Python Threading: The Complete Guide

APRIL 9, 2022 *by* JASON BROWNLEE *in* THREADING (HTTPS://SUPERFASTPYTHON.COM/CATEGORY/THREADING/)

The **Python Threading** module allows you to create and manage new threads of execution in Python.

Although the threads have been available since Python 2, it is not widely used, perhaps because of misunderstandings of the capabilities and limitations of threads in Python.

This guide provides a detailed and comprehensive review of **threading in Python**, including how threads work, how to use threads in multithreaded programming, concurrency primitives used with threads, common questions, and best practices.

This is a massive 26,000+ word guide. You may want to bookmark it so you can refer to it as you develop your concurrent programs.

Let's dive in.

Table of Contents

Python Threads

So what are threads and why do we care?

What Are Threads

A thread refers to a thread of execution in a computer program.

Each program is a process and has at least one thread that executes instructions for that process.

> *Thread: The operating system object that executes the instructions of a process.*
>
> **— PAGE 273, THE ART OF CONCURRENCY (HTTPS://AMZN.TO/3Z1XHOZ), 2009.**

When we run a Python script, it starts an instance of the Python interpreter that runs our code in the main thread. The main thread is the default thread of a Python process.

We may develop our program to perform tasks concurrently, in which case we may need to create and run new threads. These will be concurrent threads of execution without our program, such as:

- Executing function calls concurrently.
- Executing object methods concurrently.

A Python thread is an object representation of a native thread provided by the underlying operating system.

When we create and run a new thread, Python will make system calls on the underlying operating system and request a new thread be created and to start running the new thread.

This highlights that Python threads are real threads, as opposed to simulated software threads, e.g. fibers or green threads.

The code in new threads may or may not be executed in parallel (at the same time), even though the threads are executed concurrently.

There are a number of reasons for this, such as:

- The underlying hardware may or may not support parallel execution (e.g. one vs multiple CPU cores).
- The Python interpreter may or may not permit multiple threads to execute in parallel.

This highlights the distinction between code that can run out of order (concurrent) from the capability to execute simultaneously (parallel).

- **Concurrent**: Code that can be executed out of order.
- **Parallel**: Capability to execute code simultaneously.

Next, let's consider the important differences between threads and processes.

Thread vs Process

A process refers to a computer program.

Each process is in fact one instance of the Python interpreter that executes Python instructions (Python byte-code), which is a slightly lower level than the code you type into your Python program.

> *"Process: The operating system's spawned and controlled entity that encapsulates an executing application. A process has two main functions. The first is to act as the resource holder for the application, and the second is to execute the instructions of the application.*
>
> **— PAGE 271, THE ART OF CONCURRENCY (HTTPS://AMZN.TO/3Z1XHOZ), 2009.**

The underlying operating system controls how new processes are created. On some systems, that may require spawning a new process, and on others, it may require that the process is forked. The operating-specific method used for creating new processes in Python is not something we need to worry about as it is managed by your installed Python interpreter.

A thread always exists within a process and represents the manner in which instructions or code is executed.

A process will have at least one thread, called the main thread. Any additional threads that we create within the process will belong to that process.

The Python process will terminate once all (non background threads) are terminated.

- **Process**: An instance of the Python interpreter has at least one thread called the MainThread.

- **Thread**: A thread of execution within a Python process, such as the MainThread or a new thread.

Next, let's take a look at threads in Python.

Life-Cycle of a Thread

A thread in Python is represented as an instance of the **threading.Thread** class.

Once a thread is started, the Python interpreter will interface with the underlying operating system and request that a new native thread be created. The **threading.Thread** instance then provides a Python-based reference to this underlying native thread.

Each thread follows the same life-cycle. Understanding the stages of this life-cycle can help when getting started with concurrent programming in Python.

For example:

- The difference between creating and starting a thread.
- The difference between run and start.
- The difference between blocked and terminated

And so on.

A Python thread may progress through three steps of its life-cycle: a new thread, a running thread, and a terminated thread.

While running, the thread may be executing code or may be blocked, waiting on something such as another thread or an external resource. Although, not all threads may block, it is optional based on the specific use case for the new thread.

1. New Thread.
2. Running Thread.
    1. Blocked Thread (optional).
3. Terminated Thread.

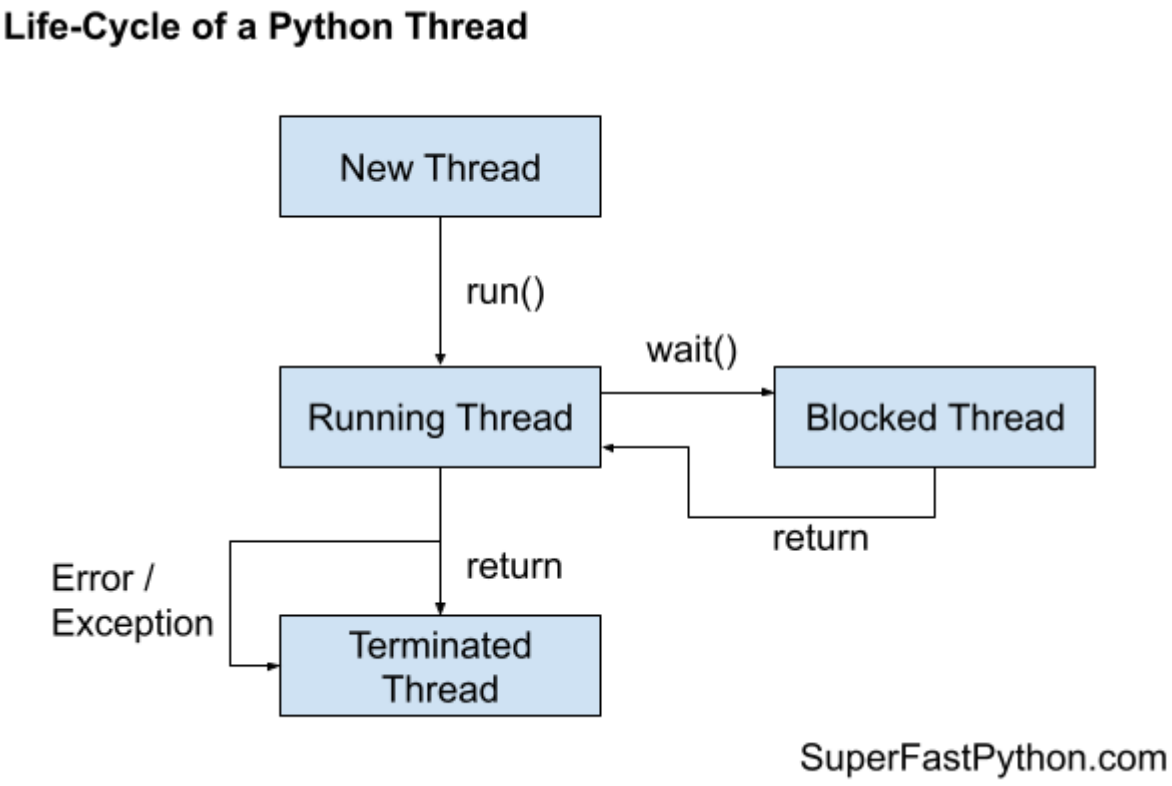
A new thread is a thread that has been constructed by creating an instance of the **threading.Thread** class.

A new thread can transition to a running thread by calling the **start()** function.

A running thread may block in many ways, such as reading or writing from a file or a socket or by waiting on a concurrency primitive such as a semaphore or a lock. After blocking, the thread will run again.

Finally, a thread may terminate once it has finished executing its code or by raising an error or exception.

The following figure summarizes the states of the thread life-cycle and how the thread may transition through these states.



THREAD-LIFE-CYCLE

You can learn more about the thread life-cycle in this tutorial:

- Thread Life-Cycle in Python (https://superfastpython.com/thread-life-cycle-in-python)

Next, let's look at how to run a function in a new thread.

Run your loops using all CPUs, download my FREE book (https://superfastpython.com/plip-incontent) to learn how.

Run a Function in a Thread

Python functions can be executed in a separate thread using the **threading.Thread** class.

In this section we will look at a few examples of how to run functions in another thread.

How to Run a Function In a Thread

To run a function in another thread:

1. Create an instance of the **threading.Thread** class.
2. Specify the name of the function via the "**target**" argument.
3. Call the **start()** function.

The function executed in another thread may have arguments in which case they can be specified as a tuple and passed to the "**args**" argument of the threading.Thread class constructor or as a dictionary to the "**kwargs**" argument.

The **start()** function will return immediately and the operating system will execute the function in a separate thread as soon as it is able.

We do not have control over when the thread will execute precisely or which CPU core will execute it. Both of these are low-level responsibilities that are handled by the underlying operating system.

Next, let's look at a worked example.

Example of Running a Function in a Thread

First, we can define a custom function that will be executed in another thread.

We will define a simple function that blocks for a moment then prints a statement.

The function can have any name we like, in this case we'll name it **task()**.

```
# a custom function that blocks for a moment
def task():
    # block for a moment
    sleep(1)
    # display a message
    print('This is from another thread')
```

Next, we can create an instance of the **threading.Thread** class and specify our function name as the "**target**" argument in the constructor.

```
...
# create a thread
thread = Thread(target=task)
```

Once created we can run the thread which will execute our custom function in a new native thread, as soon as the operating system can.

```
...
# run the thread
thread.start()
```

The **start()** function does not block, meaning it returns immediately.

We can explicitly wait for the new thread to finish executing by calling the **join()** function. This is not needed as the main thread will not exit until the new thread has completed.

```
...
# wait for the thread to finish
print('Waiting for the thread...')
thread.join()
```

You can learn more about joining a thread in this tutorial:

- How to Join a Thread in Python (https://superfastpython.com/join-a-thread-in-python/)

Tying this together, the complete example of executing a function in another thread is listed below.

```
```

```
# SuperFastPython.com
# example of running a function in another thread
from time import sleep
from threading import Thread

# a custom function that blocks for a moment
def task():
    # block for a moment
    sleep(1)
    # display a message
    print('This is from another thread')

# create a thread
thread = Thread(target=task)
# run the thread
thread.start()
# wait for the thread to finish
print('Waiting for the thread...')
thread.join()
```

Running the example first creates the **threading.Thread** then calls the **start()** function. This does not start the thread immediately, but instead allows the operating system to schedule the function to execute as soon as possible.

The main thread then prints a message waiting for the thread to complete, then calls the **join()** function to explicitly block and wait for the new thread to finish executing.

Once the custom function returns, the thread is closed. The **join()** function then returns and the main thread exists.

```
Waiting for the thread...
This is from another thread
```

Next, let's look at how we can run a function that takes arguments in a new thread.

Example of Running a Function in a Thread With Arguments

We can execute functions in another thread that take arguments.

This can be demonstrated by first updating our **task()** function from the previous section to take two arguments, one for the time in seconds to block and the second for a message to display.

```
# a custom function that blocks for a moment
def task(sleep_time, message):
    # block for a moment
    sleep(sleep_time)
    # display a message
    print(message)
```

Finally, we can update the call to the **threading.Thread** constructor to specify the two arguments in order that the function expects them as a tuple via the "**args**" argument.

```
...
# create a thread
thread = Thread(target=task, args=(1.5, 'New message from another thread'))
```

Tying this together, the complete example of executing a custom function that takes arguments in a separate thread is listed below.

```
# SuperFastPython.com
# example of running a function with arguments in another thread
from time import sleep
from threading import Thread

# a custom function that blocks for a moment
def task(sleep_time, message):
    # block for a moment
    sleep(sleep_time)
    # display a message
    print(message)

# create a thread
thread = Thread(target=task, args=(1.5, 'New message from another thread'))
# run the thread
thread.start()
# wait for the thread to finish
print('Waiting for the thread...')
thread.join()
```

Running the example creates the thread specifying the function name and the arguments to the function.

The thread is started and the function blocks for the parameterized number of seconds and prints the parameterized message.

```
Waiting for the thread...
New message from another thread
```

We now know how to execute functions in another thread via the "**target**" argument.

You can learn more about running functions in new threads in this tutorial:

- How to Run a Function in a New Thread in Python (https://superfastpython.com/run-function-in-new-thread/)

Next let's look at how we might run a function in another thread by extending the **threading.Thread** class.

Extend the Thread Class

We can also execute functions in another thread by extending the **threading.Thread** class and overriding the **run()** function.

In this section we will look at some examples of extending the **threading.Thread** class.

How to Extend the Thread Class

The **threading.Thread** class can be extended to run code in another thread.

This can be achieved by first extending the class, just like any other Python class.

For example:

```
# custom thread class
class CustomThread(Thread):
    # ...
```

Then the **run()** function of the **threading.Thread** class must be overridden to contain the code that you wish to execute in another thread.

For example:

```
# override the run function
def run(self):
    # ...
```

And that's it.

Given that it is a custom class, you can define a constructor for the class and use it to pass in data that may be needed in the run() function, stored as instance variables (attributes).

You can also define additional functions on the class to split up the work you may need to complete in another thread.

Finally, attributes can also be used to store the results of any calculation or IO performed in another thread that may need to be retrieved afterward.

Next, let's look at a worked example of extending the **threading.Thread** class.

Example of Extending the Thread Class

First, we can define a class that ends the **threading.Thread** class.

We will name the class something arbitrary such as "**CustomThread**"

```
# custom thread class
class CustomThread(Thread):
    # ...
```

We can then override the **run()** instance method and define the code that we wish to execute in another thread.

In this case, we will block for a moment and then print a message.

```
# override the run function
def run(self):
    # block for a moment
    sleep(1)
    # display a message
    print('This is coming from another thread')
```

Next, we can create an instance of our **CustomThread** class and call the **start()** function to begin executing our **run()** function in another thread. Internally, the **start()** function will call the **run()** function.

The code will then run in a new thread as soon as the operating system can schedule it.

```
...
# create the thread
thread = CustomThread()
# start the thread
thread.start()
```

Finally, we wait for the new thread to finish executing.

```
...
# wait for the thread to finish
print('Waiting for the thread to finish')
thread.join()
```

Tying this together, the complete example of executing code in another thread by extending the **threading.Thread** class is listed below.

```
# SuperFastPython.com
# example of extending the Thread class
from time import sleep
from threading import Thread

# custom thread class
class CustomThread(Thread):
    # override the run function
    def run(self):
        # block for a moment
        sleep(1)
        # display a message
        print('This is coming from another thread')

# create the thread
thread = CustomThread()
# start the thread
thread.start()
# wait for the thread to finish
print('Waiting for the thread to finish')
thread.join()
```

Running the example first creates an instance of the thread, then executes the content of the **run()** function.

Meanwhile, the main thread waits for the new thread to finish its execution, before exiting.

```
Waiting for the thread to finish
This is coming from another thread
```

You now know how to run code in a new thread by extending the **threading.Thread** class.

You can learn more about extending the **threading.Thread** class in this tutorial:

- How to Extend the Thread Class in Python (https://superfastpython.com/extend-thread-class/)

Next, let's look at how we might return values from a new thread.

Example of Extending the Thread Class and Returning Values

We may need to retrieve data from the thread, such as a return value.

There is no way for the **run()** function to return a value to the **start()** function and back to the caller.

Instead, we can return values from our **run()** function by storing them as instance variables and having the caller retrieve the data from those instance variables.

Let's demonstrate this with an example. We can update the previous example above to indirectly return a value from the extended **threading.Thread** class.

First, we can update the **run()** function to store some data as an instance variable (also called a Python attribute).

```
...
# store return value
self.value = 99
```

The updated version of the extended **threading.Thread** class with this change is listed below.

```
# custom thread class
class CustomThread(Thread):
    # override the run function
    def run(self):
        # block for a moment
        sleep(1)
        # display a message
        print('This is coming from another thread')
        # store return value
        self.value = 99
```

Next, we can retrieve the "**returned**" (stored) value from the run function in the main thread.

This can be achieved by accessing the attribute directly.

```
...
# get the value returned from run
value = thread.value
print(f'Got: {value}')
```

Tying this together, the complete example of returning a value from another thread indirectly in an extended **threading.Thread** class is listed below.

```python
# SuperFastPython.com
# example of extending the Thread class and return values
from time import sleep
from threading import Thread

# custom thread class
class CustomThread(Thread):
    # override the run function
    def run(self):
        # block for a moment
        sleep(1)
        # display a message
        print('This is coming from another thread')
        # store return value
        self.value = 99

# create the thread
thread = CustomThread()
# start the thread
thread.start()
# wait for the thread to finish
print('Waiting for the thread to finish')
thread.join()
# get the value returned from run
value = thread.value
print(f'Got: {value}')
```

Running the example creates and executes the new thread as per normal.

The **run()** function stores a return value as an instance variable, that is then accessed and reported by the main thread after the new thread has finished executing.

```
Waiting for the thread to finish
This is coming from another thread
Got: 99
```

You can learn more about returning values from a thread in this tutorial:

- How to Return Values From a Thread in Python (https://superfastpython.com/thread-return-values/)

Next, let's look at some properties of thread instances.

Thread Instance Attributes

An instance of the Thread class provides a handle of a thread of execution.

As such, it provides attributes that we can use to query properties and the status of the underlying thread.

Let's look at some examples.

Query Thread Name

Each thread has a name.

Threads are named automatically in a somewhat unique manner within each process with the form "**Thread-%d**" where **%d** is the integer indicating the thread number within the process, e.g. **Thread-1** for the first thread created.

We can access the name of a thread via the "**name**" attribute, for example:

```
...
# report the thread name
print(thread.name)
```

The example below creates an instance of the **threading.Thread** class and reports the default name of the thread.

```
# SuperFastPython.com
# example of accessing the thread name
from threading import Thread
# create the thread
thread = Thread()
# report the thread name
print(thread.name)
```

Running the example creates the thread and reports the default name assigned to the thread within the process.

```
Thread-1
```

Query Thread Daemon

A thread may be a daemon thread.

Daemon threads is the name given to background threads. By default, threads are non-daemon threads.

A Python program will only exit when all non-daemon threads have finished exiting. For example, the main thread is a non-daemon thread. This means that daemon threads can run in the background and do not have to finish or be explicitly excited for the program to end.

We can determine if a thread is a daemon thread via the "**daemon**" attribute.

```
...
# report the daemon attribute
print(thread.daemon)
```

The example creates an instance of the **threading.Thread** class and reports whether the thread is a daemon or not.

```
# SuperFastPython.com
# example of assessing whether a thread is a daemon
from threading import Thread
# create the thread
thread = Thread()
# report the daemon attribute
print(thread.daemon)
```

Running the example reports that the thread is not a daemon thread, the default for new threads.

```
False
```

Query Thread Identifier

Each thread has a unique identifier (id) within a Python process, assigned by the Python interpreter.

The identifier is a read-only positive integer value and is assigned only after the thread has been started.

We can access the identifier assigned to a **threading.Thread** instance via the "**ident**" property.

```
...
# report the thread identifier
print(thread.ident)
```

The example below creates a **threading.Thread** instance and reports the assigned identifier.

```
# SuperFastPython.com
# example of reporting the thread identifier
from threading import Thread
# create the thread
thread = Thread()
# report the thread identifier
print(thread.ident)
# start the thread
thread.start()
# report the thread identifier
print(thread.ident)
```

Running the example creates the thread instance then confirms that it does not have an identifier before it is started. The thread is started and its unique identifier is reported.

Note, your identifier will differ as the thread will have a different identifier each time the code is run.

```
None
123145502363648
```

Query Thread Native Identifier

Each thread has a unique identifier assigned by the operating system.

Python threads are real native threads, meaning that each thread we create is actually created and managed (scheduled) by the underlying operating system. As such, the operating system will assign a unique integer to each thread that is created on the system (across processes).

The native thread identifier can be accessed via the "**native_id**" property and is assigned after the thread has been started.

```
...
# report the native thread identifier
print(thread.native_id)
```

The example below creates an instance of a **threading.Thread** and reports the assigned native thread id.

```

```

```
# SuperFastPython.com
# example of reporting the native thread identifier
from threading import Thread
# create the thread
thread = Thread()
# report the native thread identifier
print(thread.native_id)
# start the thread
thread.start()
# report the native thread identifier
print(thread.native_id)
```

Running the example first creates the thread and confirms that it does not have a native thread identifier before it was started.

The thread is then started and the assigned native identifier is reported.

Note, your native thread identifier will differ as the thread will have a different identifier each time the code is run.

```
None
3061545
```

Query Thread Alive

A thread instance can be alive or dead.

An alive thread means that the **run()** method of the **threading.Thread** instance is currently executing.

This means that before the **start()** method is called and after the **run()** method has completed, the thread will not be alive.

We can check if a Thread is alive via the **is_alive()** method.

```
...
# report the thread is alive
print(thread.is_alive())
```

The example below creates a **threading.Thread** instance then checks whether it is alive.

```
# SuperFastPython.com
# example of assessing whether a thread is alive
from threading import Thread
# create the thread
thread = Thread()
# report the thread is alive
print(thread.is_alive())
```

Running the example creates a new Thread instance then reports that the thread is not alive.

False

We can update the example so that the thread sleeps for a moment and then report the alive status of the thread while it is running.

This can be achieved by setting the "**target**" argument to the **threading.Thread** to a lambda anonymous function that calls the **time.sleep()** function.

```
...
# create the thread
thread = Thread(target=lambda:sleep(1))
```

We can then start the thread by calling the **start()** function and report the alive status of the thread while it is running.

```
...
# start the thread
thread.start()
# report the thread is alive
print(thread.is_alive())
```

Finally, we can wait for the thread to finish and report the status of the thread afterward.

```
...
# wait for the thread to finish
thread.join()
# report the thread is alive
print(thread.is_alive())
```

Tying this together, the complete example of checking the alive status of a thread while it is running is listed below.

```
# SuperFastPython.com
# example of assessing whether a running thread is alive
from time import sleep
from threading import Thread
# create the thread
thread = Thread(target=lambda:sleep(1))
# report the thread is alive
print(thread.is_alive())
# start the thread
thread.start()
# report the thread is alive
print(thread.is_alive())
# wait for the thread to finish
thread.join()
# report the thread is alive
print(thread.is_alive())
```

Running the example creates the thread instance and confirms that it is not alive before the **start()** function has been called.

The thread is then started and blocked for a second, meanwhile we check the alive status in the main thread which is reported as **True**, the thread is alive.

The thread finishes and the alive status is reported again, showing that indeed the thread is no longer alive.

```
1 False
2 True
3 False
```

Now that we are familiar with some attributes of a **threading.Thread** instance, let's look at how we might configure a Thread.

Configure Threads

Instances of the **threading.Thread** class can be configured.

There are two properties of a thread that can be configured, they are the name of the thread and whether the thread is a daemon or not.

Let's take a closer look at each.

How to Configure Thread Name

Threads can be assigned custom names.

The name of a Thread can be set via the "**name**" argument in the **threading.Thread** constructor.

For example:

```
1 ...
2 # create a thread with a custom name
3 thread = Thread(name='MyThread')
```

The example below demonstrates how to set the name of a thread in the **threading.Thread** constructor.

```
# SuperFastPython.com
# example of setting the thread name in the constructor
from threading import Thread
# create a thread with a custom name
thread = Thread(name='MyThread')
# report thread name
print(thread.name)
```

Running the example creates the thread with the custom name then reports the name of the thread.

```
MyThread
```

The name of the thread can also be set via the "**name**" property.

For example:

```
...
# set the name
thread.name = 'MyThread'
```

The example below demonstrates this by creating an instance of a thread and then setting the name via the property.

```
# SuperFastPython.com
# example of setting the thread name via the property
from threading import Thread
# create a thread
thread = Thread()
# set the name
thread.name = 'MyThread'
# report thread name
print(thread.name)
```

Running the example creates the thread and sets the name then reports that the new name was assigned correctly.

```
MyThread
```

You can learn more about how to configure the thread name in this tutorial:

- How to Change the Thread Name in Python (https://superfastpython.com/thread-name/)

Next, let's take a closer look at daemon threads.

How to Configure Thread Daemon

A daemon thread (https://en.wikipedia.org/wiki/Daemon_(computing)) is a background thread.

Daemon is pronounced "*dee-mon*", like the alternate spelling "*demon*".

The ideas is that backgrounds are like "*daemons*" or spirits (from the ancient Greek) that do tasks for you in the background. You might also refer to daemon threads as daemonic.

A thread may be configured to be a daemon or not, and most threads in concurrent programming, including the main thread, are non-daemon threads (not background threads) by default.

A thread can be configured to be a daemon by setting the "**daemon**" argument to **True** in the **threading.Thread** constructor.

For example:

```
...
# create a daemon thread
thread = Thread(daemon=True)
```

The example below shows how to create a new daemon thread.

```
# SuperFastPython.com
# example of setting a thread to be a daemon via the constructor
from threading import Thread
# create a daemon thread
thread = Thread(daemon=True)
# report if the thread is a daemon
print(thread.daemon)
```

Running the example creates a new thread and configures it to be a daemon thread via the constructor.

```
True
```

We can also configure a thread to be a daemon thread after it has been constructed via the "**daemon**" property.

For example:

```
...
# configure the thread to be a daemon
thread.daemon = True
```

The example below creates a new **threading.Thread** instance then configures it to be a daemon thread via the property.

```
# SuperFastPython.com
# example of setting a thread to be a daemon via the property
from threading import Thread
# create a thread
thread = Thread()
# configure the thread to be a daemon
thread.daemon = True
# report if the thread is a daemon
print(thread.daemon)
```

Running the example creates a new thread instance then configures it to be a daemon thread, then reports the daemon status.

```
True
```

You can learn more about daemon threads in this tutorial:

- How to Use Daemon Threads in Python (https://superfastpython.com/daemon-threads-in-python/)

Now that we are familiar with how to configure new threads, let's take a closer look at the built-in main thread.

Main Thread

Each Python process is created with one default thread called the "*main thread*".

In this section we will take a closer look at the main thread.

When you execute a Python program, it is executing in the main thread.

The main thread is the default thread created for each Python process.

> *"In normal conditions, the main thread is the thread from which the Python interpreter was started.*
>
> **— THREADING — THREAD-BASED PARALLELISM (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/THREADING.HTML)**

The main thread in each Python process always has the name "**MainThread**" and is not a daemon thread.

Once the main thread exits, the Python process will exit, assuming there are no other non-daemon threads running.

> *"There is a "main thread" object; this corresponds to the initial thread of control in the Python program. It is not a daemon thread.*
>
> — **THREADING — THREAD-BASED PARALLELISM (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/THREADING.HTML)**

We can acquire a **threading.Thread** instance that represents the main thread by calling the **threading.current_thread()** function from within the main thread.

For example:

```
...
# get the main thread
thread = current_thread()
```

The example below demonstrates this and reports properties of the main thread.

```
# SuperFastPython.com
# example of getting the current thread for the main thread
from threading import current_thread
# get the main thread
thread = current_thread()
# report properties for the main thread
print(f'name={thread.name}, daemon={thread.daemon}, id={thread.ident}')
```

Running the example retrieves a **threading.Thread** instance for the current thread which is the main thread, then reports the details.

Note, your main thread will have a different identifier each time the program is run.

```
name=MainThread, daemon=False, id=4386115072
```

We can also get a **threading.Thread** instance for the main thread from any thread by calling the **threading.main_thread()** function.

For example:

```
...
# get the main thread
thread = main_thread()
```

The example below demonstrates this and reports the properties of the main thread

```
# SuperFastPython.com
# example of getting the main thread
from threading import main_thread
# get the main thread
thread = main_thread()
# report properties for the main thread
print(f'name={thread.name}, daemon={thread.daemon}, id={thread.ident}')
```

Running the example acquires the **threading.Thread** instance that represents the main thread and reports the thread properties.

Note, your main thread will have a different identifier each time the program is run.

```
name=MainThread, daemon=False, id=4644216320
```

You can learn more about the main thread in this tutorial:

- What is the Main Thread in Python (https://superfastpython.com/main-thread/)

Now that we are familiar with the main thread, let's look at some additional threading utilities.

Thread Utilities

There are a number of utilities we can use when working with threads within a Python process.

These utilities are provided as "**threading**" module functions.

We have already seen two of these functions in the previous section, specifically **threading.current_thread()** and **threading.main_thread()**.

In this section we will review a number of additional utility functions.

Number of Active Threads

We can discover the number of active threads in a Python process.

This can be achieved via the **threading.active_count()** function that returns an integer that indicates the number threads that are "*alive*".

```
...
# get the number of active threads
count = active_count()
```

We can demonstrate this with a short example.

```
# SuperFastPython.com
# report the number of active threads
from threading import active_count
# get the number of active threads
count = active_count()
# report the number of active threads
print(count)
```

Running the example reports the number of active threads in the process.

Given that there is only a single thread, the main thread, the count is one.

```
1
```

Current Thread

We can get a **threading.Thread** instance for the thread running the current code.

This can be achieved via the **threading.current_thread()** function that returns a **threading.Thread** instance.

```
...
# get the current thread
thread = current_thread()
```

We can demonstrate this with a short example.

```
# SuperFastPython.com
# retrieve the current thread within
from threading import Thread
from threading import current_thread

# function to get the current thread
def task():
    # get the current thread
    thread = current_thread()
    # report the name
    print(thread.name)

# create a thread
thread = Thread(target=task)
# start the thread
thread.start()
# wait for the thread to exit
thread.join()
```

Running the example first creates a new **threading.Thread** to run a custom function.

The custom function acquires a **threading.Thread** instance for the currently running thread and uses it to report the name of the currently running thread.

```
Thread-1
```

Thread Identifier

We can get the Python thread identifier for the current thread.

This can be achieved via the **threading.get_ident()** function.

```
...
# get the id for the current thread
identifier = get_ident()
```

Recall that each thread within a Python process has a unique identifier assigned by the interpreter.

We can demonstrate this with a short example.

```
# SuperFastPython.com
# report the id for the current thread
from threading import get_ident
# get the id for the current thread
identifier = get_ident()
# report the thread id
print(identifier)
```

Running the example gets and reports the identifier of the currently running thread.

Note, your identifier will differ as a new identifier is assigned to a thread each time the code is run.

```
4662242816
```

Native Thread Identifier

We can get the native thread identifier for the current thread assigned by the operating system.

This can be achieved via the **threading.get_native_id()** function.

```
...
# get the native id for the current thread
identifier = get_native_id()
```

Recall that each thread within a Python process has a unique native identifier assigned by the operating system.

We can demonstrate this with a short example.

```
# SuperFastPython.com
# report the native id for the current thread
from threading import get_native_id
# get the native id for the current thread
identifier = get_native_id()
# report the thread id
print(identifier)
```

Running the example gets and reports the native identifier of the currently running thread.

Note, your native identifier will differ as a new identifier is assigned to a thread each time the code is run.

```
374772
```

Enumerate Active Threads

We can get a list of all active threads within a Python process.

This can be achieved via the **threading.enumerate()** function.

Only those threads that are "*alive*" will be included in the list, meaning those threads that are currently executing their **run()** function.

The list will always include the main thread.

We can demonstrate this with a short example.

```
# SuperFastPython.com
# enumerate all active threads
import threading
# get a list of all active threads
threads = threading.enumerate()
# report the name of all active threads
for thread in threads:
    print(thread.name)
```

Running the example first retrieves a list of all active threads within the Python process.

The list of active threads is then iterated, reporting the name of each thread. In this case, the list only includes the main thread.

```
MainThread
```

Thread Exception Handling

We can manage unhandled exceptions that are raised within threads.

In this section, we will take a closer look at **threading.Thread** exception handling.

Unhandled Exception

An unhandled exception can occur in a new thread.

The effect will be that the thread will unwind and report the message on standard error.

Unwinding the thread means that the thread will stop executing at the point of the exception (or error) and that the exception will bubble up the stack in the thread until it reaches the top level, e.g. the **run()** function.

We can demonstrate this with an example.

Firstly, we can define a target function that will block for a moment then raise an exception.

```
# target function that raises an exception
def work():
    print('Working...')
    sleep(1)
    # rise an exception
    raise Exception('Something bad happened')
```

Next, we can create a thread that will call the target function and wait for the function to complete.

```
...
# create a thread
thread = Thread(target=work)
# run the thread
thread.start()
# wait for the thread to finish
thread.join()
# continue on
print('Continuing on...')
```

Tying this together, the complete example is listed below

```

```

```
# SuperFastPython.com
# example of an unhandled exception in a thread
from time import sleep
from threading import Thread

# target function that raises an exception
def work():
    print('Working...')
    sleep(1)
    # rise an exception
    raise Exception('Something bad happened')

# create a thread
thread = Thread(target=work)
# run the thread
thread.start()
# wait for the thread to finish
thread.join()
# continue on
print('Continuing on...')
```

Running the exception creates the thread and starts executing it.

The thread blocks for a moment and raises an exception.

The exception bubbles up to the **run()** function and is then reported on standard error (on the terminal).

Importantly, the failure in the thread does not impact the main thread, which continues on executing.

```
Working...
Exception in thread Thread-1:
Traceback (most recent call last):
  ...
  exception_unhandled.py", line 11, in work
    raise Exception('Something bad happened')
Exception: Something bad happened
Continuing on...
```

Exception Hook

We can specify how to handle unhandled errors and exceptions that occur within new threads via the exception hook.

By default, there is no exception hook, in which case the **sys.excepthook** function (https://docs.python.org/3/library/sys.html#sys.excepthook) is called that reports the familiar message.

We can specify a custom exception hook function that will be called whenever a **threading.Thread** fails with an unhandled Error or Exception.

This can be achieved via the **threading.excepthook** function (https://docs.python.org/3/library/threading.html#threading.excepthook).

First, we must define a function that takes a single argument that will be an instance of the **ExceptHookArgs** class, containing details of the exception and thread.

```
# custom exception hook
def custom_hook(args):
    # ...
```

We can then specify the exception hook function to be called whenever an unhandled exception bubbles up to the top level of a thread.

```
...
# set the exception hook
threading.excepthook = custom_hook
```

We can demonstrate this with an example by updating the unhandled exception in the previous section.

We can define a custom hook function that reports a single line to standard out.

```
# custom exception hook
def custom_hook(args):
    # report the failure
    print(f'Thread failed: {args.exc_value}')
```

The complete example is listed below.

```
# SuperFastPython.com
# example of an unhandled exception in a thread
from time import sleep
import threading

# target function that raises an exception
def work():
    print('Working...')
    sleep(1)
    # rise an exception
    raise Exception('Something bad happened')

# custom exception hook
def custom_hook(args):
    # report the failure
    print(f'Thread failed: {args.exc_value}')

# set the exception hook
threading.excepthook = custom_hook
# create a thread
thread = threading.Thread(target=work)
# run the thread
thread.start()
# wait for the thread to finish
thread.join()
# continue on
print('Continuing on...')
```

Running the example creates and runs the thread.

The new thread blocks for a moment, then fails with an exception.

The exception bubbles up to the top level of the thread at which point the custom exception hook function is called. Our custom message is then reported.

The main thread then continues to execute as per normal.

This can be helpful if we want to perform special actions when a thread fails unexpectedly, such as log to a file.

```
1 Working...
2 Thread failed: Something bad happened
3 Continuing on...
```

You can learn more about handling unexpected exceptions in threads in this tutorial:

- Handle Unexpected Exceptions in Threads with excepthook (https://superfastpython.com/thread-exception-handling/)

Next, let's look at the limitations of threads in Python.

Limitations of Threads in Python

The reference Python interpreter is referred to as CPython.

It is the free version of Python that you can download from python.org to develop and run Python programs.

The CPython Python interpreter generally does not permit more than one thread to run at a time.

This is achieved through a mutual exclusion (mutex) lock within the interpreter that ensures that only one thread at a time can execute Python bytecodes in the Python virtual machine.

> *In CPython, due to the Global Interpreter Lock, only one thread can execute Python code at once (even though certain performance-oriented libraries might overcome this limitation).*
>
> **— THREADING — THREAD-BASED PARALLELISM (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/THREADING.HTML)**

This lock is referred to as the Global Interpreter Lock (https://wiki.python.org/moin/GlobalInterpreterLock) or GIL for short.

> *In CPython, the global interpreter lock, or GIL, is a mutex that protects access to Python objects, preventing multiple threads from executing Python bytecodes at once. The GIL prevents race conditions and ensures thread safety.*
>
> **— GLOBAL INTERPRETER LOCK, PYTHON WIKI (HTTPS://WIKI.PYTHON.ORG/MOIN/GLOBALINTERPRETERLOCK).**

This means that although we might write concurrent code with threads in Python and run our code on hardware with many CPU cores, we may not be able to execute our code in parallel.

There are some exceptions to this.

Specifically, the GIL is released by the Python interpreter sometimes to allow other threads to run.

Such as when the thread is blocked, such as performing IO with a socket or file, or often if the thread is executing computationally intensive code in a C library, like hashing bytes.

> *Luckily, many potentially blocking or long-running operations, such as I/O, image processing, and NumPy number crunching, happen outside the GIL. Therefore it is only in multithreaded programs that spend a lot of time inside the GIL, interpreting CPython bytecode, that the GIL becomes a bottleneck.*
>
> **— GLOBAL INTERPRETER LOCK, PYTHON WIKI (HTTPS://WIKI.PYTHON.ORG/MOIN/GLOBALINTERPRETERLOCK).**

Therefore, although in most cases CPython will prevent parallel execution of threads, it is allowed in some circumstances. These circumstances represent the base use case for adopting threads in your Python programs.

Next, let's look at specific cases when you should consider using threads.

When to Use a Thread

The reference Python interpreter CPython prevents more than one thread from executing bytecode at the same time.

This is achieved using a mutex called the Global Interpreter Lock or GIL, as we learned in the previous section.

There are times when the lock is released by the interpreter and we can achieve parallel execution of our concurrent code in Python.

Examples of when the lock is released include:

- When a thread is performing blocking IO.
- When a thread is executing C code and explicitly releases the lock.

There are also ways of avoiding the lock entirely, such has:

- Using a third-party Python interpreter to execute Python code.

Let's take a look at each of these use cases in turn.

Use Threads for Blocking IO

You should use threads for IO-bound tasks.

An IO-bound task is a type of task that involves reading from or writing to a device, file, or socket connection.

The operations involve input and output (IO), and the speed of these operations is bound by the device, hard drive, or network connection. This is why these tasks are referred to as IO-bound.

CPUs are really fast. Modern CPUs, like a 4GHz CPU, can execute 4 billion instructions per second, and you likely have more than one CPU core in your system.

Doing IO is very slow compared to the speed of CPUs.

Interacting with devices, reading and writing files and socket connections involves calling instructions in your operating system (the kernel), which will wait for the operation to complete. If this operation is the main focus for your CPU, such as executing in the main thread of your Python program, then your CPU is going to wait many milliseconds or even many seconds doing nothing.

That is potentially billions of operations prevented from executing.

A thread performing an IO operation will block for the duration of the operation. While blocked, this signals to the operating system that a thread can be suspended and another thread can execute, called a context switch.

Additionally, the Python interpreter will release the GIL when performing blocking IO operations, allowing other threads within the Python process to execute.

Therefore, blocking IO provides an excellent use case for using threads in Python.

Examples of blocking IO operations include:

- Reading or writing a file from the hard drive.
- Reading or writing to standard output, input or error (stdin, stdout, stderr).

- Printing a document.
- Reading or writing bytes on a socket connection with a server.
- Downloading or uploading a file.
- Querying a server.
- Querying a database.
- Taking a photo or recording a video.
- And so much more.

Use Threads External C Code (that releases the GIL)

We may make function calls that themselves call down into a third-party C library.

Often these function calls will release the GIL as the C library being called will not interact with the Python interpreter.

This provides an opportunity for other threads in the Python process to run.

For example, when using the "**hash**" module in the Python standard library, the GIL is released when hashing the data via the **hash.update()** function (https://docs.python.org/3/library/hashlib.html#hashlib.hash.update).

> *"The Python GIL is released to allow other threads to run while hash updates on data larger than 2047 bytes is taking place when using hash algorithms supplied by OpenSSL.*
>
> **— HASHLIB — SECURE HASHES AND MESSAGE DIGESTS (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/HASHLIB.HTML)**

Another example is the NumPy library for managing arrays of data which will release the GIL when performing functions on arrays.

> *The exceptions are few but important: while a thread is waiting for IO (for you to type something, say, or for something to come in the network) python releases the GIL so other threads can run. And, more importantly for us, while numpy is doing an array operation, python also releases the GIL.*
>
> — **WRITE MULTITHREADED OR MULTIPROCESS CODE, SCIPY COOKBOOK (HTTPS://SCIPY-COOKBOOK.READTHEDOCS.IO/ITEMS/PARALLELPROGRAMMING.HTML).**

Use Threads With (Some) Third-Party Python Interpreters

Another important consideration is the use of third-party Python interpreters.

There are alternate commercial and open source Python interpreters that you can acquire and use to execute your Python code.

Some of these interpreters may implement a GIL and release it more or less than CPython. Other interpreters remove the GIL entirely and allow multiple Python concurrent threads to execute in parallel.

Examples of third-party Python interpreters without a GIL include:

- **Jython**: an open source Python interpreter written in Java.
- **IronPython**: an open source Python interpreter written in .NET.

> *... Jython does not have the straightjacket of the GIL. This is because all Python threads are mapped to Java threads and use standard Java garbage collection support (the main reason for the GIL in CPython is because of the reference counting GC system). The important ramification here is that you can use threads for compute-intensive tasks that are written in Python.*
>
> **— NO GLOBAL INTERPRETER LOCK, DEFINITIVE GUIDE TO JYTHON (HTTPS://JYTHON.READTHEDOCS.IO/EN/LATEST/CONCURRENCY/).**

Next, let's take a closer look at thread blocking function calls.

Thread Blocking Calls

A blocking call is a function call that does not return until it is complete.

All normal functions are blocking calls. No big deal.

In concurrent programming, blocking calls have special meaning.

Blocking calls are calls to functions that will wait for a specific condition and signal to the operating system that nothing interesting is going on while the thread is waiting.

The operating system may notice that a thread is making a blocking function call and decide to context switch to another thread.

You may recall that the operating system manages what threads should run and when to run them. It achieves this using a type of multitasking where a running thread is suspended and suspended thread is restored and continues running. This suspending and restoring of threads is called a context switch.

The operating system prefers to context switch away from blocked threads, allowing non-blocked threads to run.

This means if a thread makes a blocking function call, a call that waits, then it is likely to signal that the thread can be suspended and allow other threads to run.

Similarly, many function calls that we may traditionally think block may have non-blocking versions in modern non-blocking concurrency APIs, like asyncio.

There are three types of blocking function calls you need to consider in concurrent programming, they are:

- Blocking calls on concurrent primitives.
- Blocking calls for IO.
- Blocking calls to sleep.

Let's take a closer look at each in turn.

Blocking Calls on Concurrency Primitives

There are many examples of blocking function calls in concurrent programming.

Common examples include:

- Waiting for a lock, e.g. calling **acquire()** on a **threading.Lock**.
- Waiting to be notified, e.g. calling **wait()** on a **threading.Condition**.
- Waiting for a thread to terminate, e.g. calling **join()** on a **threading.Thread**.
- Waiting for a semaphore, e.g. calling **acquire()** on a **threading.Semaphore**.
- Waiting for an event, e.g. calling **wait()** on a **threading.Event**.
- Waiting for a barrier, e.g. calling **wait()** on a **threading.Barrier**.

Blocking on concurrency primitives is a normal part of concurrency programming.

Blocking Calls for I/O

Conventionally, function calls that interact with IO are often blocking function calls.

That is, they are blocking in the same sense as blocking calls in the concurrency primitives.

The wait for the IO device to respond is another signal to the operating system that the thread can be context switched.

Common examples include:

- **Hard disk drive**: Reading, writing, appending, renaming, deleting, etc. files.
- **Peripherals**: mouse, keyboard, screen, printer, serial, camera, etc.
- **Internet**: Downloading and uploading files, getting a webpage, querying RSS, etc.
- **Database**: Select, update, delete, etc. SQL queries.
- **Email**: Send mail, receive mail, query inbox, etc.

And many more examples, mostly related to sockets.

Performing IO with devices is typically very slow compared to the CPU.

The CPU can perform orders of magnitude more instructions compared to reading or writing some bytes to a file or socket.

The IO with devices is coordinated by the operating system and the device. This means the operating system can gather or send some bytes from or to the device then context switch back to the blocking thread when needed, allowing the function call to progress.

As such, you will often see comments about blocking calls when working with IO.

Blocking Calls to Sleep

The **sleep()** function is a capability provided by the underlying operating system that we can make use of within our program.

It is a blocking function call that pauses the thread to block for a fixed time in seconds.

This can be achieved via a call to **time.sleep()**.

For example:

```
...
# sleep for 5 seconds
time.sleep(5)
```

It is a blocking call in the same sense as concurrency primitives and blocking IO function calls. It signals to the operating system that the thread is waiting and is a good candidate for a context switch.

Sleeps are often used when timing is important in an application.

In concurrent programming, adding a sleep can be a useful way to simulate computational effort by a thread for a fixed interval.

We often use sleeps in worked examples when demonstrating concurrency programming, but adding sleeps to code can also aid in unit testing and debugging concurrency failure conditions, such as race conditions by forcing mistiming of events within a dynamic application.

You can learn more about blocking calls in the tutorial:

- Thread Blocking Call in Python (https://superfastpython.com//thread-blocking-call-in-python)

Next, let's take a look at thread-local data storage.

Thread-Local Data

Threads can store local data via an instance of the **threading.local** class.

First, an instance of the local class must be created.

```
...
# create a local instance
local = threading.local()
```

Then data can be stored on the local instance with properties of any arbitrary name.

For example:

```
...
# store some data
local.custom = 33
```

Importantly, other threads can use the same property names on the local but the values will be limited to each thread.

This is like a namespace limited to each thread and is called “thread-local data”. It means that threads cannot access or read the local data of other threads.

Importantly, each thread must hang on to the “**local**” instance in order to access the stored data.

We can demonstrate this with an example.

```
# SuperFastPython.com
# example of thread local storage
from time import sleep
import threading

# custom target function
def task(value):
    # create a local storage
    local = threading.local()
    # store data
    local.value = value
    # block for a moment
    sleep(value)
    # retrieve value
    print(f'Stored value: {local.value}')

# create and start a thread
threading.Thread(target=task, args=(1,)).start()
# create and start another thread
threading.Thread(target=task, args=(2,)).start()
```

Running the example first creates one thread that stores the value “1” against the property “value”, then blocks for one second.

The second thread runs and stores the value “2” against the property “value”, then blocks for two seconds.

Each thread then reports their “value” from thread-local storage, matching the value that was stored within each thread-local context.

If the threads used the same variable for storage, then the both threads would report “2” for the “value” property, which is not the case because the “value” property is unique to each thread.

```
Stored value: 1
Stored value: 2
```

You can learn more about thread local storage in this tutorial:

- How to Use Thread-Local Data in Python (https://superfastpython.com/thread-local-data/)

Now that we are familiar with thread-local storage, let's look at a mutual exclusion lock.

Thread Mutex Lock

A mutex lock is used to protect critical sections of code from concurrent execution.

You can use a mutual exclusion (mutex) lock in Python via the **threading.Lock** class.

What is a Mutual Exclusion Lock

A mutual exclusion lock (https://en.wikipedia.org/wiki/Mutual_exclusion) or mutex lock is a synchronization primitive intended to prevent a race condition.

A race condition is a concurrency failure case when two threads run the same code and access or update the same resource (e.g. data variables, stream, etc.) leaving the resource in an unknown and inconsistent state.

Race conditions often result in unexpected behavior of a program and/or corrupt data.

These sensitive parts of code that can be executed by multiple threads concurrently and may result in race conditions are called critical sections. A critical section may refer to a single block of code, but it also refers to multiple accesses to the same data variable or resource from multiple functions.

A mutex lock can be used to ensure that only one thread at a time executes a critical section of code at a time, while all other threads trying to execute the same code must wait until the currently executing thread is finished with the critical section and releases the lock.

Each thread must attempt to acquire the lock at the beginning of the critical section. If the lock has not been obtained, then a thread will acquire it and other threads must wait until the thread that acquired the lock releases it.

If the lock has not been acquired, we might refer to it as being in the "*unlocked*" state. Whereas if the lock has been acquired, we might refer to it as being in the "*locked*" state.

- **Unlocked**: The lock has not been acquired and can be acquired by the next thread that makes an attempt.
- **Locked**: The lock has been acquired by one thread and any thread that makes an attempt to acquire it must wait until it is released.

Locks are created in the unlocked state.

Now that we know what a mutex lock is, let's take a look at how we can use it in Python.

How to Use a Mutex Lock

Python provides a mutual exclusion lock via the **threading.Lock** class (https://docs.python.org/3/library/threading.html#lock-objects).

An instance of the lock can be created and then acquired by threads before accessing a critical section, and released after the critical section.

For example:

```
...
# create a lock
lock = Lock()
# acquire the lock
lock.acquire()
# ...
# release the lock
lock.release()
```

Only one thread can have the lock at any time. If a thread does not release an acquired lock, it cannot be acquired again.

The thread attempting to acquire the lock will block until the lock is acquired, such as if another thread currently holds the lock then releases it.

We can attempt to acquire the lock without blocking by setting the "**blocking**" argument to **False**. If the lock cannot be acquired, a value of **False** is returned.

```
...
# acquire the lock without blocking
lock.acquire(blocking=false)
```

We can also attempt to acquire the lock with a timeout, that will wait the set number of seconds to acquire the lock before giving up. If the lock cannot be acquired, a value of **False** is returned.

```
...
# acquire the lock with a timeout
lock.acquire(timeout=10)
```

We can also use the lock via the context manager protocol via the with statement, allowing the critical section to be a block within the usage of the lock.

For example:

```
...
# create a lock
lock = Lock()
# acquire the lock
with lock:
    # ...
```

This is the preferred usage as it makes it clear where the protected code begins and ends, and ensures that the lock is always released, even if there is an exception or error within the critical section.

We can also check if the lock is currently acquired by a thread via the **locked()** function.

```
...
# check if a lock is currently acquired
if lock.locked():
    # ...
```

Next, let's look at a worked example of using the threading.Lock class.

Example of Using a Mutex Lock

We can develop an example to demonstrate how to use the mutex lock.

First, we can define a target task function that takes a lock as an argument and uses the lock to protect a critical section. In this case, the critical section involves reporting a message and blocking for a fraction of a section.

```
# work function
def task(lock, identifier, value):
    # acquire the lock
    with lock:
        print(f'>thread {identifier} got the lock, sleeping for {value}')
        sleep(value)
```

We can then create one instance of the **threading.Lock** shared among the threads, and pass it to each thread that we intend to execute the target task function.

```
...
# create a shared lock
lock = Lock()
# start a few threads that attempt to execute the same critical section
for i in range(10):
    # start a thread
    Thread(target=task, args=(lock, i, random())).start()
```

Tying this together, the complete example of using a lock is listed below.

```
# SuperFastPython.com
# example of a mutual exclusion (mutex) lock
from time import sleep
from random import random
from threading import Thread
from threading import Lock

# work function
def task(lock, identifier, value):
    # acquire the lock
    with lock:
        print(f'>thread {identifier} got the lock, sleeping for {value}')
        sleep(value)

# create a shared lock
lock = Lock()
# start a few threads that attempt to execute the same critical section
for i in range(10):
    # start a thread
    Thread(target=task, args=(lock, i, random())).start()
# wait for all threads to finish...
```

Running the example starts ten threads that all execute a custom target function.

Each thread attempts to acquire the lock, and once they do, they report a message including their id and how long they will sleep before releasing the lock.

Your specific results may vary given the use of random numbers.

```
>thread 0 got the lock, sleeping for 0.8859193801237439
>thread 1 got the lock, sleeping for 0.02868415293867832
>thread 2 got the lock, sleeping for 0.04469783674319383
>thread 3 got the lock, sleeping for 0.20456291750962474
>thread 4 got the lock, sleeping for 0.3689208984892195
>thread 5 got the lock, sleeping for 0.21105944750222927
>thread 6 got the lock, sleeping for 0.052093068060339864
>thread 7 got the lock, sleeping for 0.871251970586552
>thread 8 got the lock, sleeping for 0.932718580790764
>thread 9 got the lock, sleeping for 0.9514093969897454
```

You can learn more about thread mutex locks in this tutorial:

- How to Use a Mutex Lock in Python (https://superfastpython.com/thread-mutex-lock/)

Now that we are familiar with the lock, let's take a look at the reentrant lock.

Thread Reentrant Lock

A reentrant lock is a lock that can be acquired more than once by the same thread.

You can use reentrant locks in Python via the **threading.RLock** class.

What is a Reentrant Lock

A reentrant mutual exclusion lock, "*reentrant mutex*" or "*reentrant lock*" for short, is like a mutex lock except it allows a thread to acquire the lock more than once.

> *"A reentrant lock is a synchronization primitive that may be acquired multiple times by the same thread. [...] In the locked state, some thread owns the lock; in the unlocked state, no thread owns it.*
>
> **— RLOCK OBJECTS, THREADING – THREAD-BASED PARALLELISM (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/THREADING.HTML#RLOCK-OBJECTS).**

A thread may need to acquire the same lock more than once for many reasons.

We can imagine critical sections spread across a number of functions, each protected by the same lock. A thread may call across these functions in the course of normal execution and may call into one critical section from another critical section.

A limitation of a (non-reentrant) mutex lock is that if a thread has acquired the lock that it cannot acquire it again. In fact, this situation will result in a deadlock as it will wait forever for the lock to be released so that it can be acquired, but it holds the lock and will not release it.

A reentrant lock will allow a thread to acquire the same lock again if it has already acquired it. This allows the thread to execute critical sections from within critical sections, as long as they are protected by the same reentrant lock.

Each time a thread acquires the lock it must also release it, meaning that there are recursive levels of acquire and release for the owning thread. As such, this type of lock is sometimes called a "*recursive mutex lock*".

Now that we are familiar with the reentrant lock, let's take a closer look at the difference between a lock and a reentrant lock in Python.

How to Use the Reentrant Lock

Python provides a lock via the **threading.RLock** class (https://docs.python.org/3/library/threading.html#rlock-objects).

An instance of the RLock can be created and then acquired by threads before accessing a critical section, and released after the critical section.

For example:

```
...
# create a reentrant lock
lock = RLock()
# acquire the lock
lock.acquire()
# ...
# release the lock
lock.release()
```

The thread attempting to acquire the lock will block until the lock is acquired, such as if another thread currently holds the lock (once or more than once) then releases it.

We can attempt to acquire the lock without blocking by setting the "**blocking**" argument to **False**. If the lock cannot be acquired, a value of **False** is returned.

```
...
# acquire the lock without blocking
lock.acquire(blocking=false)
```

We can also attempt to acquire the lock with a timeout, that will wait the set number of seconds to acquire the lock before giving up. If the lock cannot be acquired, a value of **False** is returned.

```
...
# acquire the lock with a timeout
lock.acquire(timeout=10)
```

We can also use the reentrant lock via the context manager protocol via the with statement, allowing the critical section to be a block within the usage of the lock.

For example:

```
...
# create a reentrant lock
lock = RLock()
# acquire the lock
with lock:
    # ...
```

Next, let's look at a worked example of using the **threading.RLock** class.

Example of Using a Reentrant Lock

We can develop an example to demonstrate how to use the reentrant lock.

First, we can define a function to report that a thread is done that protects the reporting process with a lock.

```
# reporting function
def report(lock, identifier):
    # acquire the lock
    with lock:
        print(f'>thread {identifier} done')
```

Next, we can define a task function that reports a message, blocks for a moment, then calls the reporting function. All of the work is protected with the lock.

```
# work function
def task(lock, identifier, value):
    # acquire the lock
    with lock:
        print(f'>thread {identifier} sleeping for {value}')
        sleep(value)
        # report
        report(lock, identifier)
```

Given that the target task function is protected with a lock and calls the reporting function that is also protected by the same lock, we can use a reentrant lock so that if a thread acquires the lock in task(), it will be able to re-enter the lock in the **report()** function.

Finally, we can create the reentrant lock, then startup threads that execute the target task function.

```
```

```
...
# create a shared reentrant lock
lock = RLock()
# start a few threads that attempt to execute the same critical section
for i in range(10):
    # start a thread
    Thread(target=task, args=(lock, i, random())).start()
# wait for all threads to finish...
```

Tying this together, the complete example of demonstrating a reentrant lock is listed below.

```
# SuperFastPython.com
# example of a reentrant lock
from time import sleep
from random import random
from threading import Thread
from threading import RLock

# reporting function
def report(lock, identifier):
    # acquire the lock
    with lock:
        print(f'>thread {identifier} done')

# work function
def task(lock, identifier, value):
    # acquire the lock
    with lock:
        print(f'>thread {identifier} sleeping for {value}')
        sleep(value)
        # report
        report(lock, identifier)

# create a shared reentrant lock
lock = RLock()
# start a few threads that attempt to execute the same critical section
for i in range(10):
    # start a thread
    Thread(target=task, args=(lock, i, random())).start()
# wait for all threads to finish...
```

Running the example starts up ten threads that execute the target task function.

Only one thread can acquire the lock at a time, and then once acquired, blocks and then reenters the same lock again to report the done message.

If a non-reentrant lock, e.g. a **threading.Lock** was used instead, then the thread would block forever waiting for the lock to become available, which it can't because the thread already holds the lock.

```
>thread 0 sleeping for 0.5784837315288808
>thread 0 done
>thread 1 sleeping for 0.19407032646041522
>thread 1 done
>thread 2 sleeping for 0.03612750793398978
>thread 2 done
>thread 3 sleeping for 0.17964358883204423
>thread 3 done
>thread 4 sleeping for 0.2800897627049981
>thread 4 done
>thread 5 sleeping for 0.9314520504231987
>thread 5 done
>thread 6 sleeping for 0.04446466830667195
>thread 6 done
>thread 7 sleeping for 0.37852383245813737
>thread 7 done
>thread 8 sleeping for 0.3529410350492209
>thread 8 done
>thread 9 sleeping for 0.7780184882739216
>thread 9 done
```

You can learn more about reentrant locks in this tutorial:

- How to Use a Reentrant Lock in Python (https://superfastpython.com/thread-reentrant-lock/)

Now that we are familiar with the reentrant lock, let's look at the condition.

Thread Condition

A condition allows threads to wait and be notified.

You can use a thread condition object in Python via the **threading.Condition** class.

What is a Threading Condition

In concurrency, a condition (also called a monitor (https://en.wikipedia.org/wiki/Monitor_(synchronization))) allows multiple threads to be notified about some result.

It combines both a mutual exclusion lock (mutex) and a conditional variable.

A mutex allow can be used to protect a critical section, but it cannot be used to alert other threads that a condition has changed or been met.

A condition can be acquired by a thread (like a mutex) after which it can wait to be notified by another thread that something has changed. While waiting, the thread is blocked and releases the lock for other threads to acquire.

Another thread can then acquire the condition, make a change, and notify one, all, or a subset of threads waiting on the condition that something has changed. The waiting thread can then wake-up (be scheduled by the operating system), re-acquire the condition (mutex), perform checks on any changed state and perform required actions.

This highlights that a condition makes use of a mutex internally (to acquire/release the condition), but it also offers additional features such as allowing threads to wait on the condition and to allow threads to notify other threads waiting on the condition.

Now that we know what a condition is, let's look at how we might use it in Python.

How to Use a Condition Object

Python provides a condition via the **threading.Condition** class (https://docs.python.org/3/library/threading.html#condition-objects).

We can create a condition object and by default it will create a new reentrant mutex lock (**threading.RLock** class) by default which will be used internally.

```
1 ...
2 # create a new condition
3 condition = threading.Condition()
```

We may have a reentrant mutex or a non-reentrant mutex that we wish to reuse in the condition for some good reason, in which case we can provide it to the constructor.

I don't recommend this unless you know your use case has this requirement. The chance of getting into trouble is high.

```
1 ...
2 # create a new condition with a custom lock
3 condition = threading.Condition(lock=my_lock)
```

In order for a thread to make use of the condition, it must acquire it and release it, like a mutex lock.

This can be achieved manually with the **acquire()** and **release()** functions.

For example, we can acquire the condition and then wait on the condition to be notified and finally release the condition as follows:

```
...
# acquire the condition
condition.acquire()
# wait to be notified
condition.wait()
# release the condition
condition.release()
```

An alternative to calling the **acquire()** and **release()** functions directly is to use the context manager, which will perform the acquire/release automatically for us, for example:

```
...
# acquire the condition
with condition:
    # wait to be notified
    condition.wait()
```

We also must acquire the condition in a thread if we wish to notify waiting threads. This too can be achieved directly with the acquire/release function calls or via the context manager.

We can notify a single waiting thread via the **notify()** function.

For example:

```
...
# acquire the condition
with condition:
    # notify a waiting thread
    condition.notify()
```

The notified thread will stop-blocking as soon as it can re-acquire the mutex within the condition. This will be attempted automatically as part of its call to **wait()**, you do not need to do anything extra.

If there are more than one thread waiting on the condition, we will not know which thread will be notified.

We can notify all threads waiting on the condition via the **notify_all()** function.

```
...
# acquire the condition
with condition:
    # notify all threads waiting on the condition
    condition.notify_all()
```

Now that we know how to use the **threading.Condition** class, let's look at some worked examples.

Example of Wait and Notify With a Condition

We will explore using a **threading.Condition** to notify a waiting thread that something has happened.

We will use a new **threading.Thread** instance to prepare some data and notify a waiting thread, and in the main thread we will kick-off the new thread and use the condition to wait for the work to be completed.

First, we will define a target task function to execute in a new thread.

The function will take the condition object and a list in which it can deposit data. The function will block for a moment, add data to the list, then notify the waiting thread.

The complete target task function is listed below.

```
# target function to prepare some work
def task(condition, work_list):
    # block for a moment
    sleep(1)
    # add data to the work list
    work_list.append(33)
    # notify a waiting thread that the work is done
    print('Thread sending notification...')
    with condition:
        condition.notify()
```

In the main thread, first we can create the condition and a list in which we can place data.

```
...
# create a condition
condition = Condition()
# prepare the work list
work_list = list()
```

Next, we can start a new thread calling our target task function and wait to be notified of the result.

```

```

```
...
# wait to be notified that the data is ready
print('Main thread waiting for data...')
with condition:
    # start a new thread to perform some work
    worker = Thread(target=task, args=(condition, work_list))
    worker.start()
    # wait to be notified
    condition.wait()
```

Note, we must start the new thread after we have acquired the mutex lock in the condition in this example.

If we did not acquire the lock first, it is possible that there would be a race condition. Specifically, if we started the new thread before acquiring the condition and waiting in the main thread, then it is possible for the new thread to execute and notify before the main thread has had a chance to start waiting. In which case the main thread would wait forever to be notified.

Finally, we can report the data once it is available.

```
...
# we know the data is ready
print(f'Got data: {work_list}')
```

Tying this together, the complete example is listed below.

```
# SuperFastPython.com
# example of wait/notify with a condition
from time import sleep
from threading import Thread
from threading import Condition

# target function to prepare some work
def task(condition, work_list):
    # block for a moment
    sleep(1)
    # add data to the work list
    work_list.append(33)
    # notify a waiting thread that the work is done
    print('Thread sending notification...')
    with condition:
        condition.notify()

# create a condition
condition = Condition()
# prepare the work list
work_list = list()
# wait to be notified that the data is ready
print('Main thread waiting for data...')
with condition:
    # start a new thread to perform some work
    worker = Thread(target=task, args=(condition, work_list))
    worker.start()
    # wait to be notified
    condition.wait()
# we know the data is ready
print(f'Got data: {work_list}')
```

Running the example first creates the condition and the work list.

The new thread is defined and started. The thread blocks for a moment, adds data to the list then notifies the waiting thread.

Meanwhile the main thread waits to be notified by the new threads, then once notified it knows the data is ready and reports the results.

```
1 Main thread waiting for data...
2 Thread sending notification...
3 Got data: [33]
```

You can learn more about conditions in the tutorial:

- How to Use A Thread Condition Object in Python (https://superfastpython.com/thread-condition/)

Next, let's look at how to use a semaphore.

Thread Semaphore

A semaphore is essentially a counter protected by a mutex lock, used to limit the number of threads that can access a resource.

You can use a semaphore in Python by **threading.Semaphore** class.

What is a Semaphore

A semaphore is a concurrency primitive (https://en.wikipedia.org/wiki/Semaphore_(programming)) that allows a limit on the number of threads that can acquire a lock protecting a critical section.

It is an extension of a mutual exclusion (mutex) lock that adds a count for the number of threads that can acquire the lock before additional threads will block. Once full, new threads can only acquire a position on the semaphore once an existing thread holding the semaphore releases a position.

Internally, the semaphore maintains a counter protected by a mutex lock that is incremented each time the semaphore is acquired and decremented each time it is released.

When a semaphore is created, the upper limit on the counter is set. If it is set to be 1, then the semaphore will operate like a mutex lock.

A semaphore provides a useful concurrency tool for limiting the number of threads that can access a resource concurrently. Some examples include:

- Limiting concurrent socket connections to a server.
- Limiting concurrent file operations on a hard drive.
- Limiting concurrent calculations.

Now that we know what a semaphore is, let's look at how we might use it in Python.

How to Use a Semaphore

Python provides a semaphore via the **threading.Semaphore** class (https://docs.python.org/3/library/threading.html#semaphore-objects).

The **threading.Semaphore** instance must be configured when it is created to set the limit on the internal counter. This limit will match the number of concurrent threads that can hold the semaphore.

For example, we might want to set it to 100:

```
...
# create a semaphore with a limit of 100
semaphore = Semaphore(100)
```

In this implementation, each time the semaphore is acquired, the internal counter is decremented. Each time the semaphore is released, the internal counter is incremented. The semaphore cannot be acquired if the semaphore has no available positions in which case, threads attempting to acquire it must block until a position becomes available.

The semaphore can be acquired by calling the **acquire()** function, for example:

```
...
# acquire the semaphore
semaphore.acquire()
```

By default, it is a blocking call, which means that the calling thread will block until a position becomes available on the semaphore.

The "**blocking**" argument can be set to **False** in which case, if a position is not available on the semaphore, the thread will not block and the function will return immediately, returning a **False** value indicating that the semaphore was not acquired or a **True** value if a position could be acquired.

```
...
# acquire the semaphore without blocking
semaphore.acquire(blocking=False)
```

The "**timeout**" argument can be set to a number of seconds that the calling thread is willing to wait for a position on the semaphore if one is not available, before giving up. Again, the **acquire()** function will return a value of **True** if a position could be acquired or False otherwise.

```
...
# acquire the semaphore with a timeout
semaphore.acquire(timeout=10)
```

Once acquired, the semaphore can be released again by calling the **release()** function.

```
...
# release the semaphore
semaphore.release()
```

Finally, the **threading.Semaphore** class supports usage via the context manager, which will automatically acquire and release the semaphore for you. As such it is the preferred usage, if appropriate for your program.

For example:

```
...
# acquire the semaphore
with semaphore:
    # ...
```

It is possible for the **release()** method to be called more times than the **acquire()** method, and this will not cause a problem. It is a way of increasing the capacity of the semaphore.

Now that we know how to use the **threading.Semaphore** in Python, let's look at a worked example.

Example of Using a Semaphore

We can explore how to use **threading.Semaphore** with a worked example.

We will develop an example with a suite of threads but a limit on the number of threads that can perform an action simultaneously. A semaphore will be used to limit the number of concurrent threads which will be less than the total number of threads, allowing some threads to block, wait for a position, then be notified and acquire a position.

First, we can define a target task function that takes the shared semaphore and a unique integer as arguments. The function will attempt to acquire the semaphore, and once a position is acquired it will simulate some processing by generating a random number and blocking for a moment, then report its data and release the semaphore.

The complete **task()** function is listed below.

```
# target function
def task(semaphore, number):
    # attempt to acquire the semaphore
    with semaphore:
        # process
        value = random()
        sleep(value)
        # report result
        print(f'Thread {number} got {value}')
```

The main thread will first create the **threading.Semaphore** instance and limit the number of concurrent processing threads to 2.

```
...
# create a semaphore
semaphore = Semaphore(2)
```

Next, we will create and start 10 threads and pass each the shared semaphore instance and a unique integer to identify the thread.

```
...
# create a suite of threads
for i in range(10):
    worker = Thread(target=task, args=(semaphore, i))
    worker.start()
```

The main thread will then wait for all worker threads to complete before exiting.

Tying this together, the complete example of using a semaphore is listed below.

```
```

```
# SuperFastPython.com
# example of using a semaphore
from time import sleep
from random import random
from threading import Thread
from threading import Semaphore

# target function
def task(semaphore, number):
    # attempt to acquire the semaphore
    with semaphore:
        # process
        value = random()
        sleep(value)
        # report result
        print(f'Thread {number} got {value}')

# create a semaphore
semaphore = Semaphore(2)
# create a suite of threads
for i in range(10):
    worker = Thread(target=task, args=(semaphore, i))
    worker.start()
# wait for all workers to complete...
```

Running the example first creates the semaphore instance then starts ten worker threads.

All ten threads attempt to acquire the semaphore, but only two threads are granted positions at a time. The threads on the semaphore do their work and release the semaphore when they are done, at random intervals.

Each release of the semaphore (via the context manager) allows another thread to acquire a position and perform its calculation, all the while allowing only two of the threads to be processed at any one time, even though all ten threads are executing their run methods.

Note, your specific values will differ given the use of random numbers.

```
Thread 1 got 0.4468840323081351
Thread 0 got 0.7288038062917327
Thread 2 got 0.4497887327563145
Thread 4 got 0.019601471581036645
Thread 3 got 0.5114751539092154
Thread 6 got 0.6191428550062478
Thread 5 got 0.9893921843198458
Thread 8 got 0.022640379341017924
Thread 7 got 0.20649643092073067
Thread 9 got 0.18636311846540998
```

You can learn more about semaphores in this tutorial:

- How to Use a Semaphore in Python (https://superfastpython.com/thread-semaphore/)

Next, let's look at event objects.

Thread Event

An event is a thread-safe boolean flag.

You can use an Event Object in Python via the **threading.Event** class.

How to Use an Event Object

Python provides an event object via the **threading.Event** class (https://docs.python.org/3/library/threading.html#event-objects).

An event is a simple concurrency primitive that allows communication between threads.

A **threading.Event** object wraps a boolean variable that can either be "*set*" (**True**) or "*not set*" (**False**). Threads sharing the event instance can check if the event is set, set the event, clear the event (make it not set), or wait for the event to be set.

The **threading.Event** provides an easy way to share a boolean variable between threads that can act as a trigger for an action.

> *"This is one of the simplest mechanisms for communication between threads: one thread signals an event and other threads wait for it.*
>
> **— EVENT OBJECTS, THREADING — THREAD-BASED PARALLELISM (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/THREADING.HTML#EVENT-OBJECTS)**

First, an event object must be created and the event will be in the "*not set*" state.

```
...
# create an instance of an event
event = threading.Event()
```

Once created we can check if the event has been set via the **is_set()** function which will return **True** if the event is set, or **False** otherwise.

For example:

```
...
# check if the event is set
if event.is_set():
    # do something...
```

The event can be set via the **set()** function. Any threads waiting on the event to be set will be notified.

For example:

```
...
# set the event
event.set()
```

The event can be marked as "*not set*" (whether it is currently set or not) via the **clear()** function.

```
...
# mark the event as not set
event.clear()
```

Finally, threads can wait for the event to set via the **wait()** function. Calling this function will block until the event is marked as set (e.g. another thread calling the **set()** function). If the event is already set, the **wait()** function will return immediately.

```
...
# wait for the event to be set
event.wait()
```

From reviewing the source code for **threading.Event** (https://github.com/python/cpython/blob/3.10/Lib/threading.py#L527), waiting threads are only notified when the **set()** function is called, not when the **clear()** function is called.

A "**timeout**" argument can be passed to the **wait()** function which will limit how long a thread is willing to wait in seconds for the event to be marked as set.

The **wait()** function will return **True** if the event was set while waiting, otherwise a value of **False** returned indicates that the event was not set and the call timed-out.

```
...
# wait for the event to be set with a timeout
event.wait(timeout=10)
```

Now that we know how to use a **threading.Event**, let's look at a worked example.

Example of Using an Event Object

We can explore how to use a **threading.Event** object.

In this example we will create a suite of threads that each will perform some processing and report a message. All threads will use an event to wait to be set before starting their work. The main thread will set the event and trigger the processing in all threads.

First, we can define a target task function that takes the shared **threading.Event** instance and a unique integer to identify the thread.

```
# target task function
def task(event, number):
    # ...
```

Next, the function will wait for the event to be set before starting the processing work.

```
...
# wait for the event to be set
event.wait()
```

Once triggered, the thread will generate a random number, block for a moment and report a message.

```
...
# begin processing
value = random()
sleep(value)
print(f'Thread {number} got {value}')
```

Tying this together, the complete target task function is listed below.

```
# target task function
def task(event, number):
    # wait for the event to be set
    event.wait()
    # begin processing
    value = random()
    sleep(value)
    print(f'Thread {number} got {value}')
```

The main thread will first create the shared **threading.Event** instance, which will be in the "*not set*" state by default.

```
...
# create a shared event object
event = Event()
```

Next, we can start five new threads specifying the target **task()** function with the event object and a unique integer as arguments.

```
...
# create a suite of threads
for i in range(5):
    thread = Thread(target=task, args=(event, i))
    thread.start()
```

Finally, the main thread will block for a moment, then trigger the processing in all of the threads via the event object.

```
...
# block for a moment
print('Main thread blocking...')
sleep(2)
# start processing in all threads
event.set()
```

Tying this all together, the complete example is listed below.

```
# SuperFastPython.com
# example of using an event object
from time import sleep
from random import random
from threading import Thread
from threading import Event

# target task function
def task(event, number):
    # wait for the event to be set
    event.wait()
    # begin processing
    value = random()
    sleep(value)
    print(f'Thread {number} got {value}')

# create a shared event object
event = Event()
# create a suite of threads
for i in range(5):
    thread = Thread(target=task, args=(event, i))
    thread.start()
# block for a moment
print('Main thread blocking...')
sleep(2)
# start processing in all threads
event.set()
# wait for all the threads to finish...
```

Running the example first creates and starts five threads. Each thread waits on the event before it starts processing.

The main thread blocks for a moment, allowing all threads to get started and start waiting on the event. The main thread then sets the event. This triggers all five threads that perform their processing and report a message.

Note, your specific results will differ given the use of random numbers.

```
Main thread blocking...
Thread 2 got 0.12310258010408259
Thread 4 got 0.2951602689055347
Thread 3 got 0.2995528547475702
Thread 0 got 0.6661975710559368
Thread 1 got 0.6687449622581129
```

You can learn more about event objects in this tutorial:

- How to Use an Event Object In Python (https://superfastpython.com/thread-event-object-in-python/)

Next, let's take a look at timer threads.

Timer Threads

A timer thread will execute a function after a time delay.

You can use a timer thread object in Python via the **threading.Timer** class.

How to Use a Timer Thread

Python provides a timer thread in the **threading.Timer** class (https://docs.python.org/3/library/threading.html#timer-objects).

The **threading.Timer** is an extension of the **threading.Thread** class, meaning that we can use it just like a normal thread instance.

It provides a useful way to execute a function after an interval of time.

First, we can create an instance of the timer and configure it. This includes the time to wait before executing in seconds, the function to execute once triggered, and any arguments to the target function.

For example:

```
...
# configure a timer thread
timer = Timer(10, task, args=(arg1, arg2))
```

The target task function will not execute until the time has elapsed.

Once created, the thread must be started by calling the **start()** function which will begin the timer.

```
...
# start the timer thread
timer.start()
```

If we decide to cancel the timer before the target function has executed, this can be achieved by calling the **cancel()** function.

For example:

```
...
# cancel the timer thread
timer.cancel()
```

Once the time has elapsed and the target function has executed, the timer thread cannot be reused.

Now that we know how to use a **threading.Timer** thread, let's look at some worked examples.

Example of Using a Timer Thread

We can explore how to use a **threading.Timer** object with a worked example.

In this example we will use a timer to delay some processing, in this case to report a custom message after a wait period.

First, we can define a target task function that takes a message, then reports it with a print statement.

```
# target task function
def task(message):
    # report the custom message
    print(message)
```

Next, we can create an instance of the **threading.Timer** class. We will configure the Timer with a delay of 3 seconds, then call the **task()** function with a single argument message of 'Hello world'.

```
...
# create a thread timer object
timer = Timer(3, task, args=('Hello world',))
```

Next, we can start the timer thread.

```
...
# start the timer object
timer.start()
```

Finally, the main thread will wait for the timer thread to complete before exiting.

```
...
# wait for the timer to finish
print('Waiting for the timer...')
```

Tying this together, the complete example is listed below.

```
# SuperFastPython.com
# example of using a thread timer object
from threading import Timer

# target task function
def task(message):
    # report the custom message
    print(message)

# create a thread timer object
timer = Timer(3, task, args=('Hello world',))
# start the timer object
timer.start()
# wait for the timer to finish
print('Waiting for the timer...')
```

Running the example first creates the **threading.Timer** object and specifies the target **task()** function with a single argument.

The timer is then started and the main thread waits for the timer thread to finish.

The timer thread runs, waits for the configured 3 seconds, then executes the **task()** function reporting the custom message.

```
Waiting for the timer...
Hello world
```

You can learn more about timer threads in this tutorial:

- How to Use a Timer Thread in Python (https://superfastpython.com/timer-thread-in-python/)

Next, let's take a closer look at thread barriers.

Thread Barrier

A barrier allows you to coordinate threads.

You can use a thread barrier in Python via the **threading.Barrier** class.

What is a Barrier

A barrier is a synchronization primitive (https://en.wikipedia.org/wiki/Barrier_(computer_science)).

It allows multiple threads to wait on the same barrier object instance (e.g. at the same point in code) until a predefined fixed number of threads arrive (e.g. the barrier is full), after which all threads are then notified and released to continue their execution.

Internally, a barrier maintains a count of the number of threads waiting on the barrier and a configured maximum number of parties (threads) that are expected. Once the expected number of parties reaches the pre-defined maximum, all waiting threads are notified.

This provides a useful mechanism to coordinate actions between multiple threads.

Now that we know what a barrier is, let's look at how we might use it in Python.

How to Use a Barrier

Python provides a barrier via the **threading.Barrier** class (https://docs.python.org/3/library/threading.html#barrier-objects).

A barrier instance must first be created and configured via the constructor specifying the number of parties (threads) that must arrive before the barrier will be lifted.

For example:

```
...
# create a barrier
barrier = threading.Barrier(10)
```

We can also perform an action once all threads reach the barrier which can be specified via the "action" argument in the constructor.

This action must be a callable such as a function or a lambda that does not take any arguments and will be executed by one thread once all threads reach the barrier but before the threads are released.

```
...
# configure a barrier with an action
barrier = threading.Barrier(10, action=my_function)
```

We can also set a default timeout used by all threads that reach the barrier and call the **wait()** function.

The default timeout can be set via the "**timeout**" argument in seconds in the constructor.

```
...
# configure a barrier with a default timeout
barrier = threading.Barrier(10, timeout=5)
```

Once configured, the barrier instance can be shared between threads and used.

A thread can reach and wait on the barrier via the **wait()** function, for example:

```
...
# wait on the barrier for all other threads to arrive
barrier.wait()
```

This is a blocking call and will return once all other threads (the pre-configured number of parties) have reached the barrier.

The wait function does return an integer indicating the number of parties remaining to arrive at the barrier. If a thread was the last thread to arrive, then the return value will be zero. This is helpful if you want the last thread or one thread to perform an action after the barrier is released, an alternative to using the "**action**" argument in the constructor.

```
...
# wait on the barrier
remaining = barrier.wait()
# after released, check if this was the last party
if remaining == 0:
    print('I was last...')
```

A timeout can be set on the call to wait in second via the "**timeout**" argument. If the timeout expires before all parties reach the barrier, a **BrokenBarrierError** will be raised in all threads waiting on the barrier and the barrier will be marked as broken.

If a timeout is used via the "**timeout**" argument or the default timeout in the constructor, then all calls to the **wait()** function may need to handle the **BrokenBarrierError**.

```
...
# wait on the barrier for all other threads to arrive
try:
    barrier.wait()
except BrokenBarrierError:
    # ...
```

We can also abort the barrier.

Aborting the barrier means that all threads waiting on the barrier via the **wait()** function will raise a **BrokenBarrierError** and the barrier will be put in the broken state.

This might be helpful if you wish to cancel the coordination effort.

```
...
# abort the barrier
barrier.abort()
```

A broken barrier cannot be used. Calls to **wait()** will raise a **BrokenBarrierError**.

A barrier can be fixed and made ready for use again by calling the **reset()** function.

This might be helpful if you cancel a coordination effort although you wish to retry it again with the same barrier instance.

```
...
# reset a broken barrier
barrier.reset()
```

Finally, the status of the barrier can be checked via attributes.

- **parties**: reports the configured number of parties that must reach the barrier for it to be lifted.
- **n_waiting**: reports the current number of threads waiting on the barrier.
- **broken**: attribute indicates whether the barrier is currently broken or not.

Now that we know how to use the barrier in Python, let's look at some worked examples.

Example of Using a Thread Barrier

We can explore how to use a **threading.Barrier** with a worked example.

In this example we will create a suite of threads, each required to perform some blocking calculation. We will use a barrier to coordinate all threads after they have finished their work and perform some action in the main thread. This is a good proxy for the types of coordination we may need to perform with a barrier.

First, let's define a target task function to execute by each thread. The function will take the barrier as an argument as well as a unique identifier for the thread.

The thread will generate a random value between 0 and 10, block for that many seconds, report the result, then wait on the barrier for all other threads to perform their computation.

The complete target task function is listed below.

```
# target function to prepare some work
def task(barrier, number):
    # generate a unique value
    value = random() * 10
    # block for a moment
    sleep(value)
    # report result
    print(f'Thread {number} done, got: {value}')
    # wait on all other threads to complete
    barrier.wait()
```

Next in the main thread we can create the barrier.

We need one party for each thread we intend to create, five in this place, as well as an additional party for the main thread that will also wait for all threads to reach the barrier.

```
...
# create a barrier
barrier = Barrier(5 + 1)
```

Next, we can create and start our five worker threads executing our target **task()** function.

```
...
# create the worker threads
for i in range(5):
    # start a new thread to perform some work
    worker = Thread(target=task, args=(barrier, i))
    worker.start()
```

Finally, the main thread can wait on the barrier for all threads to perform their calculation.

```
...
# wait for all threads to finish
print('Main thread waiting on all results...')
barrier.wait()
```

Once the threads are finished, the barrier will be lifted and the worker threads will exit and the main thread will report a message.

```
...
# report once all threads are done
print('All threads have their result')
```

Tying this together, the complete example is listed below.

```
```

```python
# SuperFastPython.com
# example of using a barrier
from time import sleep
from random import random
from threading import Thread
from threading import Barrier

# target function to prepare some work
def task(barrier, number):
    # generate a unique value
    value = random() * 10
    # block for a moment
    sleep(value)
    # report result
    print(f'Thread {number} done, got: {value}')
    # wait on all other threads to complete
    barrier.wait()

# create a barrier
barrier = Barrier(5 + 1)
# create the worker threads
for i in range(5):
    # start a new thread to perform some work
    worker = Thread(target=task, args=(barrier, i))
    worker.start()
# wait for all threads to finish
print('Main thread waiting on all results...')
barrier.wait()
# report once all threads are done
print('All threads have their result')
```

Running the example first creates the barrier then creates and starts the worker threads.

Each worker thread performs its calculation and then waits on the barrier for all other threads to finish.

Finally, the threads finish and are all released, including the main thread, reporting a final message.

Note, your specific results will differ given the use of random numbers. Try running the example a few times.

```
Main thread waiting on all results...
Thread 0 done, got: 0.029404047464883787
Thread 2 done, got: 0.5464335927441033
Thread 1 done, got: 4.99178858357096
Thread 4 done, got: 6.821886004822457
Thread 3 done, got: 7.868471944670812
All threads have their result
```

You can learn more about how to use thread barriers in this tutorial:

- How to Use a Thread Barrier in Python (https://superfastpython.com/thread-barrier-in-python/)

Next, let's take a look at best practices when using threads in Python.

Python Threading Best Practices

Now that we know threads work and how to use them, let's review some best practices to consider when bringing threads into our Python programs.

Some best practices when using threads in Python are as follows:

1. Use Context Managers
2. Use Timeouts When Waiting
3. Use a Mutex to Protect Critical Sections
4. Acquire Locks in Order

Following best practices will help you avoid common concurrency failure modes like race conditions and deadlocks.

Let's take a closer look at each in turn.

Tip 1: Use Context Managers

Acquire and release locks using a context manager, wherever possible.

Locks can be acquired manually via a call to **acquire()** at the beginning of the critical section followed by a call to **release()** at the end of the critical section.

For example:

```
...
# acquire the lock manually
lock.acquire()
# critical section...
# release the lock
lock.release()
```

This approach should be avoided wherever possible.

Traditionally, it was recommended to always acquire and release a lock in a try-finally structure.

The lock is acquired, the critical section is executed in the try block, and the lock is always released in the finally block.

For example:

```
...
# acquire the lock
lock.acquire()
try:
    # critical section...
finally:
    # always release the lock
    lock.release()
```

This was since replaced with the context manager interface that achieves the same thing with less code.

For example:

```
...
# acquire the lock
with lock:
    # critical section...
```

The benefit of the context manager is that the lock is always released as soon as the block is exited, regardless of how it is exited, e.g. normally, a return, an error, or an exception.

This applies to a number of concurrency primitives, such as:

- Acquiring a mutex lock via the **threading.Lock** class.
- Acquiring a reentrant mutex lock via the **threading.RLock** class.
- Acquiring a semaphore via the **threading.Semaphore** class.
- Acquiring a condition via the **threading.Condition** class.

Tip 2: Use Timeouts When Waiting

Always use a timeout when waiting on a blocking call.

Many calls made on concurrency primitives may block.

For example:

- Waiting to acquire a **threading.Lock** via **acquire()**.
- Waiting for a thread to terminate via **join()**.
- Waiting to be notified on a **threading.Condition** via **wait()**.
- And more.

All blocking calls on concurrency primitives take a "**timeout**" argument and return **True** if the call was successful or **False** otherwise.

Do not call a blocking call without a timeout, wherever possible.

For example:

```
...
# acquire the lock
if not lock.acquire(timeout=2*60):
    # handle failure case...
```

This will allow the waiting thread to give-up waiting after a fixed time limit and then attempt to rectify the situation, e.g. report an error, force termination, etc.

Tip 3: Use a Mutex to Protect Critical Sections

Always use a mutual exclusion (mutex) lock to protect critical sections in code.

Critical sections are sensitive parts of code that can be executed by multiple threads concurrently and may result in race conditions.

A critical section may refer to a single block of code, but it also refers to multiple accesses to the same data variable or resource from multiple functions.

A mutex lock can be used to ensure that only one thread at a time executes a critical section of code at a time, while all other threads trying to execute the same code must wait until the currently executing thread is finished with the critical section and releases the lock.

Each thread must attempt to acquire the lock at the beginning of the critical section. If the lock has not been obtained, then a thread will acquire it and other threads must wait until the thread that acquired the lock releases it.

Mutex locks are provided by the threading.Lock class and can be acquired and released automatically using the context manager interface.

For example:

```
...
# acquire the lock
with lock:
    # critical section...
```

Tip 4: Acquire Locks in Order

Acquire locks in the same order throughout the application, wherever possible.

This is called "*lock ordering*".

In some applications you may be able to abstract the acquisition of locks using a list of **threading.Lock** objects that may be iterated and acquired in order, or a function call that acquires locks in a consistent order.

When this is not possible, you may need to audit your code to confirm that all paths through the code acquire the locks in the same order.

Next, let's look at common errors when using threads.

Python Threading Common Errors

This section gives examples of general errors encountered by developers when using threads in Python.

These errors are not specific to Python, but instead refer to general concurrency failure modes.

Do you have a question about common errors when using threads?
Ask your question in the comments below and I will do my best to answer it and perhaps add it to this list of questions.

To keep things simple, there are four common threading errors; they are:

1. Race conditions
    1. Race Condition With Variables
    2. Race Condition With Timing
2. Thread Deadlocks

3. Thread Livelocks

Race Conditions

A race condition (https://en.wikipedia.org/wiki/Race_condition) is a bug in concurrency programming.

It is a failure case where the behavior of the program is dependent upon the order of execution by two or more threads. This means, the behavior of the program will not be predictable, possibly changing each time it is run.

NOTE: Race conditions are a real problem in Python when using threads, even in the presence of the global interpreter lock (GIL). The refrain that there are no race conditions in Python because of the GIL is dangerously wrong.

There are many types of race conditions, although perhaps two main categories that we can look at, they are:

1. Race conditions that involve shared variables.
2. Race conditions that involve coordinating multiple threads.

Let's take a look at each.

Race Condition With Variables

A common type of race condition is when two or more threads attempt to change the same data variable.

For example, one thread may be adding values to a variable, while another thread is subtracting values from the same variable.

Let's call them an adder thread and a subtractor thread.

At some point, the operating system may context switch from the adding thread to the subtracting thread in the middle of updating the variable. Perhaps right at the point where it was about to write an updated value with an addition, say from the current value of 100 to the

new value of 110.

```
1 ...
2 # add to the variable
3 variable = variable + 10
```

You may recall that the operating system controls what threads execute and when, and that a context switch refers to pausing the execution of a thread and storing its state, while unpausing another thread and restoring its state.

You may also notice that the adding or subtracting from the variable is composed of at least three steps:

1. Read the current value of the variable.
2. Calculate a new value for the variable.
3. Write a new value for the variable.

A context switch between threads may occur at any point in this task.

Back to our threads. The subtracting thread runs and reads the current value as 100 and reduces the value from 100 to 90.

```
1 ...
2 # subtract from the variable
3 variable = variable - 10
```

This subtraction is performed as per normal and the variable value is now 90.

The operating system context switches back to the adding thread and it picks up where it left off writing the value 110.

This means that in this case, one subtraction operation was lost and the shared balance variable has an inconsistent value. A race condition.

There are many ways to fix a race condition between two or more threads with a shared variable.

The approach taken may depend on the specifics of the application.

Nevertheless, a common approach is to protect the critical section of code. This may be achieved with a mutual exclusion lock, sometimes simply called a mutex.

You may recall that a critical section of code is code that may be executed by two or more threads concurrently and may be at risk of a race condition.

Updating a variable shared between threads is an example of a critical section of code.

A critical section can be protected by a mutex lock which will ensure that one and only one thread can access the variable at a time.

This can be achieved by first creating a **threading.Lock** instance.

```
...
# create a lock
lock = threading.Lock()
```

When a thread is about to execute the critical section, it can acquire the lock by calling the **acquire()** function. It can then execute the critical section and release the lock by calling the **release()** function on the lock.

For example:

```
...
# acquire the lock
lock.acquire()
# add to the variable
variable = variable + 10
# release the lock
lock.release()
```

We might also use the context manager on the lock, which will acquire and release the lock automatically for us.

```
...
# acquire the lock
with lock:
    # add to the variable
    variable = variable + 10
# release the lock automatically
```

If one thread has acquired the lock, another thread cannot acquire it, therefore cannot execute a critical section and in turn cannot update the shared variable.

Instead, any threads attempting to acquire the lock while it is acquired must wait until the lock is released. This waiting for the lock to be released is performed automatically within the call to **acquire()**, no need to do anything special.

You can learn more about this type of race condition in the tutorial:

- Race Condition With a Shared Variable in Python (https://superfastpython.com/thread-race-condition-shared-variable/)

Race Condition With Timing

Another common type of race condition is when two threads attempt to coordinate their behavior.

For example, consider the case of the use of a **threading.Condition** used by two threads to coordinate their behavior.

One thread may wait on the **threading.Condition** to be notified of some state change within the application via the wait function.

```
...
# wait for a state change
with condition:
    condition.wait()
```

Recall that when using a **threading.Condition**, you must acquire the condition before you can call **wait()** or **notify()**, then release it once you are done. This is easily achieved using the context manager.

Another thread may perform some change within the application and alert the waiting thread via the condition with the **notify()** function.

```
...
# alert the waiting thread
with condition:
    condition.notify()
```

This is an example of coordinated behavior between two threads where one thread signals another thread.

For the behavior to work as intended, the notification from the second thread must be sent after the first thread has started waiting. If the first thread calls **wait()** after the second thread calls **notify()**, then it will not be notified and will wait forever.

This may happen if there is a context switch by the operating system that allows the second thread that calls **notify()** to run before the first thread that calls **wait()** to run.

You may recall that the operating system controls what threads execute and when, and that a context switch refers to pausing the execution of a thread and storing its state, while unpausing another thread and restoring its state.

There are many ways to fix a race condition between two or more threads based on timing.

The approach taken may depend on the specifics of the application.

Nevertheless, one common approach is to have the waiting threads signal that they are ready before the notifying thread starts its work and calls notify.

This can be achieved with a **threading.Event**, which is like a thread-safe boolean flag variable.

A shared **threading.Event** object can be created.

```
...
# create a shared event
event = threading.Event()
```

When the waiting thread (or threads) are ready they can signal this fact via setting the event flag to true.

```
...
# signal that the waiting thread is ready
event.set()
```

Importantly, this should be performed while the waiting thread has acquired the **threading.Condition**. For example:

```
...
# wait to be notified
with condition:
    # indicate we are ready to be notified
    event.set()
    # wait to be notified
    condition.wait()
```

This will be clearer in a moment, but it ensures we avoid fixing the timing race condition by introducing a second timing race condition. Specifically between one thread setting the flag and another waiting on the flag to be set.

The notifying thread can then wait for the waiting thread to signal that it is ready before doing its work, such as changing state, and notifying the waiting thread.

This can be achieved using busy waiting. That is, a type of waiting where we check for the event flag to be set in a loop.

To lessen the computational effort of the busy wait loop, the main thread can sleep for a fraction of a second each iteration.

For example:

```
...
# busy wait for the new thread to get ready
while not event.is_set():
    sleep(0.1)
```

When the **threading.Event** flag is set by the new thread, the main thread will exit its busy wait loop.

At this time the new thread has already acquired the **threading.Condition** and will call **wait()**.

After exiting the busy wait loop, the main thread will then acquire the **threading.Condition** and notify the waiting thread.

```
...
# notify the new thread
with condition:
    condition.notify()
```

Importantly, if there was a context switch or a delay of any kind between the new thread calling **event.set()** and **condition.wait()**, it will not introduce a second race condition.

For example:

```
...
# wait to be notified
with condition:
    # indicate we are ready to be notified
    event.set()
    # add an artificial delay
    sleep(5)
    # wait to be notified
    condition.wait()
```

The reason is because the new thread has acquired the **threading.Condition** and no other thread can acquire it until it releases it or waits on it.

Therefore, even if the new thread performs a long calculation or a sleep between setting the event and waiting on the condition, the main thread will not be able to acquire the threading.Condition (and call **notify()**) until the new thread calls **wait()**.

You can learn more about this type of race condition in the tutorial:

- How to Fix a Race Condition With Timing in Python (https://superfastpython.com/thread-race-condition-timing/)

Thread Deadlocks

A deadlock is a concurrency failure mode where a thread or threads wait for a condition that never occurs.

The result is that the deadlock threads are unable to progress and the program is stuck or frozen and must be terminated forcefully.

There are many ways in which you may encounter a deadlock in your concurrent program.

Deadlocks are not developed intentionally, instead, they are an unexpected side effect or bug in concurrency programming.

Common examples of the cause of threading deadlocks include:

- A thread that waits on itself (e.g. attempts to acquire the same mutex lock twice).
- Threads that wait on each other (e.g. A waits on B, B waits on A).
- Thread that fails to release a resource (e.g. mutex lock, semaphore, barrier, condition, event, etc.).
- Threads that acquire mutex locks in different orders (e.g. fail to perform lock ordering).

Deadlocks may be easy to describe, but hard to detect in an application just from reading code.

It is important to develop an intuition for the causes of different deadlocks. This will help you identify deadlocks in your own code and trace down the causes of those deadlocks that you may encounter.

Now that we are familiar with what a deadlock is, let's look at a worked example.

A common cause of a deadlock is a thread that waits on itself.

We do not intend for this deadlock to occur, e.g. we don't intentionally write code that causes a thread to wait on itself. Instead, this occurs accidentally due to a series of function calls and variables being passed around.

A thread may wait on itself for many reasons, such as:

- Waiting to acquire a mutex lock that it has already acquired.
- Waiting to be notified on a condition by itself.
- Waiting for an event to be set by itself.
- Waiting for a semaphore to be released by itself.
- And so on.

One example that cannot occur is that a thread cannot explicitly wait for itself to terminate with a call to **join()**. This situation is detected and a **RuntimeError** is raised.

We can demonstrate a deadlock caused by a thread waiting on itself.

In this case we will develop a **task()** function that directly attempts to acquire the same mutex lock twice. That is, the task will acquire the lock, then attempt to acquire the lock again.

This will cause a deadlock as the thread already holds the lock and will wait forever for itself to release the lock so that it can acquire it again.

The **task()** function that attempts to acquire the same lock twice and trigger a deadlock is listed below.

```
# task to be executed in a new thread
def task(lock):
    print('Thread acquiring lock...')
    with lock:
        print('Thread acquiring lock again...')
        with lock:
            # will never get here
            pass
```

In the main thread, we can then create the lock.

```
...
# create the mutex lock
lock = Lock()
```

We will then create and configure a new thread to execute our **task()** function in a new thread, then start the thread and wait for it to terminate, which it never will.

```
...
# create and configure the new thread
thread = Thread(target=task, args=(lock,))
# start the new thread
thread.start()
# wait for threads to exit...
thread.join()
```

Tying this together, the complete example is listed below.

```
# SuperFastPython.com
# example of a deadlock caused by a thread waiting on itself
from threading import Thread
from threading import Lock

# task to be executed in a new thread
def task(lock):
    print('Thread acquiring lock...')
    with lock:
        print('Thread acquiring lock again...')
        with lock:
            # will never get here
            pass

# create the mutex lock
lock = Lock()
# create and configure the new thread
thread = Thread(target=task, args=(lock,))
# start the new thread
thread.start()
# wait for threads to exit...
thread.join()
```

Running the example first creates the lock.

The new thread is then confused and started and the main thread blocks until the new thread terminates, which it never does.

The new thread runs and first acquires the lock. It then attempts to acquire the same mutex lock again and blocks.

It will block forever waiting for the lock to be released. The lock cannot be released because the thread already holds the lock. Therefore the thread has deadlocked.

The program must be terminated forcefully, e.g. killed via Control-C.

```
Thread acquiring lock...
Thread acquiring lock again...
```

Perhaps the above example is too contrived.

Attempting the same lock is common if you protect a critical section with a lock and within that critical section you call another function that attempts to acquire the same lock.

For example, we can update the previous example to split the **task()** function into two functions **task1()** and **task2()**. The **task1()** function acquires the lock, does some work, then calls **task2()** that does some work and attempts to acquire the lock again.

For example:

```
# task2 to be executed in a new thread
def task2(lock):
    print('Thread acquiring lock again...')
    with lock:
        # will never get here
        pass

# task1 to be executed in a new thread
def task1(lock):
    print('Thread acquiring lock...')
    with lock:
        task2(lock)
```

This is a more realistic scenario and is easy to fall into.

For example, you may have a custom class that has a lock as a member variable to protect state within the object. Each method called on your object uses the lock to protect state, and some functions call other functions internally to reuse code.

Nevertheless, the complete updated example with the two task functions is listed below.

```
# SuperFastPython.com
# example of a deadlock caused by a thread waiting on itself
from threading import Thread
from threading import Lock

# task2 to be executed in a new thread
def task2(lock):
    print('Thread acquiring lock again...')
    with lock:
        # will never get here
        pass

# task1 to be executed in a new thread
def task1(lock):
    print('Thread acquiring lock...')
    with lock:
        task2(lock)

# create the mutex lock
lock = Lock()
# create and configure the new thread
thread = Thread(target=task1, args=(lock,))
# start the new thread
thread.start()
# wait for threads to exit...
thread.join()
```

Running the example creates the lock and then creates and starts the new thread.

The thread acquires the lock in **task1()**, simulates some work then calls **task2()**. The **task2()** function attempts to acquire the same lock and the thread is stuck in a deadlock waiting for the lock to be released by itself so it can acquire it again.

```
1 Thread acquiring lock...
2 Thread acquiring lock again...
```

This specific deadlock with a mutex lock can be avoided by using a reentrant mutex lock. This allows a thread to acquire the same lock more than once.

A reentrant lock is recommended any time you may have code that acquires a lock that may call other code that may acquire the same lock.

You can learn more about thread deadlocks in the tutorial:

- How to Identify a Deadlock in Python (https://superfastpython.com/thread-deadlock-in-python)

Thread Livelocks

A livelock is a concurrency failure case where a thread is not blocked but is unable to make progress because of the actions of another thread.

A livelock typically involves two or more threads that share concurrency primitives such as mutual exclusion (mutex) locks.

The threads may both attempt to acquire a lock or series of locks at the same time, detect that they are competing for the same resource, and then back-off. This process is repeated and neither thread is able to make progress and are "*locked*".

A livelock can be contrasted with a deadlock.

The main difference between a livelock and a deadlock is the state of the thread or threads while they are unable to make progress.

In a livelock the threads execute and are not blocked, whereas in a deadlock the threads are blocked, e.g. waiting on a concurrency primitive such as a lock, and are unable to to execute code.

- **Livelock**: Threads cannot make progress but can continue to execute code.
- **Deadlock**: Threads cannot make progress and are blocked, e.g. waiting.

Now that we know what a livelock is, let's look at a worked example.

We can make the idea of a livelock clear with a worked example.

In this example we will define a task function to be executed by worker threads. The function will involve acquiring one lock, then attempting to acquire the second lock. If the second lock is already locked, it will back-off, release the first lock and wait a fraction of a second before trying again.

If the function is executed by two threads at the same time and the threads acquire the locks in a different order then it will lead to a live lock.

For example:

- **Thread 1**: Acquires Lock1, checks Lock2 and backs off if it is not free.
- **Thread 2**: Acquires Lock2, checks Lock1 and backs off if it is not free.

First, let's define a function that attempts to acquire locks in an order and backs off if the second lock is not available.

The function will take a unique number to identify the thread, and the two locks to acquire as arguments.

```
1 # task for worker threads
2 def task(number, lock1, lock2):
3     # ...
```

The task will continue to attempt to acquire both locks forever. This can be achieved with a while-loop.

```
1 ...
2 # loop until the task is completed
3 while True:
4     # ...
```

The task attempts to acquire the first lock.

```
...
# acquire the first lock
with lock1:
    # ...
```

We need to contrive the situation by ensuring the lock is held while the other thread attempts to acquire it. Therefore we will force the task to block for a moment while holding the lock.

```
...
# wait a moment
sleep(0.1)
```

Next, the task checks if the second lock is available, and if not it gives up. Otherwise, if the lock is available, it attempts to acquire it and if so completes the task and breaks out of the outer while loop.

```
...
# check if the second lock is available
if lock2.locked():
    print(f'Task {number} cannot get the second lock, giving up...')
else:
    # acquire lock2
    with lock2:
        print(f'Task {number} made it, all done.')
        break
```

Tying this together, the complete **task()** function is listed below.

```
# task for worker threads
def task(number, lock1, lock2):
    # loop until the task is completed
    while True:
        # acquire the first lock
        with lock1:
            # wait a moment
            sleep(0.1)
            # check if the second lock is available
            if lock2.locked():
                print(f'Task {number} cannot get the second lock, giving up...')
            else:
                # acquire lock2
                with lock2:
                    print(f'Task {number} made it, all done.')
                    break
```

Finally, in the main thread, we can first create both mutex locks.

```
...
# create locks
lock1 = Lock()
lock2 = Lock()
```

We can then create and configure two threads, both executing the same task at the same time. Importantly, we will switch the order of the locks provided as arguments to each thread to force the live-lock situation.

```

```

```
...
# create threads
thread1 = Thread(target=task, args=(0, lock1, lock2))
thread2 = Thread(target=task, args=(1, lock2, lock1))
```

We can then start both worker threads and wait for the tasks to finish.

```
...
# start threads
thread1.start()
thread2.start()
# wait for threads to finish
thread1.join()
thread2.join()
```

Tying this together, the complete example of a livelock is listed below.

```
# SuperFastPython.com
# example of a livelock
from time import sleep
from threading import Thread
from threading import Lock

# task for worker threads
def task(number, lock1, lock2):
    # loop until the task is completed
    while True:
        # acquire the first lock
        with lock1:
            # wait a moment
            sleep(0.1)
            # check if the second lock is available
            if lock2.locked():
                print(f'Task {number} cannot get the second lock, giving up...')
            else:
                # acquire lock2
                with lock2:
                    print(f'Task {number} made it, all done.')
                    break

# create locks
lock1 = Lock()
lock2 = Lock()
# create threads
thread1 = Thread(target=task, args=(0, lock1, lock2))
thread2 = Thread(target=task, args=(1, lock2, lock1))
# start threads
thread1.start()
thread2.start()
# wait for threads to finish
thread1.join()
thread2.join()
```

Running the example first creates the locks.

The two worker threads are then configured and started and the main thread blocks while waiting for the tasks to complete.

The first thread acquires lock1 and blocks for a moment. The second thread acquires lock2 and blocks for a moment.

The first thread then checks if lock2 is available. It isn't so it gives up and repeats the loop. At the same time, the second thread checks if lock1 is available. It isn't so it also gives up and repeats the loop.

This process of both threads getting one lock and attempting to get the next lock and giving up is then repeated forever.

Note, you will have to manually kill the process (e.g. Control-C) in order to stop it.

```
...
Task 0 cannot get the second lock, giving up...
Task 1 cannot get the second lock, giving up...
Task 0 cannot get the second lock, giving up...
Task 1 cannot get the second lock, giving up...
Task 0 cannot get the second lock, giving up...
Task 1 cannot get the second lock, giving up...
Task 0 cannot get the second lock, giving up...
Task 1 cannot get the second lock, giving up...
Task 0 cannot get the second lock, giving up...
Task 1 cannot get the second lock, giving up...
```

You can learn more about thread livelocks in the tutorial:

- Thread Livelocks in Python (https://superfastpython.com/thread-livelock-in-python)

Next, let's look at common questions about threading in Python.

Python Threading Common Questions

This section answers common questions asked by developers when using threads in Python.

Do you have a question about threads?
Ask your question in the comments below and I will do my best to answer it and perhaps add it to this list of questions.

How to Stop a Thread?

Python does not provide the ability in the threading API to stop a thread.

Instead, you can add this functionality to your code directly.

A thread can be stopped using a shared boolean variable such as a **threading.Event**.

A **threading.Event** is a thread-safe boolean variable flag that can be either set or not set. It can be shared between threads and checked and set without fear of a race condition.

A new event can be created and then shared between threads, for example:

```
...
# create a shared event
event = Event()
```

The event is created in the '*not set*' or **False** state.

We may have a task in a custom function that is run in a new thread. The task may iterate, such as in a while-loop or a for-loop.

For example:

```
# custom task function
def task():
    # perform task in iterations
    while True:
        # ...
```

We can update our task function to check the status of an event each iteration.

If the event is set true, we can exit the task loop or return from the **task()** function, allowing the new thread to terminate.

The status of the **threading.Event** can be checked via the **is_set()** function.

For example:

```
# custom task function
def task():
    # perform task in iterations
    while True:
        # ...
        # check for stop
        if event.is_set():
            break
```

The main thread, or another thread, can then set the event in order to stop the new thread from running.

The event can be set or made **True** via the **set()** function.

For example:

```
...
# stop the thread
thread.set()
# wait for the new thread to stop
thread.join()
```

Now that we know how to stop a Python thread, let's look at a worked example.

We can develop an example that runs a function in a new thread.

The new task function will execute in a loop and once finished the new thread will terminate and the main thread will terminate.

Firstly, we can define the task function.

The function will take an instance of a **threading.Event** instance as an argument. It will execute a loop five times. Each iteration it will block for a second, then report a message in an effort to simulate work and show progress.

The task loop to check the status of the event each iteration, and if set to break the task loop.

Once the task is finished, the function will report a final message.

The **task()** function below implements this.

```
# custom task function
def task(event):
    # execute a task in a loop
    for i in range(5):
        # block for a moment
        sleep(1)
        # check for stop
        if event.is_set():
            break
        # report a message
        print('Worker thread running...')
    print('Worker closing down')
```

In the main thread can first create a new **threading.Event** instance to be shared between the threads.

```
...
# create the event
event = Event()
```

Next, in the main thread we can create a new **threading.Thread** instance that is configured to execute our **task()** function in a new thread via the "**target**" argument.

```
```

```
...
# create and configure a new thread
thread = Thread(target=task)
```

We can then immediately start the new thread via the **start()** function.

```
...
# start the new thread
thread.start()
```

The main thread will then wait for the new thread to finish by joining the thread.

```
...
# wait for the new thread to finish
thread.join()
```

The main thread will then block for a few seconds and then stop the new thread by setting the event.

```
...
# block for a while
sleep(3)
# stop the worker thread
print('Main stopping thread')
event.set()
```

Tying this together, the complete example is listed below.

```
# SuperFastPython.com
# example of stopping a new thread
from time import sleep
from threading import Thread
from threading import Event

# custom task function
def task(event):
    # execute a task in a loop
    for i in range(5):
        # block for a moment
        sleep(1)
        # check for stop
        if event.is_set():
            break
        # report a message
        print('Worker thread running...')
    print('Worker closing down')

# create the event
event = Event()
# create and configure a new thread
thread = Thread(target=task, args=(event,))
# start the new thread
thread.start()
# block for a while
sleep(3)
# stop the worker thread
print('Main stopping thread')
event.set()
# wait for the new thread to finish
thread.join()
```

Running the example first creates and starts the new thread.

The main thread then blocks for a few seconds.

Meanwhile, the new thread executes its task loop, blocking and reporting a message each iteration. It checks the event each iteration, which remains false and does not trigger the thread to stop.

The main thread wakes up, and then sets the event. It then joins the new thread, waiting for it to terminate.

The task thread checks the event which is discovered to be set (e.g, True). The thread breaks the task loop, reports a final message then terminates the new thread.

The main thread then terminates, closing the Python process.

```
1 Worker thread running...
2 Worker thread running...
3 Main stopping thread
4 Worker closing down
```

You can learn more about stopping a thread in this tutorial:

- How to Stop a Thread in Python (https://superfastpython.com/stop-a-thread-in-python)

How to Kill a Thread?

A thread can be killed by killing the process to which it belongs.

A common approach to killing a Python process is by sending it the SIGINT signal (https://en.wikipedia.org/wiki/Signal_(IPC)#SIGINT) (signal interrupt).

This can be achieved by the Control-C command (https://en.wikipedia.org/wiki/Control-C).

Alternatively, the SIGQUIT signal (https://en.wikipedia.org/wiki/Signal_(IPC)#SIGKILL) can be set to the process.

This can be achieved via the Control-\ command (https://en.wikipedia.org/wiki/Control-%5C).

An alternate approach is to use the kill command (https://en.wikipedia.org/wiki/Kill_(command)) on POSIX machines (e.g. Linux and MacOS) and specify the process id.

There are also many approaches, specific to your underlying operating system.

How Do You Wait for Threads to Finish?

You can wait for a thread to finish by joining it.

A thread can be joined in Python by calling the **Thread.join()** method (https://docs.python.org/3/library/threading.html#threading.Thread.join).

For example:

```
...
# join a thread
thread.join()
```

This has the effect of blocking the current thread until the target thread that has been joined has terminated.

The target thread that is being joined may terminate for a number of reasons, such as:

- Finishes executing its target function.
- Finishes executing its **run()** method if it extends the **Thread** class.
- Raised an error or exception.

Once the target thread has finished, the **join()** method will return and the current thread can continue to execute.

You can learn more about joining a thread in this tutorial:

- How to Join a Thread in Python (https://superfastpython.com/join-a-thread-in-python/)

How to Restart a Thread?

Python threads cannot be restarted or reused.

In fact, this is probably a limitation of the capabilities of threads provided by the underlying operating system.

Once a thread has terminated you cannot call the **start()** method on it again to reuse it.

Recall that a thread may terminate for many reasons such as raising an error or exception, or when it finishes executing its **run()** function.

Calling the **start()** function on a terminated thread will result in a RuntimeError indicating that threads can only be started once.

For example, the listing below starts a thread, lets it terminate and then attempts to start it again.

```
# SuperFastPython.com
# example of reusing a thread
from time import sleep
from threading import Thread

# custom target function
def task():
    # block for a moment
    sleep(1)
    # report a message
    print('Hello from the new thread')

# create a new thread
thread = Thread(target=task)
# start the thread
thread.start()
# wait for the thread to finish
thread.join()
# try and start the thread again
thread.start()
```

Running the example first creates a thread instance then starts its execution.

The new thread is started, blocks for a moment then reports a message.

The main thread joins the new thread and waits for it to terminate. Once terminated, the main thread attempts to start the same thread again.

The result is a **RuntimeError**, as we expected.

```
Hello from the new thread
Traceback (most recent call last):
...
    raise RuntimeError("threads can only be started once")
RuntimeError: threads can only be started once
```

Instead, to restart a thread in Python, you must create a new instance of the thread with the same configuration and then call the **start()** function.

You can learn more about restarting a terminating thread in this tutorial:

- How to Restart a Thread in Python (https://superfastpython.com/restart-a-thread-in-python/)

How to Change a Thread's Name?

You can set the thread name in the **threading.Thread** constructor or via the "**name**" property of the **threading.Thread** class.

We can get the name of a thread via the "**name**" property on the **threading.Thread** instance.

For example:

```
...
# report the name of the thread
print(thread.name)
```

The thread name will default to the format "**Thread-%d**", where "**%d**" is an integer for the thread number created by the process, counted from 1 sequentially up for each new thread. This means that the first new thread you might create would have the name "**Thread-1**".

When creating a new instance of a **threading.Thread**, we can specify the name of the thread via the "**name**" argument.

For example:

```
...
# create with a custom name
thread = Thread(name='MyThread')
```

We can also configure the thread name after the **threading.Thread** instance has been created via the "**name**" property.

For example:

```

```

```
...
# create a thread with default name
thread = Thread()
# set the name
thread.name = 'MyThread'
```

You can learn more about changing the name of a thread in this tutorial:

- How to Change the Thread Name in Python (https://superfastpython.com/thread-name/)

How to Use a Countdown Latch in Python?

A latch or countdown latch is a synchronization primitives used in concurrent programming.

It is created in the closed position and requires a count to be decremented until zero before opening.

The count is decremented by threads that pass through the latch, calling a **count_down()** function. This is a non-blocking call, allowing the caller to proceed immediately.

Other threads register interest in the latch by calling **wait()** to block on the latch until it is opened.

A latch is used to coordinate more threads with the full opening of the latch. It just so happens that the latch is designed to be opened incrementally each time other threads pass through, e.g. counting down from a specified number to zero.

A latch and a barrier are very similar, the main difference is:

- The “countdown” and “waiting” are performed by separate parties (threads) on the latch.
- The “countdown” and “waiting” are performed by the same parties (threads) on the barrier.

Python does not provide a countdown latch, but we can develop one easily using a new class and a **threading.Condition**.

We can develop a simple countdown latch class named **CountDownLatch**. It must have three elements:

- **Constructor** that require a count and initializes the internal locks.
- **CountDown** that decrements the counter in a thread safe manner and notify waiting threads if the count reaches zero.
- **Wait** that allows threads to block until the count reaches zero.

We will require a wait/notify structure within the latch. Python provides a **threading.Condition** that supports wait/notify directly.

Let's start developing our class.

```
# simple countdown latch, starts closed then opens once count is reached
class CountDownLatch():
    # ...
```

The constructor will take a count specified by the user, indicating the expected number of parties to arrive before the latch is opened.

We will store this count and then create a new **threading.Condition** used to manage changes to the count and to allow threads to wait and be notified.

```
# constructor
def __init__(self, count):
    # store the count
    self.count = count
    # control access to the count and notify when latch is open
    self.condition = Condition()
```

The **count_down()** function must first acquire the condition before doing anything to ensure any checking and changing of the internal count is thread safe.

This can be achieved using the context manager.

```
...
# acquire the lock on the condition
with self.condition:
    # ...
```

Next, we need to check if the latch is already open, and if so return immediately.

```
...
# check if the latch is already open
if self.count == 0:
    return
```

We can then decrement the counter.

```
...
# decrement the counter
self.count -= 1
```

Finally, we can check if the counter has reached zero and whether we should notify all waiting threads.

```
...
# check if the latch is now open
if self.count == 0:
    # notify all waiting threads that the latch is open
    self.condition.notify_all()
```

Tying this together, the complete **count_down()** method is listed below.

```
# count down the latch by one increment
def count_down(self):
    # acquire the lock on the condition
    with self.condition:
        # check if the latch is already open
        if self.count == 0:
            return
        # decrement the counter
        self.count -= 1
        # check if the latch is now open
        if self.count == 0:
            # notify all waiting threads that the latch is open
            self.condition.notify_all()
```

The **count_down()** function could be extended to support counting down by a specified amount. It may also raise an **Exception** if the latch is already open, if that truly is an error state (it feels like it is).

Finally, we need a **wait()** function.

Again, we must acquire the condition before we can check the internal state in a thread-safe manner.

```
...
# acquire the lock on the condition
with self.condition:
    # ...
```

We can then check if the latch is already open, in which case we can return immediately.

```
...
# check if the latch is already open
if self.count == 0:
    return
```

Otherwise, we can wait on the condition to be notified that the latch will be opened.

```
...
# wait to be notified when the latch is open
self.condition.wait()
```

Tying this together, the complete **wait()** function is listed below.

```
# wait for the latch to open
def wait(self):
    # acquire the lock on the condition
    with self.condition:
        # check if the latch is already open
        if self.count == 0:
            return
        # wait to be notified when the latch is open
        self.condition.wait()
```

The **wait()** function could be extended to support a timeout.

And that's it.

Tying this together, the complete **CountDownLatch** class is listed below.

```
# simple countdown latch, starts closed then opens once count is reached
class CountDownLatch():
    # constructor
    def __init__(self, count):
        # store the count
        self.count = count
        # control access to the count and notify when latch is open
        self.condition = Condition()

    # count down the latch by one increment
    def count_down(self):
        # acquire the lock on the condition
        with self.condition:
            # check if the latch is already open
            if self.count == 0:
                return
            # decrement the counter
            self.count -= 1
            # check if the latch is now open
            if self.count == 0:
                # notify all waiting threads that the latch is open
                self.condition.notify_all()

    # wait for the latch to open
    def wait(self):
        # acquire the lock on the condition
        with self.condition:
            # check if the latch is already open
            if self.count == 0:
                return
            # wait to be notified when the latch is open
            self.condition.wait()
```

You might want to add extra methods to the latch as a fun extension.

For example, we could add a method to get the current value of the internal count. We also might want to add a method to reset the count, but perhaps only if the latch is already open, e.g. reset from open to closed state.

We can use this countdown class by first creating an instance of the class and specifying the number of expected parties to arrive as an argument to the constructor.

```
...
# create the countdown latch
latch = CountDownLatch(5)
```

As each party arrives at the latch, we can call the **count_down()** function.

For example:

```
...
# create the countdown latch
latch = CountDownLatch(5)
```

Any threads interested in when the latch is open can call the **wait()** function.

For example:

```
...
# block until the latch is open
latch.wait()
```

You can learn more about developing a countdown latch in this tutorial:

- Thread Countdown Latch in Python (https://superfastpython.com/thread-countdown-latch)

How to Wait for a Result from a Thread?

It is common to use a new thread to perform a sub-task required by another thread.

This is often a task that involves blocking function calls, such as reading or writing a file, a socket, or similar external resource.

Python threads do not provide a direct method to get results from a new thread to a waiting thread.

Instead indirect methods can be used, such as:

1. Use instance variables on an extended **threading.Thread**.
2. Use a **queue.Queue** to share a result between threads.
3. Use a global variable to share a result between threads.

You can learn more about how to return a result from a thread in this tutorial:

- How to Return Values From a Thread in Python (https://superfastpython.com/thread-return-values/)

Before we can get the result from the new thread, the new thread must prepare the result. This means that the other thread must wait until the result is prepared and made available.

There are many ways for one to wait for a result from another thread.

Five common approaches include:

1. **Use a Sleep**: Use a busy-wait loop with a call to **time.sleep()**.
2. **Join the Thread**: Call **join()** to wait for the new thread to terminate.
3. **Use an Event**: Wait on a **threading.Event** to be set.
4. **Use a Wait/Notify**: Wait on a **threading.Condition** to be notified about each result.
5. **Use a Queue**: Wait on a **queue.Queue** for results to arrive.

You can learn more about how to wait for a result from a new thread in this tutorial:

- Wait for a Result from a Thread in Python (https://superfastpython.com/thread-wait-for-result)

How to Change the Context Switch Interval?

In a multitasking operating system, a context switch involves suspending the execution of one thread and resuming the execution of another thread.

Not all threads are able to run at the same time. Instead the operating system simulates multitasking by allowing each thread to run for a short amount of time before pausing the execution of the thread and storing its state and switching to another thread.

The process of suspending one thread and reanimating a suspended thread is called a context switch.

The switch interval in Python specifies how long the Python interpreter will allow a Python thread to run before it is forced to be made available for a context switch.

Python 3 provides an API that allows you to set the context switch interval as a time interval in seconds.

This can be checked and configured via the **sys.getswitchinterval() (https://docs.python.org/3/library/sys.html#sys.getswitchinterval)** and **sys.setswitchinterval() (https://docs.python.org/3/library/sys.html#sys.setswitchinterval)** functions.

The default value for the switch interval is 5 milliseconds set as 0.005 seconds.

This can be checked by calling the **sys.getswitchinterval()** function.

For example:

```
...
# get the switch interval
value = sys.getswitchinterval()
```

It can be changed to a new value in seconds via the **sys.setswitchinterval()** function.

For example:

```
...
# set the switch interval
sys.setswitchinterval(1.0)
```

There are two main reasons to change the value, they are:

- Allow threads to run longer before a switch to increase performance, e.g. increase the value.
- Allow threads less running time before a switch to expose race conditions, e.g. decrease the value.

Increasing the switch interval may help improve performance of an application by allowing more non-blocking code to be executed before signaling an availability for a switch.

Decreasing the interval may help to expose race conditions related to the concurrent modification of variables or via the coordination between threads.

You can learn more about context switch interval in this tutorial:

- Context Switch Interval In Python (https://superfastpython.com/context-switch-interval-in-python)

How to Sleep a Thread?

The **time.sleep()** function (https://docs.python.org/3/library/time.html#time.sleep) will cause the calling thread to block for the specified number of seconds.

For example:

```
...
# sleep for 5 seconds
time.sleep(5)
```

The **time.sleep()** function takes a floating point number of seconds to sleep.

If a fraction less than zero is provided, then this indicates the number of milliseconds for the calling thread to block.

Recall that there are 1,000 milliseconds in one second. Therefore a sleep value of 0.1 will allow the calling thread to sleep for 1/10th of a second or 100 milliseconds.

```
...
# sleep for 100 milliseconds
time.sleep(0.1)
```

We may call the **time.sleep()** function from the main thread, which is the default thread in your progress. We may also call the **time.sleep()** function from a new thread used to execute a custom function.

The calling thread may block for less than the time specified, if the process is interrupted.

This may happen if the user presses Ctrl-C to send a SIGINT (signal interrupt) to the process that owns the thread, which by default will terminate the process.

The calling thread will block for at least the specified number of seconds, but may block for longer.

This is because the underlying operating system decides what threads should run and when to run them. It may decide to let the thread block for a little longer before waking it up and allowing it to resume execution (e.g. a fraction of a second longer).

The **time.sleep()** function calls a[sleep system function (https://en.wikipedia.org/wiki/Sleep_(system_call))](https://en.wikipedia.org/wiki/Sleep_(system_call)) to block the calling thread. e.g. the sleep system call. This function call itself may have more or less precision than you require. For example, although you specify a sleep of one or a few milliseconds, the underlying system call may have a precision limited at tens of milliseconds.

This is a capability provided by the underlying operating system and may be implemented differently on different operating systems, such as Windows, Linux and MacOS.

There are many reasons why we may want a thread to sleep.

For example:

- Add a delay.
- Slow down execution of a loop.
- Allow other threads to run.
- Force race conditions during debugging.

You can learn more about sleeping a thread in the following tutorial:

- [How to Sleep a Thread in Python (https://superfastpython.com/thread-sleep-in-python)](https://superfastpython.com/thread-sleep-in-python)

Are Random Numbers Thread-Safe?

At the time of writing (Python 3.10), the **random** [module (https://docs.python.org/3/library/random.html)](https://docs.python.org/3/library/random.html) in Python is thread-safe, mostly.

Specifically, the generation of all random numbers relies on the **random.random()** function, which calls down to C-code and is thread-safe. You may recall that the **random.random()** function returns a random floating point value between 0 and 1.

Both the functions on the "**random**" module are thread safe as are the methods on an instance of the **random.Random** class.

For example:

```
...
# generate a thread-safe random number
value = random.random()
```

This means that a multithreaded program may call module functions in order to generate random numbers with a seed and sequence of random numbers shared between threads, or have a single new random.Random instance shared between threads.

The single function on the random module and the **random.Random** class that is not thread-safe is the **gauss()** function, specifically **random.gauss()** and **random.Random.gauss()**.

This function will generate a random number drawn from a Gaussian distribution (e.g. normal distribution) with a given mean and standard deviation.

For example:

```
...
# generate a thread-unsafe gaussian random number
value = random.gauss(0.0, 1.0)
```

This function is not thread-safe.

The algorithm used for generating Gaussian random numbers will generate two numbers each time the function is called. As such, it is possible for two threads to call this function at the same time and suffer a race condition, receiving the same random Gaussian values in return.

As such the **random.normalvariate()** function should be used instead in a multithreaded application.

For example:

```
...
# generate a random gaussian in a thread-safe manner
value = random.normalvariate(0.0, 1.0)
```

Alternatively, a mutual exclusion (mutex) lock may be used to protect the **random.gauss()** function via the **threading.Lock** class.

For example, a lock may be created once:

```
...
lock = threading.Lock()
```

Then used to protect each call to the **random.gauss()** function:

```
...
# acquire the mutex
with lock:
    # generate a random gaussian
    value = random.gauss(0.0, 1.0)
```

This will ensure that only one thread can execute the **random.gauss()** function at a time.

You can learn more about how to generate thread-safe random numbers in the tutorial:

- Thread-Safe Random Numbers in Python (https://superfastpython.com/random-thread-safe)

Are Lists, Dics, and Sets Thread-Safe?

Yes.

An atomic operation (https://en.wikipedia.org/wiki/Linearizability) is one or a sequence of code instructions that are completed without interruption.

A program may be interrupted for one of many reasons. In concurrent programming, a program may be interrupted via a context switch.

You may recall that the operating system controls what threads execute and when. A context switch refers to the operating system pausing the execution of a thread and storing its state, while unpausing another thread and restoring its state.

A thread cannot be context switched in the middle of an atomic operation.

This means these operations are thread-safe as we can expect them to be completed once started.

Most operations on Python's built-in data structures are atomic and therefore thread-safe.

> *"... it means that operations on shared variables of built-in data types (ints, lists, dicts, etc) that "look atomic" really are.*
>
> — **WHAT KINDS OF GLOBAL VALUE MUTATIONS ARE THREAD-SAFE? (HTTPS://DOCS.PYTHON.ORG/3/FAQ/LIBRARY.HTML#WHAT-KINDS-OF-GLOBAL-VALUE-MUTATION-ARE-THREAD-SAFE)**

You can learn more about atomic operations in this tutorial:

- Thread Atomic Operations in Python (https://superfastpython.com/thread-atomic-operations/)

Is Logging Thread-Safe?

Yes.

Logging is a way of tracking events within a program.

There are many types of events that may be logged within a program, ranging in different levels of severity, such as debugging and information to warnings, errors and critical events.

The **logging** module (https://docs.python.org/3/library/logging.html) provides infrastructure for logging within Python programs.

Logging is achieved by first configuring the log handler and then adding calls to the logging infrastructure at key points in the program.

For example, we can log to file by calling the **logging.basicConfig()** function and specifying the file name and path to log to (e.g. **application.log**), the level of logging to capture to the file (e.g. from **logging.DEBUG** to **logging.CRITICAL**).

```
...
# log everything to file
logging.basicConfig(filename='application.log', level=logging.DEBUG)
```

Events are logged via function calls based on the type of event performed, e.g. **logging.debug()** for debugging messages.

For example, we may add information messages to application code by calling **logging.info()** and passing in the string details of the event.

```
...
# log that data was loaded successfully
logger.info('All data was loaded successfully')
```

The logging module can be used directly from multiple threads.

The reason is because the logging module is thread-safe.

> *"The logging module is intended to be thread-safe without any special work needing to be done by its clients.*
>
> **— THREAD SAFETY, LOGGING — LOGGING FACILITY FOR PYTHON (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/LOGGING.HTML#THREAD-SAFETY).**

This is achieved using locks.

Internally, the logging module uses mutual exclusion (mutex) locks to ensure that logging handlers are protected from race conditions from multiple threads.

You can learn more about logging from threads in the tutorial:

- Thread-Safe Logging in Python (https://superfastpython.com/thread-safe-logging-in-python)

How Do You Use Thread Pools?

You can use thread pools in Python via the **concurrent.futures.ThreadPoolExecutor** class.

The **ThreadPoolExecutor** in Python provides a pool of reusable threads for executing ad hoc tasks.

You can submit tasks to the thread pool by calling the **submit()** function and passing in the name of the function you wish to execute on another thread.

Calling the **submit()** function will return a **Future** object that allows you to check on the status of the task and get the result from the task once it completes.

You can also submit tasks by calling the **map()** function and specifying the name of the function to execute and the iterable of items to which your function will be applied.

You can learn more about thread pools in Python via the tutorial:

- ThreadPoolExecutor in Python: The Complete Guide (https://superfastpython.com/threadpoolexecutor-in-python)

Do We Need to Have a Check for __main__?

No.

You do not need to have a check for **__main__** when using the threads in Python via the **threading.Thread** class.

You do need a check for **__main__** when using processes in Python via the **multiprocessing.Process** class.

Does Python Have Volatile Variables?

No.

Volatile variables (https://en.wikipedia.org/wiki/Volatile_(computer_programming)) in concurrency programming refer to variables whose values must be retrieved from main memory each time they are accessed.

This is the meaning of "*volatile*" used in modern compiled programming languages such as Java and C#.

In these languages, the virtual machine may maintain cached copies of variable values per thread or stack space.

Python does not have volatile variables.

The reason is because Python does not require volatile variables.

One aspect for this is because Python does not require variables to be defined before being used, as with compiled languages such as Java.

Another aspect is that the Python virtual machine will always access the value of a variable from main memory, e.g. from the heap.

Finally, accessing the value of a variable in Python is atomic. This means that it is thread safe and that the thread cannot be context switched in the middle of accessing the variable's value.

This means that we can access shared variables between threads and know we are using the current value of the variable rather than a cached version of the variable.

You can learn more about volatile variables in Python in the tutorial:

- Volatile Variables in Python (https://superfastpython.com/thread-volatile-variables-in-python/)

What is Thread Busy Waiting?

Busy waiting, also called spinning, refers to a thread that repeatedly checks a condition in a loop.

It is referred to as "*busy*" or "*spinning*" because the thread continues to execute the same code, such as an if-statement within a while-loop, achieving a wait by executing code (e.g. keeping busy).

Busy waiting can be compared to waiting by a blocking call, such as a sleep or a wait.

Busy waiting is typically undesirable in concurrent programming as the tight loop of checking a condition consumes CPU cycles unnecessarily, occupying a CPU core. As such, it is sometimes referred to as an anti-pattern of concurrent programming, a pattern to be avoided.

That being said, there are some occasions where a busy wait is a preferred solution, such as situations where there communication or signals between threads may be missed due to the unpredictable execution times or race conditions. In such cases, a busy wait can provide an alternative to other concurrency primitives.

Additionally, if the busy wait involves checking if a mutual exclusion lock is available, this is referred to as a spinlock.

We can perform busy waiting in Python with an if-statement within a while-loop.

The while-loop may loop forever and the if-statement will check the desired goal state and break the loop.

For example:

```
...
# busy wait
while True:
    # check for the goal state
    if goal_state():
        break
```

Here, we simplify the if-statement to a function call, but it could be any condition relevant to the program.

We can see that this tight loop will execute as fast as possible, checking the condition every iteration.

It is sometimes referred to as an anti-pattern because of so much calculation being dedicated to redundant checks of the same condition.

The computation used to perform the loop may be lessened some amount. Two approaches may include:

- Adding a sleep to the busy-wait loop, e.g. a call to **time.sleep()**.
- Adding a blocking wait, if a concurrency primitive like a lock or a condition is being used.

Adding a sleep, even of a fraction of a second, will allow the operating system to context switch to another thread of execution and make progress.

```
...
# busy wait
while True:
    # check condition
    if goal_state():
        break
    else:
        sleep(0.1)
```

This may be desirable for some applications, if a small gap in time between the condition being **True** and the thread checking the condition is acceptable.

Alternatively a blocking **wait()** call can be made on a shared concurrent primitive like a threading.Lock or a threading.Condition can be used. Again, a timeout can be set on the blocking call to allow the thread to periodically check the condition.

This approach can be used in a spinlock to reduce the computational burden waiting for a mutex **threading.Lock** to become available. It can also be used with a **threading.Condition**, if a race condition may mean that a notification from another thread may be missed

Like adding a sleep, this is only appropriate if the application can tolerate a small gap in time between the condition being **True** and the thread noticing the change.

```
...
# busy wait
while True:
    # acquire the condition
    with condition:
        # check condition
        if goal_state():
            break
        else:
            # block with a timeout
            condition.wait(timeout=0.1)
```

You can learn more about busy waiting in the tutorial:

- How to Use Busy Waiting in Python (https://superfastpython.com/thread-busy-waiting-in-python/)

What is a Thread Spinlock?

A spinlock (https://en.wikipedia.org/wiki/Spinlock) is a busy wait for a mutual exclusion (mutex) lock.

Busy waiting, also called spinning, refers to a thread that repeatedly checks a condition.

Busy waiting is typically undesirable in concurrent programming as the tight loop of checking a condition consumes CPU cycles unnecessarily, occupying a CPU core. As such, it is sometimes referred to as an anti-pattern of concurrent programming, a pattern to be avoided.

A spinlock is a type of lock where a thread must perform a busy wait if it is already locked.

Spinlocks are typically implemented at a low-level, such as in the underlying operating system. Nevertheless, we can implement a spinlock at a high-level in our program as a concurrency primitive.

A spinlock may be a desirable concurrency primitive that gives fine grained control over both how long a thread may wait for a lock (e.g. loop iterations or wait time) and what the thread is able to do while waiting for the lock.

This latter point of performing actions while waiting for a lock is the most desirable property of a spinlock, as this might include a suite of application specific tasks such as checking application state or logging status.

We can create a spinlock using a **threading.Lock** in Python.

This can be achieved by using a mutex lock and having a thread perform a busy wait look in order to acquire it.

The busy wait loop in a spinlock may look as follows:

```
...
# example of a busy wait loop for a spinlock
while True:
    # try and get the lock
    acquired = lock.acquire(blocking=False)
    # check if we got the lock
    if acquired:
        # stop waiting
        break
```

We can see that the busy loop will loop forever until the lock is acquired.

Each iteration of the loop, we attempt to acquire the lock without blocking. If acquired, the loop will then exit.

Later, the lock must be released.

If the critical section protected by the lock is short enough, it can be called directly from within the busy loop, after the lock is acquired.

We can contrast the above busy wait loop of the spinlock with the alternate of simply blocking until the lock becomes available.

```
...
# block until the lock can be acquired
lock.acquire()
```

In this case, the waiting thread is unable to perform any other action while waiting on the lock.

A limitation of the spinlock is the increased computational burden of executing a tight loop repeatedly.

The computational burden of the busy wait loop can be lessened by adding a blocking sleep, e.g. a call to **time.sleep()** for a fraction of a second.

For example:

```
...
# example of a busy wait loop for a spinlock with a sleep
while True:
    # try and get the lock
    acquired = lock.acquire(blocking=False)
    # check if we got the lock
    if acquired:
        # stop waiting
        break
    else:
        # sleep for a moment
        sleep(0.1)
```

Alternatively, the attempt to acquire the lock each loop iteration could use a short blocking wait with a timeout of a fraction of a second. The **acquire()** function on the **threading.Lock** takes a "**timeout**" argument that could be set to 100 milliseconds (e.g. 0.1 seconds).

```
...
# example of a busy wait loop for a spinlock with a wait
while True:
    # try and get the lock with a timeout
    acquired = lock.acquire(timeout=0.1)
    # check if we got the lock
    if acquired:
        # stop waiting
        break
```

Both approaches would reduce the number of iterations of the busy loop, causing most of the execution time spent sleeping or blocking. This would allow the operating system to context switch the thread and perhaps free-up a CPU core.

The downside of adding a sleep or a blocking wait, is that it may introduce a delay between the locking becoming available and the thread noticing this fact and acquiring it. The tolerance for such a delay will be application dependent.

A further improvement would be to give-up attempting to acquire the lock after a fixed time limit or number of iterations of the loop.

You can learn more about spinlocks in the tutorial:

- How to Use a Spinlock in Python (https://superfastpython.com/thread-spinlock-in-python/)

How Do You Create a Thread-Safe Counter?

A counter can be made thread-safe using a mutual exclusion (mutex) lock via the **threading.Lock** class.

First, an instance of a lock can be created.

For example:

```
...
# create a lock
lock = Lock()
```

Then each time the counter variable is accessed or updated, it can be protected by the lock.

This can be achieved by calling the **acquire()** function before accessing the counter variable and calling **release()** after work with the counter variable has completed.

For example:

```
...
# acquire the lock protecting the counter
lock.acquire()
# update the counter
counter += 1
# release the lock protecting the counter
lock.release()
```

A simpler approach to using the **threading.Lock** is to use the context manager which will release the lock automatically once the block is exited.

For example:

```
...
# acquire the lock protecting the counter
with lock:
    # update the counter
    counter += 1
```

We can create a class that implements this with an **increment()** function that increments the counter in a thread-safe manner.

The **ThreadSafeCounter** class is listed below.

```
# thread safe counter class
class ThreadSafeCounter():
    # constructor
    def __init__(self):
        # initialize counter
        self._counter = 0
        # initialize lock
        self._lock = Lock()

    # increment the counter
    def increment(self):
        with self._lock:
            self._counter += 1

    # get the counter value
    def value(self):
        with self._lock:
            return self._counter
```

You can learn more about how to create a thread-safe counter in the tutorial:

- Thread-Safe Counter in Python (https://superfastpython.com/thread-safe-counter-in-python)

Common Objections to Using Python Threads

Threads may not be the best solution for all concurrency problems in your program.

That being said, there may also be some misunderstandings that are preventing you from making full and best use of the capabilities of the threads in Python.

In this section, we review some of the common objections seen by developers when considering using the threads in their code.

What About the Global Interpreter Lock (GIL)?

The Global Interpreter Lock, or GIL for short, is a design decision with the reference Python interpreter.

It refers to the fact that the implementation of the Python interpreter makes use of a master lock that prevents more than one Python instruction executing at the same time.

This prevents more than one thread of execution within Python programs, specifically within each Python process, that is each instance of the Python interpreter.

The implementation of the GIL means that Python threads may be concurrent, but cannot run in parallel. Recall that concurrent means that more than one task can be in progress at the same time; parallel means more than one task actually executing at the same time. Parallel tasks are concurrent, concurrent tasks may or may not execute in parallel.

It is the reason behind the heuristic that Python threads should only be used for IO-bound tasks, and not CPU-bound tasks, as IO-bound tasks will wait in the operating system kernel for remote resources to respond (not executing Python instructions), allowing other Python threads to run and execute Python instructions.

Put another way, the GIL does not mean we cannot use threads in Python, only that some use cases for Python threads are viable or appropriate.

This design decision was made within the reference implementation of the Python interpreter (from Python.org), but may not impact other interpreters (such as PyPy, Iron Python, and Jython) that allow multiple Python instructions to be executed concurrently and in parallel.

Are Python Threads “Real Threads”?

Yes.

Python makes use of real system-level threads, also called kernel-level threads, a capability provided by modern operating systems like Windows, Linux, and MacOS.

Python threads are not software-level threads, sometimes called user-level threads or green threads (https://en.wikipedia.org/wiki/Green_threads).

Aren’t Python Threads Buggy?

No.

Python threads are not buggy.

Python threading is a first class capability of the Python platform and has been for a very long time.

Isn’t Python a Bad Choice for Concurrency?

Developers love python for many reasons, most commonly because it is easy to use and fast for development.

Python is commonly used for glue code, one-off scripts, but more and more for large-scale software systems.

If you are using Python and then you need concurrency, then you work with what you have. The question is moot.

If you need concurrency and you have not chosen a language, perhaps another language would be more appropriate, or perhaps not. Consider the full scope of functional and non-functional requirements (or user needs, wants, and desires) for your project and the capabilities of different development platforms.

Why Not Always Use Processes Instead of Threads?

Threads and Processes are quite different and choosing one over the other is intentional.

A Python program is a process that has a main thread. You can create many additional threads in a Python process. You can also fork or spawn many Python processes, each of which will have one thread, and may spawn additional threads.

More broadly, threads are lightweight and can share memory (data and variables) within a process, whereas processes are heavyweight and require more overhead and impose more limits on sharing memory (data and variables).

Typically, processes are used for CPU-bound tasks and threads are used for IO-bound tasks, and this is a good heuristic, but this does not have to be the case.

Perhaps using processes or multiprocessing instead of multithreading is a better fit for your specific problem. Perhaps try it and see.

You can learn more about multiprocessing in the tutorial:

- Multiprocessing in Python: The Complete Guide (https://superfastpython.com/multiprocessing-in-python/)

You can learn more about why you should not always use processes in the tutorial:

- Why Not Always Use Processes in Python (https://superfastpython.com/why-not-always-use-processes-in-python/)

Why Not Use AsyncIO?

AsyncIO can be an alternative to using threads.

AsyncIO is designed to support large numbers of IO operations, perhaps thousands to tens of thousands, all within a single Thread.

It requires an alternate programming paradigm, called reactive programming (https://en.wikipedia.org/wiki/Reactive_programming), which can be challenging for beginners.

Nevertheless, it may be a better alternative to using a thread pool for many applications.

Further Reading

Python Threading Books

This section lists my books on Python threading, designed to help you get started and get good, super fast.