Process and Pool Class

Process

By subclassing multiprocessing.process, we can create a process that runs independently. By extending the \_\_init\_\_ method we can initialize resource and by implementing Process.run() method we can write the code for the subprocess.

To spawn the process, we need to initialize our Process object and invoke Process.start() method. Here Process.start() will create a new process and will invoke the Process.run() method.

The code after p.start() will be executed immediately before the task completion of process p. To wait for the task completion, we can use Process.join().

Program:

import multiprocessing import time class Process(multiprocessing.Process): def \_\_init\_\_(self, id): super(Process, self).\_\_init\_\_() self.id = id

def run(self):

time.sleep(1)

print("I'm the process with id: {}".format(self.id))

if \_\_name\_\_ == '\_\_main\_\_': p = Process(0) p.start()

p.join() p = Process(1) p.start()

p.join()

Output:

I'm the process with id: 0

I'm the process with id: 1

### Pool class

Pool class can be used for parallel execution of a function for different input data. The multiprocessing.Pool() class spawns a set of processes called workers and can submit tasks using the methods apply/apply\_async and map/map\_async. For parallel mapping, you should first initialize a multiprocessing.Pool() object. The first argument is the number of workers; if not given, that number will be equal to the number of cores in the system.

In this example, we will see how to pass a function which computes the square of a number. Using Pool.map() we can map the function to the list and passing the function and the list of inputs as arguments, as follows:

import multiprocessing import time

def square(x):

return x \* x

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

pool = multiprocessing.Pool() pool = multiprocessing.Pool(processes=4) inputs = [0,1,2,3,4] outputs = pool.map(square, inputs) print("Input: {}".format(inputs)) print("Output: {}".format(outputs))

Output

Input: [0, 1, 2, 3, 4]

Output: [0, 1, 4, 9, 16]

When we use the normal map method, the execution of the program is stopped until all the workers completed the task. Using map\_async(), the AsyncResult object is returned immediately without stopping the main program and the task is done in the background. The result can be retrieved by using the AsyncResult.get() method at any time as shown below:

import multiprocessing import time def square(x):

return x \* x if \_\_name\_\_ == '\_\_main\_\_':

pool = multiprocessing.Pool() inputs = [0,1,2,3,4]

outputs\_async = pool.map\_async(square, inputs) outputs = outputs\_async.get()

print("Output: {}".format(outputs))

Output:

Output: [0, 1, 4, 9, 16]

**Pool.apply\_async** assigns a task consisting of a single function to one of the workers. It takes the function and its arguments and returns an AsyncResult object.

import multiprocessing import time def square(x):

return x \* x if \_\_name\_\_ == '\_\_main\_\_':

pool = multiprocessing.Pool()

result\_async = [pool.apply\_async(square, args = (i, )) for i in range(10)] results = [r.get() for r in result\_async] print("Output: {}".format(results))

Output: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]