

Python Threading Tutorial

Contents

Python Threading Tutorial	1
Video	1
Threading.....	1
Process	2
Time Slicing.....	2
Tutorial 1: Without Threading	3
Tutorial 2: Threading	4
Tutorial 3: Threading with Join.....	5
Tutorial 4: Daemon Threads and Tkinter	7
Assignment Submission.....	11

Time required: 60 minutes

Video

This is a 3 hour class that goes into this subject very deeply. Watch this if you want to take a deep dive into threading.

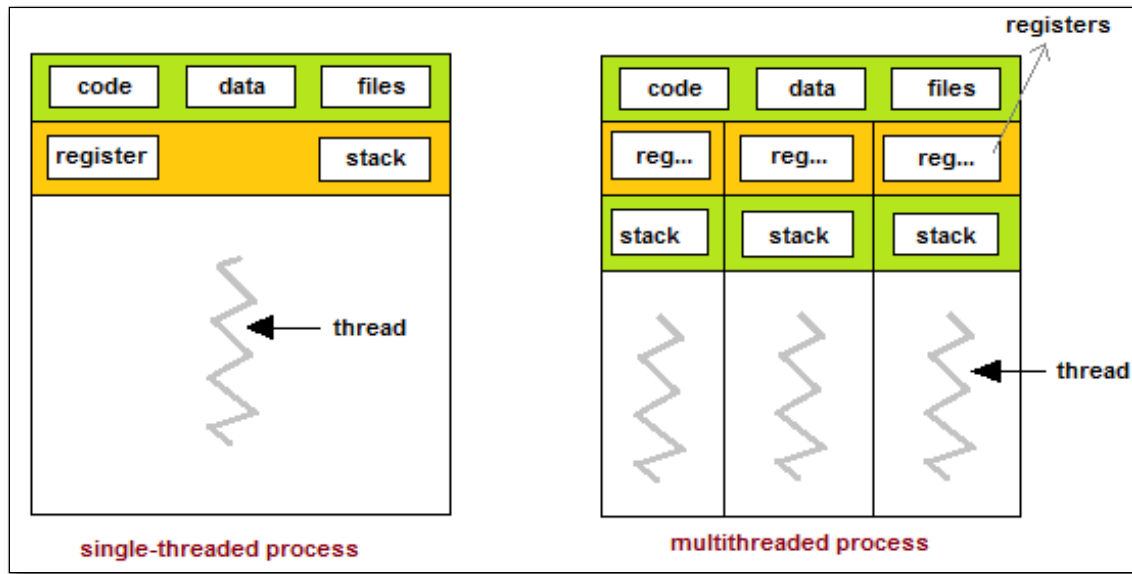
[Asynchronous Python Mastery: Threads, processes, and Async Development](#)

Threading

Threading is built into Python. Threading in Python is used to run multiple threads (tasks, function calls) at the same time. Python threads are used in cases where the execution of a task involves some waiting. One example would be interaction with a service hosted on another computer, such as a webserver. Threading allows python to execute other code while waiting.

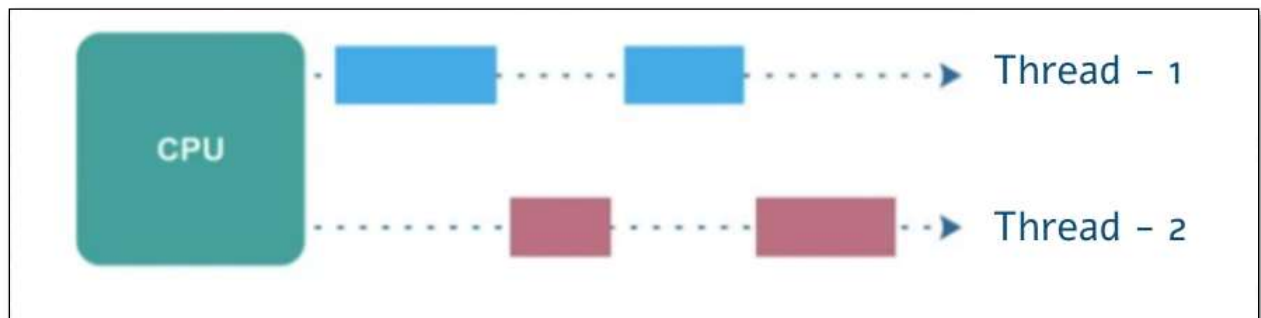
Process

A process has at least one thread and some resources set aside by the operating system. This could be a hard drive, processor cores, network, etc. The resources change as the process's needs change.



Time Slicing

Your computer can have hundreds of processes running at the same time. Time slicing or scheduling allows these threads to share process time. Time slicing is not free. The OS has to save the current status of the thread so that when it comes back to the core, it doesn't start from the beginning. There is some overhead in switching from one thread to another.



Tutorial 1: Without Threading

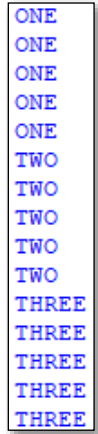
All program runs on at least one single thread. All code executes nose to tail, one after the other. Functions must run one after the other. These are called blocking calls. Nothing else can happen while the function is running.

- function1()
- function2()
- function3()

This program shows an example of functions running in a single thread.

```
1  """
2      Filename: threading_example.py
3  """
4  import threading
5
6
7  # ----- TEST FUNCTIONS -----#
8  def function1():
9      for i in range(5):
10         print("ONE ")
11
12
13  def function2():
14      for i in range(5):
15         print("TWO ")
16
17
18  def function3():
19      for i in range(5):
20         print("THREE ")
21
22
23  def main():
24      # ----- NORMAL LINEAR FUNCTIONS -----#
25      # If we call these functions, the first function call
26      # MUST complete before the next, they are executed linearly
27      function1()
28      function2()
29      function3()
30
31
32  # If a standalone program, call the main function
33  # Else, use as a module
34  if __name__ == "__main__":
35      main()
```

Example run:



ONE
ONE
ONE
ONE
ONE
TWO
TWO
TWO
TWO
TWO
THREE
THREE
THREE
THREE
THREE

Tutorial 2: Threading

Threading allows us to speed up programs by executing multiple tasks at the SAME time.

- Each task will run on its own thread
- Each thread can run simultaneously and share data with each other
- Every thread you start must do SOMETHING
- Threads will finish at different times. The OS task scheduler runs them when it has time.

We will define 3 different functions, one for each thread. Our threads will then target these functions. When we start the threads, the target functions will be run.

Modify the existing code as shown.

```

23 def main():
24     # ----- NORMAL LINEAR FUNCTIONS -----#
25     # If we call these functions, the first function call
26     # MUST complete before the next, they are executed linearly
27     # function1()
28     # function2()
29     # function3()
30
31     # ----- THREADED FUNCTIONS -----#
32     # We can execute these functions concurrently using threads!
33     # We must have a target for a thread.
34     t1 = threading.Thread(target=function1)
35     t2 = threading.Thread(target=function2)
36     t3 = threading.Thread(target=function3)
37
38     t1.start()
39     t2.start()
40     t3.start()

```

Example run (Each run may be different):

```

ONE
TWO
TWO
ONE
TWO
TWO
ONE
ONE
ONE
TWO
THREE
THREE
THREE
THREE
THREE

```

Tutorial 3: Threading with Join

Modify the existing code to add a print statement. Notice that it executes right away on the program thread.

```

23 def main():
24     # ----- NORMAL LINEAR FUNCTIONS -----#
25     # If we call these functions, the first function call
26     # MUST complete before the next, they are executed linearly
27     # function1()
28     # function2()
29     # function3()
30
31     # ----- THREADED FUNCTIONS -----#
32     # We can execute these functions concurrently using threads!
33     # We must have a target for a thread.
34     t1 = threading.Thread(target=function1)
35     t2 = threading.Thread(target=function2)
36     t3 = threading.Thread(target=function3)
37
38     t1.start()
39     t2.start()
40     t3.start()
41
42     # This pauses the main program until the thread is complete
43     # t1.join()
44     # t2.join()
45     # t3.join()
46     print("Threading rules!")

```

Example run (Each run may be different):

```

ONE
ONE
ONE
ONE
ONE
TWO
THREE
THREE
THREE
THREE
THREE
TWO
TWO
TWO
TWO
Threading rules!

```

Uncomment the **.join()** methods. The **.join()** method releases the thread and allows the program to "join" the main thread.

This is an example run with join. All three threads must complete before the next command is run.

Example run (Each run may be different):

```
ONE
ONE
TWO
TWO
ONE
ONE
ONE
Threading rules!
THREE
THREE
TWO
TWO
TWO
THREE
THREE
THREE
```

Tutorial 4: Daemon Threads and Tkinter

A daemon is a background service. If you are running a function or method in a separate thread that you want to keep going until the program stops, setting `daemon` to `True` will stop the thread when the program ends.

In a Tkinter application, the main event loop (`mainloop()`) is responsible for handling all GUI updates and user interactions. If a long-running task is executed within this main thread, it will block the event loop, causing the GUI to become unresponsive until the task completes.

Threading helps by allowing long-running tasks to run in separate threads, freeing up the main thread to continue handling GUI updates and user interactions. This ensures that the GUI remains responsive even while background tasks are being executed.

In the provided code, the **`background_task`** method runs in a separate thread created by the **`start_thread`** method. This allows the label to be updated with the current timestamp every second without blocking the main event loop, keeping the GUI responsive.

This program can be used as a template for any Python GUI program to run multiple threads.

```

1  import tkinter as tk
2  from threading import Thread, Event
3  from time import localtime, sleep
4
5
6  class ThreadingApp:
7      def __init__(self):
8          # Create the main window of the application
9          self.root = tk.Tk()
10         # Set the title of the window
11         self.root.title("Threading with Tkinter")
12         # Set the size of the window
13         self.root.geometry("150x150")
14         # Call the method to set up the GUI elements
15         self.setup_gui()
16
17         # Event to signal the thread to stop
18         self.stop_event = Event()
19
20         # Boolean flag to track the thread state
21         self.is_running = False
22
23         # Handle window close event
24         self.root.protocol("WM_DELETE_WINDOW", self.on_closing)
25         self.root.mainloop()
26
27     # ----- SETUP GUI ----- #
28     def setup_gui(self):
29         # Create and grid label widget to display text
30         self.lbl_display = tk.Label(self.root, text="Background Task Stopped")
31         self.lbl_display.grid(row=0, column=0, columnspan=2, pady=20)
32
33         # Create and grid a button that will start/stop the thread when clicked
34         self.toggle_button = tk.Button(
35             self.root,                                # Parent widget is the main window
36             text="Start Thread",                        # Text shown on the button
37             command=self.toggle_thread                 # Method called when button is clicked
38         )
39         self.toggle_button.grid(
40             row=1, column=0, columnspan=2, pady=20, padx=10)

```



```

42 # ----- BACKGROUND TASK ----- #
43 def background_task(self):
44     """This method runs in a separate thread
45     and performs a background task"""
46     # While the stop event is not set, keep running the task
47     while not self.stop_event.is_set():
48         # Update the label text with the current timestamp
49         self.get_local_time()
50         self.lbl_display.config(
51             text=f"Background Task Running\n {self.now}"
52         )
53
54         # Pause for 1 second before the next update
55         sleep(1)
56
57 # ----- TOGGLE THREAD ----- #
58 def toggle_thread(self):
59     if self.is_running:
60         self.stop_thread()
61     else:
62         self.start_thread()
63
64 # ----- START THREAD ----- #
65 def start_thread(self):
66     # Clear the stop event before starting the thread
67     self.stop_event.clear()
68     # Create a new thread object
69     self.thread = Thread(
70         target=self.background_task, # Function to run in the thread
71         daemon=True                 # Thread stops when program ends
72     )
73     # Start the thread
74     self.thread.start()
75
76     # Update the button text
77     self.toggle_button.config(text="Stop Thread")
78
79     # Update the flag value
80     self.is_running = True

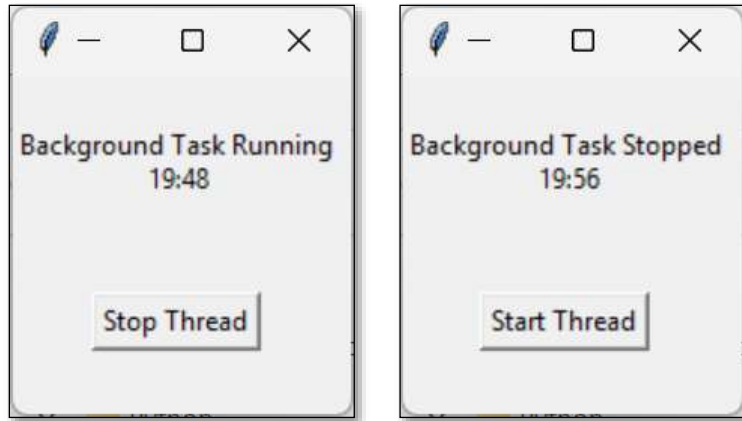
```

```

82  # ----- STOP THREAD ----- #
83  def stop_thread(self):
84      # Set the stop event to signal the thread to stop
85      # The thread will stop after the next iteration
86      self.stop_event.set()
87
88      # Update the button text and flag
89      self.toggle_button.config(text="Start Thread")
90
91      self.get_local_time()
92      self.lbl_display.config(text=f"Background Task Stopped\n {self.now}")
93
94      # Update the flag
95      self.is_running = False
96
97  # ----- GET LOCAL TIME ----- #
98  def get_local_time(self):
99      """Get the current local time and format it"""
100     now = localtime()
101     self.now = f"{now.tm_min:02d}:{now.tm_sec:02d}"
102
103  # ----- ON CLOSING ----- #
104  def on_closing(self):
105      # Stop the thread when closing the window
106      self.stop_thread()
107      # Destroy the window
108      self.root.destroy()
109
110
111  def main():
112      app = ThreadingApp()
113
114
115  if __name__ == "__main__":
116      main()

```

Example run:



Assignment Submission

1. Attach all program files.
2. Attach a screenshot of each successful program run.
3. Submit the assignment in Blackboard.