multiprocessing Basics

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

```
# multiprocessing_simple.py

import multiprocessing


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


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 come out entirely clean, depending on the order of execution, because each process is competing for access to the output stream.

```
$ python3 multiprocessing_simple.py

Worker
Worker
Worker
Worker
Worker
```

It usually more useful to be able to spawn a process with arguments to tell it what work to do. Unlike with `threading`, in order to pass arguments to a `multiprocessing` `Process` the arguments must be able to be serialized using `pickle`. This example passes each worker a number to be printed.

```
# multiprocessing_simpleargs.py

import multiprocessing


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


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

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

```
$ python3 multiprocessing_simpleargs.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, `multiprocessing_import_main.py` uses a worker function defined in a second module.

```
# multiprocessing_import_main.py

import multiprocessing
import multiprocessing_import_worker

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

The worker function is defined in `multiprocessing_import_worker.py`.

```
# multiprocessing_import_worker.py

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

Calling the main program produces output similar to the first example.

```
$ python3 multiprocessing_import_main.py

Worker
Worker
Worker
Worker
Worker
```

Determining the Current Process

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.

```
# multiprocessing_names.py

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(
        name='my_service',
        target=my_service,
    )
    worker_1 = multiprocessing.Process(
        name='worker 1',
        target=worker,
    )
    worker_2 = multiprocessing.Process(  # default name
        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_2`.

```
$ python3 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 to `True`. The default is for processes to not be daemons.

```
# multiprocessing_daemon.py

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(
        name='daemon',
        target=daemon,
    )
    d.daemon = True

    n = multiprocessing.Process(
        name='non-daemon',
        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 two second sleep.

```
$ python3 multiprocessing_daemon.py

Starting: daemon 36250
Starting: non-daemon 36256
Exiting : non-daemon 36256
```

The daemon process is terminated automatically before the main program exits, which avoids leaving orphaned processes running. This can be verified by looking for the process id value printed when the program runs, 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.

```
# multiprocessing_daemon_join.py
```

```
import multiprocessing
import time
import sys


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


def non_daemon():
    name = multiprocessing.current_process().name
    print('Starting:', name)
    print('Exiting :', name)


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

    n = multiprocessing.Process(
        name='non-daemon',
        target=non_daemon,
    )
    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.

```
$ python3 multiprocessing_daemon_join.py

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.

```
# multiprocessing_daemon_join_timeout.py

import multiprocessing
import time
import sys


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


def non_daemon():
    name = multiprocessing.current_process().name
    print('Starting:', name)
    print('Exiting :', name)


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

    n = multiprocessing.Process(
        name='non-daemon',
        target=non_daemon,
    )
    n.daemon = False

    d.start()
    n.start()

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

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

```
$ python3 multiprocessing_daemon_join_timeout.py

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, later in this chapter), 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.

```python
# multiprocessing_terminate.py

import multiprocessing
import time


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


if __name__ == '__main__':
    p = multiprocessing.Process(target=slow_worker)
    print('BEFORE:', p, p.is_alive())

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

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

    p.join()
    print('JOINED:', p, p.is_alive())
```

Note

It is important to `join()` the process after terminating it in order to give the process management code time to update the status of the object to reflect the termination.

```
$ python3 multiprocessing_terminate.py

BEFORE: <Process(Process-1, initial)> False
DURING: <Process(Process-1, started)> True
TERMINATED: <Process(Process-1, started)> True
JOINED: <Process(Process-1, stopped[SIGTERM])> False
```

Process Exit Status

The status code produced when the process exits can be accessed via the `exitcode` attribute. The ranges allowed are listed in the table below.

Multiprocessing Exit Codes

Exit Code	Meaning
`== 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 * exitcode`

```python
# multiprocessing_exitcode.py

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 an error!')


def terminated():
    time.sleep(3)


if __name__ == '__main__':
    jobs = []
    funcs = [
        exit_error,
        exit_ok,
        return_value,
        raises,
        terminated,
    ]
    for f in funcs:
        print('Starting process for', f.__name__)
        j = multiprocessing.Process(target=f, name=f.__name__)
        jobs.append(j)
        j.start()

    jobs[-1].terminate()

    for j in jobs:
        j.join()
        print('{:>15}.exitcode = {}'.format(j.name, j.exitcode))
```

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

```
$ python3 multiprocessing_exitcode.py

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 ".../lib/python3.7/multiprocessing/process.py", line 297,
in _bootstrap
    self.run()
  File ".../lib/python3.7/multiprocessing/process.py", line 99,
in run
    self._target(*self._args, **self._kwargs)
  File "multiprocessing_exitcode.py", line 28, in raises
    raise RuntimeError('There was an error!')
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.

```
# multiprocessing_log_to_stderr.py

import multiprocessing
import logging
import sys


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


if __name__ == '__main__':
    multiprocessing.log_to_stderr(logging.DEBUG)
    p = multiprocessing.Process(target=worker)
    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 desired.

```
$ python3 multiprocessing_log_to_stderr.py

[INFO/Process-1] child process calling self.run()
Doing some work
[INFO/Process-1] process shutting down
[DEBUG/Process-1] running all "atexit" finalizers with priority
>= 0
[DEBUG/Process-1] running the remaining "atexit" finalizers
[INFO/Process-1] process exiting with exitcode 0
[INFO/MainProcess] process shutting down
[DEBUG/MainProcess] running all "atexit" finalizers with
priority >= 0
[DEBUG/MainProcess] running the remaining "atexit" finalizers
```

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

```
# multiprocessing_get_logger.py

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`”.

```
$ python3 multiprocessing_get_logger.py

[INFO/Process-1] child process calling self.run()
Doing some work
[INFO/Process-1] process shutting down
[INFO/Process-1] process exiting with exitcode 0
[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.

```python
# multiprocessing_subclass.py

import multiprocessing


class Worker(multiprocessing.Process):

    def run(self):
        print('In {}'.format(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.

```
$ python3 multiprocessing_subclass.py

In Worker-1
In Worker-2
In Worker-3
In Worker-4
In Worker-5
```

multiprocessing — Manage Processes Like Threads

Passing Messages to Processes

Quick Links

This page was last updated 2018-12-09.

Navigation

multiprocessing — Manage Processes Like Threads
Passing Messages to Processes

The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?

This Site

Module Index
Index

Other Writing

Blog
The Python Standard Library By Example