

# PyMOTW

[Home](#)
[Blog](#)
[The Book](#)
[About](#)
[Site Index](#)

If you find this information useful, consider picking up a copy of my book, *The Python Standard Library By Example*.

## multiprocessing Basics

The simplest way to spawn a second is to instantiate a **Process** object with a target function and call **start()** to let it begin working.

```
import multiprocessing

def worker():
    """worker function"""
    print 'Worker'
    return

if __name__ == '__main__':
    jobs = []
    for i in range(5):
        p = multiprocessing.Process(target=worker)
        jobs.append(p)
        p.start()
```

The output includes the word “Worker” printed five times, although it may not be entirely clean depending on the order of execution.

```
$ python multiprocessing_simple.py

Worker
Worker
Worker
Worker
Worker
```

It is usually more useful to be able to spawn a process with arguments to tell it what work to do. Unlike with **threading**, to pass arguments to a **multiprocessing Process** the argument must be able to be serialized using **pickle**. This example passes each worker a number so the output is a little more interesting.

```
import multiprocessing

def worker(num):
    """thread worker function"""
    print 'Worker:', num
    return

if __name__ == '__main__':
    jobs = []
    for i in range(5):
        p = multiprocessing.Process(target=worker, args=(i,))
        jobs.append(p)
        p.start()
```

## Page Contents

- [multiprocessing Basics](#)
  - [Importable Target Functions](#)
  - [Determining the Current Process](#)
  - [Daemon Processes](#)
  - [Waiting for Processes](#)
  - [Terminating Processes](#)
  - [Process Exit Status](#)
  - [Logging](#)
  - [Subclassing Process](#)

## Navigation

### Table of Contents

**Previous:** [multiprocessing – Manage processes like threads](#)

**Next:** [Communication Between Processes](#)

## This Page

[Show Source](#)

## Examples

The output from all the example programs from PyMOTW has been generated with Python 2.7.8, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

If you are looking for examples that work under Python 3, please refer to the [PyMOTW-3](#) section of the site.

The integer argument is now included in the message printed by each worker:

```
$ python multiprocessing_simple.py

Worker: 0
Worker: 1
Worker: 2
Worker: 3
Worker: 4
```

## Importable Target Functions

One difference between the **threading** and **multiprocessing** examples is the extra protection for `__main__` used in the **multiprocessing** examples. Due to the way the new processes are started, the child process needs to be able to import the script containing the target function. Wrapping the main part of the application in a check for `__main__` ensures that it is not run recursively in each child as the module is imported. Another approach is to import the target function from a separate script.

For example, this main program:

```
import multiprocessing
import multiprocessing_import_worker

if __name__ == '__main__':
    jobs = []
    for i in range(5):
        p = multiprocessing.Process(target=worker, args=(i,))
        jobs.append(p)
        p.start()
```

uses this worker function, defined in a separate module:

```
def worker():
    """worker function"""
    print 'Worker'
    return
```

and produces output like the first example above:

```
$ python multiprocessing_import_worker.py

Worker
Worker
Worker
Worker
Worker
```

## Determining the Current Process



Now available for Python 3!



Buy the book!

Passing arguments to identify or name the process is cumbersome, and unnecessary. Each **Process** instance has a name with a default value that can be changed as the process is created. Naming processes is useful for keeping track of them, especially in applications with multiple types of processes running simultaneously.

```
import multiprocessing
import time

def worker():
    name = multiprocessing.current_process().name
    print name, 'Starting'
    time.sleep(2)
    print name, 'Exiting'

def my_service():
    name = multiprocessing.current_process().name
    print name, 'Starting'
    time.sleep(3)
    print name, 'Exiting'

if __name__ == '__main__':
    service = multiprocessing.Process(target=my_service)
    worker_1 = multiprocessing.Process(target=worker)
    worker_2 = multiprocessing.Process(target=worker)

    worker_1.start()
    worker_2.start()
    service.start()
```

The debug output includes the name of the current process on each line. The lines with Process-3 in the name column correspond to the unnamed process worker\_1.

```
$ python multiprocessing_names.py

worker 1 Starting
worker 1 Exiting
Process-3 Starting
Process-3 Exiting
my_service Starting
my_service Exiting
```

## Daemon Processes

By default the main program will not exit until all of the children have exited. There are times when starting a background process that runs without blocking the main program from exiting is useful, such as in services where there may not be an easy way to interrupt the worker, or where letting it die in the middle of its work does not lose or corrupt data (for example, a task that generates “heart beats” for a service monitoring tool).

To mark a process as a daemon, set its **daemon** attribute with a boolean value. The default is for processes to not be daemons, so passing True turns the daemon mode on.

```
import multiprocessing
import time
import sys

def daemon():
    p = multiprocessing.current_process()
    print 'Starting:', p.name, p.pid
    sys.stdout.flush()
    time.sleep(2)
    print 'Exiting :', p.name, p.pid
    sys.stdout.flush()

def non_daemon():
    p = multiprocessing.current_process()
    print 'Starting:', p.name, p.pid
    sys.stdout.flush()
    print 'Exiting :', p.name, p.pid
    sys.stdout.flush()

if __name__ == '__main__':
    d = multiprocessing.Process(target=daemon)
    d.daemon = True

    n = multiprocessing.Process(target=non_daemon)
    n.daemon = False

    d.start()
    time.sleep(1)
    n.start()
```

The output does not include the “Exiting” message from the daemon process, since all of the non-daemon processes (including the main program) exit before the daemon process wakes up from its 2 second sleep.

```
$ python multiprocessing_daemon.py

Starting: daemon 13866
Starting: non-daemon 13867
Exiting : non-daemon 13867
```

The daemon process is terminated automatically before the main program exits, to avoid leaving orphaned processes running. You can verify this by looking for the process id value printed when you run the program, and then checking for that process with a command like `ps`.

## Waiting for Processes

To wait until a process has completed its work and exited, use the `join()` method.

```

import multiprocessing
import time
import sys

def daemon():
    print 'Starting:', multiprocessing
    time.sleep(2)
    print 'Exiting :', multiprocessing

def non_daemon():
    print 'Starting:', multiprocessing
    print 'Exiting :', multiprocessing

if __name__ == '__main__':
    d = multiprocessing.Process(r
    d.daemon = True

    n = multiprocessing.Process(r
    n.daemon = False

    d.start()
    time.sleep(1)
    n.start()

    d.join()
    n.join()

```

Since the main process waits for the daemon to exit using `join()`, the “Exiting” message is printed this time.

```

$ python multiprocessing_daemon_

Starting: non-daemon
Exiting : non-daemon
Starting: daemon
Exiting : daemon

```

By default, `join()` blocks indefinitely. It is also possible to pass a timeout argument (a float representing the number of seconds to wait for the process to become inactive). If the process does not complete within the timeout period, `join()` returns anyway.

```

import multiprocessing
import time
import sys

def daemon():
    print 'Starting:', multiprocessing
    time.sleep(2)
    print 'Exiting :', multiprocessing

def non_daemon():
    print 'Starting:', multiprocessing
    print 'Exiting :', multiprocessing

if __name__ == '__main__':
    d = multiprocessing.Process(r
    d.daemon = True

```

```

n = multiprocessing.Process(t
n.daemon = False

d.start()
n.start()

d.join(1)
print 'd.is_alive()', d.is_al
n.join()

```

Since the timeout passed is less than the amount of time the daemon sleeps, the process is still “alive” after `join()` returns.

```

$ python multiprocessing_daemon_

Starting: non-daemon
Exiting : non-daemon
d.is_alive() True

```

## Terminating Processes

Although it is better to use the *poison pill* method of signaling to a process that it should exit (see [Passing Messages to Processes](#)), if a process appears hung or deadlocked it can be useful to be able to kill it forcibly. Calling `terminate()` on a process object kills the child process.

```

import multiprocessing
import time

def slow_worker():
    print 'Starting worker'
    time.sleep(0.1)
    print 'Finished worker'

if __name__ == '__main__':
    p = multiprocessing.Process(t
    print 'BEFORE:', p, p.is_aliv

    p.start()
    print 'DURING:', p, p.is_aliv

    p.terminate()
    print 'TERMINATED:', p, p.is_

    p.join()
    print 'JOINED:', p, p.is_aliv

```

**Note:** It is important to `join()` the process after terminating it in order to give the background machinery time to update the status of the object to reflect the termination.

```

$ python multiprocessing_terminat

BEFORE: <Process(Process-1, initi

```

```
DURING: <Process(Process-1, start
TERMINATED: <Process(Process-1, s
JOINED: <Process(Process-1, stopp
```

## Process Exit Status

The status code produced when the process exits can be accessed via the **exitcode** attribute.

For **exitcode** values

- == 0 - no error was produced
- > 0 - the process had an error, and exited with that code
- < 0 - the process was killed with a signal of  $-1 * \text{exitcode}$

```
import multiprocessing
import sys
import time

def exit_error():
    sys.exit(1)

def exit_ok():
    return

def return_value():
    return 1

def raises():
    raise RuntimeError('There was

def terminated():
    time.sleep(3)

if __name__ == '__main__':
    jobs = []
    for f in [exit_error, exit_ok,
              print 'Starting process f
    j = multiprocessing.Proce
    jobs.append(j)
    j.start()

    jobs[-1].terminate()

    for j in jobs:
        j.join()
        print '%s.exitcode = %s'
```

Processes that raise an exception automatically get an **exitcode** of 1.

```
$ python multiprocessing_exitcode

Starting process for exit_error
Starting process for exit_ok
Starting process for return_value
Starting process for raises
Starting process for terminated
Process raises:
Traceback (most recent call last)
```

```

File "/Library/Frameworks/Python
2.7/multiprocessing/process.py",
    self.run()
File "/Library/Frameworks/Python
2.7/multiprocessing/process.py",
    self._target(*self._args, **s
File "multiprocessing_exitcode.
    raise RuntimeError('There was
RuntimeError: There was an error!
exit_error.exitcode = 1
exit_ok.exitcode = 0
return_value.exitcode = 0
raises.exitcode = 1
terminated.exitcode = -15

```

## Logging

When debugging concurrency issues, it can be useful to have access to the internals of the objects provided by `multiprocessing`. There is a convenient module-level function to enable logging called `log_to_stderr()`. It sets up a logger object using `logging` and adds a handler so that log messages are sent to the standard error channel.

```

import multiprocessing
import logging
import sys

def worker():
    print 'Doing some work'
    sys.stdout.flush()

if __name__ == '__main__':
    multiprocessing.log_to_stderr()
    p = multiprocessing.Process(t
    p.start()
    p.join()

```

By default the logging level is set to `NOTSET` so no messages are produced. Pass a different level to initialize the logger to the level of detail you want.

```

$ python multiprocessing_log_to_s

[INFO/Process-1] child process ca
Doing some work
[INFO/Process-1] process shutting
[DEBUG/Process-1] running all "at
[DEBUG/Process-1] running the ren
[INFO/Process-1] process exiting
[INFO/MainProcess] process shutti
[DEBUG/MainProcess] running all '
[DEBUG/MainProcess] running the r

```

To manipulate the logger directly (change its level setting or add handlers), use `get_logger()`.



```
import multiprocessing
import logging
import sys

def worker():
    print 'Doing some work'
    sys.stdout.flush()

if __name__ == '__main__':
    multiprocessing.log_to_stderr()
    logger = multiprocessing.get_logger()
    logger.setLevel(logging.INFO)
    p = multiprocessing.Process(target=worker)
    p.start()
    p.join()
```

The logger can also be configured through the **logging** configuration file API, using the name `multiprocessing`.

```
$ python multiprocessing_get_logger.py

[INFO/Process-1] child process created
Doing some work
[INFO/Process-1] process shutting down
[INFO/Process-1] process exiting
[INFO/MainProcess] process shutting down
```

## Subclassing Process

Although the simplest way to start a job in a separate process is to use `Process` and pass a target function, it is also possible to use a custom subclass.

```
import multiprocessing

class Worker(multiprocessing.Process):

    def run(self):
        print 'In %s' % self.name
        return


if __name__ == '__main__':
    jobs = []
    for i in range(5):
        p = Worker()
        jobs.append(p)
        p.start()
    for j in jobs:
        j.join()
```

The derived class should override `run()` to do its work.

```
$ python multiprocessing_subclass.py

In Worker-1
In Worker-2
In Worker-3
```

```
In Worker-4  
In Worker-5
```

© Copyright Doug Hellmann. |  | Last updated on Jul 11, 2020. | Created using Sphinx. | Design based on "Leaves"  
by SmallPark | 