Python Multiprocessing: The Complete Guide

JUNE 27, 2022 *by* JASON BROWNLEE *in* MULTIPROCESSING (HTTPS://SUPERFASTPYTHON.COM/CATEGORY/MULTIPROCESSING/)

The **Python multiprocessing** module allows you to create and manage new child processes in Python.

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

This guide provides a detailed and **comprehensive guide to multiprocessing in Python**, including how processes work, how to use processes in multiprocessor programming, concurrency primitives used with processes, common questions, and best practices.

This is a massive 24,000+ 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 Processes

So what are processes and why do we care?

What Are Processes

A process (https://en.wikipedia.org/wiki/Process_(computing)) refers to a computer program.

Every Python program is a process and has one default thread called the main thread used to execute your program instructions. Each process is, in fact, one instance of the Python interpreter that executes Python instructions (Python byte-ode), which is a slightly lower level than the code you type into your Python program.

Sometimes we may need to create new processes to run additional tasks concurrently.

Python provides real system-level processes via the **multiprocessing.Process** class in the **multiprocessing** module (https://docs.python.org/3/library/multiprocessing.html).

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.

The code in new child processes 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).

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.

Processes and threads are different.

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

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/3J74TRR), 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.

You can learn more about the differences between processes and threads in the tutorial:

- Thread vs Process in Python (/thread-vs-process)

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

Life-Cycle of a Process

A process in Python is represented as an instance of the **multiprocessing.Process** class.

Once a process is started, the Python runtime will interface with the underlying operating system and request that a new native process be created. The **multiprocessing.Process** instance then provides a Python-based reference to this underlying native process.

Each process 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 process.
- The difference between run and start.
- The difference between blocked and terminated

And so on.

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

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

1. New Child Process.
2. Running Process.
    1. Blocked Process (optional).
3. Terminated Process.

A new process is a process that has been constructed and configured by creating an instance of the **multiprocessing.Process** class.

A new process can transition to a running process by calling the **start()** method. This also creates and starts the main thread for the process that actually executes code in the process.

A running process may block in many ways if its main thread is blocked, 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 process will run again.

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

A process cannot terminate until:

- All non-daemon threads have terminated, including the main thread.
- All non-daemon child processes have terminated, including the main process.

You can learn more about the life-cycle of processes in the tutorial:

- Process Life-Cycle in Python (/process-life-cycle)

Next, let's take a closer look at the difference between parent and child processes.

Child vs Parent Process

Parent processes have one or more child processes, whereas child processes is created by a parent process.

Parent Process

A parent process is a process that is capable of starting child processes.

Typically, we would not refer to a process as a parent process until it has created one or more child processes. This is the commonly accepted definition.

- **Parent Process**: Has one or more child processes. May have a parent process, e.g. may also be a child process.

Recall, a process is an instance of a computer program. In Python, a process is an instance of the Python interpreter that executes Python code.

In Python, the first process created when we run our program is called the 'MainProcess'. It is also a parent process and may be called the main parent process.

The main process will create the first child process or processes.

A child process may also become a parent process if it in turn creates and starts a child process.

Child Process

A child process is a process that was created by another process.

A child process may also be called a subprocess, as it was created by another process rather than by the operating system directly. That being said, the creation and management of all processes is handled by the underlying operating system.

- **Child Process**: Has a parent process. May have its own child processes, e.g. may also be a parent.

The process that creates a child process is called a parent process. A child process only ever has one parent process.

There are three main techniques used to create a child process, referred to as process start methods.

They are: fork, spawn, and fork server.

Depending on the technique used to start the child, the child process may or may not inherit properties of the parent process. For example, a forked process may inherit a copy of the global variables from the parent process.

A child process may become orphaned if the parent process that created it is terminated.

You can learn more about the differences between child and parent processes in the tutorial:

- Parent Process vs Child Process in Python (/parent-process-vs-child-process-in-python)

Now that we are familiar with child and parent processes, let's look at how to run code in new processes.

Run a Function in a Process

Python functions can be executed in a separate process using the **multiprocessing.Process** class.

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

How to Run a Function In a Process

To run a function in another process:

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

First, we must create a new instance of the **multiprocessing.Process** class and specify the function we wish to execute in a new process via the "**target**" argument.

```
...
# create a process
process = multiprocessing.Process(target=task)
```

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

```
...
# create a process
process = multiprocessing.Process(target=task, args=(arg1, arg2))
```

We can then start executing the process by calling the **start()** function.

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

```
...
# run the new process
process.start()
```

A new instance of the Python interrupter will be created and a new thread within the new process will be created to execute our target function.

And that's all there is to it.

We do not have control over when the process 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 of executing a function in a new process.

Example of Running a Function in a Process

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

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 process')
```

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

```
...
# create a process
process = Process(target=task)
```

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

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

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

We can explicitly wait for the new process to finish executing by calling the **join()** function.

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

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

```
```

```
# SuperFastPython.com
# example of running a function in another process
from time import sleep
from multiprocessing import Process

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

# entry point
if __name__ == '__main__':
    # create a process
    process = Process(target=task)
    # run the process
    process.start()
    # wait for the process to finish
    print('Waiting for the process...')
    process.join()
```

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

At some point a new instance of the Python interpreter is created that has a new thread which will execute our target function.

The main thread of our initial process then prints a message waiting for the new process to complete, then calls the **join()** function to explicitly block and wait for the new process to terminate.

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

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

Next, let's look at how we might run a function with arguments in a new process.

Example of Running a Function in a Process With Arguments

We can execute functions in another process that takes 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)
```

Next, we can update the call to the **multiprocessing.Process** constructor to specify the two arguments in the order that our **task()** function expects them as a tuple via the "**args**" argument.

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

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

```
# SuperFastPython.com
# example of running a function with arguments in another process
from time import sleep
from multiprocessing import Process

# 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)

# entry point
if __name__ == '__main__':
    # create a process
    process = Process(target=task, args=(1.5, 'New message from another process'))
    # run the process
    process.start()
    # wait for the process to finish
    print('Waiting for the process...')
    process.join()
```

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

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

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

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

- Run a Function in a Child Process (https://superfastpython.com/run-function-in-new-process/)

Next let's look at how we might run a function in a child process by extending the **multiprocessing.Process** class.

Confused by the multiprocessing module API?
Download my FREE PDF cheat sheet (https://marvelous-writer-6152.ck.page/4be43fa1f0)

Extend the Process Class

We can also execute functions in a child process by extending the **multiprocessing.Process** class and overriding the **run()** function.

In this section we will look at some examples of extending the **multiprocessing.Process** class.

How to Extend the Process Class

The **multiprocessing.Process** class can be extended to run code in another process.

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

For example:

```
# custom process class
class CustomProcess(multiprocessing.Process):
    # ...
```

Then the **run()** function of the **multiprocessing.Process** class must be overridden to contain the code that you wish to execute in another process.

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 such as instance variables (attributes).

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

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

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

Example of Extending the Process Class

First, we can define a class that extends the **multiprocessing.Process** class.

We will name the class something arbitrary such as "**CustomProcess**".

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

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 process')
```

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

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

```
...
# create the process
process = CustomProcess()
# start the process
process.start()
```

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

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

Tying this together, the complete example of executing code in another process by extending the **multiprocessing.Process** class is listed below.

```
# SuperFastPython.com
# example of extending the Process class
from time import sleep
from multiprocessing import Process

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

# entry point
if __name__ == '__main__':
    # create the process
    process = CustomProcess()
    # start the process
    process.start()
    # wait for the process to finish
    print('Waiting for the process to finish')
    process.join()
```

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

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

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

You can learn more about extending the **multiprocessing.Process** class in this tutorial:

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

Next, let's look at how we might return values from a child process.

Example of Extending the Process Class and Returning Values

Instance variable attributes can be shared between processes via the **multiprocessing.Value** and **multiprocessing.Array** classes.

These classes explicitly define data attributes designed to be shared between processes in a process-safe manner.

Shared variables mean that changes made in one process are always propagated and made available to other processes.

An instance of the **multiprocessing.Value** can be defined in the constructor of a custom class as a shared instance variable.

The constructor of the **multiprocessing.Value** class (https://docs.python.org/3/library/multiprocessing.html#multiprocessing.Value) requires that we specify the data type and an initial value.

The data type can be specified using ctype "type" or a typecode.

We can define an instance attribute as an instance of the **multiprocessing.Value** which will automatically and correctly be shared between processes.

We can update the above example to use a **multiprocessing.Value** directly.

Firstly, we must update the constructor of the **CustomProcess** class to initialize the **multiprocessing.Value** instance. We will define it to be an integer with an initial value of zero.

```
...
# initialize integer attribute
self.data = Value('i', 0)
```

A class constructor with this addition is listed below.

```
# override the constructor
def __init__(self):
    # execute the base constructor
    Process.__init__(self)
    # initialize integer attribute
    self.data = Value('i', 0)
```

We can then update the **run()** method to change the "**value**" attribute on the "**data**" instance variable and then to report this value via a print statement.

```
...
# store the data variable
self.data.value = 99
# report stored value
print(f'Child stored: {self.data.value}')
```

The updated **run()** function with this change is listed below.

```
# override the run function
def run(self):
    # block for a moment
    sleep(1)
    # store the data variable
    self.data.value = 99
    # report stored value
    print(f'Child stored: {self.data.value}')
```

Tying this together, the updated **CustomProcess** class is listed below.

```
# custom process class
class CustomProcess(Process):
    # override the constructor
    def __init__(self):
        # execute the base constructor
        Process.__init__(self)
        # initialize integer attribute
        self.data = Value('i', 0)

    # override the run function
    def run(self):
        # block for a moment
        sleep(1)
        # store the data variable
        self.data.value = 99
        # report stored value
        print(f'Child stored: {self.data.value}')
```

Finally, in the parent process, we can update the print statement to correctly access the value of the shared instance variable attribute.

```
...
# report the process attribute
print(f'Parent got: {process.data.value}')
```

Tying this together, the complete example is listed below.

```
```

```
# SuperFastPython.com
# example of extending the Process class and adding shared attributes
from time import sleep
from multiprocessing import Process
from multiprocessing import Value

# custom process class
class CustomProcess(Process):
    # override the constructor
    def __init__(self):
        # execute the base constructor
        Process.__init__(self)
        # initialize integer attribute
        self.data = Value('i', 0)

    # override the run function
    def run(self):
        # block for a moment
        sleep(1)
        # store the data variable
        self.data.value = 99
        # report stored value
        print(f'Child stored: {self.data.value}')

# entry point
if __name__ == '__main__':
    # create the process
    process = CustomProcess()
    # start the process
    process.start()
    # wait for the process to finish
    print('Waiting for the child process to finish')
    # block until child process is terminated
    process.join()
    # report the process attribute
    print(f'Parent got: {process.data.value}')
```

Running the example first creates an instance of the custom class then starts the child process.

The constructor initializes the instance variable to be a **multiprocessing.Value** instance with an initial value of zero.

The parent process blocks until the child process terminates.

The child process blocks, then changes the value of the instance variable and reports the change. The change to the instance variable is propagated back to the parent process.

The child process terminates and the parent process continues on. It reports the value of the instance variable which correctly reflects the change made by the child process.

This demonstrates how to share instance variable attributes among parents via the **multiprocessing.Value** class.

```
Waiting for the child process to finish
Child stored: 99
Parent got: 99
```

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

- Shared Process Class Attributes in Python (https://superfastpython.com/share-process-attributes/)

Next, let's take a closer look at process start methods.

Process Start Methods

In multiprocessing, we may need to change the technique used to start child processes.

This is called the start method.

What is a Start Method

A start method is the technique used to start child processes in Python.

There are three start methods, they are:

- **spawn**: start a new Python process.
- **fork**: copy a Python process from an existing process.
- **forkserver**: new process from which future forked processes will be copied.

Each platform has a default start method.

The following lists the major platforms and the default start methods.

- **Windows** (*win32*): spawn
- **macOS** (*darwin*): spawn
- **Linux** (*unix*): fork

Not all platforms support all start methods.

The following lists the major platforms and the start methods that are supported.

- **Windows** (*win32*): spawn
- **macOS** (*darwin*): spawn, fork, forkserver.
- **Linux** (*unix*): spawn, fork, forkserver.

How to Change The Start Method

The **multiprocessing** module provides functions for getting and setting the start method for creating child processes.

The list of supported start methods can be retrieved via the **multiprocessing.get_all_start_methods()** function (https://docs.python.org/3/library/multiprocessing.html#multiprocessing.get_all_start_methods).

The function returns a list of string values, each representing a supported start method.

For example:

```
...
# get supported start methods
methods = multiprocessing.get_all_start_methods()
```

The current start method can be retrieved via the **multiprocessing.get_start_method()** function (https://docs.python.org/3/library/multiprocessing.html#multiprocessing.get_start_method).

The function returns a string value for the currently configured start method.

For example:

```
...
# get the current start method
method = multiprocessing.get_start_method()
```

The start method can be set via the **multiprocessing.set_start_method()** function (https://docs.python.org/3/library/multiprocessing.html#multiprocessing.set_start_method).

The function takes a string argument indicating the start method to use.

This must be one of the methods returned from the **multiprocessing.get_all_start_methods()** for your platform.

For example:

```
...
# set the start method
multiprocessing.set_start_method('spawn')
```

It is a best practice, and required on most platforms that the start method be set first, prior to any other code, and to be done so within a **if __name__ == '__main__'** check called a protected entry point or top-level code environment (https://docs.python.org/3/library/__main__.html).

For example:

```
...
# protect the entry point
if __name__ == '__main__':
    # set the start method
    multiprocessing.set_start_method('spawn')
```

If the start method is not set within a protected entry point, it is possible to get a RuntimeError such as:

```
RuntimeError: context has already been set
```

It is also a good practice and required on some platforms that the start method only be set once.

In summary, the rules for setting the start method are as follows:

- Set the start method first prior to all other code.

- Set the start method only once in a program.
- Set the start method within a protected entry point.

How to Set Start Method Via Context

A multiprocessing context configured with a given start method can be retrieved via the **multiprocessing.get_context()** function (https://docs.python.org/3/library/multiprocessing.html#multiprocessing.get_context).

This function takes the name of the start method as an argument, then returns a multiprocessing context that can be used to create new child processes.

For example:

```
...
# get a context configured with a start method
context = multiprocessing.get_context('fork')
```

The context can then be used to create a child process, for example:

```
...
# create a child process via a context
process = context.Process(...)
```

It may also be possible to force the start method.

This can be achieved via the "**force**" argument provided on the **set_start_method()** implementation in the **DefaultContext**, although not documented.

For example:

```
...
# set the start method
context.set_start_method('spawn', force=True)
```

You can learn more about setting the process start method in the tutorial:

- Multiprocessing Start Methods (/multiprocessing-start-method)

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

Process Instance Attributes

An instance of the **multiprocessing.Process** class provides a handle of a new instance of the Python interpreter.

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

Let's look at some examples.

Query Process Name

Each process has a name.

The parent process has the name "MainProcess".

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

We can access the name of a process via the **multiprocessing.Process.name** attribute (https://docs.python.org/3/library/multiprocessing.html#multiprocessing.Process.name), for example:

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

The example below creates an instance of the **multiprocessing.Process** class and reports the default name of the process.

```
# SuperFastPython.com
# example of accessing the child process name
from multiprocessing import Process
# entry point
if __name__ == '__main__':
    # create the process
    process = Process()
    # report the process name
    print(process.name)
```

Running the example creates a child process and reports the default name assigned to the process.

```
Process-1
```

You can learn more about configuring the process name in the tutorial:

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

Query Process Daemon

A process may be a daemon.

Daemon process is the name given to the background process. By default, processes are non-daemon processes because they inherit the daemon value from the parent process, which is set **False** for the *MainProcess*.

A Python parent process will only exit when all non-daemon processes have finished exiting. For example, the *MainProcess* is a non-daemon process. This means that the daemon process 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 process is a daemon process via the **multiprocessing.Process.daemon** attribute (https://docs.python.org/3/library/multiprocessing.html#multiprocessing.Process.daemon).

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

The example creates an instance of the **multiprocessing.Process** class and reports whether the process is a daemon or not.

```
# SuperFastPython.com
# example of assessing whether a process is a daemon
from multiprocessing import Process
# entry point
if __name__ == '__main__':
    # create the process
    process = Process()
    # report the daemon attribute
    print(process.daemon)
```

Running the example reports that the process is not a daemon process, the default for new child processes.

```
False
```

You can learn more about configuring daemon processes in the tutorial:

- Daemon Process in Python (https://superfastpython.com/daemon-process-in-python/)

Query Process PID

Each process has a unique process identifier, called the PID, assigned by the operating system.

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

The process identifier can be accessed via the **multiprocessing.Process.pid** property (https://docs.python.org/3/library/multiprocessing.html#multiprocessing.Process.pid) and is assigned after the process has been started.

```
...
# report the pid
print(process.pid)
```

The example below creates an instance of a **multiprocessing.Process** and reports the assigned PID.

```
# SuperFastPython.com
# example of reporting the native process identifier
from multiprocessing import Process
# entry point
if __name__ == '__main__':
    # create the process
    process = Process()
    # report the process identifier
    print(process.pid)
    # start the process
    process.start()
    # report the process identifier
    print(process.pid)
```

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

The process is then started and the assigned PID is reported.

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

```
None
16302
```

Query Process Alive

A process instance can be alive or dead.

An alive process means that the **run()** method of the **multiprocessing.Process** instance is currently executing.

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

We can check if a process is alive via the **multiprocessing.Process.is_alive()** method (https://docs.python.org/3/library/multiprocessing.html#multiprocessing.Process.is_alive).

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

The example below creates a **multiprocessing.Process** instance then checks whether it is alive.

```
# SuperFastPython.com
# example of assessing whether a process is alive
from multiprocessing import Process
# entry point
if __name__ == '__main__':
    # create the process
    process = Process()
    # report the process is alive
    print(process.is_alive())
```

Running the example creates a new **multiprocessing.Process** instance then reports that the process is not alive.

```
False
```

Query Process Exit Code

A child process will have an exit code once it has terminated.

An exit code provides an indication of whether processes completed successfully or not, and if not, the type of error that occurred that caused the termination.

Common exit codes include:

- 0: Normal (exit success)
- 1: Error (exit failure)

We can write the exit code for a child process via the **multiprocessing.Process.exitcode** attribute (https://docs.python.org/3/library/multiprocessing.html#multiprocessing.Process.exitcode).

```
...
# report the process exit code
print(process.exitcode)
```

A process will not have an exit code until the process is terminated. This means, checking the exit code of a process before it has started or while it is running will return a None value.

We can demonstrate this with a worked example that executes a custom target function that blocks for one second.

First, we create a new process instance to execute our **task()** function, report the exit code, then start the process and report the exit code while the process is running.

```
...
# create the process
process = Process(target=task)
# report the exit status
print(process.exitcode)
# start the process
process.start()
# report the exit status
print(process.exitcode)
```

We can then block until the child process has terminated, then report the exit code.

```
...
# wait for the process to finish
process.join()
# report the exit status
print(process.exitcode)
```

Tying this together, the complete example is listed below.

```
# SuperFastPython.com
# example of checking the exit status of a child process
from time import sleep
from multiprocessing import Process

# function to execute in a new process
def task():
    sleep(1)

# entry point
if __name__ == '__main__':
    # create the process
    process = Process(target=task)
    # report the exit status
    print(process.exitcode)
    # start the process
    process.start()
    # report the exit status
    print(process.exitcode)
    # wait for the process to finish
    process.join()
    # report the exit status
    print(process.exitcode)
```

Running the example first creates a new process to execute our custom task function.

The status code of the child process is reported, which is **None** as expected. The process is started and blocked for one second. While running, the exit code is reported again, which again is **None** as expected.

The parent process then blocks until the child process terminates. The exit code is reported, and in this case we can see that a value of zero is reported indicating a normal or successful exit.

```
None
None
0
```

The exit code for a process can be set by calling **sys.exit() (https://docs.python.org/3/library/sys.html#sys.exit)** and specifying an integer value, such as 1 for a non-successful exit.

You can learn more about how to set exit codes in the tutorial:

- Exit Codes in Python (/exit-codes-in-python)

Next, let's look at how we might configure new child processes.

Configure Processes

Instances of the **multiprocessing.Process** class can be configured.

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

Let's take a closer look at each.

How to Configure Process Name

Processes can be assigned custom names.

The name of a process can be set via the "**name**" argument in the **multiprocessing.Process** constructor.

For example:

```
...
# create a process with a custom name
process = Process(name='MyProcess')
```

The example below demonstrates how to set the name of a process in the process class constructor.

```
# SuperFastPython.com
# example of setting the process name in the constructor
from multiprocessing import Process
# entry point
if __name__ == '__main__':
    # create a process with a custom name
    process = Process(name='MyProcess')
    # report process name
    print(process.name)
```

Running the example creates the child process with the custom name then reports the name of the process.

```
MyProcess
```

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

For example:

```
...
# set the name
process.name = 'MyProcess'
```

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

```
# SuperFastPython.com
# example of setting the process name via the property
from multiprocessing import Process
# entry point
if __name__ == '__main__':
    # create a process
    process = Process()
    # set the name
    process.name = 'MyProcess'
    # report process name
    print(process.name)
```

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

```
MyProcess
```

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

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

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

How to Configure a Daemon Process

Processes can be configured to be "*daemon*" or "*daemonic*", that is, they can be configured as background processes.

A parent Python process can only exit once all non-daemon child processes have exited.

This means that daemon child processes can run in the background and do not prevent a Python program from exiting when the "*main parts of the program*" have finished.

A process can be configured to be a daemon by setting the "**daemon**" argument to **True** in the **multiprocessing.Process** constructor.

For example:

```
```

```
...
# create a daemon process
process = Process(daemon=True)
```

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

```
# SuperFastPython.com
# example of setting a process to be a daemon via the constructor
from multiprocessing import Process
# entry point
if __name__ == '__main__':
    # create a daemon process
    process = Process(daemon=True)
    # report if the process is a daemon
    print(process.daemon)
```

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

```
True
```

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

For example:

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

The example below creates a new **multiprocessing.Process** instance then configures it to be a daemon process via the property.

```
# SuperFastPython.com
# example of setting a process to be a daemon via the property
from multiprocessing import Process
# entry point
if __name__ == '__main__':
    # create a process
    process = Process()
    # configure the process to be a daemon
    process.daemon = True
    # report if the process is a daemon
    print(process.daemon)
```

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

```
True
```

You can learn more about daemon processes in this tutorial:

- Daemon Process in Python (https://superfastpython.com/daemon-process-in-python/)

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

Main Process

The main process is the parent process that executes your program.

In this section we will take a closer look at the main process in Python.

What is the Main Process?

The main process in Python is the process started when you run your Python program.

Recall that a process in Python is an instance of the Python interpreter. We can create and run new child processes via the **multiprocessing.Process** class. (https://docs.python.org/3/library/multiprocessing.html#multiprocessing.Process)

What makes the main process unique is that it is the first process created when you run your program and the main thread of the process executes the entry point of your program.

As such the main process does not have a parent process.

The main process also has a distinct name, specifically "*MainProcess*".

How Can the Main Process Be Identified

There are a number of ways that the main process can be identified.

They are:

- The main process has a distinct name of "*MainProcess*".
- The main process does not have a parent process.
- The main process is an instance of the **multiprocessing.process._MainProcess** class (https://github.com/python/cpython/blob/3.10/Lib/multiprocessing/process.py#L391).

How to Get the Main Process?

We can get a multiprocessing.Process instance for the main process.

There are a number of ways that we can do this, depending on where exactly we are in the code.

For example, if we are running code in the main process, we can get a multiprocessing.Process instance for the main process via the **multiprocessing.current_process()** function (https://docs.python.org/3/library/multiprocessing.html#multiprocessing.current_process).

For example:

```
# SuperFastPython.com
# example of getting the main process from the main process
from multiprocessing import current_process
# get the process instance
process = current_process()
# report details of the main process
print(process)
```

Running the example gets the process instance for the main process.

```
<_MainProcess name='MainProcess' parent=None started>
```

If we are in a child process of the main process then we can get the main process via the **multiprocessing.parent_process()** function.

For example:

```
# SuperFastPython.com
# example of getting the main process from a child of the main process
from multiprocessing import parent_process
from multiprocessing import Process

# function executed in a child process
def task():
    # get the parent process instance
    process = parent_process()
    # report details of the main process
    print(process)

# entry point
if __name__ == '__main__':
    # create a new process
    process = Process(target=task)
    # start the new process
    process.start()
    # wait for the new process to terminate
    process.join()
```

Running the example starts a new child process to execute a custom **task()** function.

The child process gets the parent process instance which is the main process and reports its details.

```
<_ParentProcess name='MainProcess' parent=None unknown>
```

What is the Name of the Main Process?

The main process is assigned a distinct name when it is created.

The name of the parent process is '*MainProcess*'.

It is distinct from the default name of child processes, that are named '*Process-%d*', where *%d* is an integer starting from 1 to indicate the child number created by the parent process, e.g. *Process-1*.

Nevertheless, process names can be changed any time and should not be relied upon to identify processes uniquely.

We can confirm the name of the main process by getting the parent process instance via the **multiprocessing.current_process()** function and getting the "**name**" attribute.

For example:

```
# SuperFastPython.com
# example of reporting the name of the main process
from multiprocessing import current_process
# get the main process
process = current_process()
# report the name of the main process
print(process.name)
```

Running the example gets the **multiprocessing.Process** instance for the main process, then reports the name of the process.

```
MainProcess
```

You can learn more about the main process in the tutorial:

- Main Process in Python (/main-process-in-python)

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

Process Utilities

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

These utilities are provided as multiprocessing module functions.

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

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

Active Child Processes

We can get a list of all active child processes for a parent process.

This can be achieved via the **multiprocessing.active_children()** function (https://docs.python.org/3/library/multiprocessing.html#multiprocessing.active_children) that returns a list of all child processes that are currently running.

```
...
# get a list of all active child processes
children = active_children()
```

We can demonstrate this with a short example.

```
# SuperFastPython.com
# list all active child processes
from multiprocessing import active_children
# get a list of all active child processes
children = active_children()
# report a count of active children
print(f'Active Children Count: {len(children)}')
# report each in turn
for child in children:
    print(child)
```

Running the example reports the number of active child processes.

In this case, there are no active children processes.

```
Active Children Count: 0
```

We can update the example so that we first start a number of children processes, have the children processes block for a moment, then get the list of active children.

This can be achieved by first defining a function to execute in a new process that blocks for a moment.

The **task()** function below implements this.

```
# function to execute in a new process
def task():
    # block for a moment
    sleep(1)
```

Next, we can create a number of processes configured to run our **task()** function, then start them.

```
...
# create a number of child processes
processes = [Process(target=task) for _ in range(5)]
# start the child processes
for process in processes:
    process.start()
```

We can then report the active child processes as before.

```
...
# get a list of all active child processes
children = active_children()
# report a count of active children
print(f'Active Children Count: {len(children)}')
# report each in turn
for child in children:
    print(child)
```

Tying this together, the complete example is listed below.

```
# SuperFastPython.com
# list all active child processes
from time import sleep
from multiprocessing import active_children
from multiprocessing import Process

# function to execute in a new process
def task():
    # block for a moment
    sleep(1)

# entry point
if __name__ == '__main__':
    # create a number of child processes
    processes = [Process(target=task) for _ in range(5)]
    # start the child processes
    for process in processes:
        process.start()
    # get a list of all active child processes
    children = active_children()
    # report a count of active children
    print(f'Active Children Count: {len(children)}')
    # report each in turn
    for child in children:
        print(child)
```

Running the example first creates five processes to run our **task()** function, then starts them.

A list of all active child processes is then retrieved.

The count is reported, which is shown as five, as we expect, then the details of each process are then reported.

This highlights how we can access all active child processes from a parent process.

```
Active Children Count: 5
<Process name='Process-3' pid=16853 parent=16849 started>
<Process name='Process-4' pid=16854 parent=16849 started>
<Process name='Process-2' pid=16852 parent=16849 started>
<Process name='Process-1' pid=16851 parent=16849 started>
<Process name='Process-5' pid=16855 parent=16849 started>
```

Get The Number of CPU Cores

We may want to know the number of CPU cores available.

This can be determined via the **multiprocessing.cpu_count()** function (https://docs.python.org/3/library/multiprocessing.html#multiprocessing.cpu_count).

The function returns an integer that indicates the number of logical CPU cores available in the system running the Python program.

Recall that a central processing unit or CPU executes program instructions. A CPU may have one or more physical CPU cores for executing code in parallel. Modern CPU cores may make use of hyperthreading (https://en.wikipedia.org/wiki/Hyper-threading), allowing each physical CPU core to operate like two (or more) logical CPU cores.

Therefore, a computer system with a CPU with four physical cores may report eight logical CPU cores via the **multiprocessing.cpu_count()** function function.

It may be useful to know the number of CPU cores available to configure a thread pool or the number of processes to create to execute tasks in parallel.

The example below demonstrates how to use the number of CPU cores.

```
# SuperFastPython.com
# example of reporting the number of logical cpu cores
from multiprocessing import cpu_count
# get the number of cpu cores
num_cores = cpu_count()
# report details
print(num_cores)
```

Running the example reports the number of logical CPU cores available in the system.

In this case, my system provides eight logical CPU cores, your result may differ.

```
8
```

You can learn more about getting the number of CPU Cores in the tutorial:

- Number of CPUs in Python (/number-of-cpus-python)

The Current Process

We can get a **multiprocessing.Process** instance for the process running the current code.

This can be achieved via the **multiprocessing.current_process()** function (https://docs.python.org/3/library/multiprocessing.html#multiprocessing.current_process) that returns a **multiprocessing.Process** instance.

```
...
# get the current process
thread = current_process()
```

We can use this function to access the **multiprocessing.Process** for the *MainProcess*.

This can be demonstrated with a short example, listed below.

```
# SuperFastPython.com
# example of executing the current process
from multiprocessing import current_process
# get the current process
process = current_process()
# report details
print(process)
```

Running the example gets the process instance for the currently running process.

The details are then reported, showing that we have accessed the main process that has no parent process.

```
<_MainProcess name='MainProcess' parent=None started>
```

We can use this function within a child process, allowing the child process to access a process instance for itself.

First, we can define a custom function that gets the current process and reports its details.

The **task()** function below implements this.

```
# function executed in a new process
def task():
    # get the current process
    process = current_process()
    # report details
    print(process)
```

Next, we can create a new process to execute our custom **task()** function. Then start the process and wait for it to terminate.

```
...
# create a new process
process = Process(target=task)
# start the new process
process.start()
# wait for the child process to terminate
process.join()
```

Tying this together, the complete example is listed below.

```
# SuperFastPython.com
# example of executing the current process within a child process
from multiprocessing import Process
from multiprocessing import current_process

# function executed in a new process
def task():
    # get the current process
    process = current_process()
    # report details
    print(process)

# entry point
if __name__ == '__main__':
    # create a new process
    process = Process(target=task)
    # start the new process
    process.start()
    # wait for the child process to terminate
    process.join()
```

Running the example first creates a new process instance, then starts the process and waits for it to finish.

The new process then gets an instance of its own **multiprocessing.Process** instance and then reports the details.

The Parent Process

We may need to access the parent process for a current child process.

This can be achieved via the **multiprocessing.parent_process()** function. This will return a multiprocessing.Process instance for the parent of the current process.

The *MainProcess* does not have a parent, therefore attempting to get the parent of the MainProcess will return None.

We can demonstrate this with a worked example.

```
# SuperFastPython.com
# example of getting the parent process of the main process
from multiprocessing import parent_process
# get the the parent process
process = parent_process()
# report details
print(process)
```

Running the example attempts the **multiprocessing.Process** instance for the *MainProcess*.

The function returns None, as expected as the *MainProcess* does not have a parent process.

```
None
```

We can make the example more interesting by creating a new child process, then getting the parent of the child process, which will be the *MainProcess*.

This can be achieved by first defining a function to get the parent process and then report its details.

The *task()* function below implements this.

```
# function to execute in a new process
def task():
    # get the the parent process
    process = parent_process()
    # report details
    print(process)
```

Next, we can create a new process instance and configure it to execute our **task()** function. We can then start the new process and wait for it to terminate.

```

```

```
...
# create a new process
process = Process(target=task)
# start the new process
process.start()
# wait for the new process to terminate
process.join()
```

Tying this together, the complete example is listed below.

```
# SuperFastPython.com
# example of getting the parent process of a child process
from multiprocessing import parent_process
from multiprocessing import Process

# function to execute in a new process
def task():
    # get the the parent process
    process = parent_process()
    # report details
    print(process)

# entry point
if __name__ == '__main__':
    # create a new process
    process = Process(target=task)
    # start the new process
    process.start()
    # wait for the new process to terminate
    process.join()
```

Running the example first creates a new process configured to execute our task function, then starts the process and waits for it to terminate.

The new child process then gets a **multiprocessing.Process** instance for the parent process, then reports its details.

We can see that the parent of the new process is the MainProcess, as we expected.

```
<_ParentProcess name='MainProcess' parent=None unknown>
```

Process 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 **multiprocessing.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 processes (or 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 processes 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 process at a time executes a critical section of code at a time, while all other processes trying to execute the same code must wait until the currently executing process is finished with the critical section and releases the lock.

Each process must attempt to acquire the lock at the beginning of the critical section. If the lock has not been obtained, then a process will acquire it and other processes must wait until the process 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 process that makes an attempt.
- **Locked**: The lock has been acquired by process thread and any process 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 for use with processes via the **multiprocessing.Lock** class (https://docs.python.org/3/library/multiprocessing.html#multiprocessing.Lock).

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

For example:

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

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

The process attempting to acquire the lock will block until the lock is acquired, such as if another process 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 and for the lock to be released automatically once the block has completed.

For example:

```
...
# create a lock
lock = multiprocessing.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 process via the **locked()** function.

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

Now that we know how to use the **multiprocessing.Lock** class, let's look at a worked example.

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 second.

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

Next, we can then create one instance of the **multiprocessing.Lock** to be shared among the processes.

```
...
# create the shared lock
lock = Lock()
```

We can then create many processes, each configured to execute our **task()** function and compete to execute the critical section.

Each process will receive the shared lock as an argument as well as an integer id between 0 and 9 and a random time to sleep in seconds between 0 and 1.

We can implement this via a list comprehension, creating a list of ten configured **multiprocessing.Process** instances.

```
...
# create a number of processes with different sleep times
processes = [Process(target=task, args=(lock, i, random())) for i in range(10)]
```

Next, we can start all of the processes.

```
...
# start the processes
for process in processes:
    process.start()
```

Finally, we can wait for all of the new child processes to terminate.

```
...
# wait for all processes to finish
for process in processes:
    process.join()
```

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

```
# SuperFastPython.com
# example of a mutual exclusion (mutex) lock for processes
from time import sleep
from random import random
from multiprocessing import Process
from multiprocessing import Lock

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

# entry point
if __name__ == '__main__':
    # create the shared lock
    lock = Lock()
    # create a number of processes with different sleep times
    processes = [Process(target=task, args=(lock, i, random())) for i in range(10)]
    # start the processes
    for process in processes:
        process.start()
    # wait for all processes to finish
    for process in processes:
        process.join()
```

Running the example starts ten processes that are all configured to execute our custom function.

The child processes are then started and the main process blocks until all child processes finish.

Each child process attempts to acquire the lock within the **task()** function. Only one process can acquire the lock at a time and once they do, they report a message including their id and how long they will sleep. The process then blocks for a fraction of a second before releasing the lock.

Your specific results may vary given the use of random numbers. Try running the example a few times

```
>process 2 got the lock, sleeping for 0.34493199862842716
>process 0 got the lock, sleeping for 0.1690829274493061
>process 1 got the lock, sleeping for 0.586700038562483
>process 3 got the lock, sleeping for 0.8439760508777033
>process 4 got the lock, sleeping for 0.49642440261633747
>process 6 got the lock, sleeping for 0.7291278047802177
>process 5 got the lock, sleeping for 0.4495745681185115
>process 7 got the lock, sleeping for 0.6844618818829677
>process 8 got the lock, sleeping for 0.21518155457911792
>process 9 got the lock, sleeping for 0.30577395898093285
```

You can learn more about mutex locks in this tutorial:

- Multiprocessing Lock in Python (https://superfastpython.com/multiprocessing-mutex-lock-in-python/)

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

Process Reentrant Lock

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

You can use reentrant locks in Python via the **multiprocessing.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 process (or thread) to acquire the lock more than once.

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

Each time a process acquires the lock it must also release it, meaning that there are recursive levels of acquire and release for the owning process. 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 reentrant lock for processes via the **multiprocessing.RLock** class (https://docs.python.org/3/library/multiprocessing.html#multiprocessing.RLock).

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

For example:

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

The process attempting to acquire the lock will block until the lock is acquired, such as if another process 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 and for the lock to be released once the block is exited.

For example:

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

Now that we know how to use the **multiprocessing.RLock** class, let's look at a worked example.

Example of Using a Reentrant Lock

We can develop an example to demonstrate how to use the **multiprocessing.RLock** for processes.

First, we can define a function to report that a process is done that protects the **print()** statement with a lock.

```
# reporting function
def report(lock, identifier):
    # acquire the lock
    with lock:
        print(f'>process {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'>process {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 process acquires the lock in **task()**, it will be able to re-enter the lock in the **report()** function.

Next, we can create the reentrant lock.

```
...
# create a shared reentrant lock
lock = RLock()
```

We can then create many processes, each configured to execute our **task()** function.

Each process will receive the shared **multiprocessing.RLock** as an argument as well as an integer id between 0 and 9 and a random time to sleep in seconds between 0 and 1.

We can implement this via a list comprehension, creating a list of ten configured **multiprocessing.Process** instances.

```
...
# create processes
processes = [Process(target=task, args=(lock, i, random())) for i in range(10)]
```

Next, we can start all of the processes.

```
...
# start child processes
for process in processes:
    process.start()
```

Finally, we can wait for all of the new child processes to terminate.

```
...
# wait for child processes to finish
for process in processes:
    process.join()
```

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

```
```

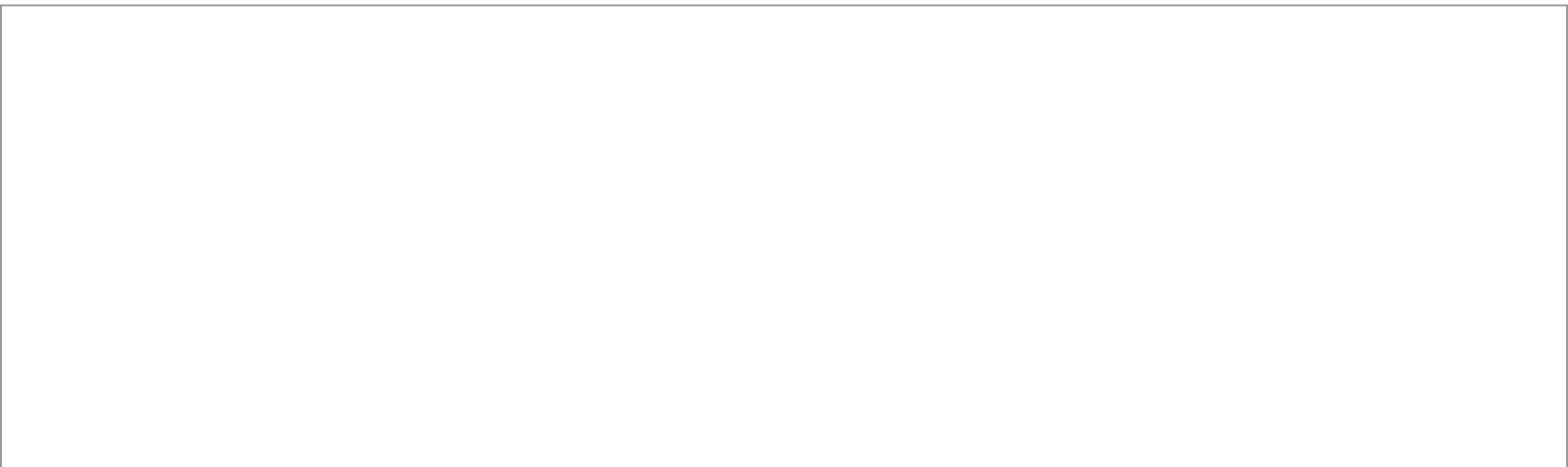
```python
# SuperFastPython.com
# example of a reentrant lock for processes
from time import sleep
from random import random
from multiprocessing import Process
from multiprocessing import RLock

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

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

# entry point
if __name__ == '__main__':
    # create a shared reentrant lock
    lock = RLock()
    # create processes
    processes = [Process(target=task, args=(lock, i, random())) for i in range(10)]
    # start child processes
    for process in processes:
        process.start()
    # wait for child processes to finish
    for process in processes:
        process.join()
```

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

Only one process can acquire the lock at a time, and then once acquired, blocks and then reenters the same lock again to report the done message via the **report()** function.

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

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



```
>process 0 sleeping for 0.9703475136810683
>process 0 done
>process 1 sleeping for 0.10372469305828702
>process 1 done
>process 2 sleeping for 0.26627777997152036
>process 2 done
>process 3 sleeping for 0.9821832886127358
>process 3 done
>process 6 sleeping for 0.005591916432016064
>process 6 done
>process 5 sleeping for 0.6150762561153148
>process 5 done
>process 4 sleeping for 0.3145220383413917
>process 4 done
>process 7 sleeping for 0.8961655132345371
>process 7 done
>process 8 sleeping for 0.5968254072867757
>process 8 done
>process 9 sleeping for 0.8139723778675512
>process 9 done
```

You can learn more about reentrant locks in this tutorial:

- Multiprocessing RLock in Python (https://superfastpython.com/multiprocessing-rlock-in-python/)

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

Process Condition Variable

A condition allows processes to wait and be notified.

You can use a process condition object in Python via the **multiprocessing.Condition** class.

What is a Process Condition Variable

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

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

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

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

Another process can then acquire the condition, make a change, and notify one, all, or a subset of processes waiting on the condition that something has changed. The waiting process 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 processes to wait on the condition and to allow processes to notify other processes 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 Variable

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

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

```
...
# create a new condition variable
condition = multiprocessing.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.

```
...
# create a new condition with a custom lock
condition = multiprocessing.Condition(lock=my_lock)
```

In order for a process 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()
```

The **wait()** function will wait forever until notified by default. We can also pass a "**timeout**" argument which will allow the process to stop blocking after a time limit in seconds.

For example:

```
...
# acquire the condition
with condition:
    # wait to be notified
    condition.wait(timeout=10)
```

The **multiprocessing.Condition** class also provides a **wait_for()** function that can be used to only unlock the waiting process if a condition is met, such as calling a function that returns a boolean value.

The name of the function that returns a boolean value can be provided to the **wait_for()** function directly, and the function also takes a "**timeout**" argument in seconds.

```
...
# acquire the condition
with condition:
    # wait to be notified and a function to return true
    condition.wait_for(all_data_collected)
```

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

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

For example:

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

The notified process 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()** or **wait_for()**, you do not need to do anything extra.

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

We can also notify a subset of waiting processes by setting the "**n**" argument to an integer number of processes to notify, for example:

```
...
# acquire the condition
with condition:
    # notify 3 waiting processes
    condition.notify(n=3)
```

Finally, we can notify all processes waiting on the condition via the **notify_all()** function.

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

A final reminder, a process must acquire the condition before waiting on it or notifying waiting processes. A failure to acquire the condition (the lock within the condition) before performing these actions will result in a **RuntimeError**.

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

Example of Wait and Notify With a Condition Variable

In this section we will explore using a **multiprocessing.Condition** to notify a waiting process that something has happened.

We will create a new child process to simulate performing some work that the main process is dependent upon. Once prepared, the child process will notify the waiting main process, then the main process will continue on.

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

The function will take the condition variable. The function will block for a moment to simulate performing a task, then notify the waiting main process.

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

```
# target function to prepare some work
def task(condition):
    # block for a moment
    sleep(1)
    # notify a waiting process that the work is done
    print('Child process sending notification...', flush=True)
    with condition:
        condition.notify()
    # do something else...
    sleep(1)
```

In the main process, first we can create the shared condition variable.

```
...
# create a condition
condition = Condition()
```

Next, we can acquire the condition variable, so that we can wait on it later.

```
...
# wait to be notified that the data is ready
print('Main process waiting for data...')
with condition:
    # ...
```

The main process will then create a new child process to perform some work, then notify the main process once the work is prepared.

Next, we can start a new child process calling our target task function and wait on the condition variable to be notified of the result.

```
...
# start a new process to perform some work
worker = Process(target=task, args=(condition,))
worker.start()
# wait to be notified
condition.wait()
```

Note, we must start the new process after we have acquired the mutex lock in the condition variable 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 child process before acquiring the condition and waiting in the main process, then it is possible for the new process to execute and notify before the main process has had a chance to start waiting. In which case the main process would wait forever to be notified.

Finally, we can report that the main process is all done.

```
...
# we know the data is ready
print('Main process all done')
```

Tying this together, the complete example is listed below.

```
# SuperFastPython.com
# example of wait/notify with a condition for processes
from time import sleep
from multiprocessing import Process
from multiprocessing import Condition

# target function to prepare some work
def task(condition):
    # block for a moment
    sleep(1)
    # notify a waiting process that the work is done
    print('Child process sending notification...', flush=True)
    with condition:
        condition.notify()
    # do something else...
    sleep(1)

# entry point
if __name__ == '__main__':
    # create a condition
    condition = Condition()
    # wait to be notified that the data is ready
    print('Main process waiting for data...')
    with condition:
        # start a new process to perform some work
        worker = Process(target=task, args=(condition,))
        worker.start()
        # wait to be notified
        condition.wait()
    # we know the data is ready
    print('Main process all done')
```

Running the example first creates the condition variable.

The condition variable is acquired, then a new child process is created and started.

The child process blocks for a moment to simulate work, then notifies the waiting main process.

Meanwhile the main process waits to be notified by the child process, then once notified it continues on.

```
1 Main process waiting for data...
2 Child process sending notification...
3 All done
```

You can learn more about condition variables in the tutorial:

- Multiprocessing Condition Variable in Python (/multiprocessing-condition-variable-in-python)

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

Process Semaphore

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

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

What is a Semaphore

A semaphore is a concurrency primitive that allows a limit on the number of processes (or 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 processes that can acquire the lock before additional processes will block. Once full, new processes can only acquire access on the semaphore once an existing process holding the semaphore releases access.

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 processes 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 for processes via the **multiprocessing.Semaphore** class (https://docs.python.org/3/library/multiprocessing.html#multiprocessing.Semaphore).

The **multiprocessing.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 processes that can hold the semaphore.

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

```
...
# create a semaphore with a limit of 100
semaphore = multiprocessing.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 access in which case, processes attempting to acquire it must block until access 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 process will block until access becomes available on the semaphore.

The "**blocking**" argument can be set to **False** in which case, if access is not available on the semaphore, the process 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 access 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 process is willing to wait for access to the semaphore if one is not available, before giving up. Again, the **acquire()** function will return a value of **True** if access 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()
```

More than one position can be made available by calling release and setting the "**n**" argument to an integer number of positions to release on the semaphore.

This might be helpful if it is known a process has died without correctly releasing the semaphore, or if one process acquires the same semaphore more than once.

Do not use this argument unless you have a clear use case, it is likely to get you into trouble with a semaphore left in an inconsistent state.

```
...
# release three positions on the semaphore
semaphore.release(n=3)
```

Finally, the **multiprocessing.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:
    # ...
```

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

Example of Using a Semaphore

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

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

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 access 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:
        # simulate computational effort
        value = random()
        sleep(value)
        # report result
        print(f'Process {number} got {value}')
```

The main process will first create the **multiprocessing.Semaphore** instance and limit the number of concurrent processes to 2.

```
...
# create the shared semaphore
semaphore = Semaphore(2)
```

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

We can implement this via a list comprehension, creating a list of ten configured **multiprocessing.Process** instances.

```
...
# create processes
processes = [Process(target=task, args=(semaphore, i)) for i in range(10)]
```

Next, we can start all of the processes.

```
...
# start child processes
for process in processes:
    process.start()
```

Finally, we can wait for all of the new child processes to terminate.

```
...
# wait for child processes to finish
for process in processes:
    process.join()
```

Tying this together, the complete example is listed below.

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

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

# entry point
if __name__ == '__main__':
    # create the shared semaphore
    semaphore = Semaphore(2)
    # create processes
    processes = [Process(target=task, args=(semaphore, i)) for i in range(10)]
    # start child processes
    for process in processes:
        process.start()
    # wait for child processes to finish
    for process in processes:
        process.join()
```

Running the example first creates the shared semaphore instance then starts ten child processes.

All ten processes attempt to acquire the semaphore, but only two processes are granted access at a time. The processes 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 process to acquire access and perform its simulated calculation, all the while allowing only two of the processes to be running within the critical section at any one time, even though all ten processes are executing their run methods.

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

```
```

```
Process 3 got 0.18383690831569133
Process 2 got 0.6897978479922813
Process 1 got 0.9585657826673842
Process 6 got 0.1590348392237605
Process 0 got 0.49322767126623646
Process 4 got 0.5704432231809451
Process 5 got 0.3505128460341568
Process 7 got 0.3135061835434463
Process 8 got 0.47103805023306533
Process 9 got 0.21838069804132387
```

You can learn more about semaphores in this tutorial:

- Multiprocessing Semaphore in Python (https://superfastpython.com/multiprocessing-semaphore-in-python/)

Next, let's look at event objects.

Process Event

An event is a process-safe boolean flag.

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

How to Use an Event Object

Python provides an event object for processes via the **multiprocessing.Event** class (https://docs.python.org/3/library/multiprocessing.html#multiprocessing.Event).

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

A multiprocessing.Event object wraps a boolean variable that can either be "***set***" (**True**) or "*not set*" (**False**). Processes 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 **multiprocessing.Event** provides an easy way to share a boolean variable between processes that can act as a trigger for an action.

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

```
...
# create an instance of an event
event = multiprocessing.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 processes 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, processes 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 process 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()
```

A "**timeout**" argument can be passed to the **wait()** function which will limit how long a process 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 **multiprocessing.Event**, let's look at a worked example.

Example of Using an Event Object

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

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

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

```
# 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
print(f'Process {number} waiting...', flush=True)
event.wait()
```

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

```
...
# begin processing
value = random()
sleep(value)
print(f'Process {number} got {value}', flush=True)
```

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
    print(f'Process {number} waiting...', flush=True)
    event.wait()
    # begin processing
    value = random()
    sleep(value)
    print(f'Process {number} got {value}', flush=True)
```

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

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

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

This can be achieved in a list comprehension.

```
...
# create a suite of processes
processes = [Process(target=task, args=(event, i)) for i in range(5)]
```

We can then start all child processes.

```
...
# start all processes
for process in processes:
    process.start()
```

Next, the main process will block for a moment, then trigger the processing in all of the child processes via the event object.

```
...
# block for a moment
print('Main process blocking...')
sleep(2)
# trigger all child processes
event.set()
```

The main process will then wait for all child processes to terminate.

```
...
# wait for all child processes to terminate
for process in processes:
    process.join()
```

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

```python
# SuperFastPython.com
# example of using an event object with processes
from time import sleep
from random import random
from multiprocessing import Process
from multiprocessing import Event

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

# entry point
if __name__ == '__main__':
    # create a shared event object
    event = Event()
    # create a suite of processes
    processes = [Process(target=task, args=(event, i)) for i in range(5)]
    # start all processes
    for process in processes:
        process.start()
    # block for a moment
    print('Main process blocking...')
    sleep(2)
    # trigger all child processes
    event.set()
    # wait for all child processes to terminate
    for process in processes:
        process.join()
```

Running the example first creates and starts five child processes.

Each child process waits on the event before it starts its work, reporting a message that it is waiting.

The main process blocks for a moment, allowing all child processes to begin and start waiting on the event.

The main process then sets the event. This triggers all five child processes that perform their simulated work and report a message.

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

```
Main process blocking...
Process 0 waiting...
Process 1 waiting...
Process 2 waiting...
Process 3 waiting...
Process 4 waiting...
Process 0 got 0.06198821143561384
Process 4 got 0.219334069761699
Process 3 got 0.7335552378594119
Process 1 got 0.7948771419640999
Process 2 got 0.8713839353896263
```

You can learn more about event objects in this tutorial:

- Multiprocessing Event Object In Python (https://superfastpython.com/multiprocessing-event-object-in-python/)

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

Process Barrier

A barrier allows you to coordinate child processes.

You can use a process barrier in Python via the **multiprocessing.Barrier** class.

What is a Barrier

A barrier is a synchronization primitive.

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

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

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

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 **multiprocessing.Barrier** class.

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

For example:

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

We can also perform an action once all processes 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 process once all processes reach the barrier but before the processes are released.

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

We can also set a default timeout used by all processes 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 = multiprocessing.Barrier(10, timeout=5)
```

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

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

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

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

The wait function returns an integer indicating the number of parties remaining to arrive at the barrier. If a process was the last to arrive, then the return value will be zero. This is helpful if you want the last process or one process 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 processes 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 processes to arrive
try:
    barrier.wait()
except BrokenBarrierError:
    # ...
```

We can also abort the barrier.

Aborting the barrier means that all processes 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 processes 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 Process Barrier

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

In this example we will create a suite of processes, each required to perform some blocking calculation. We will use a barrier to coordinate all processes after they have finished their work and perform some action in the main process. 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 process. The function will take the barrier as an argument as well as a unique identifier for the process.

The process 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 processes 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'Process {number} done, got: {value}', flush=True)
    # wait on all other processes to complete
    barrier.wait()
```

Next in the main process we can create the barrier.

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

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

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

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

Finally, the main process can wait on the barrier for all processes to perform their calculation.

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

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

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

Tying this together, the complete example is listed below.

```
# SuperFastPython.com
# example of using a barrier with processes
from time import sleep
from random import random
from multiprocessing import Process
from multiprocessing 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'Process {number} done, got: {value}', flush=True)
    # wait on all other processes to complete
    barrier.wait()

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

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

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

Finally, the processes finish and are all released, including the main process, reporting a final message.

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

```
Main process waiting on all results...
Process 3 done, got: 0.19554467220314398
Process 4 done, got: 0.345718981628913
Process 2 done, got: 2.37158455232798
Process 1 done, got: 5.970308025592141
Process 0 done, got: 7.102904442921531
All processes have their result
```

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

- Multiprocessing Barrier in Python (/multiprocessing-barrier-in-python)

Next, let's look at best practices when using multiprocessing.

Python Multiprocessing Best Practices

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

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

1. Use Context Managers
2. Use Timeouts When Waiting
3. Use Main Module Idiom
4. Use Shared ctypes
5. Use Pipes and Queues

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 (https://docs.python.org/3.10/library/stdtypes.html#typecontextmanager), 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 synchronization primitives, such as:

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

The context manager interface is also supported on other multiprocessing utilities, such as:

- Opening a connection via the multiprocessing.**connection.Connection** class
- Creating a manager via the **multiprocessing.Manager** class
- Creating a process pool via the **multiprocessing.pool.Pool** class.
- Creating a listener via the **multiprocessing.connection.Listener** class.

Tip 2: Use Timeouts When Waiting

Always use a timeout when waiting on a blocking call.

Many calls made on synchronization primitives may block.

For example:

- Waiting to acquire a **multiprocessing.Lock** via **acquire()**.
- Waiting for a process to terminate via **join()**.
- Waiting to be notified on a **multiprocessing.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 process 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 Main Module Idiom

A Python program that uses multiprocessing should protect the entry point of the program.

This can be achieved by using an if-statement to check that the entry point is the top-level environment.

For example:

```
...
# check for top-level environment
if __name__ == '__main__':
    # ...
```

This will help to avoid a RuntimeError when creating child processes using the '**spawn**' start method, the default on Windows and MacOS.

You can learn more about protecting the entry point when using multiprocessing in the tutorial:

- Fix RuntimeError When Spawning a Child Process (/multiprocessing-spawn-runtimeerror)

Additionally, it is a good practice to add freeze support as the first line of a Python program that uses multiprocessing.

Freezing a Python program is a process that transforms the Python code into C code for packaging and distribution.

When a program is frozen in order to be distributed, some features of Python are not included or disabled by default.

This is for performance and/or security reasons.

One feature that is disabled when freezing a Python program is multiprocessing.

That is, we cannot create new python processes via multiprocessing.Process instances when freezing our program for distribution.

Creating a process in a frozen application results in a **RuntimeError**.

We can add support for multiprocessing in our program when freezing code via the **multiprocessing.freeze_support()** function (https://docs.python.org/3/library/multiprocessing.html#multiprocessing.freeze_support).

For example:

```
...
# enable support for multiprocessing
multiprocessing.freeze_support()
```

This will have no effect on programs that are not frozen.

You can learn more about adding freeze support in the tutorial:

- Multiprocessing Freeze Support in Python (/multiprocessing-freeze-support-in-python)

Protecting the entry point and adding freeze support together are referred to as the "*main module*" idiom when using multiprocessing.

Using this idiom is a best practice when using multiprocessing.

```
An attempt has been made to start a new process before the
        current process has finished its bootstrapping phase.

        This probably means that you are not using fork to start your
        child processes and you have forgotten to use the proper idiom
        in the main module:

            if __name__ == '__main__':
                freeze_support()
                ...

        The "freeze_support()" line can be omitted if the program
        is not going to be frozen to produce an executable.
```

Tip 4: Use Shared ctypes

Processes do not have shared memory.

Instead, shared memory must be simulated using sockets and/or files.

If you need to share simple data variables or arrays of variables between processes, this can be achieved using shared ctypes.

Shared ctypes provide a mechanism to share data safely between processes in a process-safe manner.

You can share ctypes among processes using the multiprocessing.Value and **multiprocessing.Array** classes.

- The **multiprocessing.Value** class (https://docs.python.org/3/library/multiprocessing.html#multiprocessing.Value) is used to share a ctype of a given type among multiple processes.
- The **multiprocessing.Array** class (https://docs.python.org/3/library/multiprocessing.html#multiprocessing.Array) is used to share an array of ctypes of a given type among multiple processes.

Share ctypes provide a simple and easy to use way of sharing data between processes.

For example, a shared ctype value can be defined in a parent process, then shared with multiple child processes. All child processes and the parent process can then safely read and modify the data within the shared value.

This can be useful in a number of use cases, such as:

- A counter shared among multiple processes.
- Returning data from a child process to a parent process.
- Sharing results of computation among processes.

Shared ctypes can only be shared among processes on the one system. For sharing data across processes on

The multiprocessing.Value class will create a shared ctype with a specified data type and initial value.

For example:

```
...
# create a value
value = multiprocessing.Value(...)
```

The first argument defines the data type for the value. It may be a string type code or a Python ctype class. The second argument may be an initial value.

For example, we can define a signed integer type with the 'i' type code and an initial value of zero as follows:

```
...
# create a integer value
variable = multiprocessing.Value('i', 0)
```

We can define the same signed integer shared ctype using the **ctypes.c_int** class.

For example:

Once defined, the value can then be shared and used within multiple processes, such as between a parent and a child process.

Internally, the **multiprocessing.Value** makes use of a **multiprocessing.RLock** that ensures that access and modification of the data inside the class is mutually exclusive, e.g. process-safe.

This means that only one process at a time can access or change the data within the **multiprocessing.Value** object.

The data within the **multiprocessing.Value** object can be accessed via the "**value**" attribute.

For example:

```
...
# get the data
data = variable.value
```

The data within the **multiprocessing.Value** can be changed by the same "**value**" attribute.

For example:

```
...
# change the data
variable.value = 100
```

You can learn more about using shared ctypes in the tutorial:

- Multiprocessing Shared ctypes in Python (/multiprocessing-shared-ctypes-in-python)

Tip 5: Use Pipes and Queues

Processes can share messages with each other directly using pipes or queues.

These are process-safe data structures that allow processes to send or receive pickleable Python objects.

In multiprocessing, a pipe is a connection between two processes in Python.

Python provides a simple queue in the **multiprocessing.Pipe** class (https://docs.python.org/3/library/multiprocessing.html#multiprocessing.Pipe).

A pipe can be created by calling the constructor of the **multiprocessing.Pipe** class, which returns two **multiprocessing.connection.Connection** objects.

For example:

```
...
# create a pipe
conn1, conn2 = multiprocessing.Pipe()
```

Objects can be shared between processes using the Pipe.

The **Connection.send()** function can be used to send objects from one process to another.

The objects sent must be picklable.

For example:

```
...
# send an object
conn2.send('Hello world')
```

The **Connection.recv()** function can be used to receive objects in one process sent by another.

The objects received will be automatically un-pickled.

For example:

```
...
# receive an object
object = conn1.recv()
```

You can learn more about multiprocessing pipes in the tutorial:

- Multiprocessing Pipe in Python (/multiprocessing-pipe-in-python)

Python provides a process-safe queue in the **multiprocessing.Queue** class (https://docs.python.org/3/library/multiprocessing.html#multiprocessing.Queue).

A queue is a data structure on which items can be added by a call to put() and from which items can be retrieved by a call to get().

The **multiprocessing.Queue** provides a first-in, first-out FIFO queue, which means that the items are retrieved from the queue in the order they were added. The first items added to the queue will be the first items retrieved. This is opposed to other queue types such as last-in, first-out and priority queues.

The **multiprocessing.Queue** can be used by first creating an instance of the class. This will create an unbounded queue by default, that is, a queue with no size limit.

For example:

```
...
# created an unbounded queue
queue = multiprocessing.Queue()
```

Items can be added to the queue via a call to **put()**, for example:

```
...
# add an item to the queue
queue.put(item)
```

Items can be retrieved from the queue by calls to **get()**.

For example:

```
...
# get an item from the queue
item = queue.get()
```

You can learn more about multiprocessing queues in the tutorial:

- Multiprocessing Queue in Python (/multiprocessing-queue-in-python)

Next, let's look at common errors when using multiprocessing in Python.

Python Multiprocessing Common Errors

When you are getting started with multiprocessing in Python, you may encounter one of many common errors.

These errors are typically made because of bugs introduced by copy-and-pasting code, or from a slight misunderstanding in how new child processes work.

We will take a closer look at some of the more common errors made when creating new child processes; they are:

- **Error 1**: RuntimeError Starting New Processes
- **Error 2**: print() Does Not Work In Child Processes
- **Error 3**: Adding Attributes to Classes that Extend Process

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

Error 1: RuntimeError Starting New Processes

It is common to get a **RuntimeError** when starting a new Process in Python.

The content of the error often looks as follows:

```
    An attempt has been made to start a new process before the
    current process has finished its bootstrapping phase.

    This probably means that you are not using fork to start your
    child processes and you have forgotten to use the proper idiom
    in the main module:

        if __name__ == '__main__':
            freeze_support()
            ...

    The "freeze_support()" line can be omitted if the program
    is not going to be frozen to produce an executable.
```

This will happen on Windows and MacOS where the default start method is '**spawn**'. It may also happen when you configure your program to use the '**spawn**' start method on other platforms.

This is a common error and is easy to fix.

The fix involves checking if the code is running in the top-level environment and only then, attempt to start a new process.

This is a best practice.

The idiom for this fix, as stated in the message of the **RuntimeError**, is to use an if-statement and check if the name of the module is equal to the string '**__main__**'.

For example:

```
...
# check for top-level environment
if __name__ == '__main__':
    # ...
```

This is called "*protecting the entry point*" of the program.

Recall, that **__name__** is a variable that refers to the name of the module executing the current code.

Also, recall that '**__main__**' is the name of the top-level environment used to execute a Python program.

Using an if-statement to check if the module is the top-level environment and only starting child processes within that block will resolve the RuntimeError.

It means that if the Python file is imported, then the code protected by the if-statement will not run. It will only run when the Python file is run directly, e.g. is the top-level environment.

The if-statement idiom is required, even if the entry point of the program calls a function that itself starts a child process.

You can learn more about this common error in the tutorial:

- Fix RuntimeError When Spawning a Child Process (/multiprocessing-spawn-runtimeerror)

Error 2: print() Does Not Work In Child Processes

Printing to standard out (stdout) with the built-in **print()** function (https://docs.python.org/3/library/functions.html#print) may not work property from child processes.

For example, you may print output messages for the user or debug messages from a child process and they may never appear, or may only appear when the child process is terminated.

For example:

```
...
# report a message from a child process
print('Hello from the child process')
```

This is a very common situation and the cause is well understood and easy to workaround.

The **print()** function is a built-in function for displaying messages on standard output or stdout.

When you call **print()** from a child process created using the '**spawn**' start method, the message will not appear.

This is because the messages are block buffered by default and the buffer is not flushed by default after every message. This is unlike the main process that is interactive and will flush messages after each line, e.g. line buffered.

Instead, the buffered messages are only flushed occasionally, such as when the child process terminates and the buffer is garbage collected.

We can flush stdout automatically with each call to **print()**.

This can be achieved by setting the '**flush**' argument to **True**.

For example:

```
...
# report a message from a child process
print('Hello from the child process', flush=True)
```

An alternate approach is to call the **flush()** function on the **sys.stdout** object directly.

For example:

```
...
# report a message from a child process
print('Hello from the child process')
# flush output
sys.stdout.flush()
```

The problem with the **print()** function only occurs when using the '**spawn**' start method.

You can change the start method to '**fork**' which will cause **print()** to work as expected.

Note, the '**fork**' start method is not supported on Windows at the time of writing.

You can set the start method via the **multiprocessing.set_start_method()** function.

For example:

```
...
# set the start method to fork
set_start_method('fork')
```

You can learn more about process start methods in the tutorial:

- Multiprocessing Start Methods (https://superfastpython.com/multiprocessing-start-method)

You can learn more about fixing **print()** from child processes in the tutorial:

- How to print() from a Child Process in Python (/multiprocessing-print)

Error 3: Adding Attributes to Classes that Extend Process

Python provides the ability to create and manage new processes via the **multiprocessing.Process** class.

We can extend this class and override the **run()** function in order to run code in a new child process.

You can learn more about extending the the **multiprocessing.Process** class in the tutorial:

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

Extending the **multiprocessing.Process** and adding attributes that are shared among multiple processes will fail with an error.

For example, if we define a new class that extends the **multiprocessing.Process** class that sets an attribute on the class instance from the **run()** method executed in a new child process, then this attribute will not be accessible by other processes, such as the parent process.

This is the case even if both parent and child processes share access to the "*same*" object.

This is because class instance variables are not shared among processes by default. Instead, instance variables added to the **multiprocessing.Process** are private to the process that added them.

Each process operates on a serialized copy of the object and any changes made to that object are local to that process only, by default.

If you set class attributes in the child process and try to access them in the parent process or another process, you will get an error.

For example:

```
...
Traceback (most recent call last):
...
AttributeError: 'CustomProcess' object has no attribute 'data'
```

This error occurred because the child process operates on a copy of the class instance that is different from the copy of the class instance used in the parent process.

Instance variable attributes can be shared between processes via the **multiprocessing.Value** and **multiprocessing.Array** classes.

These classes explicitly define data attributes designed to be shared between processes in a process-safe manner.

Shared variables mean that changes made in one process are always propagated and made available to other processes.

An instance of the **multiprocessing.Value** can be defined in the constructor of a custom class as a shared instance variable.

The constructor of the **multiprocessing.Value** class requires that we specify the data type and an initial value.

The data type can be specified using ctype "**type**" or a typecode.

Typecodes are familiar and easy to use, for example '**i**' for a signed integer or '**f**' for a single floating-point value.

For example, we can define a **multiprocessing.Value** shared memory variable that holds a signed integer and is initialized to the value zero.

```
...
# initialize an integer shared variable
data = multiprocessing.Value('i', 0)
```

This can be initialized in the constructor of the class that extends the **multiprocessing.Process** class.

We can change the value of the shared data variable via the "**value**" attribute.

For example:

```
...
# change the value of the shared variable
data.value = 100
```

We can access the value of the shared data variable via the same "**value**" attribute.

For example:

```
...
# access the shared variable
value = data.value
```

The propagation of changes to the shared variable and mutual exclusion locking of the shared variable is all performed automatically behind the scenes.

You can learn more about this error and how to fix in the tutorial:

- Shared Process Class Attributes in Python (https://superfastpython.com/share-process-attributes/)

Next, let's look at common questions about multiprocessing in Python.

Python Multiprocessing Common Questions

This section answers common questions asked by developers when using multiprocessing in Python.

Do you have a question about processes?
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 Safely Stop a Process?

A process can be stopped using a shared boolean variable such as a **multiprocessing.Event**.

A **multiprocessing.Event** is a process-safe boolean variable flag that can be either set or not set. It can be shared between processes safely and correctly and checked and set without fear of a race condition.

A new **multiprocessing.Event** can be created and then shared between processes, for example:

```
...
# create a shared event
event = multiprocessing.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 process. 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 a **multiprocessing.Event** each iteration.

If the event is set **True**, we can exit the task loop or return from the **task()** function, allowing the new child process to terminate.

The status of the **multiprocessing.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 parent process, or another process, can then set the event in order to stop the new process from running.

The **multiprocessing.Event** can be set or made **True** via the **set()** function.

For example:

```
...
# set the event
event.set()
# wait for the new process to stop
process.join()
```

Now that we know how to stop a Python process, let's look at a worked example.

```
# SuperFastPython.com
# example of stopping a new process
from time import sleep
from multiprocessing import Process
from multiprocessing 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 process running...', flush=True)
    print('Worker closing down', flush=True)

# entry point
if __name__ == '__main__':
    # create the event
    event = Event()
    # create and configure a new process
    process = Process(target=task, args=(event,))
    # start the new process
    process.start()
    # block for a while
    sleep(3)
    # stop the worker process
    print('Main stopping process')
    event.set()
    # wait for the new process to finish
    process.join()
```

Running the example first creates and starts the new child process.

The main parent process then blocks for a few seconds.

Meanwhile, the new process 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 process to stop.

The main parent process wakes up, and then sets the event. It then joins the new process, waiting for it to terminate.

The task process checks the event which is discovered to be set (e.g. **True**). The process breaks the task loop, reports a final message then terminates the new process.

The parent process then terminates, closing the Python program.

You can learn more about safely stopping a process in this tutorial:

- How to Safely Stop a Process in Python (https://superfastpython.com/safely-stop-a-process-in-python/)

How to Kill a Process?

Killing a process is drastic.

If possible, it is better to safely stop a child process.

Nevertheless, a process can be forcefully stopped immediately.

The downside is that the process will not have an opportunity to clean-up, making it possibly unsafe.

Specifically:

- The finally clauses in try-except-finally or try-finally patterns will not be executed.
- The exit handlers will not be executed.

This means that open resources like files and sockets will not be closed in a safe and controlled manner. It also means that any application state will not be updated.

Nevertheless, sometimes we may need to immediately kill a child process.

There are two main approaches to kill a process, they are:

- Terminate a process via **Process.terminate()**.
- Kill a process via **Process.kill()**.

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

A process can be killed by calling the **Process.terminate()** function.

The call will only terminate the target process, not child processes.

The method is called on the **multiprocessing.Process** instance for the process that you wish to terminate. You may have this instance from creating the process or it may be acquired via module methods such as the **multiprocessing.active_children()** function.

For example:

```
...
# terminate the process
process.terminate()
```

The method takes no arguments and does not block.

The function will terminate the process using the **SIGTERM** (signal terminate) signal on most platforms, or the equivalent on windows.

A process can be killed by calling the **Process.kill()** function.

The call will only terminate the target process, not child processes.

The method is called on the **multiprocessing.Process** instance for the process that you wish to terminate. You may have this instance from creating the process or it may be acquired via module methods such as the **multiprocessing.active_children()** function.

For example:

```
...
# kill the process
process.kill()
```

The method takes no arguments and does not block.

The function will terminate the process using the **SIGKILL** (signal kill) signal on most platforms, or the equivalent on windows. This signal cannot be ignored and cannot be handled.

You can learn more about forcefully killing a process in the tutorial:

- Kill a Process in Python (https://superfastpython.com/kill-a-process-in-python/)

How Do You Wait for Processes to Finish?

A process can be joined in Python by calling the **join()** method on the process instance.

For example:

```
...
# join a process
process.join()
```

This has the effect of blocking the current process until the target process that has been joined has terminated.

The target process 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 Process class.
- Raised an error or exception.

Once the target process has finished, the **join()** method will return and the current process can continue to execute.

The **join()** method requires that you have a **multiprocessing.Process** instance for the process you wish to join.

This means that if you created the process, you may need to keep a reference to the **multiprocessing.Process** instance.

Alternatively, you may use a multiprocessing module function to get an instance for a process, such as **multiprocessing.active_children()** or **multiprocessing.parent_process()**.

The **join()** method also takes a "**timeout**" argument that specifies how long the current process is willing to wait for the target process to terminate, in seconds.

Once the timeout has expired and the target process has not terminated, the **join()** process will return.

```
...
# join the process with a timeout
process.join(timeout=10)
```

When using a timeout, it will not be clear whether the **join()** method returned because the target process terminated or because of the timeout. Therefore, we can call the **Process.is_alive()** function to confirm the target process is no longer running.

For example:

```
...
# join the process with a timeout
process.join(timeout=10)
# check if the target process is still running
if process.is_alive():
    # timeout expired, process is still running
else:
    # process has terminated
```

You can learn more about waiting for a process to finish in the tutorial:

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

How to Restart a Process?

Python processes cannot be restarted or reused.

In fact, this is probably a limitation of the capabilities of processes provided by the underlying operating system.

Once a process has terminated you cannot call the **start()** method on it again to reuse it.

Recall that a process 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 process will result in an **AssertionError** indicating that the process can only be started once.

```
AssertionError: cannot start a process twice
```

Instead, to restart a process in Python, you must create a new instance of the process with the same configuration and then call the **start()** function.

We can explore what happens when we attempt to start a terminated process in Python.

In this example we will create a new process to execute a target task function, wait for the new process to terminate, then attempt to restart it again. We expect it to fail with an AssertionError.

First, let's define a target task function to execute in a new process.

The function will first block for a moment, then will report a message to let us know that the new process is executing.

```
# custom target function
def task():
    # block for a moment
    sleep(1)
    # report a message
    print('Hello from the new process')
```

Next, the main process will create an instance of the **multiprocessing.Process** class and configure it to execute our custom **task()** function via the "**target**" keyword.

```
...
# create a new process
process = Process(target=task)
```

We then start executing the process which will internally execute the **run()** function and in turn call our custom **task()** function.

```
...
# start the process
process.start()
```

Next, the parent process joins the new child process which will block (not return) until the new process has terminated.

```
...
# wait for the process to finish
process.join()
```

Finally, the parent process will attempt to start the terminated process again.

```
...
# try and start the process again
process.start()
```

Tying this together, the complete example is listed below.

```
```

```
# SuperFastPython.com
# example of restarting a process
from time import sleep
from multiprocessing import Process

# custom target function
def task():
    # block for a moment
    sleep(1)
    # report a message
    print('Hello from the new process')

# entry point
if __name__ == '__main__':
    # create a new process
    process = Process(target=task)
    # start the process
    process.start()
    # wait for the process to finish
    process.join()
    # try and start the process again
    process.start()
```

Running the example first creates a process instance then starts its execution.

The new process is started, blocks for a moment then reports a message.

The parent process joins the new process and waits for it to terminate. Once terminated, the parent process attempts to start the same process again.

The result is an **AssertionError**, as we expected.

```
Hello from the new process
Traceback (most recent call last):
  ...
AssertionError: cannot start a process twice
```

This highlights that indeed we cannot call the **start()** method (e.g. restart) a process that has already terminated.

You can learn more about restarting a process in the tutorial:

- How to Restart a Process in Python (https://superfastpython.com/restart-a-process-in-python/)

How to Return a Value From a Process?

There are no direct methods to return a value from a process.

Instead, it can be achieved using indirect methods.

There are a number of indirect methods to choose from.

The three most convenient and widely used methods to return a value from a process are as follows:

1. Use a **multiprocessing.Value** object.
2. Use a **multiprocessing.Pipe** object.
3. Use a **multiprocessing.Queue** object.

You can learn more about techniques for returning a value from a process in the tutorial:

- Multiprocessing Return Value From Process (/multiprocessing-return-value-from-process)

Let's take a look at some of these approaches for returning a value.

We can return a variable from a process using a **multiprocessing.Value** object.

These classes explicitly define data attributes designed to be shared between processes in a process-safe manner.

A process-safe manner means that only one process can read or access the variable at a time. Shared variables mean that changes made in one process are always propagated and made available to other processes.

An instance of the **multiprocessing.Value** can be defined in the constructor of a custom class as a shared instance variable.

The constructor of the **multiprocessing.Value** class requires that we specify the data type and an initial value.

The data type can be specified using ctype "**type**" or a typecode.

Typecodes are familiar and easy to use, for example 'i' for a signed integer or 'f' for a single floating-point value.

For example, we can define a Value shared memory variable that holds a signed integer and is initialized to the value zero.

```
...
# initialize an integer shared variable
data = multiprocessing.Value('i', 0)
```

This variable can then be initialized in a parent process and shared with a child process

We can change the value of the shared data variable via the "**value**" attribute.

For example:

```
...
# change the value of the shared variable
data.value = 100
```

We can access the value of the shared data variable via the same "**value**" attribute.

For example:

```
...
# access the shared variable
value = data.value
```

The propagation of changes to the shared variable and mutual exclusion locking of the shared variable is all performed automatically behind the scenes.

You can learn more about using shared ctypes in the tutorial:

- Multiprocessing Shared ctypes in Python (/multiprocessing-shared-ctypes-in-python)

We can return a variable from a process using the **multiprocessing.Queue** class (https://docs.python.org/3/library/multiprocessing.html#multiprocessing.Queue).

A queue is a data structure (https://en.wikipedia.org/wiki/Queue_(abstract_data_type)) on which items can be added by a call to **put()** and from which items can be retrieved by a call to **get()**.

The **multiprocessing.Queue** provides a first-in, first-out FIFO queue, which means that the items are retrieved from the queue in the order they were added. The first items added to the queue will be the first items retrieved. This is opposed to other queue types such as last-in, first-out and priority queues.

The **multiprocessing.Queue** can be used by first creating an instance of the class. This will create an unbounded queue by default, that is, a queue with no size limit.

For example:

```
...
# created an unbounded queue
queue = multiprocessing.Queue()
```

Items can be added to the queue via a call to **put()**, for example:

```
...
# add an item to the queue
queue.put(item)
```

By default, the call to **put()** will block and will not use a timeout.

Items can be retrieved from the queue by calls to **get()**.

For example:

```
...
# get an item from the queue
item = queue.get()
```

By default, the call to **get()** will block until an item is available to retrieve from the queue and will not use a timeout.

You can learn more about multiprocessing queues in the tutorial:

- Multiprocessing Queue in Python (https://superfastpython.com/multiprocessing-queue-in-python)

How to Share Data Between Processes?

In multiprocessing programming, we typically need to share data and program state between processes.

This can be achieved using shared memory via shared ctypes.

The **ctypes** module (https://docs.python.org/3/library/ctypes.html) provides tools for working with C data types.

The C programming language and related languages like C++ underlie many modern programming languages and are the languages used to implement most modern operating systems.

As such, the data types used in C are somewhat standardized across many platforms.

The ctypes module allows Python code to read, write, and generally interoperate with data using standard C data types.

Python provides the capability to share ctypes between processes on one system.

This is primarily achieved via the following classes:

- **multiprocessing.Value**: manage a shared value.
- **multiprocessing.Array**: manage an array of shared values.

The **multiprocessing.Value** class (https://docs.python.org/3/library/multiprocessing.html#multiprocessing.Value) is used to share a ctype of a given type among multiple processes.

The **multiprocessing.Array** class (https://docs.python.org/3/library/multiprocessing.html#multiprocessing.Array) is used to share an array of ctypes of a given type among multiple processes.

Share ctypes provide a simple and easy to use way of sharing data between processes.

For example, a shared ctype value can be defined in a parent process, then shared with multiple child processes. All child processes and the parent process can then safely read and modify the data within the shared value.

This can be useful in a number of use cases, such as:

- A counter shared among multiple processes.
- Returning data from a child process to a parent process.
- Sharing results of computation among processes.

Shared ctypes can only be shared among processes on the one system. For sharing data across processes on multiple systems, a **multiprocessing.Manager** should be used.

You can learn more about how to share data between processes in the tutorial:

- Multiprocessing Shared ctypes in Python (/multiprocessing-shared-ctypes-in-python)

How Do You Exit a Process?

The **sys.exit()** function (https://docs.python.org/3/library/sys.html#sys.exit) will exit the Python interpreter.

When called, the **sys.exit()** function will raise a **SystemExit** exception.

This exception is (typically) not caught and bubbles up to the top of a stack of the running thread, causing it and the process to exit.

The **sys.exit()** function takes an argument that indicates the success or failure of the exit status.

A value of None (the default) or zero indicates a successful , whereas a larger value indicates an unsuccessful exit.

Importantly, finally operations in try-except-finally and try-finally patterns are executed. This allows a program to clean-up before exiting.

In concurrency programming, we may make calls to **sys.exit()** to close our program.

Each Python process has an exit code.

The exit code of a process is accessible via the **multiprocessing.Process.exitcode** attribute (https://docs.python.org/3/library/multiprocessing.html#multiprocessing.Process.exitcode).

The process exitcode is set automatically, for example:

- If the process is still running, the exitcode will be None.
- If the process exited normally, the exitcode will be 0.
- If the process terminated with an uncaught exception, the exitcode will be 1.

The exitcode can also be set via a call to **sys.exit()**.

For example, a child process may exit with a call to **sys.exit()** with no arguments.

The child process will terminate and the exitcode will be set to 0.

The **sys.exit()** function is used by simply making the function call.

A normal exit can be achieved by calling the function with no argument, e.g. defaulting to a value of **None**.

For example:

```
...
# exit normally
sys.exit()
```

A normal exit can also be achieved by passing the value of None or 0 as an argument.

```
...
# exit normally
sys.exit(0)
```

An unsuccessful exit can be signaled by passing a value other than 0 or None.

This may be an integer exit code, such as 1.

For example:

```
...
# exit with error
sys.exit(1)
```

You can learn more about calling **sys.exit()** in child processes in the tutorial:

- Exit a Process with sys.exit() in Python (/exit-process)

What is the Main Process?

The main process in Python is the process started when you run your Python program.

There are a number of ways that the main process can be identified.

They are:

- The main process has a distinct name of "*MainProcess*".
- The main process does not have a parent process.
- The main process is an instance of the **multiprocessing.process._MainProcess** class.

You can learn more about the main process in the tutorial:

- Main Process in Python (/main-process-in-python)

How Do You Use a Multiprocessing Queue?

Python provides a process-safe queue in the **multiprocessing.Queue** class (https://docs.python.org/3/library/multiprocessing.html#multiprocessing.Queue).

A queue is a data structure (https://en.wikipedia.org/wiki/Queue_(abstract_data_type)) on which items can be added by a call to **put()** and from which items can be retrieved by a call to **get()**.

The **multiprocessing.Queue** provides a first-in, first-out FIFO queue, which means that the items are retrieved from the queue in the order they were added. The first items added to the queue will be the first items retrieved. This is opposed to other queue types such as last-in, first-out and priority queues.

The **multiprocessing.Queue** can be used by first creating an instance of the class. This will create an unbounded queue by default, that is, a queue with no size limit.

For example:

```
...
# created an unbounded queue
queue = multiprocessing.Queue()
```

A create can be created with a size limit by specifying the "**maxsize**" argument to a value larger than zero.

For example:

```
...
# created a size limited queue
queue = multiprocessing.Queue(maxsize=100)
```

Once a size limited queue is full, new items cannot be added and calls to **put()** will block until space becomes available.

Items can be added to the queue via a call to **put()**, for example:

```
...
# add an item to the queue
queue.put(item)
```

By default, the call to **put()** will block and will not use a timeout.

For example, the above is equivalent to the following:

```
...
# add an item to the queue
queue.put(item, block=True, timeout=None)
```

The call to **put()** will block if the queue is full. We can choose to not block when adding items by setting the "**block**" argument to **False**. If the queue is full, then a **queue.Full** exception will be raised which may be handled.

For example:

```
...
# add an item to a size limited queue without blocking
try:
    queue.put(item, block=False)
except queue.Full:
    # ...
```

This can also be achieved with the **put_nowait()** function that does the same thing.

For example:

```
...
# add an item to a size limited queue without blocking
try:
    queue.put_nowait(item)
except queue.Full:
    # ...
```

Alternatively we may wish to add an item to a size limited queue and block with a timeout. This can be achieved by setting the "**timeout**" argument to a positive value in seconds. If an item cannot be added before the timeout expires, then a queue.Full exception will be raised which may be handled.

For example:

```
...
# add an item to a size limited queue with a timeout
try:
    queue.put(item, timeout=5)
except queue.Full:
    # ...
```

Items can be retrieved from the queue by calls to **get()**.

For example:

```
...
# get an item from the queue
item = queue.get()
```

By default, the call to **get()** will block until an item is available to retrieve from the queue and will not use a timeout. Therefore, the above call is equivalent to the following:

```
...
# get an item from the queue
item = queue.get(block=True, timeout=0)
```

We can retrieve items from the queue without blocking by setting the "**block**" argument to **False**. If an item is not available to retrieve, then a **queue.Empty** exception will be raised and may be handled.

```
...
# get an item from the queue without blocking
try:
    item = queue.get(block=False)
except queue.Empty:
    # ...
```

This can also be achieved with the **get_nowait()** function that does the same thing.

For example:

```
...
# get an item from the queue without blocking
try:
    item = queue.get_nowait()
except queue.Empty:
    # ...
```

Alternatively, we may wish to retrieve items from the queue and block with a time limit. This can be achieved by setting the "**timeout**" argument to a positive value in seconds. If the timeout expires before an item can be retrieved, then a **queue.Empty** exception will be raised and may be handled.

For example:

```
...
# get an item from the queue with a timeout
try:
    item = queue.get(timeout=10)
except queue.Empty:
    # ...
```

The number of items in the queue can be checked by the **qsize()** function.

For example:

```
...
# check the size of the queue
size = queue.qsize()
```

We can check if the queue contains no values via the **empty()** function.

For example:

```
...
# check if the queue is empty
if queue.empty():
    # ...
```

We may also check if the queue is full via the **full()** function, if it is size limited when configured.

For example:

```
...
# check if the queue is full
if queue.full():
    # ...
```

You can learn more about how to use process-safe queues in the tutorial:

- Multiprocessing Queue in Python (/multiprocessing-queue-in-python)

How Do You Use a Multiprocessing Pipe?

In multiprocessing, a pipe is a connection between two processes in Python.

It is used to send data from one process which is received by another process.

Under the covers, a pipe is implemented using a pair of connection objects, provided by the **multiprocessing.connection.Connection** class (https://docs.python.org/3/library/multiprocessing.html#multiprocessing.connection.Connecti

on).

Creating a pipe will create two connection objects, one for sending data and one for receiving data. A pipe can also be configured to be duplex so that each connection object can both send and receive data.

Python provides a simple queue in the **multiprocessing.Pipe** class (https://docs.python.org/3/library/multiprocessing.html#multiprocessing.Pipe).

A pipe can be created by calling the constructor of the **multiprocessing.Pipe** class, which returns two **multiprocessing.connection.Connection** objects.

For example:

```
...
# create a pipe
conn1, conn2 = multiprocessing.Pipe()
```

By default, the first connection (conn1) can only be used to receive data, whereas the second connection (conn2) can only be used to send data.

The connection objects can be made duplex or bidirectional.

This can be achieved by setting the "**duplex**" argument to the constructor to **True**.

For example:

```
...
# create a duplex pipe
conn1, conn2 = multiprocessing.Pipe(duplex=True)
```

In this case, both connections can be used to send and receive data.

Objects can be shared between processes using the Pipe.

The **Connection.send()** function can be used to send objects from one process to another.

The objects sent must be picklable.

For example:

```
...
# send an object
conn2.send('Hello world')
```

The **Connection.recv()** function can be used to receive objects in one process sent by another.

The objects received will be automatically un-pickled.

For example:

```
...
# receive an object
object = conn1.recv()
```

The function call will block until an object is received.

The status of the pipe can be checked via the **Connection.poll()** function.

This will return a boolean as to whether three is data to be received and read from the pipe.

For example:

```
...
# check if there is data to receive
if conn1.poll():
    # ...
```

A timeout can be set via the "**timeout**" argument. If specified, the call will block until data is available. If no data is available before the timeout number of seconds has elapsed, then the function will return.

For example:

```
...
# check if there is data to receive
if conn1.poll(timeout=5):
    # ...
```

You can learn more about multiprocessing pipes in the tutorial:

- Multiprocessing Pipe in Python (/multiprocessing-pipe-in-python)

How Do You Change The Process Start Method?

A start method is the technique used to start child processes in Python.

There are three start methods, they are:

- **spawn**: start a new Python process.
- **fork**: copy a Python process from an existing process.
- **forkserver**: new process from which future forked processes will be copied.

The start method can be set via the **multiprocessing.set_start_method()** function (https://docs.python.org/3/library/multiprocessing.html#multiprocessing.set_start_method).

The function takes a string argument indicating the start method to use.

This must be one of the methods returned from the **multiprocessing.get_all_start_methods()** for your platform.

For example:

```
...
# set the start method
multiprocessing.set_start_method('spawn')
```

It is a best practice, and required on most platforms that the start method be set first, prior to any other code, and to be done so within a **if __name__ == '__main__'** check called a protected entry point or top-level code environment (https://docs.python.org/3/library/__main__.html).

For example:

```
...
# protect the entry point
if __name__ == '__main__':
    # set the start method
    multiprocessing.set_start_method('spawn')
```

If the start method is not set within a protected entry point, it is possible to get a **RuntimeError** such as:

```
RuntimeError: context has already been set
```

It is also a good practice and required on some platforms that the start method only be set once.

You can learn more about setting the start method in the tutorial:

- Multiprocessing Start Methods (/multiprocessing-start-method)

How Do You Get The Process PID?

PID is an acronym for Process ID or Process identifier.

A process identifier is a unique number assigned to a process by the underlying operating system.

Each time a process is started, it is assigned a unique positive integer identifier and the identifier may be different each time the process is started.

The pod uniquely identifies one process among all active processes running on the system, managed by the operating system.

As such, the pid can be used to interact with a process, e.g. to send a signal to the process to interrupt or kill a process.

We can get the pid of a process via its **multiprocessing.Process** instance.

When we create a child process, we may hang on to the process instance.

Alternatively, we may get the process instance for the parent process via the **multiprocessing.parent_process()** function or for child process via the **multiprocessing.active_children()** function.

We may also get the process instance for the current process via the **multiprocessing.current_process()** function.

For example:

```
...
# get the process instance
process = multiprocessing.current_process()
```

Once we have the process instance, we get the pid via the **multiprocessing.Process.pid** attribute (https://docs.python.org/3/library/multiprocessing.html#multiprocessing.Process.pid).

For example:

```
...
# get the pid
pid = process.pid
```

We can also get the pid for the current process via the **os.getpid() function** (https://docs.python.org/3/library/os.html#os.getpid).

For example:

```
...
# get the pid for the current process
pid = os.getpid()
```

We may also get the pid for the parent process via the **os.getppid() function** (https://docs.python.org/3/library/os.html#os.getppid).

For example:

```
...
# get the pid for the parent process
pid = os.getppid()
```

Do We Need to Check for __main__?

Yes, when using the '**spawn**' start method.

It is common to get a **RuntimeError** when starting a new Process in Python.

The content of the error often looks as follows:

```
    An attempt has been made to start a new process before the
    current process has finished its bootstrapping phase.

    This probably means that you are not using fork to start your
    child processes and you have forgotten to use the proper idiom
    in the main module:

        if __name__ == '__main__':
            freeze_support()
            ...

    The "freeze_support()" line can be omitted if the program
    is not going to be frozen to produce an executable.
```

This will happen on Windows and MacOS where the default start method is '**spawn**'. It may also happen when you configure your program to use the 'spawn' start method on other platforms.

This is a common error and is easy to fix.

The fix involves checking if the code is running in the top-level environment and only then, attempt to start a new process.

This is a best practice.

The idiom for this fix, as stated in the message of the **RuntimeError**, is to use an if-statement and check if the name of the module is equal to the string '**__main__**'.

For example:

```
...
# check for top-level environment
if __name__ == '__main__':
    # ...
```

This is called "*protecting the entry point*" of the program.

Recall, that **__name__** is a variable that refers to the name of the module executing the current code.

Also, recall that '**__main__**' is the name of the top-level environment used to execute a Python program.

Using an if-statement to check if the module is the top-level environment and only starting child processes within that block will resolve the **RuntimeError**.

It means that if the Python file is imported, then the code protected by the if-statement will not run. It will only run when the Python file is run directly, e.g. is the top-level environment.

The if-statement idiom is required, even if the entry point of the program calls a function that itself starts a child process.

This is only used for child processes created by the main process. Child processes themselves can create child processes without this check.

You can learn more about protecting the entry point when using multiprocessing in the tutorial:

- Fix RuntimeError When Spawning a Child Process (/multiprocessing-spawn-runtimeerror)

How Do You Use Process Pools?

You can use process pools in Python via the **concurrent.futures.ProcessPoolExecutor** class (https://docs.python.org/3/library/concurrent.futures.html#concurrent.futures.ProcessPoolExecutor).

The **ProcessPoolExecutor** in Python provides a pool of reusable processes for executing ad hoc tasks.

You can submit tasks to the process pool by calling the **submit()** function and passing in the name of the function you wish to execute on another process.

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 process pools in Python via the tutorial:

- ProcessPoolExecutor in Python: The Complete Guide (https://superfastpython.com/processpoolexecutor-in-python/)

How to Log From Multiple Processes?

Logging is a way of tracking events within a program.

The **logging** module (https://docs.python.org/3/library/logging.html) provides infrastructure for logging within Python programs.

There are perhaps three main approaches to logging from multiple processes, they are:

- Use the logging module separately from each process.
- Use **multiprocessing.get_logger()**.
- Use custom process-safe logging.

We can log directly from each process using the logging module.

This requires that you configure the logger in each process, including the target (e.g. stream or file) and log level.

For example:

```
...
# create a logger
logger = logging.getLogger()
# log all messages, debug and up
logger.setLevel(logging.DEBUG)
```

Messages can then be logged directly.

For example:

```
...
# report a message
logging.info('Hello world.')
```

This is an easy approach in that it does not require anything sophisticated.

The downside is that you have to duplicate code in order to configure a separate logger for each new process.

The major limitation of this approach is that log messages may be lost or corrupted. This is because multiple processes will attempt to write log messages to the same target, e.g. stream or file.

I don't recommend this approach.

Alternately, the multiprocessing module has its own logger with the name "**multiprocessing**".

This logger is used within objects and functions within the multiprocessing module to log messages, such as debug messages that processes are running or have shutdown.

We can get this logger and use it for logging.

The logger used by the multiprocessing module can be acquired via the **multiprocessing.get_logger()** function. Once acquired, it can be configured, e.g. to specify a handler and a log level.

For example:

```
...
# get the multiprocessing logger
logger = get_logger()
# configure a stream handler
logger.addHandler(logging.StreamHandler())
# log all messages, debug and up
logger.setLevel(logging.DEBUG)
```

Messages logged by the multiprocessing module itself will appear in this log at the appropriate log level, e.g. there are many debug messages in the module.

This logger is not shared among processes and is not process-safe.

It is simply using the logger module directly as in the previous example, although for the "**multiprocessing**" logging namespace.

As such it has the same downside that the logger must be configured again within each child process and that log messages may be lost or corrupted.

I don't recommend this approach either.

Effective logging from multiple processes requires custom code and message passing between processes.

For example, a robust and easy to maintain approach involves sending all log messages to one process and configuring one process to be responsible for receiving log messages and storing them in a central location.

This can be achieved using a **multiprocessing.Queue** and a **logging.handlers.QueueHandler**.

For an example of how to log safely from multiple processes, see the tutorial:

- Multiprocessing Logging in Python (/multiprocessing-logging-in-python)

What is Process-Safe?

Process-safe refers to program code that can be executed free of concurrency errors by multiple processes concurrently.

It is the concept of "*thread-safe*" applied to processes, where processes are the unit of concurrency instead of threads.

Thread-safety is a major concern of concurrent programming using threads. This is because threads have shared memory within the process, meaning that concurrent access of the same data or variables can lead to race conditions.

Processes do not have direct shared memory and therefore are not subject to the same concerns of race conditions.

Nevertheless, processes do simulate shared memory using socket connections and files and may need to protect simulated shared program state or data from race conditions due to timing and concurrent modification.

Happily some tools used for inter-process communication provide some process-safety, such as queues.

You can learn more about process-safety in the tutorial:

- Process-Safe in Python (/process-safe-in-python)

Why Not Always Use Processes?

Thread-based concurrency in Python is limited.

Only a single thread is able to execute at a time.

This is because of the Global Interpreter Lock or GIL (https://wiki.python.org/moin/GlobalInterpreterLock) that requires that each thread acquire a lock on the interpreter before executing, preventing all other threads executing at the same time.

This means that although we may have tens, hundreds, or even thousands of concurrent threads in our Python application, only one thread may execute in parallel.

Process-based concurrency is not limited in the same way as thread-based concurrency.

Both threads and processes can execute concurrently (out of order), but only python processes are able to execute in parallel (simultaneously), not Python threads (with some caveats).

This means that if we want out Python code to run on all CPU cores and make the best use of our system hardware, we should use process-based concurrency.

If process-based concurrency offers true parallelism in Python, why not always use processes?

Firstly, only one thread can run at a time within a Python process under most situations, except:

1. When you are using a Python interpreter that does not use a GIL.
2. When you are performing IO-bound tasks that release the GIL.
3. When you are performing CPU-bound tasks that release the GIL.

For example, when you are reading or writing from a file or socket the GIL is released allowing multiple threads to run in parallel.

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.

Additionally, many CPU-bound tasks that are known to not rely on state in the Python interpreter will release the GIL, such as C-code in third-party libraries called from Python.

For example:

- When performing compression operations, e.g. in zlib.
- When calculating cryptographic hashes, e.g. in hashlib.
- When performing encoding/decoding images and video, e.g, in OpenCV.
- When performing matrix operations, e.g. in NumPy and SciPy.

- And many more.

Secondly, processes and process-based concurrency also have limitations compared to threads.

For example:

1. We may have thousands of threads, but perhaps only tens of processes.
2. Threads are small and fast, whereas processes are large and slow to create and start.
3. Threads can share data quickly and directly, whereas processes must pickle and transmit data to each other.

You can learn more about why we should not always use process-based concurrency in Python in the tutorial:

- Why Not Always Use Processes in Python (/why-not-always-use-processes-in-python)

Common Objections to Using Python Multiprocessing

Processes 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 processes in Python.

In this section, we review some of the common objections seen by developers when considering using the processes in their code.

What About the Global Interpreter Lock (GIL)?

The GIL is generally not relevant when using processes such as the Process class.

The Global Interpreter Lock, or GIL for short (https://wiki.python.org/moin/GlobalInterpreterLock), 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.

As such, the GIL is a consideration when using threads in Python such as the **threading.Thread** class. It is not a consideration when using the multiprocessing.Process class (unless you use additional threads within each task).

Are Python Processes “Real Processes”?

Yes.

Python makes use of real system-level processes, also called spawning processes or forking processes, a capability provided by modern operating systems like Windows, Linux, and MacOS.

Are Python Processes Buggy?

No.

Python processes are not buggy.

Python processes are a first-class capability of the Python platform and have been for a very long time.

Isn’t Python a Bad Choice for Concurrency?

No.

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 Use Threads?

Threads and processes are quite different and choosing one over the other must be quite 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 main 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 in Python, 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.

You can learn more about threads in the tutorial:

- Threading in Python: The Complete Guide (https://superfastpython.com/threading-in-python)

Why Not Use AsyncIO?

AsyncIO can be an alternative to using a **threading.Thread**, but is probably not a good alternative for the multiprocessing.Process class.

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, which can be challenging for beginners.

When using the **multiprocessing.Process** class, you are typically executing CPU-bound tasks, which are not appropriate when using the AsyncIO module.

Further Reading

Books

Some books and materials dedicated to helping you learn the multiprocessing module API more fully include:

- Multiprocessing Module API Cheat Sheet (https://superfastpython.gumroad.com/l/rlgsc)
- Multiprocessing API Interview Questions (https://superfastpython.gumroad.com/l/pmiq)
- Multiprocessing Jump-Start (https://superfastpython.com/pmj-further-reading) (my 7-day course)

Other Books

Some more general books that have small sections on multiprocessing include:

- Effective Python (https://amzn.to/3GpopJ1), Brett Slatkin, 2019.
  - See: **Chapter 7: Concurrency and Parallelism**

- High Performance Python (https://amzn.to/3wRD5MX), Ian Ozsvald and Micha Gorelick, 2020.
  - See: **Chapter 9: The multiprocessing Module**
- Python in a Nutshell (https://amzn.to/3m7SLGD), Alex Martelli, et al., 2017.
  - See: **Chapter: 14: Threads and Processes**

APIs

- threading — Thread-based parallelism (https://docs.python.org/3/library/threading.html)
- Multiprocessing — Process-based parallelism (https://docs.python.org/3/library/multiprocessing.html)
- logging — Logging facility for Python (https://docs.python.org/3/library/logging.html)

References

- Thread (computing), Wikipedia (https://en.wikipedia.org/wiki/Thread_(computing)).
- Process (computing), Wikipedia (https://en.wikipedia.org/wiki/Process_(computing)).

Conclusions

This is a large guide, and you have discovered in great detail how multiprocessing works in Python and how to best use processes in your project.

Did you find this guide useful?
I'd love to know, please share a kind word in the comments below.

Have you used Processes on a project?
I'd love to hear about it, please let me know in the comments.

Do you have any questions?
Leave your question in a comment below and I will reply fast with my best advice.

Comments