**What is multiprocessing?**

Multiprocessing refers to the ability of a system to support more than one processor at the same time. Applications in a multiprocessing system are broken to smaller routines that run independently. The operating system allocates these threads to the processors improving performance of the system.

**Why multiprocessing?**

Consider a computer system with a single processor. If it is assigned several processes at the same time, it will have to interrupt each task and switch briefly to another, to keep all of the processes going.  
This situation is just like a chef working in a kitchen alone. He has to do several tasks like baking, stirring, kneading dough, etc.

So the gist is that: The more tasks you must do at once, the more difficult it gets to keep track of them all, and keeping the timing right becomes more of a challenge.  
This is where the concept of multiprocessing arises!  
**A multiprocessing system can have:**

# multiprocessor, i.e. a computer with more than one central processor.

# multi-core processor, i.e. a single computing component with two or more independent actual processing units (called “cores”).

Here, the CPU can easily executes several tasks at once, with each task using its own processor.

It is just like the chef in last situation being assisted by his assistants. Now, they can divide the tasks among themselves and chef doesn’t need to switch between his tasks.

**Multiprocessing in Python**

In Python, the [**multiprocessing**](https://docs.python.org/3/library/multiprocessing.html) module includes a very simple and intuitive API for dividing work between multiple processes.  
Let us consider a simple example using multiprocessing module:

# # importing the multiprocessing module

# import multiprocessing

# 

# def print\_cube(num):

# """

# function to print cube of given num

# """

# print("Cube: {}".format(num \* num \* num))

# 

# def print\_square(num):

# """

# function to print square of given num

# """

# print("Square: {}".format(num \* num))

# 

# if \_\_name\_\_ == "\_\_main\_\_":

# # creating processes

# p1 = multiprocessing.Process(target=print\_square, args=(10, ))

# p2 = multiprocessing.Process(target=print\_cube, args=(10, ))

# 

# # starting process 1

# p1.start()

# # starting process 2

# p2.start()

# 

# # wait until process 1 is finished

# p1.join()

# # wait until process 2 is finished

# p2.join()

# 

# # both processes finished

# print("Done!")

**Output**

Square: 100

Cube: 1000

Done!

Let us try to understand the above code:

# To import the multiprocessing module, we do:

* import multiprocessing

# To create a process, we create an object of ****Process**** class. It takes following arguments:

# ****target****: the function to be executed by process

# ****args****: the arguments to be passed to the target function

Note: **Process** constructor takes many other arguments also which will be discussed later. In above example, we created 2 processes with different target functions:

p1 = multiprocessing.Process(target=print\_square, args=(10, ))

p2 = multiprocessing.Process(target=print\_cube, args=(10, ))

# To start a process, we use ****start**** method of ****Process**** class.

* p1.start()
* p2.start()

# Once the processes start, the current program also keeps on executing. In order to stop execution of current program until a process is complete, we use ****join**** method.

* p1.join()
* p2.join()

As a result, the current program will first wait for the completion of **p1** and then **p2**. Once, they are completed, the next statements of current program are executed.

Let us consider another program to understand the concept of different processes running on same python script. In this example below, we print the ID of the processes running the target functions:

|  |
| --- |
| **# importing the multiprocessing module**  import multiprocessing  import os    def worker1():      # printing process id      print("ID of process running worker1: {}".format(os.getpid()))    def worker2():      # printing process id      print("ID of process running worker2: {}".format(os.getpid()))    if \_\_name\_\_ == "\_\_main\_\_":      # printing main program process id      print("ID of main process: {}".format(os.getpid()))        # creating processes      p1 = multiprocessing.Process(target=worker1)      p2 = multiprocessing.Process(target=worker2)        # starting processes      p1.start()      p2.start()        # process IDs      print("ID of process p1: {}".format(p1.pid))      print("ID of process p2: {}".format(p2.pid))        # wait until processes are finished      p1.join()      p2.join()        # both processes finished      print("Both processes finished execution!")        # check if processes are alive      print("Process p1 is alive: {}".format(p1.is\_alive()))      print("Process p2 is alive: {}".format(p2.is\_alive())) |

# Output

ID of main process: 28628

ID of process running worker1: 29305

ID of process running worker2: 29306

ID of process p1: 29305

ID of process p2: 29306

Both processes finished execution!

Process p1 is alive: False

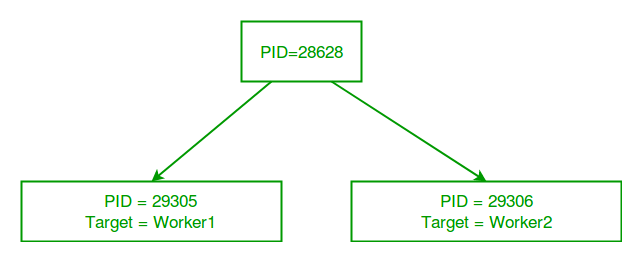
Process p2 is alive: False

# The main python script has a different process ID and multiprocessing module spawns new processes with different process IDs as we create ****Process**** objects ****p1**** and ****p2****. In above program, we use ****os.getpid()**** function to get ID of process running the current target function.

Notice that it matches with the process IDs of **p1** and **p2** which we obtain using **pid** attribute of **Process** class.

# Each process runs independently and has its own memory space.

# As soon as the execution of target function is finished, the processes get terminated. In above program we used ****is\_alive**** method of ****Process**** class to check if a process is still active or not.

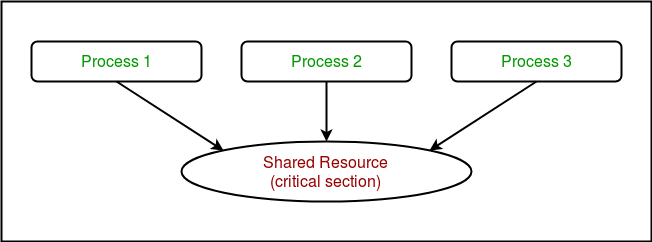
Consider the diagram below to understand how new processes are different from main Python script:  


# Synchronization and Pooling of processes in Python

# ****Synchronization between processes****

Process synchronization is defined as a mechanism which ensures that two or more concurrent processes do not simultaneously execute some particular program segment known as **critical section**.

*Critical section refers to the parts of the program where the shared resource is accessed.*

For example, in the diagram below, 3 processes try to access shared resource or critical section at the same time.  


Concurrent accesses to shared resource can lead to **race condition**.

*A race condition occurs when two or more processes can access shared data and they try to change it at the same time. As a result, the values of variables may be unpredictable and vary depending on the timings of context switches of the processes.*

Consider the program below to understand the concept of race condition:

|  |
| --- |
| # Python program to illustrate  # the concept of race condition  # in multiprocessing  import multiprocessing    # function to withdraw from account  def withdraw(balance):      for \_ in range(10000):          balance.value = balance.value - 1    # function to deposit to account  def deposit(balance):      for \_ in range(10000):          balance.value = balance.value + 1    def perform\_transactions():        # initial balance (in shared memory)      balance = multiprocessing.Value('i', 100)        # creating new processes      p1 = multiprocessing.Process(target=withdraw, args=(balance,))      p2 = multiprocessing.Process(target=deposit, args=(balance,))        # starting processes      p1.start()      p2.start()        # wait until processes are finished      p1.join()      p2.join()        # print final balance      print("Final balance = {}".format(balance.value))    if \_\_name\_\_ == "\_\_main\_\_":      for \_ in range(10):            # perform same transaction process 10 times          perform\_transactions() |

If you run above program, you will get some unexpected values like this:

Final balance = 1311

Final balance = 199

Final balance = 558

Final balance = -2265

Final balance = 1371

Final balance = 1158

Final balance = -577

Final balance = -1300

Final balance = -341

Final balance = 157

In above program, 10000 withdraw and 10000 deposit transactions are carried out with initial balance as 100. The expected final balance is 100 but what we get in 10 iterations of **perform\_transactions** function is some different values.

This happens due to concurrent access of processes to the shared data **balance**. This unpredictability in balance value is nothing but **race condition**.

Let us try to understand it better using the sequence diagrams given below. These are the different sequences which can be produced in above example for a single withdraw and deposit action.

* This is a possible sequence which gives wrong answer as both processes read the same value and write it back accordingly.

|  |  |  |
| --- | --- | --- |
| **P1** | **P2** | **BALANCE** |
| **read(balance)** current=100 |  | 100 |
|  | **read(balance)** current=100 | 100 |
| balance=current-1=99 **write(balance)** |  | 99 |
|  | balance=current+1=101 **write(balance)** | 101 |

* These are 2 possible sequences which are desired in above scenario.

|  |  |  |
| --- | --- | --- |
| **P1** | **P2** | **BALANCE** |
| **read(balance)** current=100 |  | 100 |
| balance=current-1=99 **write(balance)** |  | 99 |
|  | **read(balance)** current=99 | 99 |
|  | balance=current+1=100 **write(balance)** | 100 |

|  |  |  |
| --- | --- | --- |
| **P1** | **P2** | **BALANCE** |
|  | **read(balance)** current=100 | 100 |
|  | balance=current+1=101 **write(balance)** | 101 |
| **read(balance)** current=101 |  | 101 |
| balance=current-1=100 **write(balance)** |  | 100 |

**Using Locks**

**multiprocessing** module provides a **Lock** class to deal with the race conditions. **Lock** is implemented using a **Semaphore** object provided by the Operating System.

*A semaphore is a synchronization object that controls access by multiple processes to a common resource in a parallel programming environment. It is simply a value in a designated place in operating system (or kernel) storage that each process can check and then change. Depending on the value that is found, the process can use the resource or will find that it is already in use and must wait for some period before trying again. Semaphores can be binary (0 or 1) or can have additional values. Typically, a process using semaphores checks the value and then, if it using the resource, changes the value to reflect this so that subsequent semaphore users will know to wait.*

**Consider the example given below:**

|  |
| --- |
| # Python program to illustrate  # the concept of locks  # in multiprocessing  import multiprocessing    # function to withdraw from account  def withdraw(balance, lock):      for \_ in range(10000):          lock.acquire()          balance.value = balance.value - 1          lock.release()    # function to deposit to account  def deposit(balance, lock):      for \_ in range(10000):          lock.acquire()          balance.value = balance.value + 1          lock.release()    def perform\_transactions():        # initial balance (in shared memory)      balance = multiprocessing.Value('i', 100)        # creating a lock object      lock = multiprocessing.Lock()        # creating new processes      p1 = multiprocessing.Process(target=withdraw, args=(balance,lock))      p2 = multiprocessing.Process(target=deposit, args=(balance,lock))        # starting processes      p1.start()      p2.start()        # wait until processes are finished      p1.join()      p2.join()        # print final balance      print("Final balance = {}".format(balance.value))    if \_\_name\_\_ == "\_\_main\_\_":      for \_ in range(10):            # perform same transaction process 10 times          perform\_transactions() |

Output:

Final balance = 100

Final balance = 100

Final balance = 100

Final balance = 100

Final balance = 100

Final balance = 100

Final balance = 100

Final balance = 100

Final balance = 100

Final balance = 100

Let us try to understand the above code step by step:

* Firstly, a **Lock** object is created using:
* lock = multiprocessing.Lock()
* Then, **lock** is passed as target function argument:
* p1 = multiprocessing.Process(target=withdraw, args=(balance,lock))
* p2 = multiprocessing.Process(target=deposit, args=(balance,lock))

In the critical section of target function, we apply lock using **lock.acquire()** method. As soon as a lock is acquired, no other process can access its critical section until the lock is released using **lock.release()** method.

lock.acquire()

balance.value = balance.value - 1

lock.release()

As you can see in the results, the final balance comes out to be 100 every time (which is the expected final result).

**Pooling between processes**

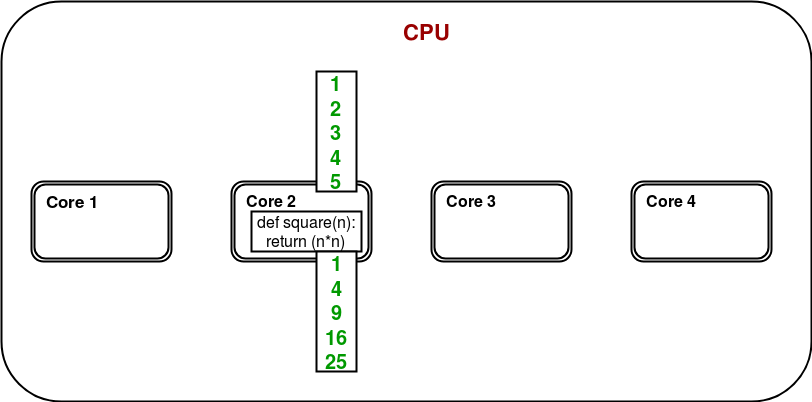
Let us consider a simple program to find squares of numbers in a given list.

|  |
| --- |
| # Python program to find  # squares of numbers in a given list  def square(n):      return (n\*n)    if \_\_name\_\_ == "\_\_main\_\_":        # input list      mylist = [1,2,3,4,5]        # empty list to store result      result = []        for num in mylist:          result.append(square(num))        print(result) |

Output:

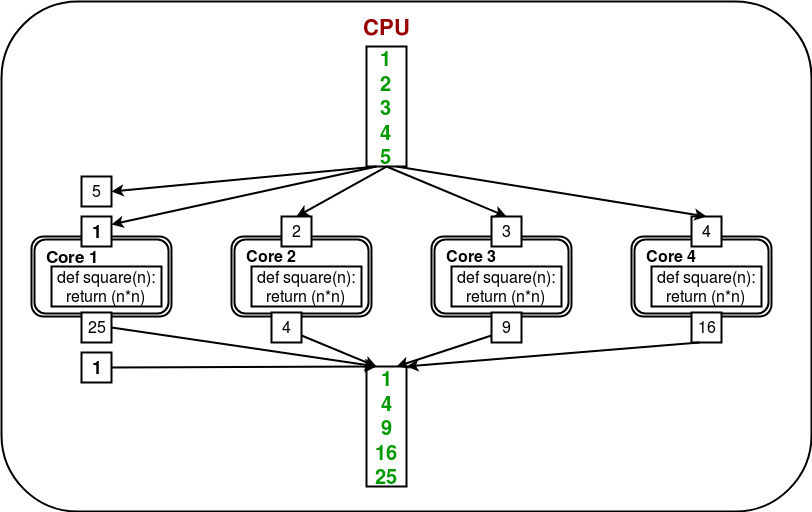
[1, 4, 9, 16, 25]

It is a simple program to calculate squares of elements of a given list. In a multi-core/multi-processor system, consider the diagram below to understand how above program will work:



Only one of the cores is used for program execution and it’s quite possible that other cores remain idle.

In order to utilize all the cores, **multiprocessing** module provides a **Pool** class. The **Pool** class represents a pool of worker processes. It has methods which allows tasks to be offloaded to the worker processes in a few different ways. Consider the diagram below:



Here, the task is offloaded/distributed among the cores/processes automatically by **Pool** object. User doesn’t need to worry about creating processes explicitly.

Consider the program given below:

|  |
| --- |
| # Python program to understand  # the concept of pool  import multiprocessing  import os    def square(n):      print("Worker process id for {0}: {1}".format(n, os.getpid()))      return (n\*n)    if \_\_name\_\_ == "\_\_main\_\_":      # input list      mylist = [1,2,3,4,5]        # creating a pool object      p = multiprocessing.Pool()        # map list to target function      result = p.map(square, mylist)        print(result) |

Output:

Worker process id for 2: 4152

Worker process id for 1: 4151

Worker process id for 4: 4151

Worker process id for 3: 4153

Worker process id for 5: 4152

[1, 4, 9, 16, 25]

Let us try to understand above code step by step:

* We create a **Pool** object using:
* p = multiprocessing.Pool()

There are a few arguments for gaining more control over offloading of task. These are:

* + **processes:** specify the number of worker processes.
  + **maxtasksperchild:** specify the maximum number of task to be assigned per child.

All the processes in a pool can be made to perform some initialization using these arguments:

* + **initializer:** specify an initialization function for worker processes.
  + **initargs:** arguments to be passed to initializer.
* Now, in order to perform some task, we have to map it to some function. In the example above, we map **mylist** to **square** function. As a result, the contents of **mylist** and definition of **square** will be distributed among the cores.
* result = p.map(square, mylist)
* Once all the worker processes finish their task, a list is returned with the final result.

# Multithreading in Python

# ****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)

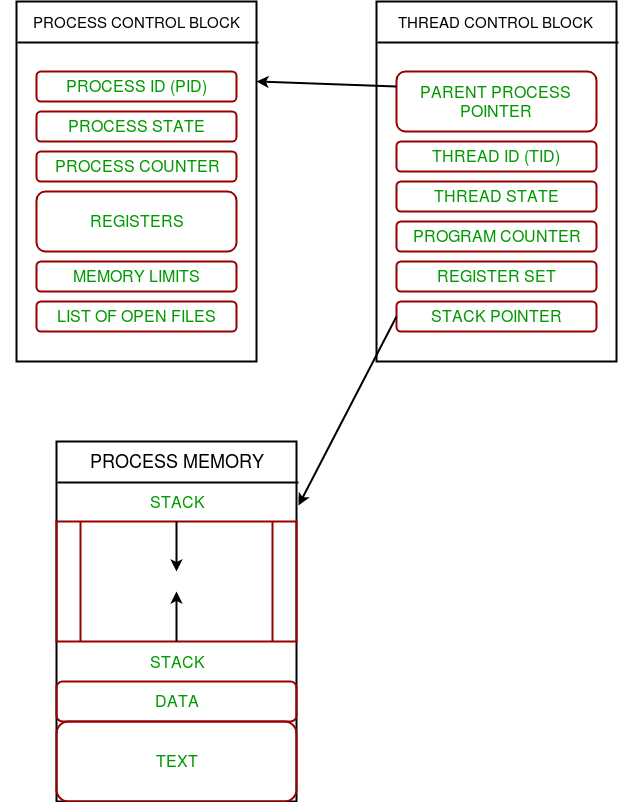
A **thread** is an entity within a process that can be scheduled for execution. Also, it is the smallest unit of processing that can be performed in an OS (Operating System).

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.

Consider the diagram below to understand the relation between process and its thread:

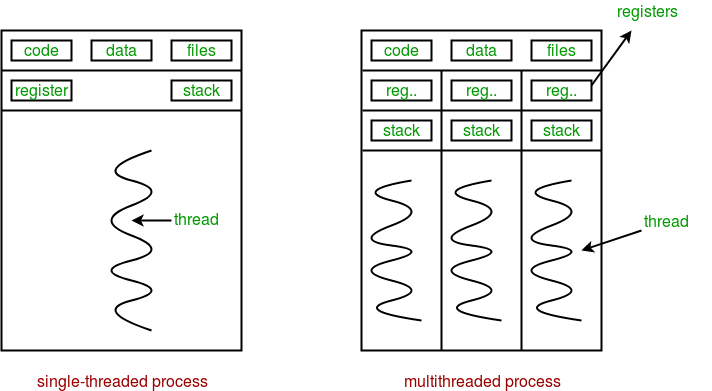


**Multithreading**

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**.

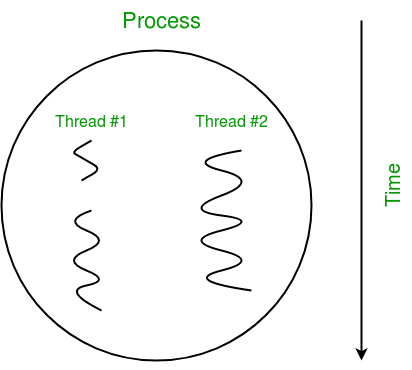
Consider the diagram below to understand how multiple threads exist in memory:



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

*In a simple, single-core CPU, it is achieved using frequent switching between threads. This is termed as****context switching****. In context switching, the state of a thread is saved and state of another thread is loaded whenever any interrupt (due to I/O or manually set) takes place. Context switching takes place so frequently that all the threads appear to be running parallely (this is termed as****multitasking****).*

Consider the diagram below in which a process contains two active threads:



**Multithreading in Python**

There are two modules which support the usage of threads in Python:

* thread
* threading

The thread module has been considered as "deprecated" for quite a long time. Users have been encouraged to use the threading module instead. So,in Python 3 the module "thread" is not available anymore. But that's not really true: It has been renamed to "\_thread" for backwards incompatibilities in Python3.   
The module "thread" treats a thread as a function, while the module "threading" is implemented in an object oriented way

**threading** module provides a very simple and intuitive API for spawning multiple threads in a program.

Let us consider a simple example using threading module:

|  |
| --- |
| # Python program to illustrate the concept  # of threading  # importing the threading module  import threading    def print\_cube(num):      """      function to print cube of given num      """      print("Cube: {}".format(num \* num \* num))    def print\_square(num):      """      function to print square of given num      """      print("Square: {}".format(num \* num))    if \_\_name\_\_ == "\_\_main\_\_":      # creating thread      t1 = threading.Thread(target=print\_square, args=(10,))      t2 = threading.Thread(target=print\_cube, args=(10,))        # starting thread 1      t1.start()      # starting thread 2      t2.start()        # wait until thread 1 is completely executed      t1.join()      # wait until thread 2 is completely executed      t2.join()        # both threads completely executed      print("Done!") |

Square: 10

Cube: 1000

Done!

Let us try to understand the above code:

* To import the threading module, we do:
* import threading
* To create a new thread, we create an object of **Thread** class. It takes following arguments:
  + **target**: the function to be executed by thread
  + **args**: the arguments to be passed to the target function

In above example, we created 2 threads with different target functions:

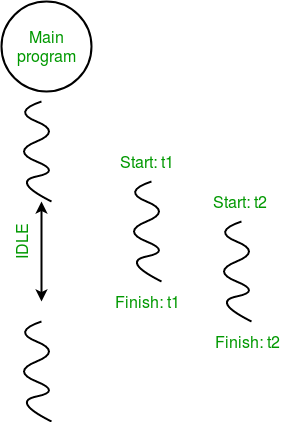
t1 = threading.Thread(target=print\_square, args=(10,))

t2 = threading.Thread(target=print\_cube, args=(10,))

* To start a thread, we use **start** method of **Thread** class.
* t1.start()
* t2.start()
* Once the threads start, the current program (you can think of it like a main thread) also keeps on executing. In order to stop execution of current program until a thread is complete, we use **join** method.
* t1.join()
* t2.join()

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

Consider the diagram below for a better understanding of how above program works:



Consider the python program given below in which we print thread name and corresponding process for each task:

|  |
| --- |
| # Python program to illustrate the concept  # of threading  import threading  import os    def task1():      print("Task 1 assigned to thread: {}".format(threading.current\_thread().name))      print("ID of process running task 1: {}".format(os.getpid()))    def task2():      print("Task 2 assigned to thread: {}".format(threading.current\_thread().name))      print("ID of process running task 2: {}".format(os.getpid()))    if \_\_name\_\_ == "\_\_main\_\_":        # print ID of current process      print("ID of process running main program: {}".format(os.getpid()))        # print name of main thread      print("Main thread name: {}".format(threading.main\_thread().name))        # creating threads      t1 = threading.Thread(target=task1, name='t1')      t2 = threading.Thread(target=task2, name='t2')        # starting threads      t1.start()      t2.start()        # wait until all threads finish      t1.join()      t2.join() |

ID of process running main program: 11758

Main thread name: MainThread

Task 1 assigned to thread: t1

ID of process running task 1: 11758

Task 2 assigned to thread: t2

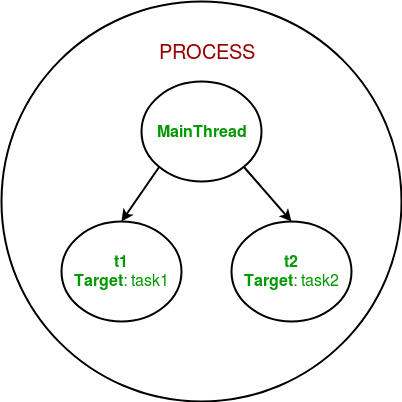
ID of process running task 2: 11758

Let us try to understand the above code:

* We use **os.getpid()** function to get ID of current process.
* print("ID of process running main program: {}".format(os.getpid()))

As it is clear from the output, the process ID remains same for all threads.

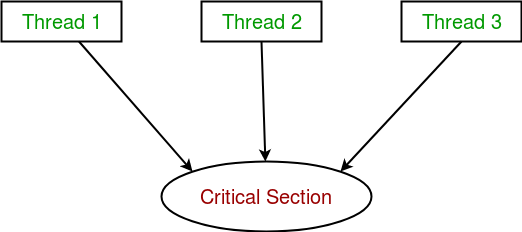
* We use **threading.main\_thread()** function to get the main thread object. In normal conditions, the main thread is the thread from which the Python interpreter was started. **name** attribute of thread object is used to get the name of thread.
* print("Main thread name: {}".format(threading.main\_thread().name))
* We use the **threading.current\_thread()** function to get the current thread object.
* print("Task 1 assigned to thread: {}".format(threading.current\_thread().name))

The diagram given below clears the above concept:  


**Synchronization between threads**

Thread synchronization is defined as a mechanism which ensures that two or more concurrent threads do not simultaneously execute some particular program segment known as **critical section**.

*Critical section refers to the parts of the program where the shared resource is accessed.*

For example, in the diagram below, 3 threads try to access shared resource or critical section at the same time.  


Concurrent accesses to shared resource can lead to **race condition**.

*A race condition occurs when two or more threads can access shared data and they try to change it at the same time. As a result, the values of variables may be unpredictable and vary depending on the timings of context switches of the processes.*

Consider the program below to understand the concept of race condition:

|  |
| --- |
| import threading    # global variable x  x = 0    def increment():      """      function to increment global variable x      """      global x      x += 1    def thread\_task():      """      task for thread      calls increment function 100000 times.      """      for \_ in range(100000):          increment()    def main\_task():      global x      # setting global variable x as 0      x = 0        # creating threads      t1 = threading.Thread(target=thread\_task)      t2 = threading.Thread(target=thread\_task)        # start threads      t1.start()      t2.start()        # wait until threads finish their job      t1.join()      t2.join()    if \_\_name\_\_ == "\_\_main\_\_":      for i in range(10):          main\_task()          print("Iteration {0}: x = {1}".format(i,x)) |

Output

Iteration 0: x = 175005

Iteration 1: x = 200000

Iteration 2: x = 200000

Iteration 3: x = 169432

Iteration 4: x = 153316

Iteration 5: x = 200000

Iteration 6: x = 167322

Iteration 7: x = 200000

Iteration 8: x = 169917

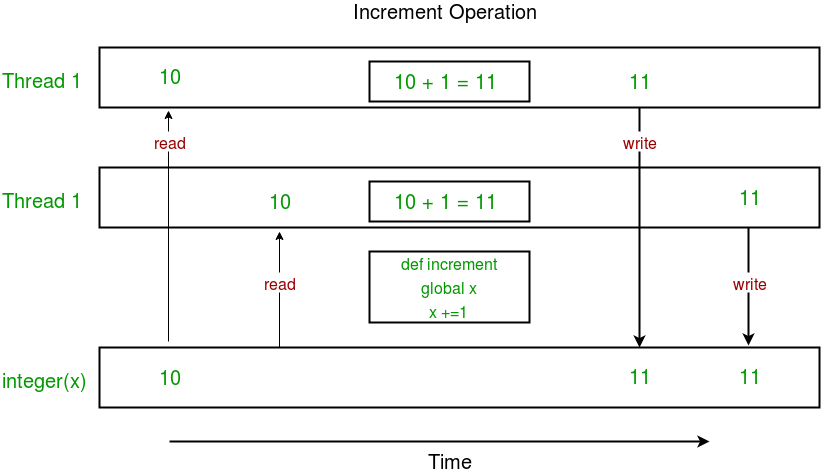
Iteration 9: x = 153589

In above program:

* Two threads **t1** and **t2** are created in **main\_task** function and global variable **x** is set to 0.
* Each thread has a target function **thread\_task** in which **increment** function is called 100000 times.
* **increment** function will increment the global variable **x** by 1 in each call.

The expected final value of **x** is 200000 but what we get in 10 iterations of **main\_task** function is some different values.

This happens due to concurrent access of threads to the shared variable **x**. This unpredictability in value of **x** is nothing but **race condition**.

Given below is a diagram which shows how can **race condition** occur in above program:  


Notice that expected value of **x** in above diagram is 12 but due to race condition, it turns out to be 11!  
  
Hence, we need a tool for proper synchronization between multiple threads.

**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.

*A semaphore is a synchronization object that controls access by multiple processes/threads to a common resource in a parallel programming environment. It is simply a value in a designated place in operating system (or kernel) storage that each process/thread can check and then change. Depending on the value that is found, the process/thread can use the resource or will find that it is already in use and must wait for some period before trying again. Semaphores can be binary (0 or 1) or can have additional values. Typically, a process/thread using semaphores checks the value and then, if it using the resource, changes the value to reflect this so that subsequent semaphore users will know to wait.*

**Lock** class provides following methods:

* **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.
* **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.

Consider the example given below:

|  |
| --- |
| import threading    # global variable x  x = 0    def increment():      """      function to increment global variable x      """      global x      x += 1    def thread\_task(lock):      """      task for thread      calls increment function 100000 times.      """      for \_ in range(100000):          lock.acquire()          increment()          lock.release()    def main\_task():      global x      # setting global variable x as 0      x = 0        # creating a lock      lock = threading.Lock()        # creating threads      t1 = threading.Thread(target=thread\_task, args=(lock,))      t2 = threading.Thread(target=thread\_task, args=(lock,))        # start threads      t1.start()      t2.start()        # wait until threads finish their job      t1.join()      t2.join()    if \_\_name\_\_ == "\_\_main\_\_":      for i in range(10):          main\_task()          print("Iteration {0}: x = {1}".format(i,x)) |

Output:

Iteration 0: x = 200000

Iteration 1: x = 200000

Iteration 2: x = 200000

Iteration 3: x = 200000

Iteration 4: x = 200000

Iteration 5: x = 200000

Iteration 6: x = 200000

Iteration 7: x = 200000

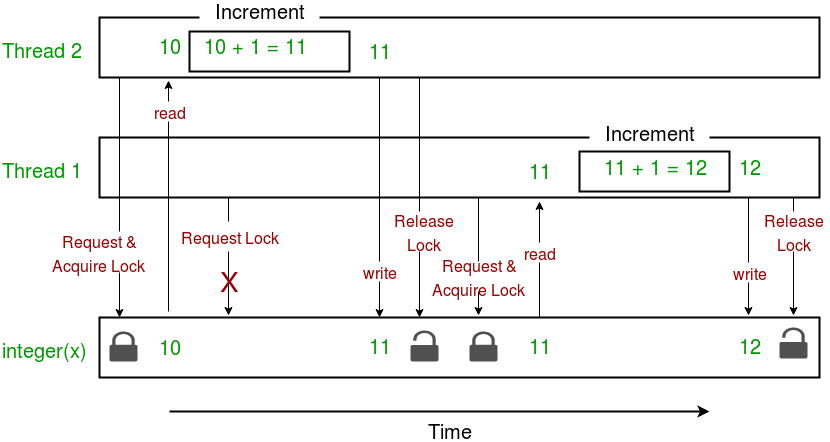
Iteration 8: x = 200000

Iteration 9: x = 200000

Let us try to understand the above code step by step:

* Firstly, a **Lock** object is created using:
* lock = threading.Lock()
* Then, **lock** is passed as target function argument:
* t1 = threading.Thread(target=thread\_task, args=(lock,))
* t2 = threading.Thread(target=thread\_task, args=(lock,))
* In the critical section of target function, we apply lock using **lock.acquire()** method. As soon as a lock is acquired, no other thread can access the critical section (here, **increment** function) until the lock is released using **lock.release()** method.
* lock.acquire()
* increment()
* lock.release()

As you can see in the results, the final value of **x** comes out to be 200000 every time (which is the expected final result).

Here is a diagram given below which depicts the implementation of locks in above program:  


This brings us to the end of this tutorial series on **Multithreading in Python**.  
Finally, here are are a few advantages and disadvantages of multithreading:

**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.

**Disadvantages:**

* As number of threads increase, complexity increases.
* Synchronization of shared resources (objects, data) is necessary.
* It is difficult to debug, result is sometimes unpredictable.
* Potential deadlocks which leads to starvation, i.e. some threads may not be served with a bad design
* Constructing and synchronizing threads is CPU/memory intensive.

## RLock Objects

A reentrant lock is a synchronization primitive that may be acquired multiple times by the same thread. Internally, it uses the concepts of “owning thread” and “recursion level” in addition to the locked/unlocked state used by primitive locks. In the locked state, some thread owns the lock; in the unlocked state, no thread owns it.

To lock the lock, a thread calls its [acquire()](https://docs.python.org/3/library/threading.html#threading.RLock.acquire) method; this returns once the thread owns the lock. To unlock the lock, a thread calls its [release()](https://docs.python.org/3/library/threading.html#threading.Lock.release) method. [acquire()](https://docs.python.org/3/library/threading.html#threading.Lock.acquire)/[release()](https://docs.python.org/3/library/threading.html#threading.Lock.release) call pairs may be nested; only the final [release()](https://docs.python.org/3/library/threading.html#threading.Lock.release) (the [release()](https://docs.python.org/3/library/threading.html#threading.Lock.release) of the outermost pair) resets the lock to unlocked and allows another thread blocked in [acquire()](https://docs.python.org/3/library/threading.html#threading.Lock.acquire) to proceed. class threading.RLock implements reentrant lock objects.

## Thread-Local Data

Thread-local data is data whose values are thread specific. To manage thread-local data, just create an instance of [local](https://docs.python.org/3/library/threading.html#threading.local) (or a subclass) and store attributes on it:

mydata = threading.local()

mydata.x = 1

The instance’s values will be different for separate threads.

# Event Object:

# This is the mechanisms for communication between threads: one thread signals an event and other threads wait for it. An event object manages an internal flag that can be set to true with the set() method and reset to false with the clear() method. The wait() method blocks until the flag is true.

# class threading.Event

# This Class implements event objects. An event manages a flag that can be set to true with the set() method and reset to false with the clear() method. The wait() method blocks until the flag is true. The flag is initially false.

## Timer Objects

This class represents an action that should be run only after a certain amount of time has passed — a timer.[Timer](https://docs.python.org/3/library/threading.html#threading.Timer) is a subclass of [Thread](https://docs.python.org/3/library/threading.html#threading.Thread) and as such also functions as an example of creating custom threads.

Timers are started, as with threads, by calling their start() method. The timer can be stopped (before its action has begun) by calling the [cancel()](https://docs.python.org/3/library/threading.html#threading.Timer.cancel) method. The interval the timer will wait before executing its action may not be exactly the same as the interval specified by the user.

For example:

**def** hello():

print("hello, world")

t = Timer(30.0, hello)

t.start() *# after 30 seconds, "hello, world" will be printed*

## Barrier Objects

This class provides a simple synchronization primitive for use by a fixed number of threads that need to wait for each other. Each of the threads tries to pass the barrier by calling the [wait()](https://docs.python.org/3/library/threading.html#threading.Barrier.wait) method and will block until all of the threads have made their [wait()](https://docs.python.org/3/library/threading.html#threading.Barrier.wait) calls. At this point, the threads are released simultaneously.

The barrier can be reused any number of times for the same number of threads.

As an example, here is a simple way to synchronize a client and server thread:

b = Barrier(2, timeout=5)

**def** server():

start\_server()

b.wait()

**while** **True**:

connection = accept\_connection()

process\_server\_connection(connection)

**def** client():

b.wait()

**while** **True**:

connection = make\_connection()

process\_client\_connection(connection)

## Locks, conditions, and semaphores can be used  [with](https://docs.python.org/3/reference/compound_stmts.html#with) statement to simplify the logic

**with** some\_lock:

*# do something...*

is equivalent to:

some\_lock.acquire()

**try**:

*# do something...*

**finally**:

some\_lock.release()

# ------------------------------------------------------------------------------------------------------------------

# From: Avinash Verma 3

**Sent:** Tuesday, April 24, 2018 10:30 AM  
**To:** Vishal Verma 2; Ashish Jain 6; Mayank Nagar 2; Tarun Dubey; Jeetendra Singh; Shalabh Mongia; Talwinder Singh; Rohit Lohiya; Ankit Arora; Rohit Kacker 2; Sundar Krishnadu; Prateek Pathak; [bathula.reddy@rbs.com](mailto:bathula.reddy@rbs.com); Pavan Kumar Kothuri 2; Charanjeet Singh; Preeti Chitkara; Avinash Saraf  
**Cc:** Vikram Verma; Anindita Bose 2; Ananya Anand; Mohit Sawhney; Prabhjinder Singh  
**Subject:** RE: [AI/ML Learning Group] Advanced Python Topics and Functional Programming

Hi Vishal/All

Adding my assignment link below:

<https://github.com/avivrm/beginpython.git>

|  |
| --- |
| [avivrm/beginpython](https://github.com/avivrm/beginpython.git)  github.com  Contribute to beginpython development by creating an account on GitHub. |

Will continue to update as per assignment. Feel free to make comments and also open discussion in issue tracker of repo.

-regards

Avinash Verma

**From:** Vishal Verma 2   
**Sent:** Monday, April 23, 2018 11:45 AM  
**To:** Avinash Verma 3 <[averma63@sapient.com](mailto:averma63@sapient.com)>; Ashish Jain 6 <[ajain212@sapient.com](mailto:ajain212@sapient.com)>; Mayank Nagar 2 <[mnagar22@sapient.com](mailto:mnagar22@sapient.com)>; Tarun Dubey <[tdubey@sapient.com](mailto:tdubey@sapient.com)>; Jeetendra Singh <[jsingh51@sapient.com](mailto:jsingh51@sapient.com)>; Shalabh Mongia <[smongia@sapient.com](mailto:smongia@sapient.com)>; Talwinder Singh <[tsingh24@sapient.com](mailto:tsingh24@sapient.com)>; Rohit Lohiya <[rlohiya@sapient.com](mailto:rlohiya@sapient.com)>; Ankit Arora <[aarora34@sapient.com](mailto:aarora34@sapient.com)>; Rohit Kacker 2 <[rkacker2@sapient.com](mailto:rkacker2@sapient.com)>; Sundar Krishnadu <[skrishnadu@sapient.com](mailto:skrishnadu@sapient.com)>; Prateek Pathak <[ppathak9@sapient.com](mailto:ppathak9@sapient.com)>; [bathula.reddy@rbs.com](mailto:bathula.reddy@rbs.com); Pavan Kumar Kothuri 2 <[pkothuri2@sapient.com](mailto:pkothuri2@sapient.com)>; Charanjeet Singh <[csingh2@sapient.com](mailto:csingh2@sapient.com)>; Preeti Chitkara <[pchitkara2@sapient.com](mailto:pchitkara2@sapient.com)>; Avinash Saraf <[asaraf3@sapient.com](mailto:asaraf3@sapient.com)>  
**Cc:** Vikram Verma <[vverma22@sapient.com](mailto:vverma22@sapient.com)>; Anindita Bose 2 <[abose4@sapient.com](mailto:abose4@sapient.com)>; Ananya Anand <[aanand2@sapient.com](mailto:aanand2@sapient.com)>; Mohit Sawhney <[msawhney@sapient.com](mailto:msawhney@sapient.com)>; Prabhjinder Singh <[psingh91@sapient.com](mailto:psingh91@sapient.com)>  
**Subject:** Re: [AI/ML Learning Group] Advanced Python Topics and Functional Programming

Learning Never Exhausts the Mind - de Vinci

Lets dive deep in the key concepts and do more hands on!!!

## Overview

         Advanced Python techniques and topics:

o    functions as parameters

o    sorting and keys

o    map, reduce, filter

o    lambda functions

o    xranges, iterators and generators

         Efficiency, simplicity and abstraction

         Functional programmming

## Motivation

         We’ll consider functions that are applied to entire lists: sum, min, max, slicing, and we will compare them to the code we would write to how we would generate them with just while loops.

         We should notice important commonality with relatively minor differences

         Functional programming, which we discuss in this lecture, allows us to exploit this commonality and write simpler expressions at a higher level of abstraction

         It also leads us into a different style of programming called functional programming

## Functions as parameters

         Sort comparison function must produce -1 for less than, 0 for equal, and +1 for greater than.

         **|**  sort(**...**)

         **|**      L**.**sort(cmp**=None**, key**=None**, reverse**=False**) **--** stable sort **\***IN PLACE**\***;

         **|**      cmp(x, y) **->** **-**1, 0, 1

         We write this function and then pass it as an argument to the sort function associated with the list object.

         In other words, the function is treated exactly the same way as any other argument passed to a function.

         We will look at example of sorting names by last name and then by first.

## Map: Apply a function to each element of a list

         Suppose we want to count the number of values in a list of lists. We can use map to apply the len function to each sublist.

         >>> v **=** [ [2, 3, 5, 7], [11,13,17,19], [23, 29], [31,37] ]

         >>> print map( len, v)

         [4, 4, 2, 2]

         In order to get our final answer, we just apply `sum`:

         >>> print sum(map(len,v))

         12

         Now suppose we want to find the maximum distance of a list of points from the origin. Here we’ll have to write a function

         **def** **dist2D**( p ):

             **return** (p[0]**\*\***2 **+** p[1]**\*\***2)**\*\***0.5



         pts **=** [ (4.5, 3), (2.1,**-**1), (6.8,**-**3), (1.4, 2.9) ]

         print map( dist2D, pts)

         print max( map(dist2D,pts) )

## Lambda functions: Anonymous functions

         We can avoid writing a separate function here by writing an anonymous function called a lambda function.

o    Aside: the notion of a lambda function goes all the way back to the origin of computer science

         Our first example is just squaring the values of a list

         >>> map( **lambda** x: x**\*\***2, [ 1, 2, 3, 4 ] )

         [ 1, 4, 9, 16 ]

         Now, we can sum the squares from 1 to n

         >>> n **=** 100

         >>> sum( map( **lambda** x: x**\*\***2, range(1,n**+**1)))

         Our second example implements the dist2D function anonymously:

         >>> max( map( **lambda** p: (p[0]**\*\***2 **+** p[1]**\*\***2)**\*\***0.5, pts) )

         7.432361670424818

## Exercises:

1.    Use map to generate a new list where all values are replaced by their absolute values. You don’t need to use a lambda function.

2.    Use map and a lambda function to convert a list of Fahrenheit temperatures to a list of Celsius temperatures.

3.    Use map and a lambda function to find the maximum x coordinate (the 0-th coordinate) in a list of points. You will need to apply max to the result of the map

## Reduce: Combine entries of a list into a single summary value

         The functions sum and max reduce a list to a single value.

         Python contains a special function called reduce that allows you to apply functions that already exist or that you write - lambda functions or named functions - to create this behavior in other ways

o    These functions must involve two arguments

         The simplest is to just recreate the summation function:

         >>> reduce( **lambda** x,y: x**+**y, xrange(1,101))

         5050

         Note that this also gives us another (!) way to concantentate strings.

## Filter: Extract / eliminate values from a list

         Let’s think about how to eliminate all of the negative values from a list, using a for loop. We will work on this in class

         Once again we can we can simplify this using the built-in Python construct called filter

         >>> v **=** [ 1, 9, **-**4, **-**8, 10, **-**3 ]

         >>> filter( **lambda** x: x**>**0, v)

         [1, 9, 10]

         Here the lambda function must produce a boolean value and if that value is True the list item is preserved.

## Exercises

1.    Use filter to eliminate all words that are shorter than 4 letters from a list of words

2.    Use filter to determine the percentage of Fahrenheit temperatures in a list are within the range 32 to 80

3.    Use reduce to find the lower left corner (minimum x and minimum y value) for a list of point locations

## List Comprehensions

         Instead of map and filter some people prefer another example of functional programming in Python called list comprehensions

         Here is an example to generate the squares of the first n integers:

         n **=** 8

         **>>>** [ i**\***i **for** i **in** xrange(1,n**+**1) ]

         [1, 4, 9, 16, 25, 36, 49, 64]

         The form of this is an expression followed by a for loop statement.

         We can get the effect of filter by adding a conditional at the end:

         >>> v **=** [ 1, 9, **-**4, **-**8, 10, **-**3 ]

         >>> [ x **for** x **in** v **if** x**>**0 ]

         [1, 9, 10]

         Here, the values are only generated in the resultant list when the if condition passes.

         We can combine these as well. As a slightly silly example, we can eliminate the negative values and square the positive values

         >>> v **=** [ 1, 9, **-**4, **-**8, 10, **-**3 ]

         >>> [ x**\***x **for** x **in** v **if** x**>**0 ]

         [1, 81, 100]

         We can get even more sophisticated by nesting for loops. Here is an example where we generate all pairs of numbers between 1 and 4, except for the pairs where the numbers are equal

         >>> [ (i,j) **for** i **in** xrange(1,5) **for** j **in** xrange(1,5) **if** i **!=** j ]

         [(1, 2), (1, 3), (1, 4), (2, 1), (2, 3), (2, 4), (3, 1), (3, 2),

          (3, 4), (4, 1), (4, 2), (4, 3)]

## Exercises

1.    Write a list comprehension statement to convert a list of Fahrenheit temperatures to Celsius

2.    Write a list comprehension statement to generate a list of all pairs of odd positive integer values less than 10 where the first value is less than the second value.

- Vishal

**From:** Vishal Verma 2  
**Sent:** Thursday, April 19, 2018 11:31 AM  
**To:** Avinash Verma 3; Ashish Jain 6; Mayank Nagar 2; Tarun Dubey; Jeetendra Singh; Shalabh Mongia; Talwinder Singh; Rohit Lohiya; Ankit Arora; Rohit Kacker 2; Sundar Krishnadu; Prateek Pathak; [bathula.reddy@rbs.com](mailto:bathula.reddy@rbs.com); Pavan Kumar Kothuri 2; Charanjeet Singh; Preeti Chitkara; Avinash Saraf  
**Cc:** Vikram Verma; Anindita Bose 2; Ananya Anand; Mohit Sawhney; Prabhjinder Singh  
**Subject:** [AI/ML Learning Group] More assignments = More Fun + Probability Cheat Sheet

Don't let perfect be the enemy of Good

Keep the good work going. These exercises are to unblock you while thinking in Python. Discuss with in your group. Let me know if they are hard or not relevant.

If you feel you are not motivated enough, I will work on it. If lack of time is the issue, try resting the clock. Start doing things 15 mins earlier.

Suggestions and inputs are required to make this learning experience more interesting...

Exercises:

|  |  |  |  |
| --- | --- | --- | --- |
| **S.No** | **Program** | **Level** | **Hint** |
| 11 | Write a program which accepts a sequence of comma separated 4 digit binary numbers as its input and then check whether they are divisible by 5 or not. The numbers that are divisible by 5 are to be printed in a comma separated sequence.  Example:  0100,0011,1010,1001  Then the output should be:  1010  Notes: Assume the data is input by console. | 2 | In case of input data being supplied to the question, it should be assumed to be a console input. |

12

Write a program, which will find all such numbers between 1000 and 3000 (both included) such that each digit of the number is an even number.

The numbers obtained should be printed in a comma-separated sequence on a single line.

2

In case of input data being supplied to the question, it should be assumed to be a console input.

13

Question:

Write a program that accepts a sentence and calculate the number of letters and digits.

Suppose the following input is supplied to the program:

hello world! 123

Then, the output should be:

LETTERS 10

DIGITS 3

2

In case of input data being supplied to the question, it should be assumed to be a console input.

14

Write a program that accepts a sentence and calculate the number of upper case letters and lower case letters.

Suppose the following input is supplied to the program:

Hello world!

Then, the output should be:

UPPER CASE 1

LOWER CASE 9

2

In case of input data being supplied to the question, it should be assumed to be a console input.

15

Write a program that computes the value of a+aa+aaa+aaaa with a given digit as the value of a.

Suppose the following input is supplied to the program:

9

Then, the output should be:

11106

2

In case of input data being supplied to the question, it should be assumed to be a console input.

16

Use a list comprehension to square each odd number in a list. The list is input by a sequence of comma-separated numbers.

Suppose the following input is supplied to the program:

1,2,3,4,5,6,7,8,9

Then, the output should be:

1,3,5,7,9

2

In case of input data being supplied to the question, it should be assumed to be a console input.

17

Write a program that computes the net amount of a bank account based a transaction log from console input. The transaction log format is shown as following:

D 100

W 200

¡­

D means deposit while W means withdrawal.

Suppose the following input is supplied to the program:

D 300

D 300

W 200

D 100

Then, the output should be:

500

2

In case of input data being supplied to the question, it should be assumed to be a console input

18

A website requires the users to input username and password to register. Write a program to check the validity of password input by users.

Following are the criteria for checking the password:

1. At least 1 letter between [a-z]

2. At least 1 number between [0-9]

1. At least 1 letter between [A-Z]

3. At least 1 character from [$#@]

4. Minimum length of transaction password: 6

5. Maximum length of transaction password: 12

Your program should accept a sequence of comma separated passwords and will check them according to the above criteria. Passwords that match the criteria are to be printed, each separated by a comma.

Example

If the following passwords are given as input to the program:

ABd1234@1,a F1#,2w3E\*,2We3345

Then, the output of the program should be:

ABd1234@1

3

In case of input data being supplied to the question, it should be assumed to be a console input.

19

You are required to write a program to sort the (name, age, height) tuples by ascending order where name is string, age and height are numbers. The tuples are input by console. The sort criteria is:

1: Sort based on name;

2: Then sort based on age;

3: Then sort by score.

The priority is that name > age > score.

If the following tuples are given as input to the program:

Tom,19,80

John,20,90

Jony,17,91

Jony,17,93

Json,21,85

Then, the output of the program should be:

[('John', '20', '90'), ('Jony', '17', '91'), ('Jony', '17', '93'), ('Json', '21', '85'), ('Tom', '19', '80')]

3

In case of input data being supplied to the question, it should be assumed to be a console input.

We use itemgetter to enable multiple sort keys.

20

Define a class with a generator which can iterate the numbers, which are divisible by 7, between a given range 0 and n.

3

Consider using yield