Context Managers

Austin Bingham
COFOUNDER - SIXTY NORTH

@austin_bingham

Robert Smallshire
COFOUNDER - SIXTY NORTH

@robsmallshire

Overview

Learn what context managers are

Implementing context managers

How Python uses context managers

Interaction with exceptions

Managing resources in the presence of exceptions

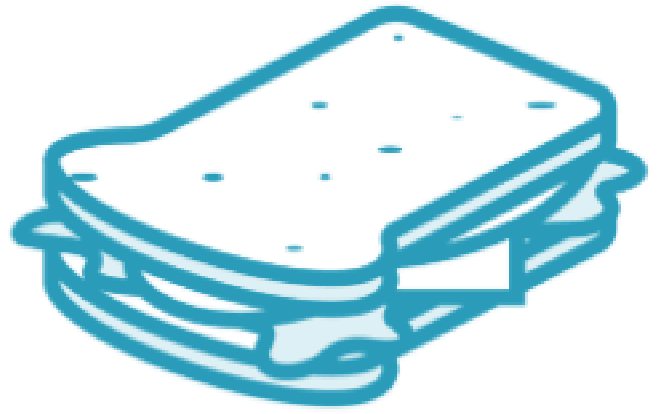
What Is a Context Manager?

An object designed to be used in a with statement

```
with expression:
    with-block
```

Must evaluate to a context manager

Methods around a Context



A context manager implements two methods.

The first is called before entering the with-block.

The second is called after exiting the with-block.

Context Manager Methods

setup	teardown
construction	destruction
allocation	deallocation
enter	exit

Context manager

An object that ensures that resources are properly and automatically handled.

The "enter" method ensures that the resource is ready for use.

The "exit" method ensures that the resource is cleaned up.

Files Are Context Managers

```
>>> with open('important_data.txt', 'w') as f:
...     f.write('The secret password is 12345')
...
28
>>> f.closed
True
>>>
```

The with-statement ensures that the file is properly closed, even when there is an exception in the with-block.

Context Manager Protocol

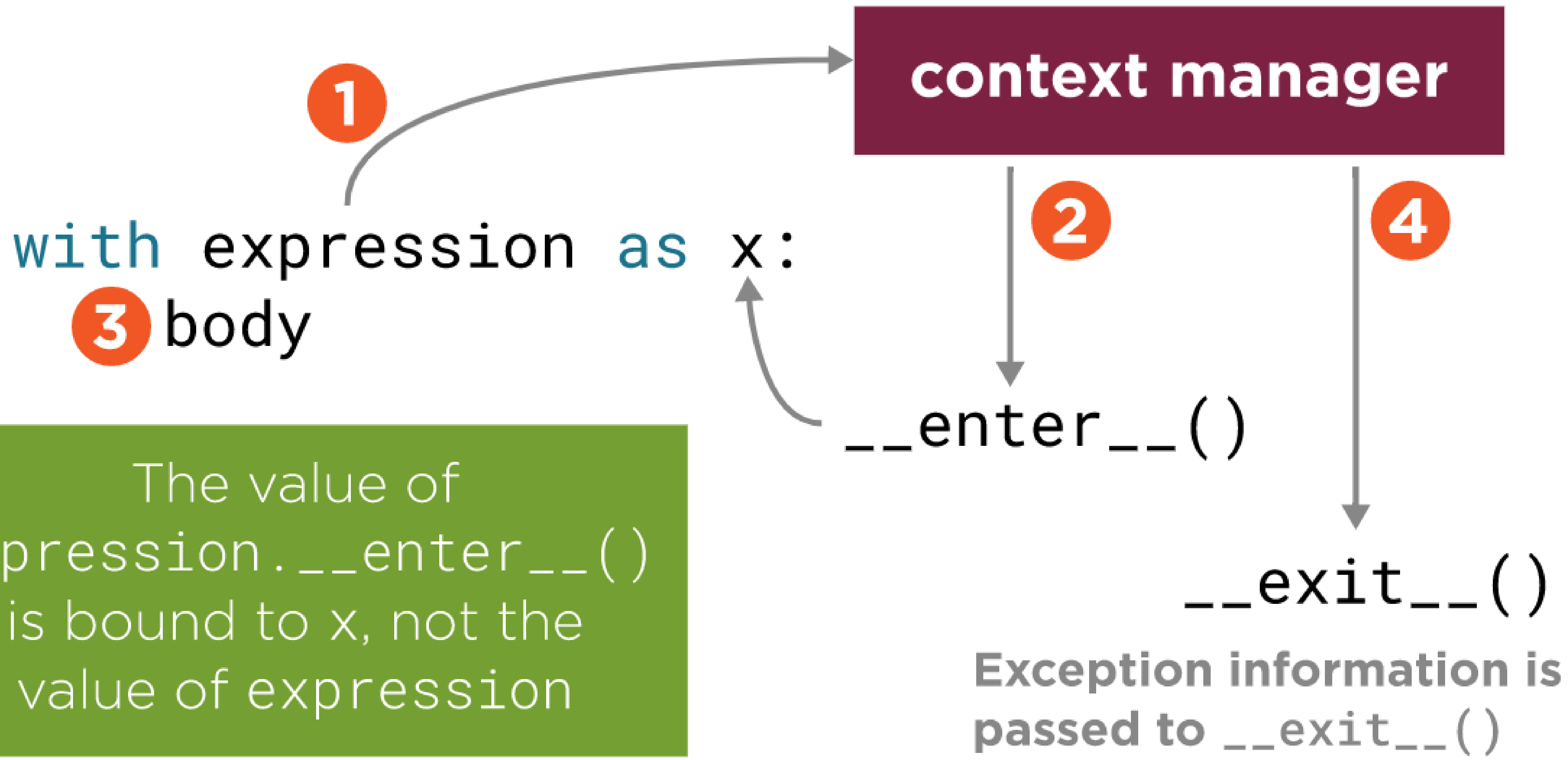
Context manager protocol

A context manager needs to support a specific protocol.

`__enter__()`

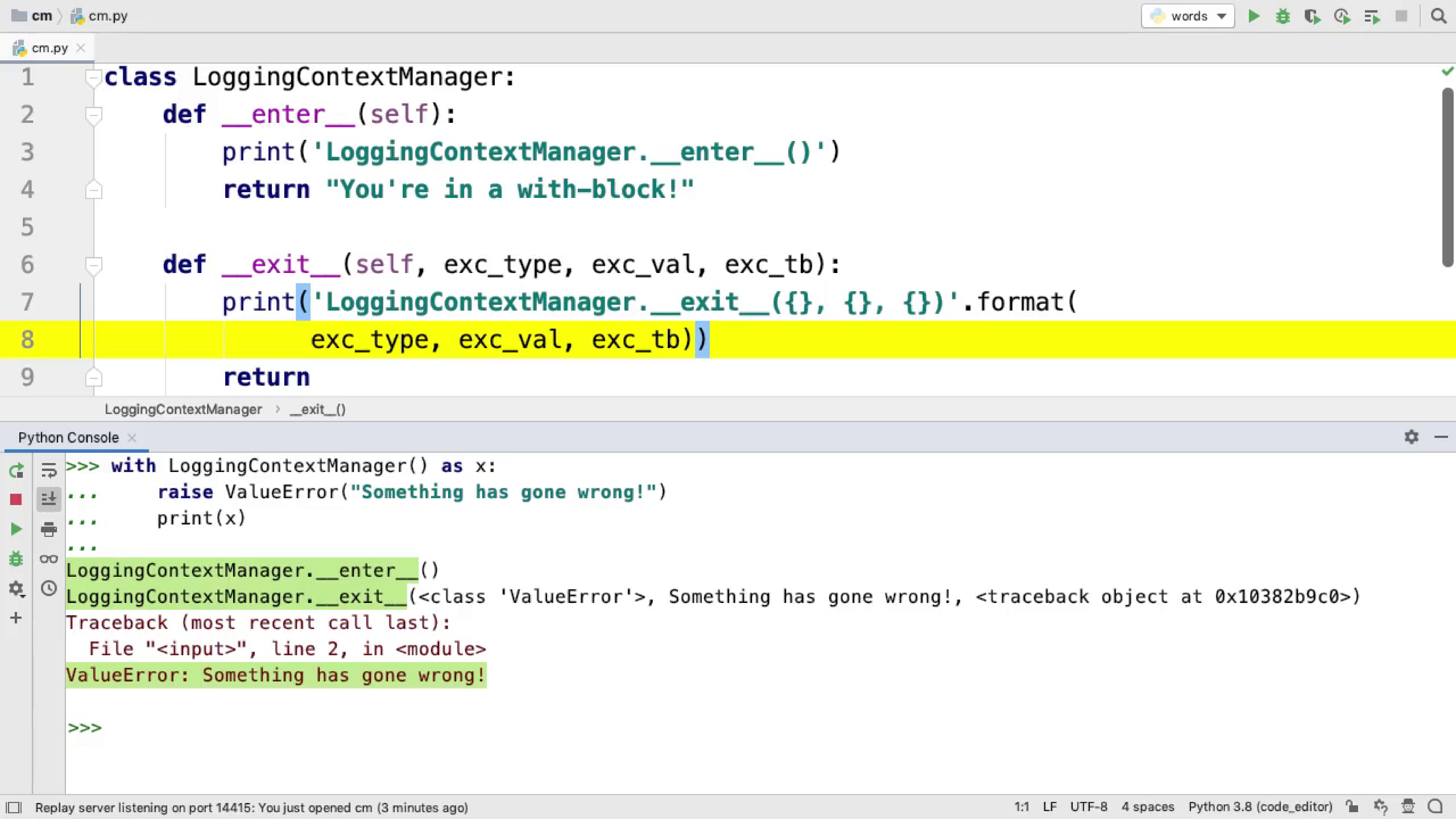
`__exit__()`

Context Manager Algorithm



The value of expression.__enter__() is bound to x, not the value of expression

Exception information is passed to __exit__()



```python
class LoggingContextManager:
    def __enter__(self):
        print('LoggingContextManager.__enter__()')
        return "You're in a with-block!"

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

```
>>> with LoggingContextManager() as x:
...     raise ValueError("Something has gone wrong!")
...     print(x)
...
LoggingContextManager.__enter__()
LoggingContextManager.__exit__(<class 'ValueError'>, Something has gone wrong!, <traceback object at 0x10382b9c0>)
Traceback (most recent call last):
  File "<input>", line 2, in <module>
ValueError: Something has gone wrong!

>>>
```

Replay server listening on port 14415: You just opened cm (3 minutes ago)   1:1 LF UTF-8 4 spaces Python 3.8 (code_editor)

`__enter__()`

Called on a context manager just before entering the with-block

The return value is bound to the as-variable

May return any value it wants, including `None`

Commonly returns the context manager itself

Returning `self` from `__enter__()`

```
>>> with open('password.txt', 'w') as f:
...     f.write('12345')
...
5
>>> f = open('a_file', 'w')
>>> with f as g:
...     print(f is g)
...
True
>>>
```

`__exit__()`

`__exit__()`

Executed after the with-block terminates

Handles exceptional exits from the with-block

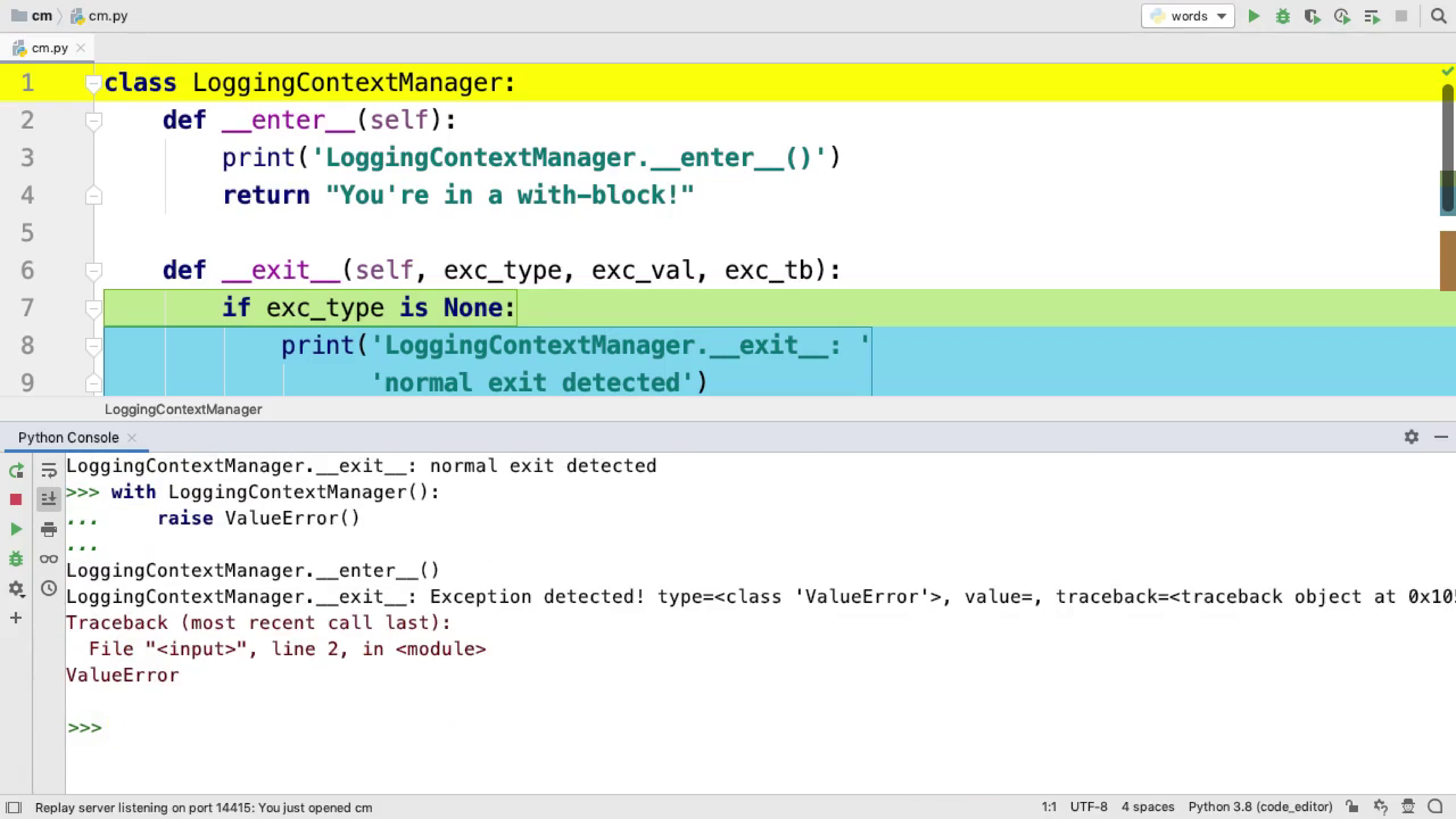
It receives the exception type, value, and traceback

The arguments are `None` when there is no exception

Exception Response

`__exit__()` often needs to choose an action based on whether an exception was raised

A common idiom is to check the type of exception



```python
class LoggingContextManager:
    def __enter__(self):
        print('LoggingContextManager.__enter__()')
        return "You're in a with-block!"

    def __exit__(self, exc_type, exc_val, exc_tb):
        if exc_type is None:
            print('LoggingContextManager.__exit__: '
                  'normal exit detected')
```

```
LoggingContextManager.__exit__: normal exit detected
>>> with LoggingContextManager():
...     raise ValueError()
...
LoggingContextManager.__enter__()
LoggingContextManager.__exit__: Exception detected! type=<class 'ValueError'>, value=, traceback=<traceback object at 0x10
Traceback (most recent call last):
  File "<input>", line 2, in <module>
ValueError

>>>
```

`__exit__()` and Propagating Exceptions

By default, `__exit__()` will propagate exceptions from the with-block to the enclosing context.

Propagating Exceptions from `__exit__()`

```
>>> try:
...     with LoggingContextManager():
...         raise ValueError('The system is down!')
... except ValueError:
...     print('*** ValueError escaped the with-block ***')
...
LoggingContextManager.__enter__()
LoggingContextManager.__exit__: Exception detected! type=<class 'ValueError'>, v
alue=The system is down!, traceback=<traceback object at 0x103c179c0>
*** ValueError escaped the with-block ***
>>>
```

If `__exit__()` returns a "falsy" value, exceptions will be propagated

It answers the question "Should I swallow exception?"

By default it propagates exceptions

This is because functions return `None` by default

Raising Exceptions in `__exit__()`

`__exit__()` should not re-raise the exception is receives from the with-block

To ensure that the exception is propagated, simply return `False`

`__exit__()` should only raise an exception if something goes wrong in the function itself

Expansion of the with-statement

PEP 343

The "with" statement

```python
mgr = (EXPR)
exit = type(mgr).__exit__  # Not calling it yet
value = type(mgr).__enter__(mgr)
exc = True
try:
    try:
        VAR = value  # Only if "as VAR" is present
        BLOCK
    except:
        # The exceptional case is handled here
        exc = False
        if not exit(mgr, *sys.exc_info()):
            raise
        # The exception is swallowed if exit() returns true
finally:
    # The normal and non-local-goto cases are handled here
    if exc:
        exit(mgr, None, None, None)
```

Summary

Context managers are objects that have both `__enter__` and `__exit__` methods

The main use of context managers is for properly managing resources

The expression in a with-statement must evaluate to a context manager object

A with-statement calls its context manager's `__enter__` method before entering the with-block

The return value of `__enter__` is bound to the as-variable of the with-statement, if it's defined

Summary

`__exit__` is called after the with-block is complete

If the with-block exits with an exception, the exception information is passed to `__exit__`

`__exit__` can control the propagation of an exception by return `False` to propagate it or `True` to swallow it

The with-statement is syntactic sugar for a much larger and more complicated body of code