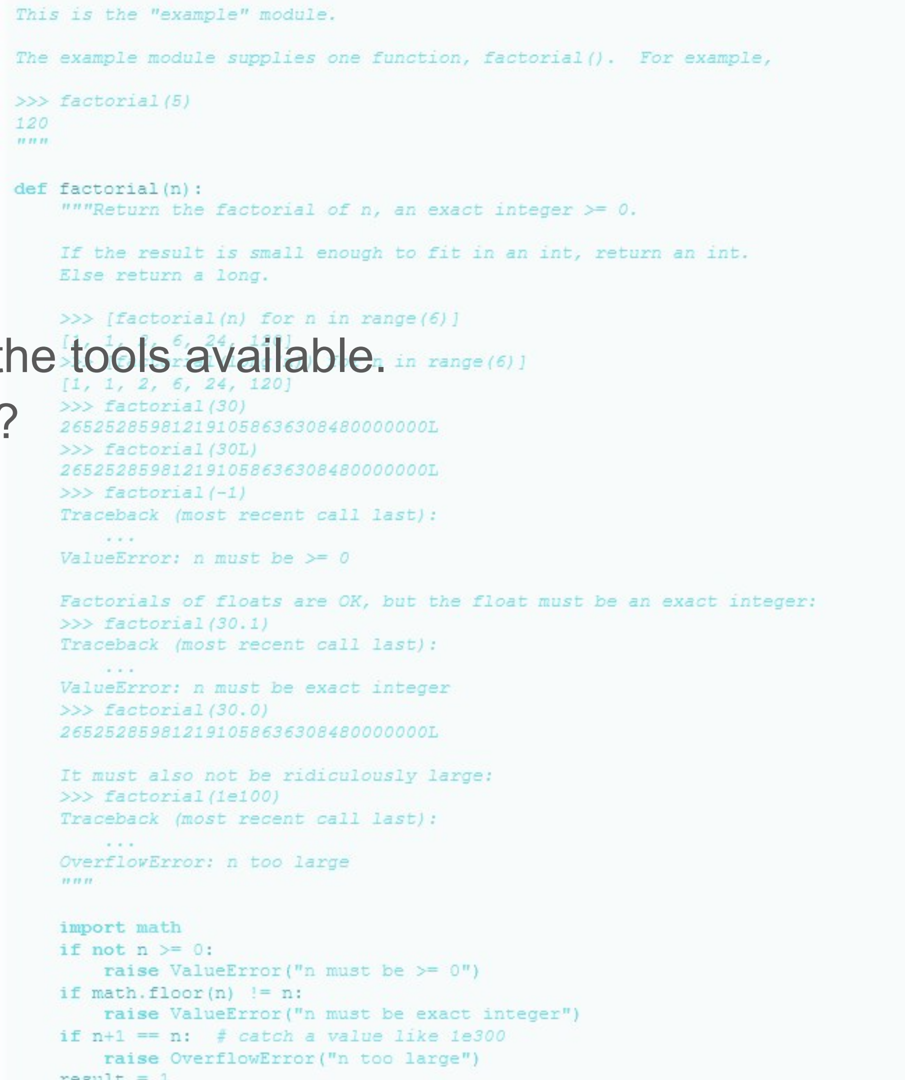
```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)

if __name__ == '__main__':
    unittest.main()
```

How to fit more testing into your day

A basic intro to testing with Python

Summary

- Motivation.
- Forms of testing and a brief intro. to the tools available.
- I don't test anything, where do I start?
- Where to go from here.



```
This is the "example" module.

The example module supplies one function, factorial().  For example,

>>> factorial(5)
120
"""

def factorial(n):
    """Return the factorial of n, an exact integer >= 0.

    If the result is small enough to fit in an int, return an int.
    Else return a long.

    >>> [factorial(n) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    >>> [factorial(long(n)) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    >>> factorial(30)
    265252859812191058636308480000000L
    >>> factorial(30L)
    265252859812191058636308480000000L
    >>> factorial(-1)
    Traceback (most recent call last):
        ...
    ValueError: n must be >= 0

    Factorials of floats are OK, but the float must be an exact integer:
    >>> factorial(30.1)
    Traceback (most recent call last):
        ...
    ValueError: n must be exact integer
    >>> factorial(30.0)
    265252859812191058636308480000000L

    It must also not be ridiculously large:
    >>> factorial(1e100)
    Traceback (most recent call last):
        ...
    OverflowError: n too large
    """

    import math
    if not n >= 0:
        raise ValueError("n must be >= 0")
    if math.floor(n) != n:
        raise ValueError("n must be exact integer")
    if n+1 == n:  # catch a value like 1e300
        raise OverflowError("n too large")
```

Motivation

- Going from a state of no testing to some basic testing with understanding of where we can go from here.

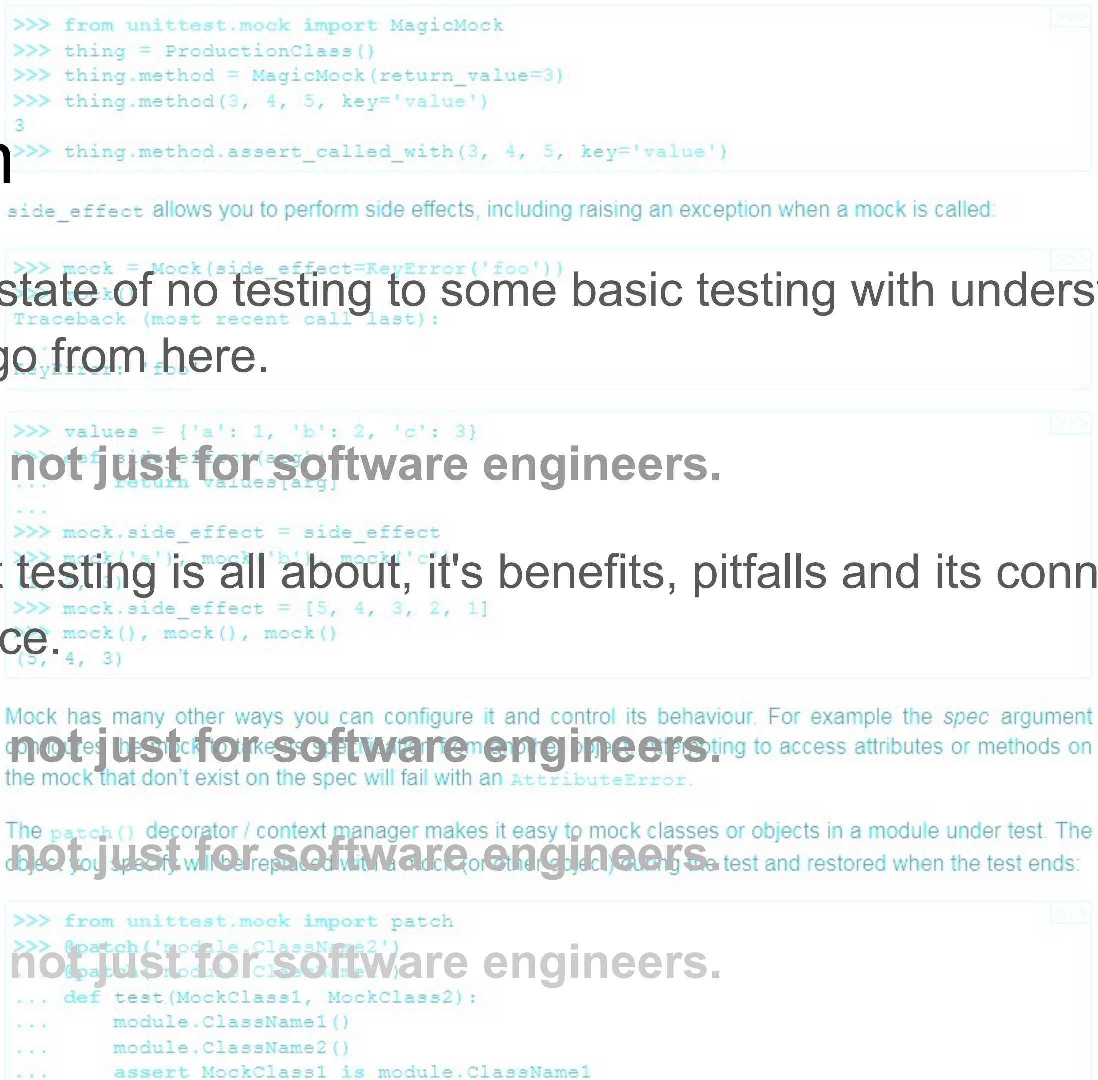
Testing is not just for software engineers.

- Knowing what testing is all about, it's benefits, pitfalls and its connection to quality assurance.

Testing is not just for software engineers.

Testing is not just for software engineers.

Testing is not just for software engineers.



```
>>> from unittest.mock import MagicMock
>>> thing = ProductionClass()
>>> thing.method = MagicMock(return_value=3)
>>> thing.method(3, 4, 5, key='value')
3
>>> thing.method.assert_called_with(3, 4, 5, key='value')
```

`side_effect` allows you to perform side effects, including raising an exception when a mock is called:

```
>>> mock = Mock(side_effect=KeyError('foo'))
>>> mock()
Traceback (most recent call last):
 ...
KeyError: 'foo'
```

```
>>> values = {'a': 1, 'b': 2, 'c': 3}
>>> def side_effect(arg):
...     return values[arg]
...
>>> mock.side_effect = side_effect
>>> mock('a'), mock('b'), mock('c')
(1, 2, 3)
>>> mock.side_effect = [5, 4, 3, 2, 1]
>>> mock(), mock(), mock()
(5, 4, 3)
```

Mock has many other ways you can configure it and control its behaviour. For example the *spec* argument configures the mock to take its specification from another object. Attempting to access attributes or methods on the mock that don't exist on the spec will fail with an `AttributeError`.

The `patch()` decorator / context manager makes it easy to mock classes or objects in a module under test. The object you specify will be replaced with a mock (or other object) during the test and restored when the test ends:

```
>>> from unittest.mock import patch
>>> @patch('module.ClassName2')
... @patch('module.ClassName1')
... def test(MockClass1, MockClass2):
...     module.ClassName1()
...     module.ClassName2()
...     assert MockClass1 is module.ClassName1
```

Forms of testing

- Doctesting.
- Unit testing.
- Integration testing.
- Acceptance testing.
- Code style checks.
- Continuous integration.
- Regression testing.
- Functional/non-functional testing.

```
diff = np.diff(array)
diff = np.hstack([np.where(diff != 1)[0], diff.size])
ref = 0
slices = []
for dd in range(len(diff)):
    slices.append(slice(array[ref], array[diff[dd]] + 1))
    ref = diff[dd] + 1
return slices
```

Unit testing

Integration testing

Continuous Integration

```
def group_indices(array):
    """
    Group an array representing indices into an iterable of slice objects.

    Parameters
    ----------
    array : :class:`numpy.ndarray`
        Numpy array representing indices.
```

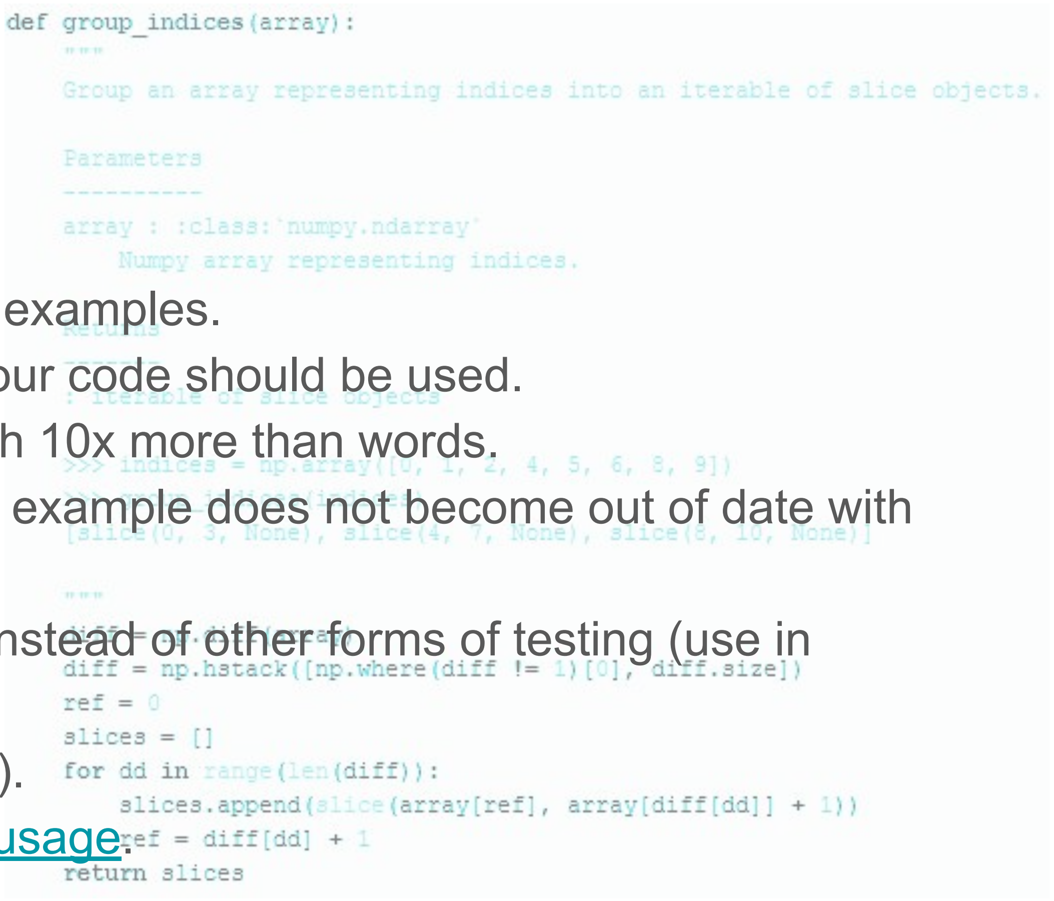
Acceptance testin

Coding standards

Doctests

Python doctest module

- Used to test interactive python examples.
- A way of demonstrating how your code should be used.
- Sometimes an example is worth 10x more than words.
- Ensures that your documented example does not become out of date with code changes.
- Generally should not be used instead of other forms of testing (use in addition).
- Example here (example failure).
- Can be integrated with sphinx usage.

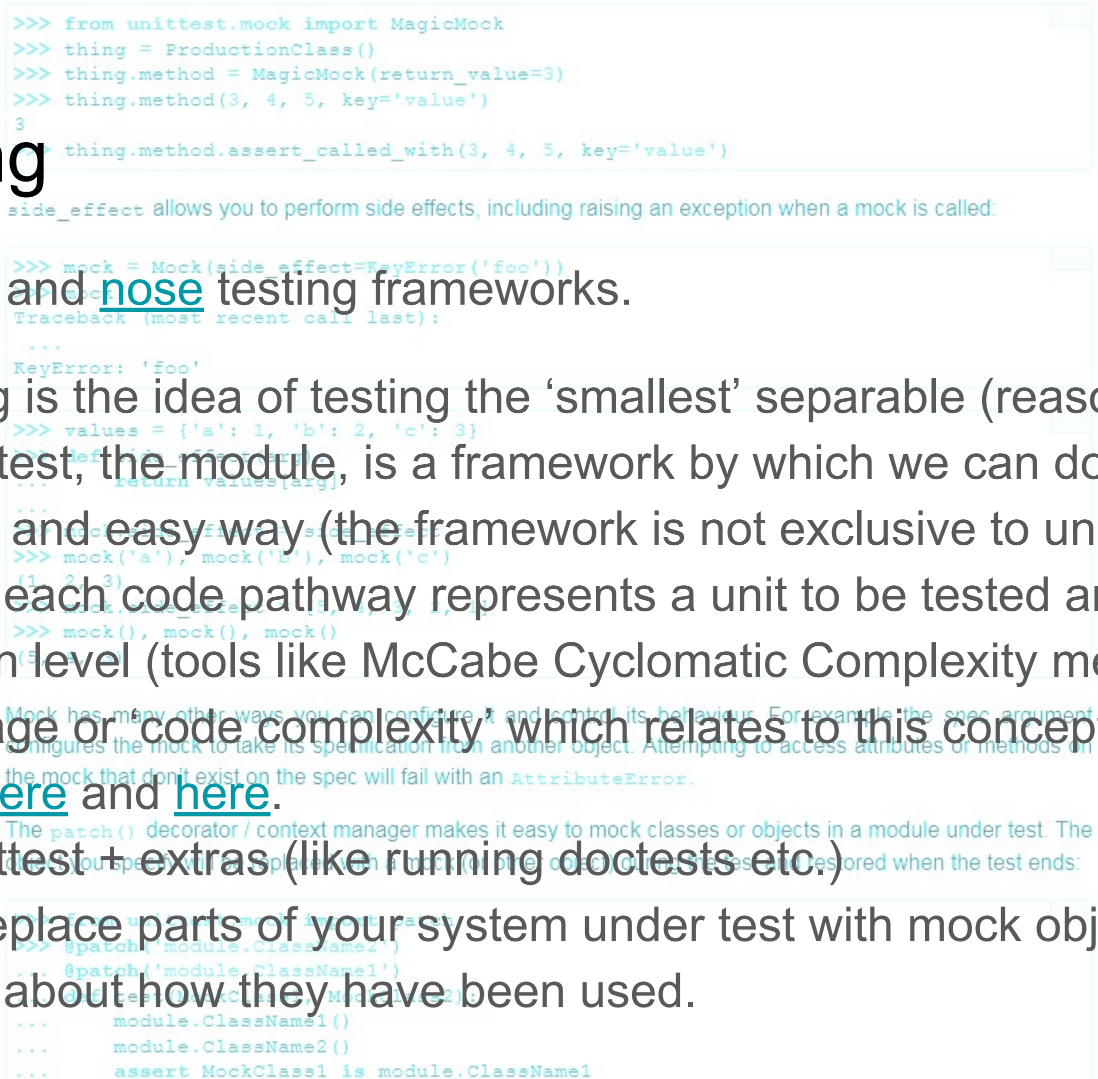


```
def group_indices(array):
    """
    Group an array representing indices into an iterable of slice objects.

    Parameters
    ----------
    array : :class:`numpy.ndarray`
        Numpy array representing indices.

    Returns
    -------
    : iterable of slice objects

    >>> indices = np.array([0, 1, 2, 4, 5, 6, 8, 9])
    >>> group_indices(indices)
    [slice(0, 3, None), slice(4, 7, None), slice(8, 10, None)]

    """
    diff = np.diff(array)
    diff = np.hstack([np.where(diff != 1)[0], diff.size])
    ref = 0
    slices = []
    for dd in range(len(diff)):
        slices.append(slice(array[ref], array[diff[dd]] + 1))
        ref = diff[dd] + 1
    return slices
```

Unit testing

Python unittest and nose testing frameworks.

- Unit testing is the idea of testing the ‘smallest’ separable (reasonable) source ‘unit’. Unittest, the module, is a framework by which we can do this testing in a standard and easy way (the framework is not exclusive to unit testing).
- Generally, each code pathway represents a unit to be tested and is done at the function level (tools like McCabe Cyclomatic Complexity metric calculate test coverage or ‘code complexity’ which relates to this concept).
- Example here and here.
- nose = unittest + extras (like running doctests etc.)
- mock, to replace parts of your system under test with mock objects and make assertions about how they have been used.



```
>>> from unittest.mock import MagicMock
>>> thing = ProductionClass()
>>> thing.method = MagicMock(return_value=3)
>>> thing.method(3, 4, 5, key='value')
3
>>> thing.method.assert_called_with(3, 4, 5, key='value')
```

side_effect allows you to perform side effects, including raising an exception when a mock is called:

```
>>> mock = Mock(side_effect=KeyError('foo'))
>>> mock()
Traceback (most recent call last):
 ...
KeyError: 'foo'
```

```
>>> values = {'a': 1, 'b': 2, 'c': 3}
>>> def side_effect(arg):
...     return values[arg]
...
>>> mock.side_effect = side_effect
>>> mock('a'), mock('b'), mock('c')
(1, 2, 3)
>>> mock.side_effect = [5, 4, 3, 2, 1]
>>> mock(), mock(), mock()
(5, 4, 3)
```

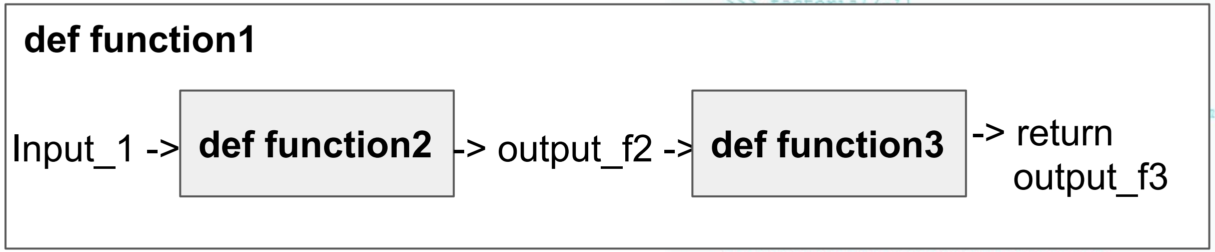
Mock has many other ways you can configure it and control its behaviour. For example the spec argument configures the mock to take its specification from another object. Attempting to access attributes or methods on the mock that don't exist on the spec will fail with an AttributeError.

The patch() decorator / context manager makes it easy to mock classes or objects in a module under test. The object you specify will be replaced with a mock (or other object) during the test and restored when the test ends:

```
>>> from unittest.mock import patch
>>> @patch('module.ClassName2')
... @patch('module.ClassName1')
... def test(MockClass1, MockClass2):
...     module.ClassName1()
...     module.ClassName2()
...     assert MockClass1 is module.ClassName1
```

Integration testing

- Essentially testing that things successfully interact with one another (as we expect). Example: function1 which is a top level API which underneath calls function1 and function2, does so successfully:



- Should still be small as reasonable and run very quickly.
- Example here.

```
This is the "example" module.

The example module supplies one function, factorial().  For example,

>>> factorial(5)
120
"""

def factorial(n):
    """Return the factorial of n, an exact integer >= 0.

    If the result is small enough to fit in an int, return an int.
    Else return a long.

    >>> [factorial(n) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    >>> [factorial(long(n)) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    >>> factorial(30)
    265252859812191058636308480000000L
    >>> factorial(30L)
    265252859812191058636308480000000L
    >>> factorial(-1)
[illegible]
                                                 must be an exact integer:
[illegible]
    It must also not be ridiculously large:
    >>> factorial(1e100)
    Traceback (most recent call last):
        ...
    OverflowError: n too large
    """

    import math
    if not n >= 0:
        raise ValueError("n must be >= 0")
    if math.floor(n) != n:
        raise ValueError("n must be exact integer")
    if n+1 == n:  # catch a value like 1e300
        raise OverflowError("n too large")
    result = 1
```

Code style checks

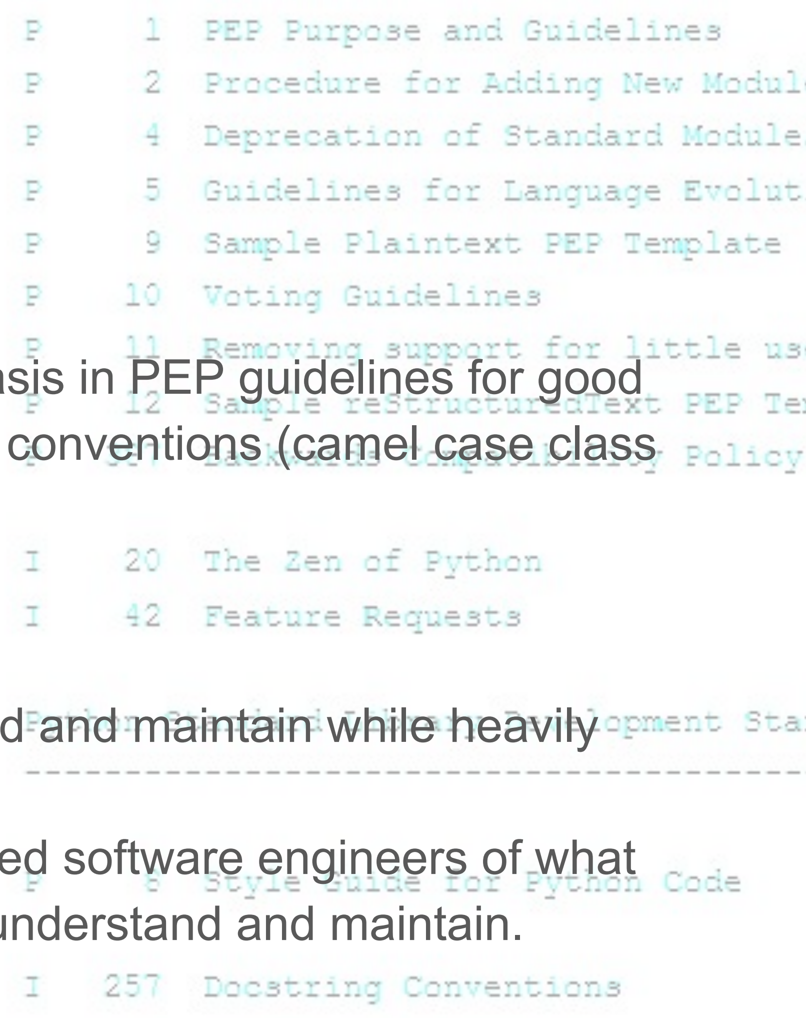
pycodestyle (pep8), pyflakes, flake8, pylint, …

- Typically, for Python these normally have basis in PEP guidelines for good style. Such as those for line length, naming conventions (camel case class names, ...).

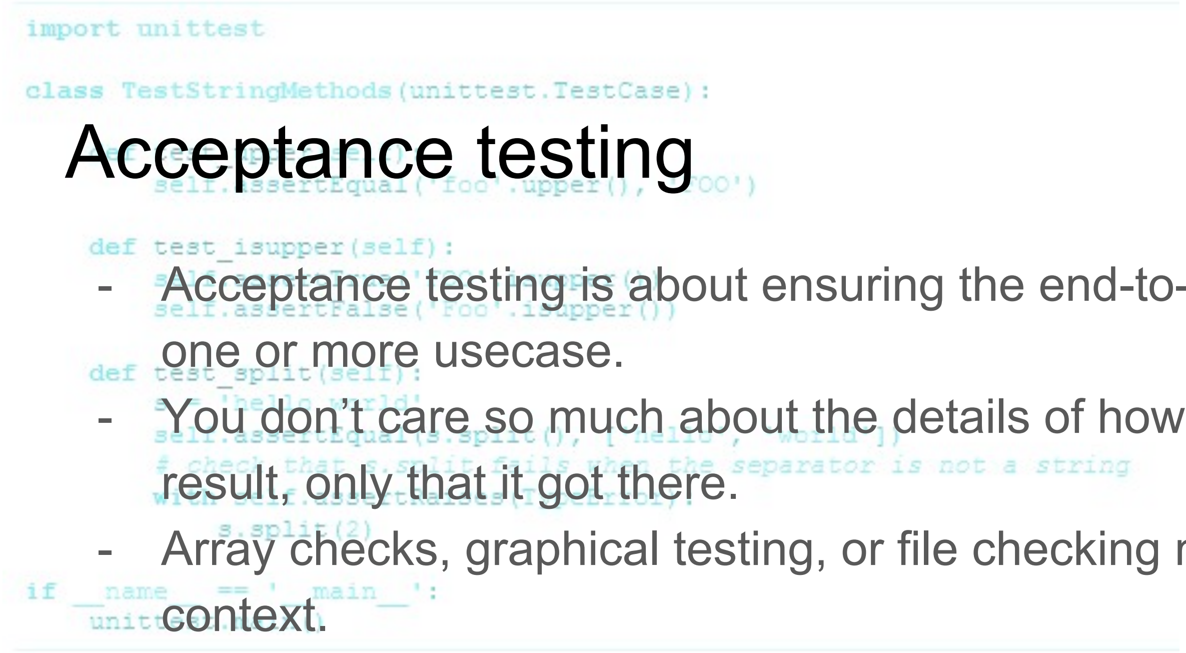
Why are these important?

- Familiar code is easy to read and understand and maintain while heavily personalised code preferences are not.
- These standards have basis from experienced software engineers of what has proven to be easier and neater to both understand and maintain.

The zen on python (pep 20) is worth a read!



```
P      1  PEP Purpose and Guidelines
P      2  Procedure for Adding New Modul
P      4  Deprecation of Standard Module
P      5  Guidelines for Language Evolut
P      9  Sample Plaintext PEP Template
P     10  Voting Guidelines
P     11  Removing support for little us
P     12  Sample reStructuredText PEP Te
P    387  Backwards Compatibility Policy

I     20  The Zen of Python
I     42  Feature Requests

Python Standard Library Development Sta
---------------------------------------

P      8  Style Guide for Python Code

I    257  Docstring Conventions
```



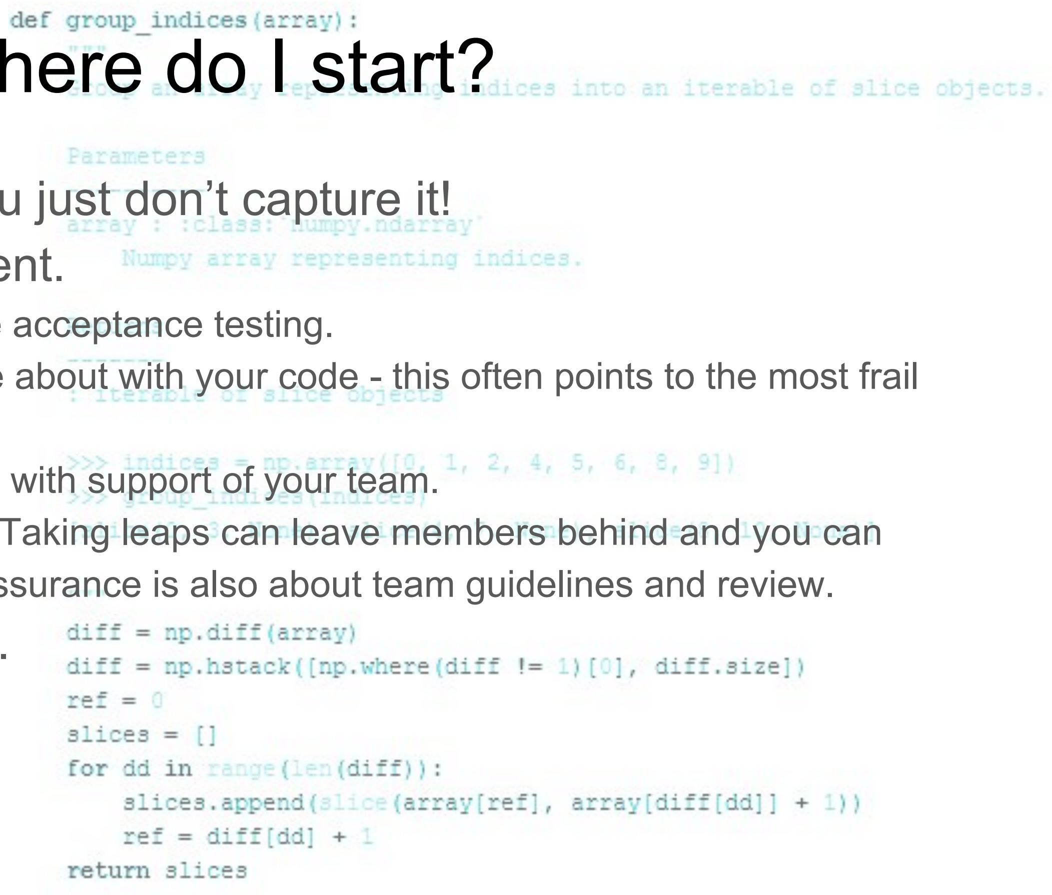
```python
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)

if __name__ == '__main__':
    unittest.main()
```

Acceptance testing

- Acceptance testing is about ensuring the end-to-end running of your code for one or more usecase.
- You don’t care so much about the details of how or with what it got to the end result, only that it got there.
- Array checks, graphical testing, or file checking may be relevant in this context.
- These may not run fast, though perhaps there are different levels of acceptance testing (full slow real ones and perhaps more frequently run fictitious end-to-end ones?).
- No guarantee that given an even slightly different usecase that it won’t break your program or worse (return incorrect results) - that’s what unit tests and integration tests are for.

I don’t test anything, where do I start?

1. You test your code already, you just don’t capture it!
2. Take small steps of improvement.
    a. Consider top-level testing first like acceptance testing.
    b. What are you most uncomfortable about with your code - this often points to the most frail pieces.
    c. Traverse along the path of testing with support of your team.
        i. Take small steps together. Taking leaps can leave members behind and you can become isolated. Quality assurance is also about team guidelines and review.
3. Have a play with my examples.



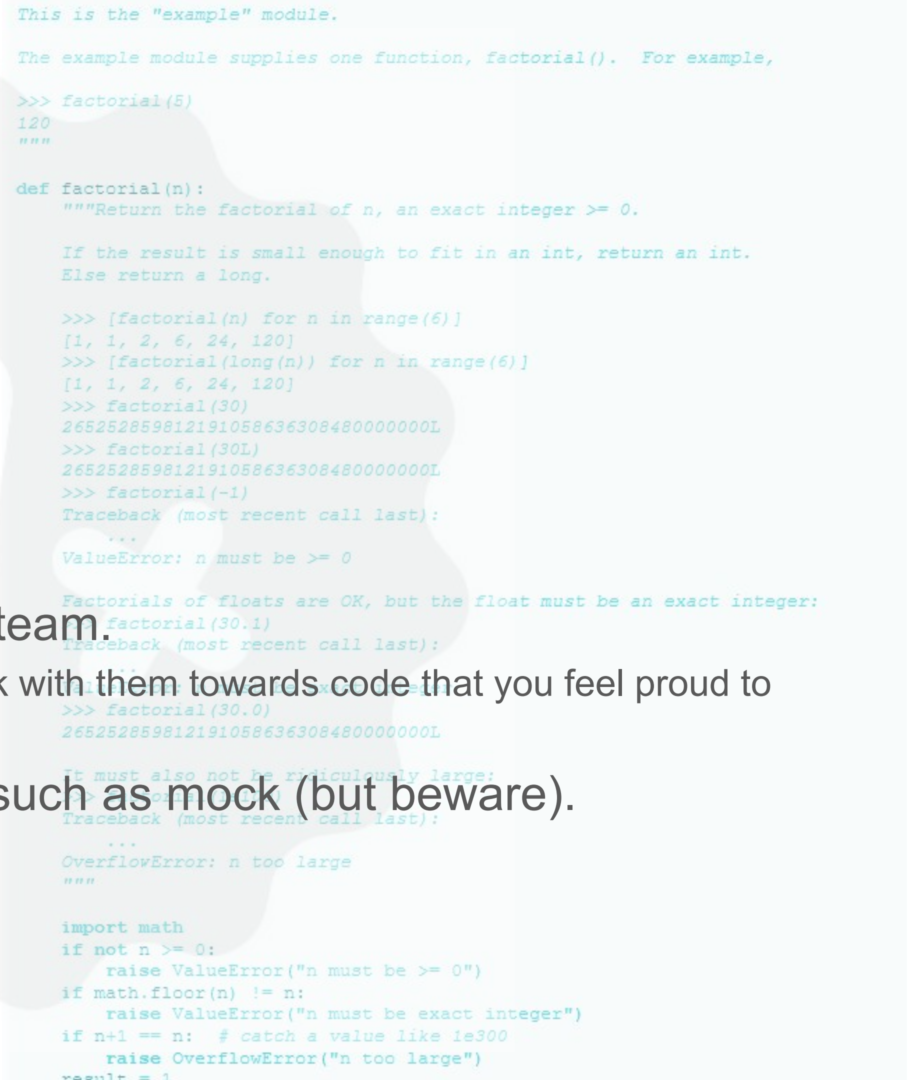
```python
def group_indices(array):
    """
    Group an array representing indices into an iterable of slice objects.

    Parameters
    ----------
    array : :class:`numpy.ndarray`
        Numpy array representing indices.

    Returns
    -------
    : iterable of slice objects

    >>> indices = np.array([0, 1, 2, 4, 5, 6, 8, 9])
    >>> group_indices(indices)
    [slice(0, 3, None), slice(4, 7, None), slice(8, 10, None)]

    """
    diff = np.diff(array)
    diff = np.hstack([np.where(diff != 1)[0], diff.size])
    ref = 0
    slices = []
    for dd in range(len(diff)):
        slices.append(slice(array[ref], array[diff[dd]] + 1))
        ref = diff[dd] + 1
    return slices
```

Where to go from here

- Increasing test granularity.

Acceptance -> integration -> unittest

- Improve project testing framework.
    - Things evolve.
- Learn from others and develop as a team.
    - Discuss concerns of teammates and work with them towards code that you feel proud to share.
- Start looking at more advance tools such as mock (but beware).



```
This is the "example" module.

The example module supplies one function, factorial().  For example,

>>> factorial(5)
120
"""

def factorial(n):
    """Return the factorial of n, an exact integer >= 0.

    If the result is small enough to fit in an int, return an int.
    Else return a long.

    >>> [factorial(n) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    >>> [factorial(long(n)) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    >>> factorial(30)
    265252859812191058636308480000000L
    >>> factorial(30L)
    265252859812191058636308480000000L
    >>> factorial(-1)
    Traceback (most recent call last):
        ...
    ValueError: n must be >= 0

    Factorials of floats are OK, but the float must be an exact integer:
    >>> factorial(30.1)
    Traceback (most recent call last):
        ...
    ValueError: n must be exact integer
    >>> factorial(30.0)
    265252859812191058636308480000000L

    It must also not be ridiculously large:
    >>> factorial(1e100)
    Traceback (most recent call last):
        ...
    OverflowError: n too large
    """

    import math
    if not n >= 0:
        raise ValueError("n must be >= 0")
    if math.floor(n) != n:
        raise ValueError("n must be exact integer")
    if n+1 == n:  # catch a value like 1e300
        raise OverflowError("n too large")
    result = 1
```