CONTEXT MANAGERS

try…finally…

The `finally` section of a `try` always executes

```
try:
    …
except:
    …
finally:
    …
```

always executes

even if an exception occurs in `except` block

Works even if inside a function and a `return` is in the `try` or `except` blocks

Very useful for writing code that should execute no matter what happens

But this can get cumbersome!

There has to be a better way!

Pattern

create some object

do some work with that object

clean up the object after we're done using it

We want to make this easy

→ automatic cleanup after we are done using the object

Context Managers PEP 343

object returned from context (optional)

```
with context as obj_name:
    # with block (can use obj_name)

# after the with block, context is cleaned up automatically
```

Example

```
with open(file_name) as f:
    # file is now open

# file is now closed
```

enter the context (optional) an object is returned

exit the context

The context management protocol

Classes implement the context management protocol by implementing two methods:

`__enter__`	setup, and optionally return some object
`__exit__`	tear down / cleanup

```
with CtxManager() as obj:
    # do something
# done with context
```

```
mgr = CtxManager()
obj = mgr.__enter__()
try:
    # do something
finally:
    # done with context
    mgr.__exit__()
```

over simplified

exception handling

Use Cases

Very common usage is for opening a file (creating resource) and closing the file (releasing resource)

Context managers can be used for much more than creating and releasing resources

Common Patterns

- Open – Close
- Lock – Release
- Change – Reset
- Start – Stop
- Enter – Exit

Examples

- file context managers
- Decimal contexts

How Context Protocol Works

```
class MyClass:
    def __init__(self):
        # init class

    def __enter__(self):
        return obj

    def __exit__(self, + …):
        # clean up obj
```

works in conjunction with a `with` statement

```
my_obj = MyClass()
```

works as a regular class

`__enter__`, `__exit__` were not called

```
with MyClass() as obj:
```

→ creates an instance of `MyClass`

→ calls `my_instance.__enter__()`

→ return value from `__enter__` is assigned to `obj`

(not the instance of `MyClass` that was created)

→ no associated symbol, but an instance exists

→ `my_instance`

after the `with` block, or if an exception occurs inside the `with` block:

→ `my_instance.__exit__` is called

Scope of with block

The with block is not like a function or a comprehension

The scope of anything in the with block (including the object returned from __enter__)

is in the same scope as the with statement itself

```
# module.py

with open(fname) as f:
    row = next(f)

print(f)

print(row)
```

f is a symbol in global scope

row is also in the global scope

f is closed, but the symbol exists

row is available and has a value

The `__enter__` Method

```
def __enter__(self):
```

This method should perform whatever setup it needs to

It can optionally return an object → `as returned_obj`

That's all there is to this method

The `__exit__` Method

More complicated…

Remember the `finally` in a `try` statement? → always runs even if an exception occurs

`__exit__` is similar → runs even if an exception occurs in `with` block

But should it handle things differently if an exception occurred?

→ maybe → so it needs to know about any exceptions that occurred

→ it also needs to tell Python whether to silence the exception, or let it propagate

The __exit__ Method

```
with MyContext() as obj:
    raise ValueError

print ('done')
```

Scenario 1

`__exit__` receives error, performs some clean up and silences error

`print` statement runs

no exception is seen

Scenario 2

`__exit__` receives error, performs some clean up and let's error propagate

`print` statement does not run

the `ValueException` is seen

The __exit__ Method

Needs three arguments:
- → the exception type that occurred (if any, None otherwise)
- → the exception value that occurred (if any, None otherwise)
- → the traceback object if an exception occurred (if any, None otherwise)

Returns True or False:
- → True = silence any raised exception
- → False = do not silence a raised exception

```
def __exit__(self, exc_type, exc_value, exc_trace):
    # do clean up work here
    return True # or False
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-14-39a69b57f322> in <module>()
      1 with MyContext() as obj:
----> 2     raise ValueError
```

Code Exercises