

# Python multiprocessing module

James Abel

3/16/22

# Why Use Python Multiprocessing

- almost all computers these days have at least 2 cores and many have 4, 6, 8, or more
- improve program responsiveness (improve latency)
- improve performance by executing across multiple processor cores (improve throughput)
- utilize non-saturated computing resources
- examples:
  - one processor core handles UI while others handle compute
    - does not require managing thread execution time as in cooperative multi-threading
    - GIL not a factor for multiprocessing
  - spread compute across multiple cores

# Python's multiprocessing module

- supports a variety of coding techniques
  - object-oriented or function based
  - a variety of ways to pass data
  - manage execution across hardware processors/cores
- data types include native `ctype` or (pickleable) Python objects
- selecting the most appropriate pattern can be rather daunting
- good news: a relatively small number of design patterns can handle many common cases

This talk will mainly discuss `multiprocessing.Process` using object oriented programming (OOP) and `multiprocessing.Pool` with a function based pattern.

# Using multiprocessing.Process

- supports an object-oriented pattern
  - similar to the `multithreading.Thread` API
  - pass parameters in via `__init__` (optional)
  - `.start()` is called to start the process
  - `.run()` is the method that actually runs in the separate process
- can also be run procedurally via `Process(target=<function>)`  
*(not a focus in these examples)*
- return values via communication provided in the `multiprocessing` module
  - e.g. `Queue`, `Value`, etc.

First, create a worker that calculates “e”

```
from multiprocessing import Event

def calculate_e(exit_event: Event) -> float:
    """
    calculate "e" to do some work
    """
    k = 1.0
    e_value = 0.0
    iteration = 0
    # exit when exit_event is set
    while iteration % 1000 != 0 or not exit_event.is_set():
        e_value += 1.0 / k
        k *= iteration + 1
        iteration += 1
    return e_value
```

## Wrap the work in a multiprocessing.Process class

```
from multiprocessing import Process, Event, SimpleQueue
from workers import calculate_e

class CalculateE(Process):
    def __init__(self):
        self._result_queue = SimpleQueue() # from multiprocessing
        self._result = None # result placed here
        self.exit_event = Event() # will tell process to stop
        super().__init__(name="calculate_e_process") # named

    def run(self):
        returned_e_value = calculate_e(self.exit_event)
        # return the value in the Queue
        self._result_queue.put(returned_e_value)

    def get(self) -> float:
        if self._result is None:
            self._result = self._result_queue.get() # blocks
        return self._result
```

# main

```
import time
from multiprocessing_talk import CalculateE

e_process = CalculateE()
e_process.start()
time.sleep(3) # do other useful stuff ...
e_process.exit_event.set() # tell process to stop
print(e_process.get()) # print e
```

# Using multiprocessing.Pool

- manages mapping execution to the underlying hardware (processor cores)
- works well with a more functional programming style, including ‘map
- manages return data
- workers have to manage everything in the `Process`, including things like logging
  - if workers don't use logging then keep them simple



## multiprocessing.Pool Example

```
import time
from multiprocessing import Pool
from multiprocessing.managers import SyncManager
from workers import calculate_e

sync_manager = SyncManager()
sync_manager.start()

with Pool() as pool:
    # for pool, use Event from SyncManager
    e_exit_event = sync_manager.Event()
    # same "e" calculation as before, Pool() manages return data
    e_process = pool.apply_async(calculate_e, args=(e_exit_event,))
    time.sleep(3) # do other stuff
    e_exit_event.set() # tell calculate_e to stop
    print(e_process.get())
```

# Logging

The created `Process` ends up with a “new” logging instance (not inherited). It must be configured. Configuration can be passed in via a parameter.

The `balsa` logging package provides `Balsa.config_as_dict()` to a pickle-able logger configuration. This is provided to `Balsa.balsa_clone()` in the process’s `.run()` method to create the logger for that process. This also handles shared resources such as log files.

## Logging main

```
from balsa import Balsa
from multiprocessing_talk import CalculateE

balsa = Balsa("myapp", "myname")
balsa.init_logger()
balsa_config = balsa.config_as_dict()

e_process = CalculateE(balsa_config) # pass in log config
e_process.start()
e_process.exit_event.set()
e_process.join()
```

## Logging Process (logging code only)

```
from multiprocessing import Process
from balsa import balsa_clone

class CalculateE(Process):
    def __init__(self, name: str, logging_config: dict):
        self.name = name # must be unique across processes
        self.logging_config = logging_config
        super().__init__()

    def run(self):
        # must be done in .run()
        balsa_log = balsa_clone(self.logging_config, self.name)
        balsa_log.init_logger()
        # do the work and use the logging module ...
```

# Demo!

```
python -m multiprocessing_talk
```

This demo simultaneously calculates information about a directory (in this case the Python directory) while also calculating “e”. The directory work takes about 5-6 seconds, and the “e” calculation is done in parallel. Also in parallel, status messages are printed to the console.

# Common Pitfalls/Considerations

- trying to access data “directly” across processes without using multiprocessing communication “channels”
  - several mechanisms exist to facilitate communication, but you should choose wisely
  - communicate using `ctypes` or `pickle`-able data
- logging implementation that doesn’t take multiprocessing into account
- mismanaging compute resources (cores)
  - mapping a program on to multiple processes that isn’t actually parallelizable
  - too much start/stop or communication overhead
- usually best to parallelize across “real” cores
  - parallelize across “HyperThread” (AKA Simultaneous Multi-Threading or SMT) processors with care

# Design Decisions

- one (or more) disparate processes or mapping a single function across many data instances
- how to pass data
- how to address process-specific aspects such as logging
- consider first prototyping a design pattern appropriate for your application

# Summary

- Parallel processing has historically been considered difficult, but it doesn't have to be
- Using `multiprocessing.Process` can be straightforward
- Choosing the most appropriate design pattern is important
- Understand limitations of multiprocessing
- Use the facilities available that enable multiprocessing



## Additional Resources

longtaskrunnin - similar techniques applied to a PyQt application.

# Thank you!

Thanks to Christopher Brousseau for reviewing this talk and code!