Chapter 33
Futures

33.1 Introduction

A future is a thread (or process) that promises to return a value in the future; once the associated behaviour has completed. It is thus a *future* value. It provides a very simple way of firing off behaviour that will either be time consuming to execute or which may be delayed due to expensive operations such as Input/Output and which could slow down the execution of other elements of a program. This chapter discusses futures in Python.

33.2 The Need for a Future

In a normal method or function invocation, the method or function is executed in line with the invoking code (the *caller*) having to wait until the function or method (the *callee*) returns. Only after this is the caller able to continue to the next line of code and execute that. In many (most) situations this is exactly what you want as the next line of code may depend on a result returned from the previous line of code etc.

However, in some situations the next line of code is independent of the previous line of code. For example, let us assume that we are populating a User Interface (UI). The first line of code may read the name of the user from some external data source (such as a database) and then display it within a field in the UI. The next line of code may then add todays data to another field in the UI. These two lines of code are independent of each other and could be run concurrently/in parallel with each other.

In this situation we could use either a `Thread` or a `Process` to run the two lines of code independently of the caller, thus achieving a level of concurrency and allowing the caller to carry onto the third line of code etc.

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-25943-3_33

However, neither the `Thread` or the `Process` by default provide a simple mechanism for obtaining a result from such an independent operation. This may not be a problem as operations may be self-contained; for example they may obtain data from the database or from today's date and then updated a UI. However, in many situations the calculation will return a result which needs to be handled by the original invoking code (the caller). This could involve performing a long running calculation and then using the result returned to generate another value or update another object etc.

A Future is an abstraction that simplifies the definition and execution of such concurrent tasks. Futures are available in many different languages including Python but also Java, Scala, C++ etc. When using a Future; a callable object (such as a function) is passed to the Future which executes the behaviour either as a separate Thread or as a separate Process and then can return a result once it is generated. The result can either be handled by a call back function (that is invoked when the result is available) or by using a operation that will wait for a result to be provided.

33.3 Futures in Python

The `concurrent.futures` library was introduced into Python in version 3.2 (and is also available in Python 2.5 onwards). The `concurrent.futures` library provides the `Future` class and a high level API for working with Futures.

The `concurrent.futures.Future` class encapsulates the asynchronous execution of a callable object (e.g. a function or method).

The `Future` class provides a range of methods that can be used to obtain information about the state of the future, retrieve results or cancel the future:

- `cancel()` Attempt to cancel the Future. If the Future is currently being executed and cannot be cancelled then the method will return `False`, otherwise the call will be cancelled and the method will return `True`.
- `cancelled()` Returns `True` if the Future was successfully cancelled.
- `running()` Returns `True` if the Future is currently being executed and cannot be cancelled.
- `done()` Returns `True` if the Future was successfully cancelled or finished running.
- `result(timeout=None)` Return the value returned by the Future. If the Future hasn't yet completed then this method will wait up to timeout seconds. If the call hasn't completed in timeout seconds, then a `TimeoutError` will be raised. timeout can be an `int` or `float`. If timeout is not specified or `None`, there is no limit to the wait time. If the future is cancelled before completing then the `CancelledError` will be raised. If the call raised, this method will raise the same exception.

It should be noted however, that Future instances should not be created directly, rather they should be created via the `submit` method of an appropriate *executor*.

33.3.1 Future Creation

Futures are created and executed by Executors. An `Executor` provides two methods that can be used to execute a `Future` (or Futures) and one to shut down the executor.

At the root of the executor class hierarchy is the `concurrent.futures.Executor` abstract class. It has two subclasses:

- the `ThreadPoolExecutor` and
- the `ProcessPoolExecutor`.

The `ThreadPoolExecutor` uses *threads* to execute the futures while the `ProcessPoolExecutor` uses separate *processes*. You can therefore choose how you want the Future to be executed by specifying one or other of these executors.

33.3.2 Simple Example Future

To illustrate these ideas, we will look at a very simple example of using a `Future`.

To do this we will use a simple `worker` function; similar to that used in the previous chapters:

```
from time import sleep

# define function to be used with future
def worker(msg):
    for i in range(0, 10):
        print(msg, end='', flush=True)
        sleep(1)
    return i
```

The only difference with this version of `worker` is that it also returns a *result* which is the number of times that the `worker` printed out the message.

We can of course invoke this method inline as follows:

```
res = worker('A')
print(res)
```

We can make the invocation of this method into a `Future`. To do this we use a `ThreadPoolExecutor` imported from the `concurrent.futures` module. We will then submit the `worker` function to the pool for execution. This returns a reference to a `Future` which we can use to obtain the result:

```
from time import sleep
from concurrent.futures import ThreadPoolExecutor

print('Setting up the ThreadPoolExecutor')

pool = ThreadPoolExecutor(1)

# Submit the function ot the pool to run
# concurrently - obtain a future from pool
print('Submitting the worker to the pool')
future = pool.submit(worker, 'A')

print('Obtained a reference to the future object', future)

# Obtain the result from the future - wait if necessary
print('future.result():', future.result())

print('Done')
```

The output from this is:

```
Setting up the ThreadPoolExecutor
Submitting the worker to the pool
AAObtained a reference to the future object <Future at
0x1086ea8d0 state=running>
AAAAAAAAfuture.result(): 9
Done
```

Notice how the output from the main program and the worker is interwoven with two 'A's being printed out before the message starting 'Obtained a…'.

In this case a new `ThreadPoolExecutor` is being created with one thread in the pool (typically there would be multiple threads in the pool but one is being used here for illustrative purposes).

The `submit()` method is then used to submit the function worker with the parameter 'A' to the `ThreadPoolExecutor` for it to schedule execution of the function. The `submit()` method returns a Future object.

The main program then waits for the future object to return a result (by calling the `result()` method on the future). This method can also take a timeout.

To change this example to use Processes rather than Threads all that is needed is to change the pool executor to a `ProcessPoolExecutor`:

```
from concurrent.futures import ProcessPoolExecutor

print('Setting up the ThreadPoolExecutor')
pool = ProcessPoolExecutor(1)

print('Submitting the worker to the pool')
future = pool.submit(worker, 'A')

print('Obtained a reference to the future object', future1)
print('future.result():', future.result())
print('Done')
```

The output from this program is very similar to the last one:

```
Setting up the ThreadPoolExecutor
Submitting the worker to the pool
Obtained a reference to the future object <Future at
0x109178630 state=running>
AAAAAAAAAAfuture.result(): 9
Done
```

The only difference is that in this particular run the message starting 'Obtained a..' is printed out before any of the 'A's are printed; this may be due to the fact that a `Process` initially takes longer to set up than a `Thread`.

33.4 Running Multiple Futures

Both the `ThreadPoolExecutor` and the `ProcessPoolExecutor` can be configured to support multiple Threads/Processes via the pool. Each task that is submitted to the pool will then run within a separate Thread/Process. If more tasks are submitted than there are Threads/Processes available, then the submitted task will wait for the first available Thread/Process and then be executed. This can act as a way of managing the amount of concurrent work being done.

For example, in the following example, the `worker()` function is submitted to the pool four times, but the pool is configured to use threads. Thus the fourth worker will need to wait until one of the first three completes before it is able to execute:

```
from concurrent.futures import ThreadPoolExecutor

print('Starting...')
pool = ThreadPoolExecutor(3)
future1 = pool.submit(worker, 'A')
future2 = pool.submit(worker, 'B')
future3 = pool.submit(worker, 'C')
future4 = pool.submit(worker, 'D')
print('\nfuture4.result():', future4.result())
print('All Done')
```

When this runs we can see that the Futures for A, B and C all run concurrently but D must wait until one of the others finishes:

```
Starting...
ABCACBCABCBABCACBACABCBACABCBADDDDDDDDDD
future4.result(): 9
All Done
```

The main thread also waits for `future4` to finish as it requests the result which is a blocking call that will only return once the future has completed and generates a result.

Again, to use Processes rather than Threads all we need to do is to replace the `ThreadPoolExecutor` with the `ProcessPoolExecutor`:

```
from concurrent.futures import ProcessPoolExecutor

print('Starting...')
pool = ProcessPoolExecutor(3)
future1 = pool.submit(worker, 'A')
future2 = pool.submit(worker, 'B')
future3 = pool.submit(worker, 'C')
future4 = pool.submit(worker, 'D')
print('\nfuture4.result():', future4.result())
print('All Done')
```

33.4.1 Waiting for All Futures to Complete

It is possible to wait for all futures to complete before progressing. In the previous section it was assumed that `future4` would be the last future to complete; but in many cases it may not be possible to know which *future* will be the last to complete. In such situations it is very useful to be able to wait for all the futures to complete before continuing. This can be done using the `concurrent.futures.wait` function. This function takes a collection of futures and optionally a `timeout` and a `return_when` indicator.

```
wait(fs, timeout=None, return_when=ALL_COMPLETED)
```

where:

- `timeout` can be used to control the maximum number of seconds to wait before returning. timeout can be an `int` or `float`. If timeout is not specified or `None`, there is no limit to the wait time.
- `return_when` indicates when this function should return. It must be one of the following constants:
  - `FIRST_COMPLETED` The function will return when any future finishes or is cancelled.

- FIRST_EXCEPTION The function will return when any future finishes by raising an exception. If no future raises an exception, then it is equivalent to ALL_COMPLETED.
- ALL_COMPLETED The function will return when all futures finish or are cancelled.

The wait() function returns two sets done and not_done. The first set contains the futures that completed (finished or were cancelled) before the wait completed. The second set, the not_dones, contains uncompleted futures.

We can use the wait() function to modify out previous example so that we no longer rely on future4 finishing last:

```
from concurrent.futures import ProcessPoolExecutor
from concurrent.futures import wait
from time import sleep

def worker(msg):
    for i in range(0,10):
        print(msg,end='', flush=True)
        sleep(1)
    return i

print('Starting...setting up pool')
pool = ProcessPoolExecutor(3)
futures = []

print('Submitting futures')
future1 = pool.submit(worker, 'A')
futures.append(future1)
future2 = pool.submit(worker, 'B')
futures.append(future2)
future3 = pool.submit(worker, 'C')
futures.append(future3)
future4 = pool.submit(worker, 'D')
futures.append(future4)

print('Waiting for futures to complete')
wait(futures)
print('\nAll Done')
```

The output from this is:

```
Starting...setting up pool
Submitting futures
Waiting for futures to complete
ABCABCABCABCABCBCACBACBABCADDDDDDDDDD
All Done
```

Note how each future is added to the list of futures which is then passed to the wait() function.

33.4.2 Processing Results as Completed

What if we want to process each of the results returned by our collection of futures? We could loop through the futures list in the previous section once all the results have been generated. However, this means that we would have to wait for them all to complete before processing the list.

In many situations we would like to process the results as soon as they are generated without being concerned if that is the first, third, last or second etc.

The `concurrent.futures.as_completed()` function does preciously this; it will serve up each future in turn as soon as they are completed; with all futures eventually being returned but without guaranteeing the order (just that as soon as a future is finished generating a result it will be immediately available).

For example, in the following example, the `is_even()` function sleeps for a random number of seconds (ensuring that different invocations of this function will take different durations) then calculates a result:

```
from concurrent.futures import ThreadPoolExecutor, as_completed
from time import sleep
from random import randint

def is_even(n):
    print('Checking if', n , 'is even')
    sleep(randint(1, 5))
    return str(n) + ' ' + str(n % 2 == 0)

print('Started')
data = [1, 2, 3, 4, 5, 6]
pool = ThreadPoolExecutor(5)
futures = []

for v in data:
    futures.append(pool.submit(is_even, v))

for f in as_completed(futures):
    print(f.result())

print('Done')
```

The second `for` loop will loop through each future as they complete printing out the result from each, as shown below:

```
Started
Checking if 1 is even
Checking if 2 is even
Checking if 3 is even
Checking if 4 is even
Checking if 5 is even
Checking if 6 is even
1 False
4 True
5 False
3 False
2 True
6 True
Done
```

As you can see from this output although the six futures were started in sequence the results returned are in a different order (with the returned order being 1, 4, 5, 3, 2 and finally 6).

33.5 Processing Future Results Using a Callback

An alternative to the `as_complete()` approach is to provide a function that will be called once a result has been generated. This has the advantage that the main program is never paused; it can continue doing whatever is required of it.

The function called once the result is generated is typically known as a *callback* function; that is the future calls *back* to this function when the result is available.

Each future can have a separate call back as the function to invoke is set on the future using the `add_done_callback()` method. This method takes the name of the function to invoke.

For example, in this modified version of the previous example, we specify a call back function that will be used to print the futures result. This call back function is called `print_future_result()`. It takes the future that has completed as its argument:

```
from concurrent.futures import ThreadPoolExecutor
from time import sleep
from random import randint

def is_even(n):
    print('Checking if', n, 'is even')
    sleep(randint(1, 5))
    return str(n) + ' ' + str(n % 2 == 0)

def print_future_result(future):
    print('In callback Future result: ', future.result())

print('Started')
data = [1, 2, 3, 4, 5, 6]

pool = ThreadPoolExecutor(5)

for v in data:
    future = pool.submit(is_even, v)
    future.add_done_callback(print_future_result)

print('Done')
```

When we run this, we can see that the call back function is called after the main thread has completed. Again, the order is unspecified as the `is_even()` function still sleeps for a random amount of time.

```
Started
Checking if 1 is even
Checking if 2 is even
Checking if 3 is even
Checking if 4 is even
Checking if 5 is even
Done
In callback Future result:  1 False
Checking if 6 is even
In callback Future result:  5 False
In callback Future result:  4 True
In callback Future result:  3 False
In callback Future result:  2 True
In callback Future result:  6 True
```

33.6 Online Resources

See the following online resources for information on Futures:

- https://docs.python.org/3/library/concurrent.futures.html The Python standard Library documentation on Futures.
- https://pymotw.com/3/concurrent.futures The Python Module of the Week page on Futures.
- https://www.blog.pythonlibrary.org/2016/08/03/python-3-concurrency-the-concurrent-futures-module an alternative tutorial on Python Futures.

33.7 Exercises

In mathematics, the *factorial* of a positive integer n, denoted by n!, is the product of all positive integers less than or equal to n. For example,

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

Note that the value of 0! is 1.

Write a `Future` that will calculate the factorial of any number with the result being printed out via a call back function.

There are several ways in which the factorial value can be calculated either using a for loop or a recursive function. In either case sleep for a millisecond between each calculation.

Start multiple Futures for different factorial values and see which comes back first.