

Using the New Python 3.2 Concurrent Programming Features

April 13, 2011

The most recent release of Python (3.2) has introduced a new module called [concurrent.futures](#) that offers Python programmers new, streamlined and flexible means for handling common concurrent programming tasks -- thread and process submission, results handling, synchronization of execution and worker threads/process pooling.

Three classes -- Futures, Executors and ExecutorPools -- constitute the basis of this new package, and the previously mentioned common concurrent programming tasks are accomplished through the interaction of these three classes.

Programmers familiar with Java will find these new features very familiar. The design of the concurrent.futures module is [directly inspired](#) by the namesake structures available in Java's [java.util.concurrent package](#).

What Are Python Futures, Executors and Executor Pools?

Concurrent programming is, by the nature of the model, a more challenging task than single-threaded sequential programming. Programmers need to manage distribution of the tasks, non-deterministic execution flows, and synchronization of the completion of the concurrent tasks.

The purpose of the Futures class, as a design concept, is to mitigate some of the cognitive burdens of concurrent programming. Futures, as a higher abstraction of the thread of execution, offer means for initiation, execution and tracking of the completion of the concurrent tasks.

One can think of Futures as objects that model a running task, unlike a synchronously executing function, that will produce a result at some point in the future. Futures offer methods to query the status of the running task and, if necessary, to shut it down.

Futures are not meant to be directly instantiated and executed by a programmer. One can think of Futures as interfaces that can be queried but not instantiated directly. Futures are instantiated by submitting tasks (functions with optional parameters) to Executors. Executors are launchers that initialize and start the Futures.

Executors, in Python, are abstractions that are accessed through their subclasses: Thread or ProcessExecutorPools.

Use of pools of threads and processes is another best design and implementation practice of concurrent programming. Instantiation of threads and process is a resource-demanding task, so it is better to pool these resources and use them as repeatable launchers or "executors" (hence the Executors concept) for parallel or concurrent tasks.

Python Futures, Executors and ExecutorPools in Action

Let's look at some examples.

In this article, I will use a search over several files as an example of how can one use Futures and Executors in Python to execute tasks suitable for concurrent execution.

Starting with the simplest tasks, I will introduce some of the most interesting functions of the `concurrent.future` package, and we will move up a bit in increasing order of complexity.

Let's start with the simple example of running individually instantiated search tasks for three different strings in three different files:

```
def basic_search():
    with concurrent.futures.ThreadPoolExecutor(max_workers=5) as
    executor:
        future1 = executor.submit( search, "test1.txt", "Test")
        future2 = executor.submit( search, "test2.txt", "synergy")
        future3 = executor.submit( search, "test3.txt", "Something Else")
    print( future1.result() )
    print( future2.result() )
    print( future3.result() )
```

In the example above, in the method `basic_search()` I instantiate a `ThreadPoolExecutor` named *executor* with the keyword *with*:

```
with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
```

Notice also that I instantiated executor pool with a `max_workers=5` parameter. This indicates the maximum size of the worker threads in the pool that will handle the execution of the concurrent tasks at the time.

Using executor pool, I submit three search jobs by passing into an executor's `submit` function, as parameters, the name of the search function (`file_search`) and two arguments for the search function itself (file name and string to search for):

```
future1 = executor.submit( file_search, "test1.txt", "Test")
```

The call to each `submit` returns a future. I get the results of each search task on the files by calling the `result()` method on the future objects that were returned by the executor.

In this case, to process the results, I just print them:

```
print( future1.result() )
```

In the example above, using the `file_search` function, I am simply returning a string indicating if the search string was found with the index or match, but the actual scenario for the search task and return types can be far more complex--futures as result of their execution may even return other futures. In addition, future objects returned from the executor can also be queried for the execution status. This is accomplished with the methods `cancelled()`, `running()` and `done()`. Futures may also be "asked" to halt with the method `cancel()`. (I purposefully use the term "asked" because a program is, by design contract of most threading libraries and underlying implementations, never guaranteed immediate control over the thread's status of execution).

The above example, despite its intended triviality, demonstrates the typical pattern of use for objects provided in `concurrent.futures` package:

1. Call `ThreadPoolExecutor` to get an instance of the Executor.
2. Submit one or more routines into it.
3. Get as a result one or more `Future` objects.
4. Query `Future` object for the result of the execution.

Python Futures with Callbacks

In addition to the basic scenario with getting the result from the future, futures also support attachment of the callback--the function that gets assigned to the future and that gets called when the future completes execution.

Building up from the original basic example, here is how the parallel file search would be processed with callbacks. In the example below, the function `process_result` is attached to the search futures as callback:

```
def with_callback():
    with concurrent.futures.ThreadPoolExecutor(max_workers=5) as
    executor:
        executor.submit( search, "test1.txt", "Test").add_done_callback(
            process_result )
        executor.submit( search, "test2.txt", "Tset").add_done_callback(
            process_result )
        executor.submit( search, "test3.txt", "Something").add_done_callback(
            process_result )
```

Note the use of the `add_done_callback` method. It attaches the method `process_result` to a future that is returned from the call to `submit` on an executor.

The `process_result` method is an ordinary function that I implement to print the result of search returned from a future. The only requirement for callback processing function is to accept a single future object as its parameter.

```
def process_result( future ):
    print ( "callback on result: " + future.result() )
```

Mapped Submission in Python 3.2

In addition to submitting a function with parameters to an executor, we can alternatively "map" a function onto a list of arguments (more precisely, map a function onto an *iterable* list of arguments).

In the example below, I use the executor to asynchronously apply the function `search_file` to each of the file names provided by iterable `files` (a sequence of file name strings) and get the future `result` as the outcome of each `map` application within an iteration of the `for` loop.

```
def mapped_search():
    with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
        for result in executor.map( search_file, files ):
            print( result )
```

Waiting for All Futures to Finish

Finally, let's look at how the *wait* function, available as the module level function, can be used to process results of all completed threads at the common synchronization point.

In the function *parallel_search* shown below, I submit a list of files into the executor to get a list of futures. I then wait for all futures to complete using the function *wait*.

```
def parallel_search():
    with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
        futures = [ executor.submit( file_search, search_target, "Synergy" )
                    for search_target in files ]

    results = concurrent.futures.wait( futures )

    for completed in results.done():
        print( completed.result() )
```

Function *wait* by default indefinitely waits for all futures to complete, but it can alternatively accept the *Timeout* parameter. The parameter represents the number of seconds and specifies how long to wait before throwing an exception. It also can accept the *return_when* parameter, which specifies the criteria for waiting. The parameter can be set to one of the following three conditions: *ALL_COMPLETED* (set by default), *FIRST_COMPLETED* and *FIRST_EXCEPTION*.

The function *wait* returns a named tuple consisting of *done* and *not_done* sets of futures.

Done contains all the futures that have completed by the time wait has returned. These two sets are most pertinent with *FIRST_COMPLETED* or *FIRST_EXCEPTION* option set.

Processing Futures Upon Their Completion

Use the *as_completed* method if your concurrent programming task demands processing of individual futures as they complete their execution and in order of completion. The function *as_completed* accepts a list of futures and returns *iterable* which returns futures iteratively as each of the futures completes and in order of completion. *as_completed* accepts the optional *Timeout* parameter as well.

```
for future in concurrent.futures.as_completed( search_futures ):
    process_results( future )
```

Other Python 3.2 Features

This article focused on threads, but the *concurrent.futures* package also offers *ProcessPoolExecutor* objects for instantiation and execution of processes instead of threads. The pattern of use is the same as with

threads, except for the different executor object use and a requirement that all submitted tasks must be *picklable*.

In the terminology of the operating system, the term "process" refers to an operating system level task that has its own stack and heap space and that is managed by the operating system scheduler. "Threads," however, can be purely within-process tasks that may or may not be visible to the operating system at all. Visibility of the threads depends on the implementation of the operating system, the compiler, the threading library used and the virtual machine that runs the interpreted program -- if applicable. System resources wise, tasks are more intensive and more expensive to manage and instantiate.

Futures also capture exceptions, and they can be checked for using the method *exception()* of the future object. The *wait()* and *as_completed()* functions can raise *TimeoutErrors* when *Timeout* is specified, so these need to be checked as well.