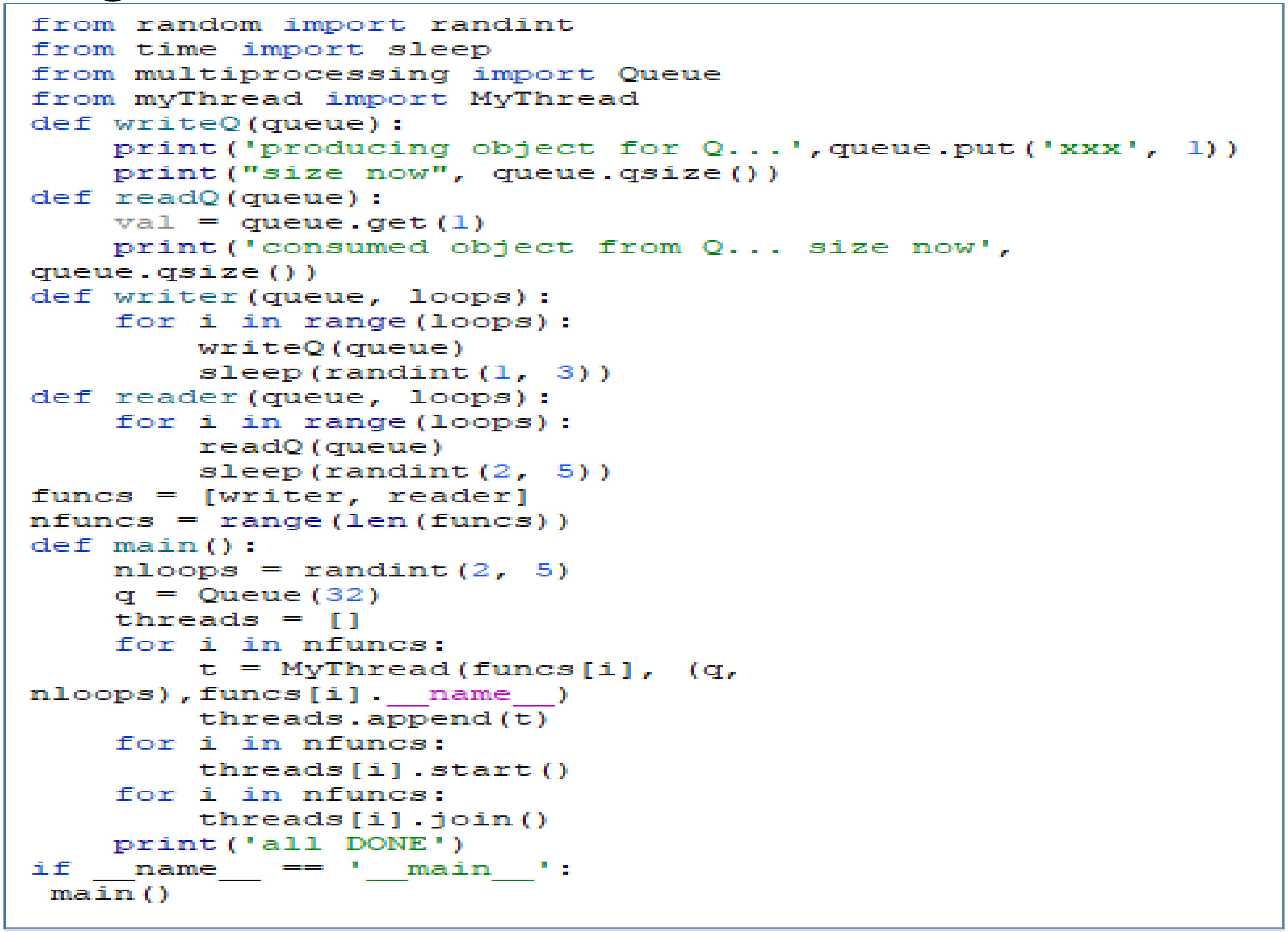
In producer-consumer problem,

i. a producer of goods or services creates goods and places in a data structure such as queue,
ii. a consumer consumes the goods produced by the producer,
iii. the amount of time between producing and consuming goods is non-deterministic.

- Common Queue Module Attributes:

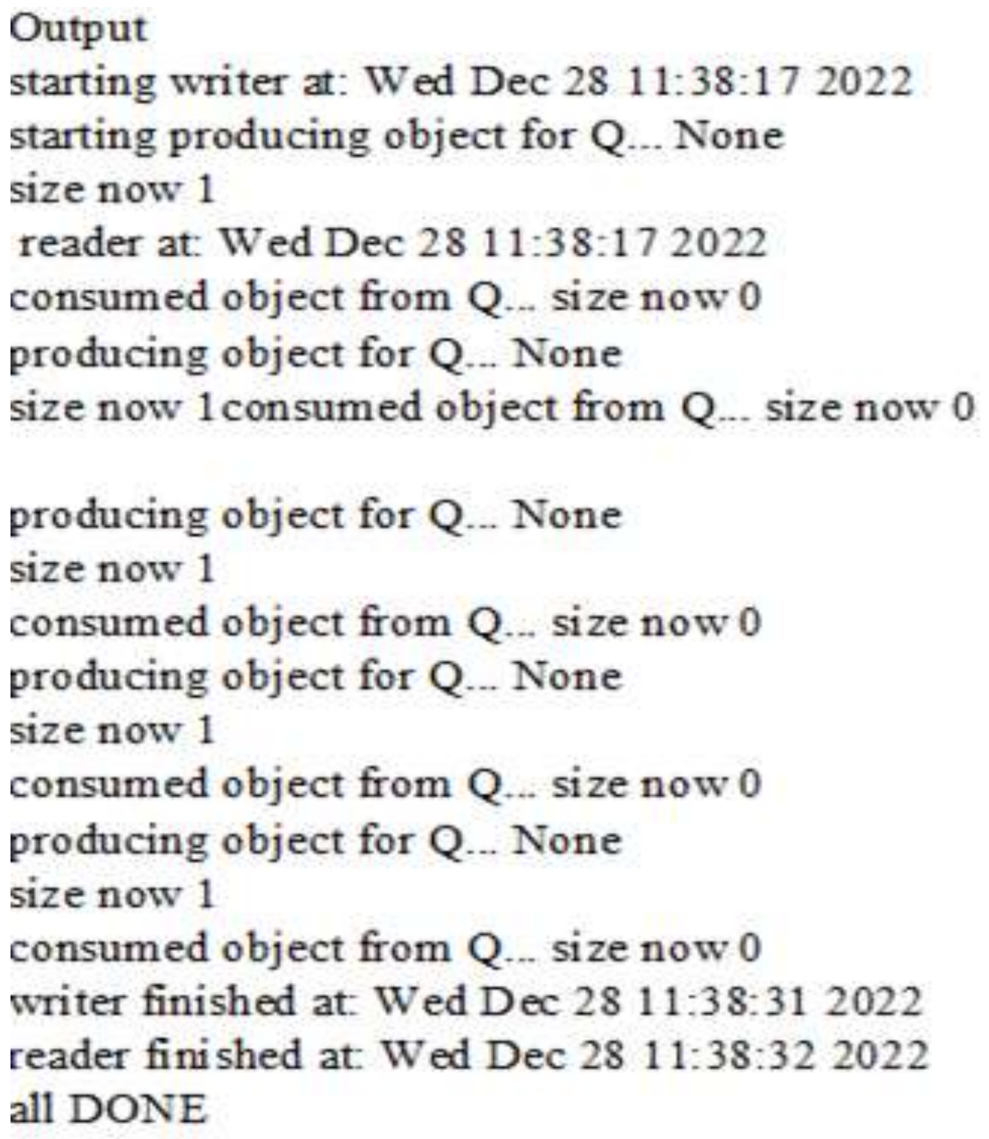
Function/Method	**Description**
Queue Module Function	
queue(size)	Creates a Queue object of given size
Queue Object Methods	
qsize()	Returns queue size (approximate, whereas queue may be getting updated by other threads)
empty()	Returns True if queue empty, False otherwise
full()	Returns True if queue full, False otherwise
put(*item*,*block*=0)	Puts *item* in queue; if *block* given(not 0),block until room is available
get(block=0)	Gets *item* from queue; if *block* given(not 0),block until an item is available

Program Producer-consumer Problem



```
from random import randint
from time import sleep
from multiprocessing import Queue
from myThread import MyThread
def writeQ(queue):
    print('producing object for Q...',queue.put('xxx', 1))
    print("size now", queue.qsize())
def readQ(queue):
    val = queue.get(1)
    print('consumed object from Q... size now',
queue.qsize())
def writer(queue, loops):
    for i in range(loops):
        writeQ(queue)
        sleep(randint(1, 3))
def reader(queue, loops):
    for i in range(loops):
        readQ(queue)
        sleep(randint(2, 5))
funcs = [writer, reader]
nfuncs = range(len(funcs))
def main():
    nloops = randint(2, 5)
    q = Queue(32)
    threads = []
    for i in nfuncs:
        t = MyThread(funcs[i], (q,
nloops),funcs[i].__name__)
        threads.append(t)
    for i in nfuncs:
        threads[i].start()
    for i in nfuncs:
        threads[i].join()
    print('all DONE')
if __name__ == '__main__':
 main()
```

Output producer-consumer problem



```
Output
starting writer at: Wed Dec 28 11:38:17 2022
starting producing object for Q... None
size now 1
 reader at: Wed Dec 28 11:38:17 2022
consumed object from Q... size now 0
producing object for Q... None
size now 1consumed object from Q... size now 0

producing object for Q... None
size now 1
consumed object from Q... size now 0
producing object for Q... None
size now 1
consumed object from Q... size now 0
producing object for Q... None
size now 1
consumed object from Q... size now 0
writer finished at: Wed Dec 28 11:38:31 2022
reader finished at: Wed Dec 28 11:38:32 2022
all DONE
```

6. Related Modules

- Threading-Related Standard Library Modules

Module	Description
thread	Basic, lower-level thread module
threading	Higher-level threading and synchronization objects
mutex	Mutual exclusion objects
Queue	Synchronized FIFO queue for multiple threads
SocketServer	Create/manage threaded TCP or UDP servers