Concurrency in Python

Motivation: End of Moore's law

In 1965, Gordon E. Moore predicted that the number of transistors in a dense integrated circuit will continue to double approximately every 2 years. Around 2015, the rate of advancement started to slow down. Chip makers started developing multi-core CPUs (2 or more processing units that can read and execute instructions). In order to take advantage of these hardwares, programs must be developed to employ multi-concurrency techniques to allow them to execute instructions on multiple core simultaneously.

Concurrency (single-machine)

It is the execution of multiple instruction sequences at the same time. They must be independent from each other, in terms of order of execution and shared resources (as few as possible).

Parallel Programming vs. Asynchronous Programming

Parallel programming in multi-core systems is best suited for tasks that are CPU-intensive, i.e. tasks for which most of the time is spent solving the problem instead of reading to or writing from a device - these are called **CPU-bound tasks** (perform better if CPU is better). Examples: string operations, search algorithms, graphics processing, number crunching algorithms...

If the tasks are about reading to or writing from a device (i.e. performing inputs and ouputs), they are most suited for asynchronous programming. These are called **IO-intensive tasks** For example: database reads / writes, web service calls, copying, downloading, uploading data.

Threading

Definitions

Process: the execution context of a running program or a running instance of a computer program. Every executing process has:

- processor state
- system resources
- a section of memory that is assigned to it
- security attributes in a process state.

A process is composed of one or more threads of execution.

Thread: the smallest sequence of instructions that can be managed, scheduled and executed by the operating system. A program can be composed of a single thread or multiple threads of execution.

Thread pool: a thread pool maintains multiple threads waiting for tasks to be allocated for concurrent execution by the supervising program.

Scheduler: an operating system module that selects the next jobs to be admitted into the system and the next process to run.

Context switch: the process of saving and restoring the state of a thread or process

Daemon process: it is a child process that does not prevent its parent process from exiting (e.g. background program)

Parallel programming using threads vs. Parallel programming using processors: if a newly-selected thread is from a different process, a full process switch occurs, which is fairly expensive process. If it's from the same process, a thread switch is executed which is less expensive.

Thread interference, also commonly referred as **race condition**: it happens when both threads are in a race to read or update the same variable.

Thread synchronization: It is best practice to keep shared memory to a minimum. Python has mechanism to synchronize access to shared resources.

Lock

The most fundamental of these synchronization mechanisms is the **lock**. Two states: locked and unlocked. Once a lock is acquired by a thread, it cannot be unlocked by another thread.

- lock is acquired by a thread and released by the same thread
- if another thread tries to acquire a lock that is not released, that thread goes into a **block state**, i.e. its execution is paused until the lock is released.

We can create a lock for shared resources.

Warning: if an exception occurs between lock.acquire and lock.release(), we may never release the lock and other threads will wait forever. Use try / finally block for avoiding this situation. We can also use context manager with a with statement (lock is automaticaly acquired once entered in the with statement, i.e. lock.acquire(), and automatically released when it leaves the with statement, i.e. lock.release().

```
lock = threadling.Lock()

with lock:
  access shared resources
```

A Re-entryLock or **RLock** for short is a lock that can be acquired more than once, even by the same thread. A regular lock can only be acquired once, even by the same thread.

Semaphore

Another widely-used thread synchronization mechanism is the **semaphore**.

```
semaphore = threading.Semaphore()
semaphore.acquire() #decrements the counter
... access the shared resource
semaphore.release()
```

A **semaphore maintains a set of permits**. A lock can be used to permit only one thread to run a critical section of code at the same time whereas a semaphore can allow one or more threads to run at the same time. We can initialize the semaphore to set the maximum number of threads to run at the same time.

```
num_permits = 3
semaphore = threading.BoundedSemaphore(num_permits)

semaphore.acquire() #decrements the counter
... up to 3 threads can access the shared resource
semaphore.release()
```

We can also use the with statement with the semaphore.

```
with semaphore:
  some code
```

Event

One thread signals an event, and the other threads wait for it. An event has an internal flag (True / False).

```
event = threading.Event()

# a client thread can wait for the flag to be set
event.wait() # blocks if flag is false

# a server thread can set or reset it
event.set() # sets the flag to true
event.clear() # resets the flag to false
```

Condition

A condition combines the properties of a lock at an event.

Queue

A queue makes it easier to exchange messages across multiple threads. 4 main methods:

- put() : Puts an item into the queue
- get() : Removes an item from the queue and returns it
- task_done(): Marks an item that was gotten from the queue as completed / processed
- join(): Blocks until all the items in the queue have been processed.

```
from threading import thread
from queue import Queue
def producer (queue):
  for i in range(10):
    item = make_an_item_available(i)
    queue.put(item)

def consumer (queue):
  while True:
    item = queue.get()
    # do something
    queue.task_done() # mark the item as done

queue = Queue()
t1 = Thread (target = producer, args = (queue,))
t2 = Thread (target = consumer, args = (queue,))
t1.start()
t2.start()
```

Creating threads

2 patterns:

- instantiate an object of the thread class

```
t = threading.Thread (target = some_func, args = (val,))
t.start()
t.join()
```

- use of a class that inherits the threading.Thread class

```
import threading
class Fibonacci (threading.Thread):
def __init__(self, num):
  Thread.__init__(self)
  self.num = num

  def run(self):
    #Fibonacci number calculation
    ...
    print fib[self.num]
myFibTask1 = FibonacciThread(9)
myFibTask2 = FibonacciThread(12)
myFibTask1.start()
myFibTask2.start()
myFibTask1.join()
myFibTask2.join()
```

Thread lifecycle

Thread Stack Space

Use of threading in python

It is best practice to use threading in Python only for input/output (I/O)-bound operations, because of the GIL (Global Interpreter Lock). The GIL is a lock that prevents multiple native threads from executing Python code at the same time. The GIL exists to protect against thread interferences and race conditions. So only one thread can operate at a time. In case we create multiple threads, they will alternate, that is why it benefits I/O operations but not CPU-bound operations.

There are mainly 2 main GIL workarounds:

- GIL-less python interpreters
  - Jython
  - IronPython

- Use of Python Multiprocessing

Multi-processing

In running Python programs, there is one interpreter per process. There is one GIL for every Python process.

Processes vs. Threads

Benefits:

- sidesteps GIL
- less need for synchronization
- can be paused and terminated
- more resilient

Limitations:

- Higher memory footprint
- Expensive context switch

Multi-processing API

```
import multiprocessing

def do_smthg (val):
  print ("doing smthg")
  return
if __name__ == '__main__':
  val = "some args"
  t = multiprocessing.Process (target = do_smthg, args = (val,))
  t.start()
  t.join()
```

Note that arguments passed to the process constructor must be **picklable**.

Pickling is the process whereby a Python object hierarchy is converted into a byte stream. "unpickling" is the inverse operation. It is also known as object serialization / deserialization or object martialing and unmartialing.

Picklable objects include:

- None, True, False
- Integers, floats, complex numbers
- Normal and Unicode strings
- Collections containing only picklable objects
- Top level functions
- Classes with pickable attributes

2 methods for managing the aliveness of processes:

- is_alive() returns True/False
- terminate() terminates a process

p.exitcode returns either:

- 0 if the process terminated without error
- more than 0 if it terminated with an error (value of error is returned)
- less than 0 if the process was killed with a signal of -1 multipled by the signal code

Process.terminate() gotchas:

- shared resources may be put in an inconsistent state
- Finally clauses an exit handlers will not be run

Proces Pools

```
class multiprocessing.Pool ([num_processes [, initializer [, initargs [, masktasksperchild]]]])
```

Note that `initargs` does not have to picklable.

Usage:

- pool.map(function, args) [**Synchronous version**]
  `pool = multiprocessing.Pool (processes = pool_size, initializer = start_process) outputs = pool.map (function, args) pool.close() p`
- pool.map_async (function, args, chunksize, callback) [**Asynchronous version**]

It is non blocking and returns an object. When we need the result, we call AsynResult.get([timeout]) and returns the result when it arrives. It blocks if results are not readily available.

```
map_async(func, iterable[apply_async(func, iterable[, chunksize[, callback]]) returns AsynResult
```

- the pool class also has the apply and apply_async method

```
apply (func, [,args[,kwargs]])
apply_async(func, [,args[,kwargs[, callback[,error_callback]]]])
```

Inter-process Communication Channels

Queue and Pipes are ways for sharing data between multi processed functions. When using multiple processes, one generally uses message passing for communication between processes and avoids having to use any synchronization primitives like locks.

For passing messages one can use Pipe() (for a connection between two processes) or a queue (which allows multiple producers and consumers).

Pipe

The Pipe() function returns a pair of connection objects connected by a pipe which by default is duplex (two-way). The two connection objects returned by Pipe() represent the two ends of the pipe. Each connection object has send() and recv() methods (among others). Note that data in a pipe may become corrupted if two processes (or threads) try to read from or write to the same end of the pipe at the same time. Of course there is no risk of corruption from processes using different ends of the pipe at the same time.

Queue

The Queue class is a near clone of queue.Queue. For example:

```
from multiprocessing import Process, Queue

def f(q):
    q.put([42, None, 'hello'])

if __name__ == '__main__':
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()
    print(q.get())    # prints "[42, None, 'hello']"
    p.join()
```

Queues are thread and process safe (safer for data integrity)

Sharing State Between Processes

2 methods exist:

*** Shared memory (for single variable)**

- multiprocessing.Value

  multiprocessing.Value(typecode*or*type, *args, lock=True)¶

- multiprocessing.Array

*** Manager Process**

It enables to share more types of data and allows communications across machines but is more resource-consuming. Note that a manager process spawns off a new process.

Others

Threads uniquely run in the same unique memory heap where as Processes run in separate memory heaps. This makes sharing information harder with processes and object instances. One problem arises because threads use the same memory heap, multiple threads can write to the same location in the memory heap which is why the global interpreter lock(GIL) in CPython was created as a mutex to prevent it from happening.

CPython is the reference(standard) implementation of the Python Programming Language. Written in C and Python, CPython is the default and most widely used implementation of the language. It can be defined as both an interpreter and a compiler as it compiles Python code into byte-code before interpreting it. A particular feature of CPython is that it makes use of a **global interpreter lock (GIL)** on each CPython interpreter process, which means that **within a single process only one thread may be processing Python byte-code at any one time.** This does not mean that there is no point in multi threading; the most common multi-threading scenario is where threads are mostly waiting on external processes to complete.

Concurrency of Python code can only be achieved with separate CPython interpreter processes managed by a multitasking operating system. This complicates communication between concurrent Python processes, though the multiprocessing module mitigates this somewhat; it means that applications that really can benefit from concurrent Python-code execution can be implemented with a limited amount of overhead.

The multiprocessing library uses separate memory space, multiple CPU cores, bypasses GIL limitations in CPython, child processes are kill able(ex. function calls in program) and is much easier to use. Some caveats of the module are a larger memory footprint and IPC's a little more complicated with more overhead. Python's multiprocessing library offers two ways to implement Process-based parallelism:-

- Process
- Pool

Process implementation

It's used when function based parallelism is required, where I could define different functionality with parameters that they receive and run those different functions in parallel which are doing totally various kind of computations.

```
from multiprocessing import Process

def f1(name):
    print('hello', name)
def f2(name):
    print('hello', name)
if __name__ == '__main__':
    procs = []
    p1 = Process(target=f1, args=('bob',))
    p1.start()
    procs.append(p1)
    p2 = Process(target=f2, args=('jerry',))
    p2.start()
    procs.append(p2)
    for p in procs:
         p.join()
```

Here two functions to pay attention are .start() and .join()

- **start()** helps in starting a process and that too asynchronously.
- **join()** method on a Process does block until the process has finished, but because we called .start() on both p1 and p2 before joining, then both processes will run asynchronously. The interpreter will, however, wait until p1 finishes before attempting to wait for p2 to finish. Note: A process cannot join itself because this would cause a deadlock. It is an error to attempt to join a process before it has been started.

Pool implementation

It offers a convenient means of parallelizing the execution of a function across multiple input values, distributing the input data across processes i.e. data based parallelism. The following example demonstrates the common practice of defining such functions in a module so that child processes can successfully import that module. from multiprocessing import Pool

```
def func(x):
    return x*x

if __name__ == '__main__':
    with Pool(5) as p:
        print(p.map(func, [1, 2, 3]))
```

Here pool.map() is a completely different kind of animal, because it distributes a bunch of arguments to the same function (asynchronously), across the pool processes, and then waits until all function calls have completed before returning the list of results. Four such variants functions provided with pool are:

- apply Call func with arguments args. It blocks until the result is ready.
- apply_async It is better suited for performing work in parallel.
- map A parallel equivalent of the map() built-in function (it supports only one iterable argument though, for multiple iterables). But it blocks.
- map_async It is better suited for performing work as map in parallel.

But make sure that in async functions, the order of the results is not guaranteed to be the same as the order of actual input results need to be.

How to save output of multi processed functions

But also most of the times, we are not just worried, about running our codes in parallel, but also saving their output at the end. Now how to obtain the combined output from various functions at the end of parallely processed functions, their is a work around for such situations.

One can use multiprocessing.Manager(). Managers provide a way to create data which can be shared between different processes, including sharing over a network between processes running on different machines. A manager object controls a server process which manages shared objects.

Attached is a code example using it:

```
import sys
import multiprocessing
from multiprocessing import Process

def func1(a, res):
 print('func1: starting')
 for i in range(10000000): pass
 print('func1: finishing')
 res.update({a:1})
 print("appended")
def func2(a, res):
 print('func2: starting')
 for i in range(10000000): pass
 print('func2: finishing')
 res.update({a:2})
 print("appended")
def runInParallel(dict):
 manager = multiprocessing.Manager()
 res = manager.dict()
 proc = []
 print(dict)
 for d in dict:
  print("h1")
  func = getattr(sys.modules[__name__], d)
  dict[d].append(res)
  print(dict[d])
  p = Process(target=func, args= tuple(dict[d],))
  print("h3")
  p.start()
  proc.append(p)
 for p in proc:
  p.join()
 return res

dict = {'func1' : [1], 'func2': [2]}
ans = runInParallel(dict)
```

Pool VS Process

Below information might help you understanding the difference between Pool and Process in Python multiprocessing class:

Pool: When you have junk of data, you can use Pool class. Only the process under execution are kept in the memory. I/O operation: It waits till the I/O operation is completed & does not schedule another process. This might increase the execution time. Uses FIFO scheduler.

Process:
When you have a small data or functions and less repetitive tasks to do. It puts all the process in the memory. Hence in the larger task, it might cause to loss of memory. I/O operation: The process class suspends the process executing I/O operations and schedule another process parallel. Uses FIFO scheduler.

Abstracting Concurrency Mechanisms

Abstracting concurrency mechanisms

Using executor interface. The executor class is an abstract class so we'll need to instantiate one of its concrete sub-classes: ThreadPoolExecutor or ProcessPoolExecutor.

The Executor API has 3 methods:

- submit: schedules a function to run. Returns a future object and is not blocking.
- map: pass an iterable of arguments
- shutodwn: stop accepting tasks and shutdown

if we use a `with` block, there is no need to call shutdown `with executor: ... use executor`

A future is an object that acts as a proxy for a result that is yet to be computed. See 3 methods:

- cancel
- done
- exception
- add*done*callback(fn)

Modules functions:

- concurrentfutures.wait()
- concurrentfutures.as_completed()

Asynchronous programming

Single-threaded asynchrony

In the traditional model, IO-bound tasks are managed by multiple threads that take turns to execute tasks. But this model does not scale well to hundreds of thousands of tasks, because of the costs of scheduling and switching costs. A more efficient solution is to have one thread handle multiple IO-bound tasks. When the task needs to wait for some IO operation to complete, instead of blocking, the thread suspends the task and moves on to a task that is ready to execute. When the IO gets completed, the thread gets notified, and then it can resume the task that it suspended. If this task suspension and resumption sounds familiar, that's because it's a similar concept to what the operating system does with threads. The difference here is that we don't pay the higher memory cost for multiple threads, and we don't have the overhead of context switching. **We can switch between tasks much quicker and more efficiently**.

Asyncio module

The asyncio module provides tools for **building concurrent applications using coroutines**. While the threading module implements concurrency through application threads and multiprocessing implements concurrency using system processes, **asyncio uses a single-threaded, single-process approach in which parts of an application cooperate to switch tasks explicitly at optimal times**. Most often this context switching occurs when the program would otherwise block waiting to read or write data, but asyncio also includes support for scheduling code to run at a specific future time, to enable one coroutine to wait for another to complete, for handling system signals, and for recognizing other events that may be reasons for an application to change what it is working on.

Event loop

In order to implement this asyncio model, most languages and frameworks turn to an event loop. In the simplest of terms, **an event loop is responsible for taking items from an event queue and handling it**. An event could be a change of state on a file when new data is available to read, a timeout occurring, some new data arriving on a socket, etc. The thread goes into a loop and checks for an event it needs to response to. Its response may include executing a callback or some other code that relied on the occurrence of the event. The code currently being executed may generate more events that need to be watched for. When that happens, the loop suspends execution of that code and continues executing other code until the event occurs. There are several ways of implementing the event loop and mechanism for pausing execution, being notified of the completion of the IO task, and then resuming execution.

Event-driven architecture is a software design that orchestrates behavior around the production, detection and consumption of events.

See Nginx and NodeJS

Cooperative Multitasking with Event Loops and Coroutines

In NodeJS programming, the event loop is invisible to the programmer and is implemented within the VM execution engine and in the libev library. But in Python **the event**

loop is explicit. The event loop in Python is responsible for scheduling and executing tasks and callbacks in response to IO events, system events, and application context changes. To get an instance of an event loop, we call the `asyncio.get_event_loop` method. This method returns an object of abstract*event*loop. As the name implies, abstract*event*loop is an abstract class which has concrete subclasses. The get*event loop determines the appropriate concrete implementation for the platform we're running on and returns it. However, the only methods which we concern ourselves with are those exposed in the abstract*event*loop class. After we get the event*loop instance, we can start it by calling either:

- the abstract `EventLoop.run_forever()` method

If we start an event loop by calling run_forever, then we can stop it by calling `AbstractEventLoop.stop()`. This causes the event loop to exit at the next suitable opportunity. Once an event loop is in the stop state, then we can close it by calling close. This method closes a non-running event loop by clearing the queues and shutting down the executor.

- or the `AbsractEventLoop.run_until_complete()` method.

In the previous section of this module, I mentioned that the task running in the loop must be suspended when it encounters an IO operation or any other long-running operation that can be offloaded to another actor. While this happens implicitly in some other platforms, in Python **the running task itself is responsible for suspending itself and yielding control to the caller** so that the caller can run other tasks. When the IO operation completes, the call can then restore the task back to the state it was in before it suspended it and resume execution. **This is called cooperative multitasking**, and this is where **coroutines** come in. There are two constructs in Python call coroutines, and it's important to understand which one is being referred to when you hear or read the word coroutine. They are:

- the coroutine function
- and the coroutine object.