Futures

This lesson gets into the details of working with threading.Future (not to be confused with asycncio.Future).

Futures

You can think of **`Future`** as an entity that represents a deferred computation that may or may not have been completed. It is an object that represents the outcome of a computation to be completed in future. We can query about the status of the deferred computation using the methods exposed by the **`Future`** class. Some of the useful ones are:

- **done**: Returns true if the execution of the callable was successfully completed or cancelled.
- **cancel**: Attempts to cancel execution of a callable. Note if the task is already finished or executing the method returns False.
- **cancelled**: Returns True if the task was successfully cancelled.
- **running**: Returns True if the task is currently running and can't be cancelled.

You should treat futures as read-only entities and never create them. They are returned by sub classes of **`Executor`** class.

In the example below, we'll exercise all the above methods. Note that we attempt to cancel the task but since it is already executing the framework is unable to cancel it.

```
from concurrent.futures import ThreadPoolExecutor
import time


def square(item):
    # simulate a computation by sleeping
    time.sleep(5)
    return item * item


if __name__ == '__main__':
    executor = ThreadPoolExecutor(max_workers=10)

    future = executor.submit(square, 7)

    print("is running : " + str(future.running()))
    print("is done : " + str(future.done()))
    print("Attempt to cancel : " + str(future.cancel()))
    print("is cancelled : " + str(future.cancelled()))

    executor.shutdown()
```

```
from concurrent.futures import ThreadPoolExecutor
import time


def square(item):
    # simulate a computation by sleeping
    time.sleep(5)
    return item * item


if __name__ == '__main__':
    executor = ThreadPoolExecutor(max_workers=10)

    future = executor.submit(square, 7)

    print("is running : " + str(future.running()))
    print("is done : " + str(future.done()))
    print("Attempt to cancel : " + str(future.cancel()))
    print("is cancelled : " + str(future.cancelled()))

    executor.shutdown()
```

Exception in Future

If an exception occurs in the callable, it can be retrieved using the **`exception()`** method. Examine **line#14** in the runnable code below, where we retrieve the exception occurred in the callable and print it. Note that if you asked for **`result()`** the exception from the callable would be thrown and the program would exit.

```
from concurrent.futures import ThreadPoolExecutor


def square(item):
    item = None
    return item * item


if __name__ == '__main__':
    executor = ThreadPoolExecutor(max_workers=1)
    lst = list()

    future = executor.submit(square, 7)
    ex = future.exception()
    print(ex)

    executor.shutdown()
```

Cancelling a task

We can cancel a task if it is not already running or finished. In the example below, we submit two tasks for execution but we restrict the max number of thread to be one so that the execution of the two tasks is serial. The first task sleeps for five seconds and we attempt to cancel the second task within those five seconds. We are able to successfully cancel the second task since it has not been started. The output of the program below only prints the square of the first integer.

```
from concurrent.futures import ThreadPoolExecutor
import time


def square(item):
    time.sleep(5)
    print(item * item)


if __name__ == '__main__':
    executor = ThreadPoolExecutor(max_workers=1)

    future1 = executor.submit(square, 7)
    future2 = executor.submit(square, 9)

    print("Attempt to cancel : " + str(future2.cancel()))
    print("is running : " + str(future2.running()))
    print("is done : " + str(future2.done()))
    print("is cancelled : " + str(future2.cancelled()))

    executor.shutdown()
```

Adding Callbacks

So far we have retrieved results from futures using the **`result()`** method. However, this call is blocking and there may be situations where we don't want our program to block. The solution to this dilemma is to add a callback to the future which is invoked when the future has completed or is canceled. The **`add_done_callback()`** takes a callable as the only argument. In the example below, we attach two callbacks to the future we submit. The callbacks are invoked in the order in which they are added.

Adding callbacks to futures

```
    future = executor.submit(square, 7)
    future.add_done_callback(my_special_callback)
    future.add_done_callback(my_other_special_callback)
```

```
from concurrent.futures import ThreadPoolExecutor


def square(item):
    return item * item


def my_special_callback(ftr):
    res = ftr.result()
    print("my_special_callback invoked " + str(res))


def my_other_special_callback(ftr):
    res = ftr.result()
    print("my_other_special_callback invoked " + str(res * res))


if __name__ == '__main__':
    executor = ThreadPoolExecutor(max_workers=10)

    future = executor.submit(square, 7)
    future.add_done_callback(my_special_callback)
    future.add_done_callback(my_other_special_callback)

    executor.shutdown(wait=False)
```

Back
Pool Executors

Next
Miscellaneous Functions

✓ Mark as Completed

Report an

Ask a Question
(https://discuss.educative.io/tag/futures__concurrent-package__python-concurrency-

issue for-senior-engineering-interviews)