(/learn)

With

This lesson introduces the use of with and context management with synchronization primitives.

With

Programs often use resources other than CPU time, including access to local disks, network sockets, and databases etc. The usage pattern is usually a **`try-except-finally`** block. Any cleanup actions are performed in the **`finally`** block. An alternative to the usual boilerplate code is to use the **`with`** statement. The **`with`** statement wraps the execution of a block of statements in a context defined by a ***context manager*** object.

Context Management Protocol

A context manager object abides by the context management protocol, which states that an object defines the following two methods. Python calls these two methods at appropriate times in the resource management cycle:

- **`__enter__()`**
- **`__exit__()`**

The with statement is used as:

```
with context-expression [as target]:
    statement#1
    statement#2
        .
        .
        .
    statement#n
```

- **`__enter__()`** should return an object that is assigned to the variable after **as** in the above template. By default the returned object is None, and is optional. A common pattern is to return self and keep the functionality required within the same class.

- **`__exit__()`** is called on the original Context Manager object, not the object returned by **`__enter__()`**. If, however, we return self in the **`__enter__()`** method, then it is obviously the same object.

- If an error is raised in **`__init__()`** or **`__enter__()`** then the code block is never executed and **`__exit__()`** is not called.

- Once the code block is entered, **`__exit__`** is always called, even if an exception is raised in the code block.

- In case an exception is raised when executing the block of code wrapped by the **`with`** statement, three values consisting of the exception types, its value and traceback are passed as arguments to the **`__exit__()`** method. These parameters are None if no exceptions occur. Lastly, if an exception was raised and the **`__exit__()`** method returns True, the exception is suppressed. On the contrary, if **`__exit__()`** returns false then the exception is re-raised.

Context managers can be used in scenarios to save and restore global state, lock and unlock resources, close opened files, etc.

Example

The most common use of the **with** statement happens when we manipulate files. Without the **with** statement, file manipulation would look as follows:

```
    file = None
    try:
        file = open("test.txt")
    except Exception as e:
        print(e)

    finally:
        if file is not None:
            file.close()
```

Using the **with** statement the above code is simplified as:

```
with open("test.txt") as file:
    data = file.read()
```

The **with** statement helps simplify some common resource management patterns by abstracting their functionality and allowing them to be factored out and reused. The code becomes expressive and easier to read. In fact, we can write a class that implements the enter and exit methods and makes it compatible with the **with** statement. An example is shown below:

```python
class ExampleClass(object):

    def __init__(self, val):
        print("init")
        self.val = val

    def display(self):
        print(self.val)

    def __enter__(self):
        print("enter invoked")
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        print("exit invoked")


if __name__ == "__main__":
    with ExampleClass("hello world") as example:
        example.display()
```

```python
class ExampleClass(object):

    def __init__(self, val):
        print("init")
        self.val = val

    def display(self):
        print(self.val)

    def __enter__(self):
        print("enter invoked")
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        print("exit invoked")


if __name__ == "__main__":
    with ExampleClass("hello world") as example:
        example.display()
```

Using With Statement in Multithreading

Some classes in the **`threading`** module such as **`Lock`**, support the context management protocol and can be used with the **`with`** statement. In the example below, we reproduce an example from an earlier section and use the **`with`** statement with the **`Lock`** object **`my_lock`**. Note, we don't need to explicitly **`acquire()`** and **`release()`** the lock object. The context manager automatically takes care of managing the lock for us.

```
sharedState = [1, 2, 3]
my_lock = Lock()


def thread1_operations():

    with my_lock:
        print("{0} has acquired the lock".format(current_threa
d().getName()))

        time.sleep(3)  #
        sharedState[0] = 777

        print("{0} about to release the lock".format(current_t
hread().getName()))

    print("{0} released the lock".format(current_thread().getN
ame()))


def thread2_operations():
    print("{0} is attempting to acquire the lock".format(curre
nt_thread().getName()))

    with my_lock:
        print("{0} has acquired the lock".format(current_threa
d().getName()))

        print(sharedState[0])
        print("{0} about to release the lock".format(current_t
hread().getName()))

    print("{0} released the lock".format(current_thread().getN
ame()))


if __name__ == "__main__":
    # create and run the two threads
```

```
    thread1 = Thread(target=thread1_operations, name="thread1"
)
    thread1.start()

    thread2 = Thread(target=thread2_operations, name="thread2"
)
    thread2.start()

    # wait for the two threads to complete
    thread1.join()
    thread2.join()
```

```
import time
from threading import Lock
from threading import Thread
from threading import current_thread

sharedState = [1, 2, 3]
my_lock = Lock()


def thread1_operations():

    with my_lock:
        print("{0} has acquired the lock".format(current_thread().getName()))

        time.sleep(3)  #
        sharedState[0] = 777

        print("{0} about to release the lock".format(current_thread().getName()))

    print("{0} released the lock".format(current_thread().getName()))


def thread2_operations():
    print("{0} is attempting to acquire the lock".format(current_thread().getName()

    with my_lock:
        print("{0} has acquired the lock".format(current_thread().getName()))

        print(sharedState[0])
        print("{0} about to release the lock".format(current_thread().getName()))

    print("{0} released the lock".format(current_thread().getName()))


if __name__ == "__main__":
    # create and run the two threads
    thread1 = Thread(target=thread1_operations, name="thread1")
    thread1.start()

    thread2 = Thread(target=thread2_operations, name="thread2")
    thread2.start()

    # wait for the two threads to complete
    thread1.join()
    thread2.join()
```

Back

Next

Barrier

Quiz 1

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/with__threading-module__python-concurrency-for-senior-engineering-interviews)