PYTHON THREADING

GOAL

The purpose of this lab is to demonstrate a step-by-step introduction in using threads in Python.

The following subjects will be covered:

- **Creating Multiple Threads with Python**
- **Execution Scheduling**
- **Thread Lifecycle**
- **Daemon Threads**
- **Data Race**
- **Mutual Exclusion**

Proceed through the pages of this document until you have gone through all the exercises.

EXERCISE 01: MULTIPLE THREADS

[Filename: **01_Exercise_multiple_threads.py**]

We'll start with a demonstration, using Python's threading module to create several concurrent threads.

- This program defines a simple function on line 7, called CPU waster, which has a while loop that will spin forever. It doesn't do any useful work, but the thread running that function will stay alive forever and continuously use CPU cycles.
- Lines 12 through 15 print out information about the program, including its process ID number and the total number of threads in the process.
- The for loop on line 14 prints information about each of those threads.
- After that, it creates and starts 12 time waster threads, using a for loop on line 18-19.

01_Exercise_multiple_threads.py

```
1  #""" Threads that waste CPU cycles """
2
3  import os
4  import threading
5
6  # a simple function that wastes CPU cycles forever
7  def cpu_waster():
8      while True:
9          pass
10
11 # display information about this process
12 print('\n  Process ID: ', os.getpid())
13 print('Thread Count: ', threading.active_count())
14 for thread in threading.enumerate():
15     print(thread)
16
17 print('\nStarting 12 CPU Wasters...')
18 for i in range(12):
19     threading.Thread(target=cpu_waster).start()
20
21 # display information about this process
22 print('\n  Process ID: ', os.getpid())
23 print('Thread Count: ', threading.active_count())
24 for thread in threading.enumerate():
25     print(thread)
26
```

EXERCISE 02: EXECUTION SCHEDULING

[Filename: **02_Exercise_execution_scheduling.py**]

To demonstrate how scheduling can impact execution we will run this Python program that **creates two threads** named Anna and George that continuously chop vegetables for about one second.

- In the vegetable_chopper function beginning on line 8, the program uses the threading module's current_thread and getName methods to retrieve the name of the current thread. Then it initializes a local variable to count the number of vegetables this thread chops.
- The while loop on line 11 will execute as long as the chopping variable that was initialized on line 6 is True. Within each loop iteration it'll print a message and increment the value of the local vegetable_count variable.
- Finally after the while loop finishes, the vegetable_chopper function prints out how many total vegetables it chopped on line 14.
- Down in the main section on lines 17 and 18, we create and start two threads to execute the vegetable_chopper function and we pass in the strings Anna and George to the optional name parameter. Python allows you to set a thread's name to whatever you want for your own use to identify the threads.

02_Exercise_execution_scheduling.py

```
1  # """ Two threads chopping vegetables """
2
3  import threading
4  import time
5
6  chopping = True
7
8  def vegetable_chopper():
9      name = threading.current_thread().getName()
10     vegetable_count = 0
11     while chopping:
12         print(name, 'chopped a vegetable!')
13         vegetable_count += 1
14     print(name, 'chopped', vegetable_count, 'vegetables.')
15
16 if __name__ == '__main__':
17     threading.Thread(target=vegetable_chopper, name='Anna').start()
18     threading.Thread(target=vegetable_chopper, name='George').start()
19
20     time.sleep(1)    # chop vegetables for 1 second
21     chopping = False # stop both threads from chopping
22
```

EXERCISE 03: THREAD LIFECYCLE

[Filename: **03_Exercise_thread_lifecycle.py**]

To demonstrate the life cycle of a Python thread, from creation to termination, we will run this example program.

- The program recreates the interactions between Anna and George, where George spawns Anna as a second thread to help slice sausages to make soup.
- **There are two ways to create a thread and specify its activity in Python**. In the previous Python examples, we put the code for our thread to execute into a function. And then passed that function to the thread constructor method as a callable object using the target parameter.
- The other way to create a thread in Python is to define a custom subclass that inherits from the thread class and overrides its run method.
- That second approach is what we've done with the class named ChefAnna on line 6. It inherits from threading.Thread and overrides two of its methods, init and run.
- **These are the only two methods you should override from the thread class**. Within the init method, we simply use the super function to execute the parent thread class' init method on line 9. Python will raise an error if you don't do that.
- Since George only sleeps for half a second after starting Anna's thread, but Anna sleeps for three whole seconds, George will be done well before Anna.
- But he needs to wait until Anna finishes to continue on. This is where the join method comes into play, causing George to wait until after Anna has completed everything she needs to do before he can continue on.
- George calls the join method on line 31. Notice that we're calling the join method on the Anna thread object from within the main George thread. **That'll block George's execution at that point until Anna terminates.**
- After Anna's thread terminates, the main George thread will be able to continue on to print its final message on line 34 that they're both done.

03_Exercise_thread_lifecycle.py

```python
# """ Two threads cooking soup """

import threading
import time

class ChefAnna(threading.Thread):

    def __init__(self):
        super(ChefAnna, self).__init__()

    def run(self):
        print('Anna started & waiting for sausage to thaw...')
        time.sleep(3)
        print('Anna is done cutting sausage.')

# main thread (e.g. ChefGeorge)
if __name__ == '__main__':
    print("George started & requesting Anna's help.")
    anna = ChefAnna()
    print('  Anna alive?:', anna.is_alive())

    print('George tells Anna to start.')
    anna.start()
    print('  Anna alive?:', anna.is_alive())

    print('George continues cooking soup.')
    time.sleep(0.5)
    print('  Anna alive?:', anna.is_alive())

    print('George patiently waits for Anna to finish and join...')
    anna.join()
    print('  Anna alive?:', anna.is_alive())

    print('George and Anna are both done!')

```

EXERCISE 04: DAEMON THREADS

[Filename: **04_Exercise_daemon_thread.py**]

To demonstrate a daemon thread, we define a function called kitchen cleaner on line 6, which represents a periodic background task like garbage collection.

- The kitchen cleaner uses an infinite while loop to continuously print a message that Olivia cleaned the kitchen once every second.
- Down in the program's main section, we create and start a new kitchen cleaner thread named Anna on lines 12 and 14.
- Then the main thread prints a series of messages that George is cooking, which are split up by sleep statements and then finally a message that George is done on line 22.

04_Exercise_daemon_thread.py

```
1  # """ George finishes cooking while Anna cleans """
2
3  import threading
4  import time
5
6  def kitchen_cleaner():
7      while True:
8          print('Anna cleaned the kitchen.')
9          time.sleep(1)
10
11 if __name__ == '__main__':
12     anna = threading.Thread(target=kitchen_cleaner)
13     anna.daemon = True
14     anna.start()
15
16     print('George is cooking...')
17     time.sleep(0.6)
18     print('George is cooking...')
19     time.sleep(0.6)
20     print('George is cooking...')
21     time.sleep(0.6)
22     print('George is done!')
23
```

- Comment line 13 and run the program again to see what is happening.

- We set Anna to be a daemon thread before she gets started by typing anna.daemon = true.
- **When a new thread is created, it'll inherit the daemon status from its parent**. In our case, the main thread is normal non-daemon thread, so by default, any threads that it creates will be non-daemon threads.
- **You must set the daemon property to configure a thread to be daemon or non-daemon before starting it.** Otherwise Python will raise runtime error.
- Daemon threads do not gracefully exit like normal threads. When all of the non-daemon threads in a program are done executing, any remaining daemon threads will be abandoned as Python exits.

EXERCISE 05: DATA RACE

[Filename: **05_Exercise_data_race.py**]

In the following code, we can see an example of a data race, where two threads have access to the same variable, namely garlic_count, without providing for protecting the variable.

- First run this code, and then change the range of line 9 to 100000.
- Run the code several times. What do you observe and why do you think this is happening?

05_Exercise_data_race.py

```
1  # """ Two shoppers adding items to a shared notepad """
2
3  import threading
4
5  garlic_count = 0
6
7  def shopper():
8      global garlic_count
9      for i in range(10):
10         garlic_count += 1
11
12 if __name__ == '__main__':
13     george = threading.Thread(target=shopper)
14     anna = threading.Thread(target=shopper)
15     george.start()
16     anna.start()
17     george.join()
18     anna.join()
19     print('We should buy', garlic_count, 'garlic.')
20
```

EXERCISE 06: MUTUAL EXCLUSION

[Filename: **06_Exercise_mutual_exclusion.py**]

To demonstrate how to manually enforce mutual exclusion with locks in Python, we'll modify the example program from earlier with two shoppers that have a data race as they can currently increment the amount of garlic to buy.

- Locks are included as part of Python's threading module which is imported on line 3.
- We may create a new lock object using the constructor method from that module, Threading.Lock.
- Now, to keep the two shopper threads from modifying the garlic count variable at the same time, we call the pencil.acquire method before increasing the garlic_counter on line 14 and then we call the pencil.release method immediately after the for loop has finished.
- This prevents the data race and we get the expected output.

06_Exercise_mutual_exclusion.py

```
1  # """ Two shoppers adding items to a shared notepad """
2
3  import threading
4  import time
5
6  garlic_count = 0
7  pencil = threading.Lock()
8
9  def shopper():
10     global garlic_count
11     for i in range(10):
12         print(threading.current_thread().getName(), 'is thinking.')
13         time.sleep(0.5)
14         pencil.acquire()
15         garlic_count += 1
16         pencil.release()
17
18 if __name__ == '__main__':
19     george = threading.Thread(target=shopper)
20     anna = threading.Thread(target=shopper)
21     george.start()
22     anna.start()
23     george.join()
24     anna.join()
25     print('We should buy', garlic_count, 'garlic.')
26
```