Python Multithreading vs Multiprocessing: -

Use multiprocessing when you have to cpu intensive task and would take benefit of multiple core (more core of cpu) and avoid the global interpreter lock. ( effectively side-stepping the [Global Interpreter Lock](https://docs.python.org/3/glossary.html#term-global-interpreter-lock) by using subprocesses instead of threads.)

  Set deamon - True to create daemon process in threading

Multithreading thread are **light weight processes**, they require less memory overhead and hence are cheaper than processes.

* The **register set** and **local variables** of each threads are stored in the stack.
* The **global variables** (stored in the heap) and the **program codes** are shared among all the threads

Use multithreading when you have a task of multiple I/O or network usage.

How to stop thread that is running even timeout defined in join exceeds? Using Event

from threading import Thread

from threading import Event

import time

class Connection(Thread):

   StopEvent = 0

   def \_\_init\_\_(self,args):

       Thread.\_\_init\_\_(self)

       self.StopEvent = args

   def run(self):

       for i in range(1,10):

           if(self.StopEvent.wait(0)):

               print('Asked to stop')

               break;

           print(f'Child process is sleeping with count {i}')

           time.sleep(3)

       print('Child Thread is exiting')

stop = Event()

conn = Connection(stop)

conn.start()

print('Main Thread is start to wait for 5 sec for Child Thread')

conn.join(5)

print('Main Thread cant wait for more then 5 sec and now stopping the child thread')

stop.set()

conn.join()

print('Main Thread is exiting')

We can **never** set active thread to daemon or non-daemon (vice-versa as well)

If we want to change the **main** thread which is always non-daemon in nature to daemon nature then we will get a ***RuntimeError*** because when the program is started at a time main thread is also started so the main thread is an active thread and the active thread is not set to the daemon.

[multiprocessing](https://docs.python.org/3/library/multiprocessing.html#module-multiprocessing) supports two types of communication channel between processes:

* Queue
* Pipe

The [Pipe()](https://docs.python.org/3/library/multiprocessing.html#multiprocessing.Pipe) function returns a pair of connection objects connected by a pipe which by default is duplex (two-way). For example:

from multiprocessing import Process, Pipe

def f(conn):

    conn.send([42, None, 'hello'])

    conn.close()

if \_\_name\_\_ == '\_\_main\_\_':

    parent\_conn, child\_conn = Pipe()

    p = Process(target=f, args=(child\_conn,))

    p.start()

    print(parent\_conn.recv())   # prints "[42, None, 'hello']"

    p.join()

 Synchronization between processes : Using lock to ensure that diffentent thread/process are not writing/doing similar operation at same time.

Sharing state with processes:-

* Shared memory
* Server Process (Manager object)

from multiprocessing import Process, Manager  
  
def f(d,l):  
 d['name'] = 'MP'  
 d['purpose'] = 'Shared processs data'  
 d['written'] = 'Programmer'  
 l.reverse()  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 with Manager() as manager:  
 d = manager.dict()  
 l = manager.list(range(10))  
  
 p = Process(target=f, args=(d,l))  
 p.start()  
 p.join()  
  
 print(d)  
 print(l)

Pool Vs Process: -

Management: -

The Pool class is easier to use than the Process class because you do not have to manage the processes by yourself. It creates the processes, splits the input data, and returns the result in a list. It also waits for the workers to finish their tasks, i.e., you do not have to call the join() method explicitly.

Memory: -

**While the Process keeps all the processes in the memory, the Pool keeps only those that are under execution. Therefore, if you have a large number of tasks, and if they have more data and take a lot of space too, then using process class might waste a lot of memory**.

The overhead of creating a Pool is more. Therefore, when there are a small number of tasks, and they are not repetitive, it is advisable to use a Process in this case.

I/O Operation: -

Both the Process and the Pool class use FIFO (First In First Out) scheduler. However, **if the current process is waiting for, or executing an I/O operation, then the Process class halts the current one and schedules another one from the task queue. The Pool class, on the other hand, waits for the process to complete its I/O operation**, i.e., it does not schedule another one until the current has finished its execution. Because of this, the execution time might increase. **Process is preferred over Pool when your task is I/O bound** (A program is I/O bound if it spends most of its time waiting for the I/O operation to complete).

import multiprocessing as mp  
import time  
  
def test(fname):  
 with open(fname,"w") as f:  
 f.write("Hi")  
 f.write("Hi 1")  
 f.write("Hi 2")  
 f.write("Hi 3")  
 f.write("Hi 4")  
  
  
def process\_performance(filename):  
 start\_time = time.time()  
 # filename = "text.txt"  
 p1 = mp.Process(target=test, args=(filename,))  
 p2 = mp.Process(target=test, args=("text-2.txt",))  
  
 p1.start()  
 p2.start()  
 p1.join()  
 p2.join()  
 end\_time = time.time()  
 print(f"Process :: I/O :: Total time taken {end\_time - start\_time}")  
  
  
def pool\_performance(filename):  
 start\_time = time.time()  
 pool = mp.Pool()  
 a = pool.apply\_async(test, args=(filename,))  
 b = pool.apply\_async(test, args=("text-2.txt",))  
 a.wait()  
 b.wait()  
 end\_time = time.time()  
 print(f"Pool :: I/O :: Total time taken {end\_time - start\_time}")  
  
  
if \_\_name\_\_ == "\_\_main\_\_" :  
 filename = "text.txt"  
 process\_performance(filename)  
 pool\_performance(filename)

**Process** :: I/O :: Total time taken 0.10791778564453125  
**Pool** :: I/O :: Total time taken 0.18415570259094238

Pytest:

Free and open source

Pytest can run multiple tests in parallel, which reduces the execution time of the test suite.

Pytest has its own way to detect the test file and test functions automatically, if not mentioned explicitly.

Pytest allows us to skip a subset of the tests during execution.

Pytest allows us to run a subset of the entire test suite.

Running pytest without mentioning a filename will run all files of format **test\_\*.py or \*\_test.py** in the current directory and subdirectories. Pytest automatically identifies those files as test files. We **can** make pytest run other filenames by explicitly mentioning them.

pytest -v # Will run all the test with verbose mode

To run specific files test just pass it

pytest <test\_file\_name>

pytest test\_compare.py

Pytest provides two ways to run the subset of the test suite.

* Select tests to run based on substring matching of test names.
* Select tests groups to run based on the markers applied.

To execute the test containing string:   
pytest -k <string>  
pytest -k great -v # This will execute all test which contain great in their name or test file name.

(venv) C:\Users\SVIJAY\PycharmProjects\Learning>pytest -k great -v

================================================================================== test session starts ===================================================================================

platform win32 -- Python 3.8.6, pytest-6.2.3, py-1.10.0, pluggy-0.13.1 -- c:\users\svijay\pycharmprojects\pythonproject\venv\scripts\python.exe

cachedir: .pytest\_cache

rootdir: C:\Users\SVIJAY\PycharmProjects\Learning

collected 6 items / 2 deselected / 4 selected

test\_compare.py::test\_greater PASSED [ 25%]

test\_compare.py::test\_greater\_equal PASSED [ 50%]

test\_great.py::test\_get PASSED [ 75%]

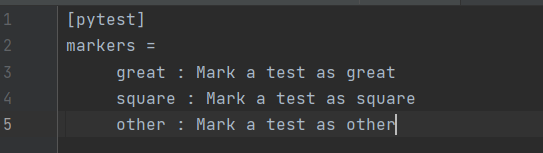
test\_great.py::test\_false PASSED [100%]

============================================================================ 4 passed, 2 deselected in 0.12s =============================================================================

(venv) C:\Users\SVIJAY\PycharmProjects\Learning>

Pytest allow to use markers on test cases. Markers are used to set various features/attributes to test functions. Pytest provides many inbuilt markers such as xfail, skip and parametrize. We need to import Pytest module in test file.

We may need to create pytest.ini file to register the marker to avoid warning while running the test.



To run the marked tests, we can use the following syntax –

pytest -m <markername> -v

Fixtures: -

Fixtures are functions which run before each test case to which it is applied. It is used to feed some data to the tests like DB connection, url to validate and some sort of input values. Therefore, instead of running the same code for every test, we can attach fixture function to the tests and it will run and return the data to the test before executing each test.

import pytest  
  
@pytest.fixture  
def input\_value():  
 val = 36  
 return val  
  
def test\_divisible\_by\_3(input\_value):  
 assert input\_value % 3 == 0  
  
def test\_divisible\_by\_12(input\_value):  
 assert input\_value % 12 == 0  
  
def test\_divisible\_by\_10(input\_value):  
 assert input\_value % 10 == 0

Here, we have a fixture function named input\_value, which supplies the input to the tests.

pytest looks for **conftest.py** modules throughout the directory structure. Each conftest.py provides configuration for the file tree pytest finds it in. You can use any fixtures that are defined in a particular conftest.py throughout the file’s parent directory and in any subdirectories. This is a great place to put your most widely used fixtures.

Conftest.py

import pytest  
  
@pytest.fixture  
def input\_value():  
 val = 36  
 return val

test\_division.py

import pytest  
  
# @pytest.fixture  
# def input\_value():  
# val = 36  
# return val  
  
def test\_divisible\_by\_3(input\_value):  
 assert input\_value % 3 == 0  
  
def test\_divisible\_by\_12(input\_value):  
 assert input\_value % 12 == 0  
  
def test\_divisible\_by\_10(input\_value):  
 assert input\_value % 10 == 0

Parameterizing of a test is done to run the test against multiple sets of inputs. We can do this by using the following marker –

@pytest.mark.parametrize

How to use parameter in your test case can be understand using below example

import pytest  
  
@pytest.mark.parametrize("num, output",[(1,11),(2,22),(3,34),(4,44)])  
def test\_multiplication(num, output):  
 assert 11\*num == output

@pytest.mark.xfail : Pytest will execute the xfailed test, but it will not be considered as part failed or passed tests. Details of these tests will not be printed even if the test fails

@pytest.mark.skip : We can skip a test using skip mark.

import pytest  
  
@pytest.mark.xfail  
@pytest.mark.great  
def test\_greater():  
 num = 100  
 assert num > 180  
  
@pytest.mark.great  
def test\_greater\_equal():  
 num = 100  
 assert num >= 100  
  
@pytest.mark.skip  
@pytest.mark.other  
def less\_test():  
 num = 100  
 assert num < 130

if we want to stop the execution of test suite when n number of test fails. This can be done in pytest using maxfail.

pytest --maxfail = <num>

pytest -k great -v --maxfail = 1

By default, pytest runs tests in sequential order. If we have multiple tests to run then it will lead to considerable time. **We can run tests parallel with Pytest and will use pytest-xdist lib to achive this.**

Just install pytest-xdist and run tests by using the syntax pytest -n <num>

pytest -n 3

Another interesting use case for fixtures is in guarding access to resources. Imagine that you’ve written a test suite for code that deals with API calls. You want to ensure that the test suite doesn’t make any real network calls, even if a test accidentally executes the real network call code. pytest provides a monkeypatch fixture to replace values and behaviors, which you can use to great effect:

import pytest  
import requests  
  
@pytest.fixture  
def input\_value():  
 val = 36  
 return val  
  
@pytest.fixture(autouse=True)  
def disable\_network\_calls(monkeypatch):  
 def stunted\_get():  
 raise RuntimeError("Network access not allowed during testing!")  
 monkeypatch.setattr(requests, "get", lambda \*args, \*\*kwargs: stunted\_get())

By placing **disable\_network\_calls()** in conftest.py and adding the **autouse=True** option, you ensure that network calls will be disabled in every test across the suite. Any test that executes code calling requests.get() will raise a RuntimeError indicating that an unexpected network call would have occurred.

Marking tests is useful for categorizing tests by subsystem or dependencies. If some of your tests require access to a database, for example, then you could create a @pytest.mark.database\_access mark for them.

When the time comes to run your tests, you can still run them all by default with the pytest command. If you’d like to run only those tests that require database access, then you can use pytest -m database\_access. To run all tests except those that require database access, you can use pytest -m "not database\_access". You can even use an autouse fixture to limit database access to those tests marked with database\_access.

Use the --**durations** option to the pytest command to include a duration report in your test results. --durations expects an integer value n and will report the slowest n number of tests. The output will follow your test results:

$ pytest --durations=3

3.03s call test\_code.py::test\_request\_read\_timeout

1.07s call test\_code.py::test\_request\_connection\_timeout

0.57s call test\_code.py::test\_database\_read

**pytest-cov**

If you measure how well your tests cover your implementation code, you likely use the [coverage](https://coverage.readthedocs.io/) package. [pytest-cov](https://pytest-cov.readthedocs.io/en/latest/) integrates coverage, so you can run pytest --cov to see the test coverage report.

**pytest-bdd**

pytest can be used to run tests that fall outside the traditional scope of unit testing. [Behavior-driven development](https://en.wikipedia.org/wiki/Behavior-driven_development) (BDD) encourages writing plain-language descriptions of likely user actions and expectations, which you can then use to determine whether to implement a given feature. [pytest-bdd](https://pytest-bdd.readthedocs.io/en/latest/) helps you use [Gherkin](http://docs.behat.org/en/v2.5/guides/1.gherkin.html) to write feature tests for your code.