## Part 2: Context Managers

Bookmark this page

## Part 2: Context Managers

You've seen the 'with' statement – probably when working with files. Now you'll learn what that is all about. A large source of repetition in code deals with the handling of external resources. As an example, how many times do you think you might type something like the following code:

```
file_handle = open('filename.txt', 'r')
file_content = file_handle.read()
file_handle.close()
# do some stuff with the contents
```

Not only is this a couple extra lines of code to write, it's also prone to error. What happens if you forget to call close()? What happens if reading the file raises an exception? Perhaps you really should write it something more like this:

```
try:
    file_handle = open(...)
    file_content = file_handle.read()
except IOError:
    print("The file couldn't be opened")
finally:
    file_handle.close()
```

And that is getting ugly, and hard to get right.

### Handling General Resources

Leaving an open file handle laying around is bad enough. What if the resource is a network connection, or a database cursor? Starting in version 2.5, Python provides a structure for reducing the repetition needed to handle resources like this: context managers. Context managers encapsulate the setup, error handling, and teardown of resources in a few simple steps. The key is to use the 'with' statement. Since the introduction of 'with' in Pep343 the above seven lines of defensive code have been replaced with this simple form:

```
with open('filename', 'r') as file_handle:
    file_content = file_handle.read()
# do something with file_content
```

The 'open' builtin is defined as a context manager. The resource it returns (file_handle) is automatically and reliably closed when the code block ends.

At this point in Python history, many functions you might expect to behave this way do:

* open() works as a context manager.
* network connections via socket() do as well.
* most implementations of database wrappers can open connections or cursors as context managers.

But what if you are working with a library that doesn't support this, urllib for instance? Close it automatically. There are a couple of ways you can go. If the resource in question has a close() method, then you can simply use the closing context manager from contextlib to handle the issue:

```
from urllib import request
from contextlib import closing

with closing(request.urlopen('http://google.com')) as web_connection:
    # do something with the open resource
# and by here, it will be closed automatically
```

But what if the thing doesn't have a close() method, or you're creating the thing and it shouldn't have a close() method? Do it yourself. If you need to support resource management of some sort, you can write a context manager of your own with the context manager protocol. The interface is simple. It must be a class that implements two more of the nifty python special methods.

```
__enter__()
```

Called when the with statement is run, it should return something to work with in the created context.

```
__exit__(self, e_type, e_val, e_traceback):
```

Clean-up that needs to happen is implemented here. The arguments will be the exception raised in the context. If the exception will be handled here, return True. If not, return False. Let's see this in action to get a sense of what happens. Consider this code:

```
class Context(object):
    """
    From Doug Hellmann, PyMOTW
    https://pymotw.com/3/contextlib/#module-contextlib
    """

    def __init__(self, handle_error):
        print('__init__({})'.format(handle_error))
        self.handle_error = handle_error

    def __enter__(self):
        print('__enter__()')
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        print('__exit__({}, {}, {})'.format(exc_type, exc_val, exc_tb))
        return self.handle_error
```

This class doesn't do much of anything, but playing with it can help clarify the order in which things happen:

```
In []: %paste
    In []: with Context(True) as foo:

        ....: print('This is in the context')
        ....: raise RuntimeError('this is the error message')

## -- End pasted text --
__init__(True)
__enter__()
This is in the context
__exit__(<class 'RuntimeError'>, this is the error message,
    <traceback object at 0x1047873c8>)
```

Because the __exit__ method returns True, the raised error is handled.

What if we try with False?

```
In []: with Context(False) as foo:
    ...: print("this is in the context")
    ...: raise RuntimeError('this is the error message')
    ...:
__init__(False)
__enter__()
this is in the context
__exit__(<class 'RuntimeError'>, this is the error message, <traceback object at 0x10349e888>)
---------------------------------------------------------------------------
RuntimeError Traceback (most recent call last)
<ipython-input-3-8837b3d7f123> in <module>()
    1 with Context(False) as foo:
    2 print("this is in the context")
----> 3 raise RuntimeError('this is the error message')

RuntimeError: this is the error message
```

So this time, the context manager did not catch the error – so it was raised in the usual way. In real life, a context

manager could have pretty much any error raised in its context. And the context manager will likely only be able to properly handle particular exceptions – so the __exit__ method takes all the information about the exception as parameters:

```
    def __exit__(self, exc_type, exc_val, exc_tb)
        exc_type: the type of the Exception
        exc_val: the value of the Exception
        exc_tb: the Exception Traceback object
```

The type of exception lets you check if this is an excpetion you know how to handle:

```
if exc_type is RuntimeError:
```

```
# Deal with it.
```

The value is the exception object itself and the traceback is a full traceback object. Traceback objects hold all the information about the context in which and error occurred. It's pretty advanced stuff, so you can mostly ignore it, but if you want to know more, there are tools for working with them in the traceback module.

https://docs.python.org/3/library/traceback.html

### The contextmanager decorator

Similar to writing iterable classes, there's a fair bit of bookkeeping involved. It turns out you can take advantage of generator functions to do the bookkeeping for you.  contextlib.contextmanager() will turn a generator function into context manager.

```
from contextlib import contextmanager

@contextmanager
def context(boolean):
    print("__init__ code here")
    try:
        print("__enter__ code goes here")
        yield object()
    except Exception as e:
         print("errors handled here")
         if not boolean:
             raise e
     finally:
         print("__exit__ cleanup goes here")
```

The code is similar to the class defined previously and using it has similar results. We can handle errors or, we can allow them to propagate:

```
In [51]: with context(False):
     ....: print("in the context")
     ....: raise RuntimeError("error raised")
 __init__ code here
 __enter__ code goes here
 in the context
 errors handled here
 __exit__ cleanup goes here
 ---------------------------------------------------------------------------
 RuntimeError Traceback (most recent call last)
```