

Passing Messages to Processes

As with threads, a common use pattern for multiple processes is to divide a job up among several workers to run in parallel. Effective use of multiple processes usually requires some communication between them, so that work can be divided and results can be aggregated. A simple way to communicate between processes with multiprocessing is to use a Queue to pass messages back and forth. Any object that can be serialized with [pickle](#) can pass through a Queue.

```
# multiprocessing_queue.py

import multiprocessing

class MyFancyClass:

    def __init__(self, name):
        self.name = name

    def do_something(self):
        proc_name = multiprocessing.current_process().name
        print('Doing something fancy in {} for {}'.format(
            proc_name, self.name))

def worker(q):
    obj = q.get()
    obj.do_something()

if __name__ == '__main__':
    queue = multiprocessing.Queue()

    p = multiprocessing.Process(target=worker, args=(queue,))
    p.start()

    queue.put(MyFancyClass('Fancy Dan'))

    # Wait for the worker to finish
    queue.close()
    queue.join_thread()
    p.join()
```

This short example only passes a single message to a single worker, then the main process waits for the worker to finish.

```
$ python3 multiprocessing_queue.py

Doing something fancy in Process-1 for Fancy Dan!
```

A more complex example shows how to manage several workers consuming data from a `JoinableQueue` and passing results back to the parent process. The *poison pill* technique is used to stop the workers. After setting up the real tasks, the main program adds one “stop” value per worker to the job queue. When a worker encounters the special value, it breaks out of its processing loop. The main process uses the task queue’s `join()` method to wait for all of the tasks to finish before processing the results.

```
# multiprocessing_producer_consumer.py

import multiprocessing
import time

class Consumer(multiprocessing.Process):

    def __init__(self, task_queue, result_queue):
```

```

def __init__(self, task_queue, result_queue):
    multiprocessing.Process.__init__(self)
    self.task_queue = task_queue
    self.result_queue = result_queue

def run(self):
    proc_name = self.name
    while True:
        next_task = self.task_queue.get()
        if next_task is None:
            # Poison pill means shutdown
            print('{}: Exiting'.format(proc_name))
            self.task_queue.task_done()
            break
        print('{}: {}'.format(proc_name, next_task))
        answer = next_task()
        self.task_queue.task_done()
        self.result_queue.put(answer)

class Task:

    def __init__(self, a, b):
        self.a = a
        self.b = b

    def __call__(self):
        time.sleep(0.1) # pretend to take time to do the work
        return '{self.a} * {self.b} = {product}'.format(
            self=self, product=self.a * self.b)

    def __str__(self):
        return '{self.a} * {self.b}'.format(self=self)

if __name__ == '__main__':
    # Establish communication queues
    tasks = multiprocessing.JoinableQueue()
    results = multiprocessing.Queue()

    # Start consumers
    num_consumers = multiprocessing.cpu_count() * 2
    print('Creating {} consumers'.format(num_consumers))
    consumers = [
        Consumer(tasks, results)
        for i in range(num_consumers)
    ]
    for w in consumers:
        w.start()

    # Enqueue jobs
    num_jobs = 10
    for i in range(num_jobs):
        tasks.put(Task(i, i))

    # Add a poison pill for each consumer
    for i in range(num_consumers):
        tasks.put(None)

    # Wait for all of the tasks to finish
    tasks.join()

    # Start printing results
    while num_jobs:
        result = results.get()
        print('Result:', result)
        num_jobs -= 1

```

Although the jobs enter the queue in order, their execution is parallelized so there is no guarantee about the order they will be completed.

```
$ python3 -u multiprocessing/producer_consumer.py
```

```
$ python3 -u multiprocessing_producer_consumer.py
Creating 8 consumers
Consumer-1: 0 * 0
Consumer-2: 1 * 1
Consumer-3: 2 * 2
Consumer-4: 3 * 3
Consumer-5: 4 * 4
Consumer-6: 5 * 5
Consumer-7: 6 * 6
Consumer-8: 7 * 7
Consumer-3: 8 * 8
Consumer-7: 9 * 9
Consumer-4: Exiting
Consumer-1: Exiting
Consumer-2: Exiting
Consumer-5: Exiting
Consumer-6: Exiting
Consumer-8: Exiting
Consumer-7: Exiting
Consumer-3: Exiting
Result: 6 * 6 = 36
Result: 2 * 2 = 4
Result: 3 * 3 = 9
Result: 0 * 0 = 0
Result: 1 * 1 = 1
Result: 7 * 7 = 49
Result: 4 * 4 = 16
Result: 5 * 5 = 25
Result: 8 * 8 = 64
Result: 9 * 9 = 81
```

Signaling between Processes

The Event class provides a simple way to communicate state information between processes. An event can be toggled between set and unset states. Users of the event object can wait for it to change from unset to set, using an optional timeout value.

```
# multiprocessing_event.py

import multiprocessing
import time

def wait_for_event(e):
    """Wait for the event to be set before doing anything"""
    print('wait_for_event: starting')
    e.wait()
    print('wait_for_event: e.is_set()->', e.is_set())

def wait_for_event_timeout(e, t):
    """Wait t seconds and then timeout"""
    print('wait_for_event_timeout: starting')
    e.wait(t)
    print('wait_for_event_timeout: e.is_set()->', e.is_set())

if __name__ == '__main__':
    e = multiprocessing.Event()
    w1 = multiprocessing.Process(
        name='block',
        target=wait_for_event,
        args=(e,),
    )
    w1.start()

    w2 = multiprocessing.Process(
        name='nonblock',
        target=wait_for_event_timeout,
```

```

        args=(e, 2),
    )
    w2.start()

    print('main: waiting before calling Event.set()')
    time.sleep(3)
    e.set()
    print('main: event is set')

```

When `wait()` times out it returns without an error. The caller is responsible for checking the state of the event using `is_set()`.

```

$ python3 -u multiprocessing_event.py

main: waiting before calling Event.set()
wait_for_event: starting
wait_for_event_timeout: starting
wait_for_event_timeout: e.is_set()-> False
main: event is set
wait_for_event: e.is_set()-> True

```

Controlling Access to Resources

In situations when a single resource needs to be shared between multiple processes, a Lock can be used to avoid conflicting accesses.

```

# multiprocessing_lock.py

import multiprocessing
import sys

def worker_with(lock, stream):
    with lock:
        stream.write('Lock acquired via with\n')

def worker_no_with(lock, stream):
    lock.acquire()
    try:
        stream.write('Lock acquired directly\n')
    finally:
        lock.release()

lock = multiprocessing.Lock()
w = multiprocessing.Process(
    target=worker_with,
    args=(lock, sys.stdout),
)
nw = multiprocessing.Process(
    target=worker_no_with,
    args=(lock, sys.stdout),
)

w.start()
nw.start()

w.join()
nw.join()

```

In this example, the messages printed to the console may be jumbled together if the two processes do not synchronize their access of the output stream with the lock.

```

$ python3 multiprocessing_lock.py

Lock acquired via with
Lock acquired directly

```

Synchronizing Operations

Condition objects can be used to synchronize parts of a workflow so that some run in parallel but others run sequentially, even if they are in separate processes.

```
# multiprocessing_condition.py

import multiprocessing
import time

def stage_1(cond):
    """perform first stage of work,
    then notify stage_2 to continue
    """
    name = multiprocessing.current_process().name
    print('Starting', name)
    with cond:
        print('{} done and ready for stage 2'.format(name))
        cond.notify_all()

def stage_2(cond):
    """wait for the condition telling us stage_1 is done"""
    name = multiprocessing.current_process().name
    print('Starting', name)
    with cond:
        cond.wait()
        print('{} running'.format(name))

if __name__ == '__main__':
    condition = multiprocessing.Condition()
    s1 = multiprocessing.Process(name='s1',
                                target=stage_1,
                                args=(condition,))

    s2_clients = [
        multiprocessing.Process(
            name='stage_2[{}]'.format(i),
            target=stage_2,
            args=(condition,),
        )
        for i in range(1, 3)
    ]

    for c in s2_clients:
        c.start()
        time.sleep(1)
    s1.start()

    s1.join()
    for c in s2_clients:
        c.join()
```

In this example, two process run the second stage of a job in parallel, but only after the first stage is done.

```
$ python3 -u multiprocessing_condition.py

Starting stage_2[1]
Starting stage_2[2]
Starting s1
s1 done and ready for stage 2
stage_2[1] running
stage_2[2] running
```

Controlling Concurrent Access to Resources

Sometimes it is useful to allow more than one worker access to a resource at a time, while still limiting the overall number. For example, a connection pool might support a fixed number of simultaneous connections, or a network application might support a fixed number of concurrent downloads. A Semaphore is one way to manage those connections.

```
# multiprocessing_semaphore.py

import random
import multiprocessing
import time

class ActivePool:

    def __init__(self):
        super(ActivePool, self).__init__()
        self.mgr = multiprocessing.Manager()
        self.active = self.mgr.list()
        self.lock = multiprocessing.Lock()

    def makeActive(self, name):
        with self.lock:
            self.active.append(name)

    def makeInactive(self, name):
        with self.lock:
            self.active.remove(name)

    def __str__(self):
        with self.lock:
            return str(self.active)

def worker(s, pool):
    name = multiprocessing.current_process().name
    with s:
        pool.makeActive(name)
        print('Activating {} now running {}'.format(
            name, pool))
        time.sleep(random.random())
        pool.makeInactive(name)

if __name__ == '__main__':
    pool = ActivePool()
    s = multiprocessing.Semaphore(3)
    jobs = [
        multiprocessing.Process(
            target=worker,
            name=str(i),
            args=(s, pool),
        )
        for i in range(10)
    ]

    for j in jobs:
        j.start()

    while True:
        alive = 0
        for j in jobs:
            if j.is_alive():
                alive += 1
                j.join(timeout=0.1)
                print('Now running {}'.format(pool))
        if alive == 0:
            # all done
            break
```

In this example, the ActivePool class simply serves as a convenient way to track which processes are running at a given moment. A real resource pool would probably allocate a connection or some other value to the newly active process, and

reclaim the value when the task is done. Here, the pool is just used to hold the names of the active processes to show that only three are running concurrently.

```
$ python3 -u multiprocessing_semaphore.py

Activating 0 now running ['0', '1', '2']
Activating 1 now running ['0', '1', '2']
Activating 2 now running ['0', '1', '2']
Now running ['0', '1', '2']
Now running ['0', '1', '2']
Now running ['0', '1', '2']
Now running ['0', '1', '2']
Activating 3 now running ['0', '1', '3']
Activating 4 now running ['1', '3', '4']
Activating 6 now running ['1', '4', '6']
Now running ['1', '4', '6']
Now running ['1', '4', '6']
Activating 5 now running ['1', '4', '5']
Now running ['1', '4', '5']
Now running ['1', '4', '5']
Now running ['1', '4', '5']
Activating 8 now running ['4', '5', '8']
Now running ['4', '5', '8']
Now running ['4', '5', '8']
Now running ['4', '5', '8']
Now running ['4', '5', '8']
Now running ['4', '5', '8']
Activating 7 now running ['5', '8', '7']
Now running ['5', '8', '7']
Activating 9 now running ['8', '7', '9']
Now running ['8', '7', '9']
Now running ['8', '9']
Now running ['8', '9']
Now running ['9']
Now running ['9']
Now running ['9']
Now running ['9']
Now running []
```

Managing Shared State

In the previous example, the list of active processes is maintained centrally in the `ActivePool` instance via a special type of list object created by a `Manager`. The `Manager` is responsible for coordinating shared information state between all of its users.

```
# multiprocessing_manager_dict.py

import multiprocessing
import pprint

def worker(d, key, value):
    d[key] = value

if __name__ == '__main__':
    mgr = multiprocessing.Manager()
    d = mgr.dict()
    jobs = [
        multiprocessing.Process(
            target=worker,
            args=(d, i, i * 2),
        )
        for i in range(10)
    ]
    for j in jobs:
        j.start()
    for j in jobs:
        j.join()
    print('Results:', d)
```

By creating the list through the manager, it is shared and updates are seen in all processes. Dictionaries are also supported.

```
$ python3 multiprocessing_manager_dict.py
```

```
Results: {0: 0, 1: 2, 2: 4, 3: 6, 4: 8, 5: 10, 6: 12, 7: 14,
8: 16, 9: 18}
```

Shared Namespaces

In addition to dictionaries and lists, a Manager can create a shared Namespace.

```
# multiprocessing_namespaces.py

import multiprocessing

def producer(ns, event):
    ns.value = 'This is the value'
    event.set()

def consumer(ns, event):
    try:
        print('Before event: {}'.format(ns.value))
    except Exception as err:
        print('Before event, error:', str(err))
    event.wait()
    print('After event:', ns.value)

if __name__ == '__main__':
    mgr = multiprocessing.Manager()
    namespace = mgr.Namespace()
    event = multiprocessing.Event()
    p = multiprocessing.Process(
        target=producer,
        args=(namespace, event),
    )
    c = multiprocessing.Process(
        target=consumer,
        args=(namespace, event),
    )

    c.start()
    p.start()

    c.join()
    p.join()
```

Any named value added to the Namespace is visible to all of the clients that receive the Namespace instance.

```
$ python3 multiprocessing_namespaces.py
```

```
Before event, error: 'Namespace' object has no attribute 'value'
After event: This is the value
```

It is important to know that updates to the contents of mutable values in the namespace are not propagated automatically.

```
# multiprocessing_namespaces_mutable.py

import multiprocessing

def producer(ns, event):
    # DOES NOT UPDATE GLOBAL VALUE!
    ns.my_list.append('This is the value')
    event.set()
```



```

def consumer(ns, event):
    print('Before event:', ns.my_list)
    event.wait()
    print('After event :', ns.my_list)

if __name__ == '__main__':
    mgr = multiprocessing.Manager()
    namespace = mgr.Namespace()
    namespace.my_list = []

    event = multiprocessing.Event()
    p = multiprocessing.Process(
        target=producer,
        args=(namespace, event),
    )
    c = multiprocessing.Process(
        target=consumer,
        args=(namespace, event),
    )

    c.start()
    p.start()

    c.join()
    p.join()

```

To update the list, attach it to the namespace object again.

```

$ python3 multiprocessing_namespaces_mutable.py

Before event: []
After event : []

```

Process Pools

The Pool class can be used to manage a fixed number of workers for simple cases where the work to be done can be broken up and distributed between workers independently. The return values from the jobs are collected and returned as a list. The pool arguments include the number of processes and a function to run when starting the task process (invoked once per child).

```

# multiprocessing_pool.py

import multiprocessing

def do_calculation(data):
    return data * 2

def start_process():
    print('Starting', multiprocessing.current_process().name)

if __name__ == '__main__':
    inputs = list(range(10))
    print('Input  :', inputs)

    builtin_outputs = map(do_calculation, inputs)
    print('Built-in:', builtin_outputs)

    pool_size = multiprocessing.cpu_count() * 2
    pool = multiprocessing.Pool(
        processes=pool_size,
        initializer=start_process,
    )
    pool_outputs = pool.map(do_calculation, inputs)

```

```

pool.close() # no more tasks
pool.join() # wrap up current tasks

print('Pool      : ', pool_outputs)

```

The result of the `map()` method is functionally equivalent to the built-in `map()`, except that individual tasks run in parallel. Since the pool is processing its inputs in parallel, `close()` and `join()` can be used to synchronize the main process with the task processes to ensure proper cleanup.

```

$ python3 multiprocessing_pool.py

Input      : [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Built-in: <map object at 0x1007b2be0>
Starting ForkPoolWorker-3
Starting ForkPoolWorker-4
Starting ForkPoolWorker-5
Starting ForkPoolWorker-6
Starting ForkPoolWorker-1
Starting ForkPoolWorker-7
Starting ForkPoolWorker-2
Starting ForkPoolWorker-8
Pool       : [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

```

By default, `Pool` creates a fixed number of worker processes and passes jobs to them until there are no more jobs. Setting the `maxtasksperchild` parameter tells the pool to restart a worker process after it has finished a few tasks, preventing long-running workers from consuming ever more system resources.

```

# multiprocessing_pool_maxtasksperchild.py

import multiprocessing

def do_calculation(data):
    return data * 2

def start_process():
    print('Starting', multiprocessing.current_process().name)

if __name__ == '__main__':
    inputs = list(range(10))
    print('Input      : ', inputs)

    builtin_outputs = map(do_calculation, inputs)
    print('Built-in:', builtin_outputs)

    pool_size = multiprocessing.cpu_count() * 2
    pool = multiprocessing.Pool(
        processes=pool_size,
        initializer=start_process,
        maxtasksperchild=2,
    )
    pool_outputs = pool.map(do_calculation, inputs)
    pool.close() # no more tasks
    pool.join() # wrap up current tasks

    print('Pool      : ', pool_outputs)

```

The pool restarts the workers when they have completed their allotted tasks, even if there is no more work. In this output, eight workers are created, even though there are only 10 tasks, and each worker can complete two of them at a time.

```

$ python3 multiprocessing_pool_maxtasksperchild.py

Input      : [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Built-in: <map object at 0x1007b21d0>
Starting ForkPoolWorker-1
Starting ForkPoolWorker-2
Starting ForkPoolWorker-4
Starting ForkPoolWorker-5

```

```
Starting ForkPoolWorker-5
Starting ForkPoolWorker-6
Starting ForkPoolWorker-3
Starting ForkPoolWorker-7
Starting ForkPoolWorker-8
Pool      : [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

[← multiprocessing Basics](#)

[Implementing MapReduce →](#)

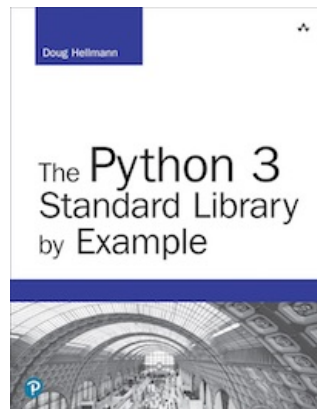
Quick Links

[Signaling between Processes](#)
[Controlling Access to Resources](#)
[Synchronizing Operations](#)
[Controlling Concurrent Access to Resources](#)
[Managing Shared State](#)
[Shared Namespaces](#)
[Process Pools](#)

This page was last updated 2017-07-30.

Navigation

[multiprocessing Basics](#)
[Implementing MapReduce](#)





[Get the book](#)

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](#)



© Copyright 2019, Doug Hellmann



Other Writing

 [Blog](#)
 [The Python Standard Library By Example](#)