Context Managers

The `with` statement

- Sometimes you don't want to catch/silence exceptions, but want to be sure some cleanup is done regardless of what happens

In [ ]:
```python
def count_lines(filename):
    """Count the number of lines in a file"""
    file = open(filename, 'r')
    try:
        s = 1
        return(len(file.readline()) + s)
    finally:
        print('finally')
        file.close()

# if file fails to open, exception will be thrown before try/finally block, and
# anything else that can go wrong will go wrong inside the block, so the file
# will be open by the time we get to the finally block…so what's the problem?
```

The `with` statment (cont'd)

- **`with`** introduces a new block, like try, but with a very different purpose in mind!
    - **`with`** statement sets up a temporary context and reliably tears it down, under the control of a context manager object

In [ ]:
```python
with open('myfile', 'w') as file:
    file.write('Now is the time')
    print('inside with block, file.closed =', file.closed)

print('outside with block, file.closed =', file.closed)
```

- **`with`** statement designed to simplify **`try/finally`** pattern
- prevents errors
- reduces boilerplate code

- makes APIs safer

Context Managers

- the context manager runs some code before the with clause is executed and runs some cleanup code afterwards
    - in the case of open(), the file is opened prior to the with block being entered, and closed at the end of the block
    - in this case, the context revolves around an open file object, which is made available to the block via the name given in the as clause
        - in other words, all operations inside the with clause are said to be executed within the context of the open file
    - in other words, there need not be such an object, and in that case, the as clause is optional
- context manager protocol consists of the `__enter__()` and `__exit__()` methods

In [ ]:

```
# A context manager to suppress exceptions
class SuppressErrors():
    def __init__(self, *exceptions):
        """Populate list of exceptions to suppress.

        If list is empty, suppress ALL exceptions because all exceptions
        inherit from the base class Exception.
        """
        # Add some instrumentation so we can see how this works
        print('in __init__() method')
        if not exceptions:
            exceptions = (Exception,)
        self.exceptions = exceptions

    # __enter__() called just prior to execution of code inside with block
    def __enter__(self):
        """Nothing to do here. Exception handling occurs in __exit__()."""
        return 'test'

    # Takes 3 args and is called when code block finishes
    def __exit__(self, exc_class, exc_instance, tb):
        """This method "suppresses" exceptions.

        Exception suppression is performed by way of the return value.
        If it completes without a return value, the original exception
        will be re-raised. Returning True catches the exception and
        suppresses it.
        """
        print("in __exit__() method")
        if isinstance(exc_instance, self.exceptions):
            import traceback
            traceback.print_tb(tb)
            return True
        return False
```

In [ ]:

```
with SuppressErrors(DivideByZeroException) as something:
    print(f'something is "{something}"')
    3 / 0
print('all is well')
```

A Frivolous but Fun Example from Fluent Python

In [ ]:

```
class LookingGlass:
    def __enter__(self):
        import sys
        self.original_write = sys.stdout.write
        sys.stdout.write = self.reverse_write
        return 'JABBERWOCKY'

    def reverse_write(self, text):
        self.original_write(text[::-1])

    def __exit__(self, exc_type, exc_value, traceback):
        import sys
        sys.stdout.write = self.original_write
        if exc_type is ZeroDivisionError:
            print('Please DO NOT divide by zero!')
            return True

with LookingGlass() as what:
    print('Lewis Carroll')
    print(what)
    3 / 0

print('back to normal')
```

In [ ]:

```
manager = LookingGlass()
print(manager)
monster = manager.__enter__()
```

```
print(monster == 'JABBERWOCKY')
print(monster)
print(manager)
manager.__exit__(None, None, None)
print(monster)
```

Context Manager Logging

In [ ]:

```
import logging

class LogBlock:
    def __init__(self, logger, level=logging.INFO):
        self._logger = logger
        self._level = level

    def __enter__(self):
        self._logger.log(self._level, 'Enter')

    def __exit__(self, ex_type, ex_value, ex_tb):
        if ex_type is None:
            self._logger.log(self._level, 'Exit (no exception)')
        else:
            self._logger.log(self._level, 'Exit (with exception %s)', ex_type)
            return True

print('This is before the with statement')

with LogBlock(logging.getLogger('mylogger'), logging.ERROR):
    print('Now inside the block')
    print('still inside block')

with LogBlock(logging.getLogger('mylogger'), logging.ERROR):
    print('Now inside the 2nd block')
    print('still inside 2nd block')
    raise ValueError
```

Lab: Context Managers

- Write a context manager that prints out balanced HTML nodes. Use the test code below.

Test code:

```
with Node('html'):
    with Node('body'):
        with Node('h1'):
             print('Page Title')
```

You should see the following result:

```
<html>
<body>
<h1>
Page Title
</h1>
</body>
</html>
```

contextlib

- This module provides utilities for common tasks involving the with statement, e.g.,
    - printing to stderr
    - closing something upon completion of block

In [ ]:

```
import sys

from contextlib import redirect_stdout

print("before with statement")

with redirect_stdout(sys.stderr):
    print("NOTE! the output of help goes to stderr")
    help(pow)

print("after with statement")
```

In [ ]:

```
import contextlib

class Something():
    def __init__(self):
        print('initialize!')

    def close(self):
        print('closing!')

with contextlib.closing(Something()) as foo:
    print('foo is', foo)
```

In [ ]:

```

```