Unit Testing in Python

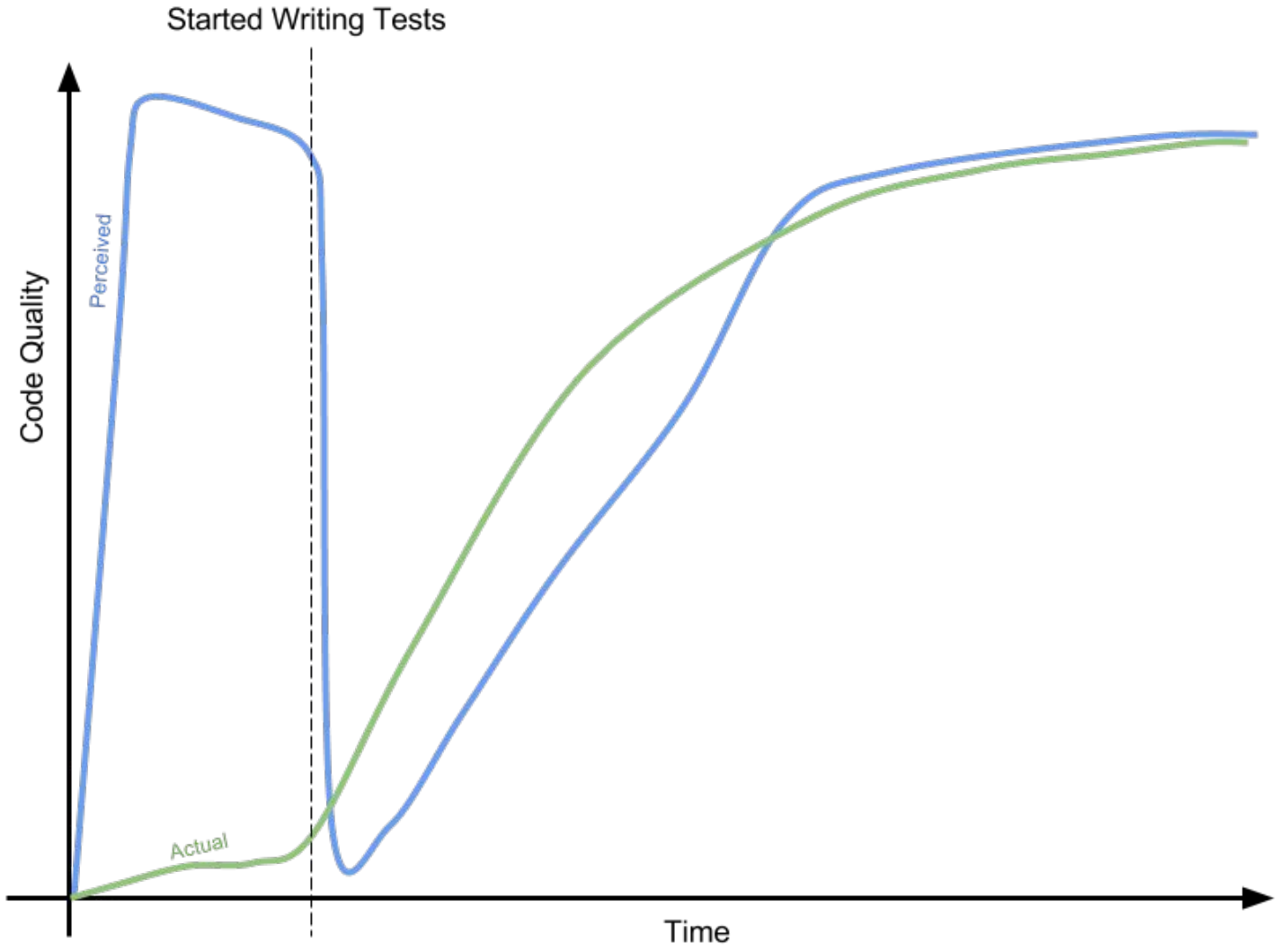
Mock Trials and Tribulations

Michelle Wynn
wynn@zymergen.com
March 15, 2017
hosted by Zymergen

Unit Testing in Python: Mock Trials and Tribulations

- **Introduction to Unit Tests in Python**
  - What is a Unit Test
  - Unit Tests Best Practices in Python
- Mock: What is it good for?
  - Examples Using Mock
  - Common Pitfalls to Avoid
- Conclusion

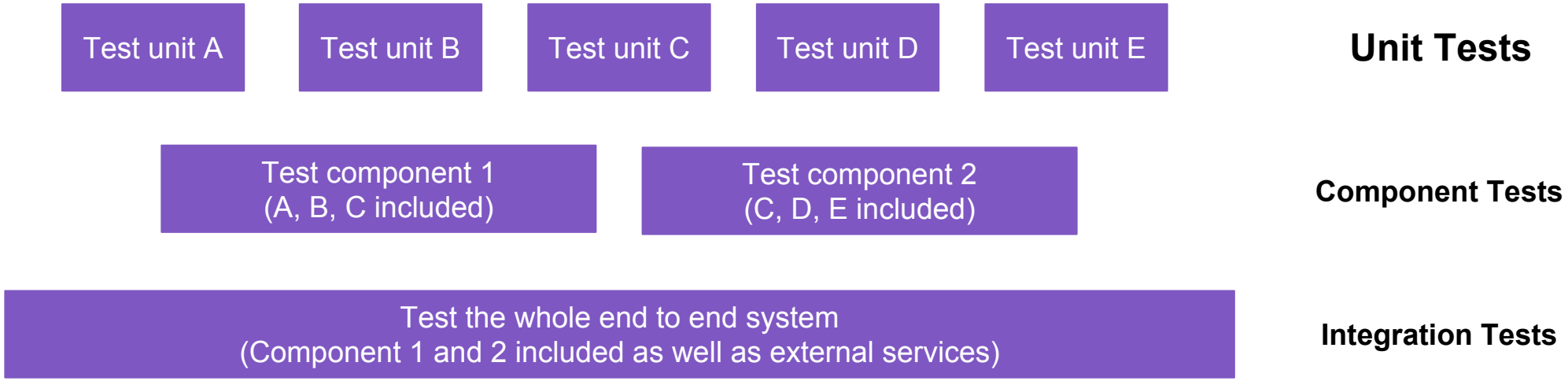
Why Are Tests Important?



Unit Testing in Python: Mock Trials and Tribulations

- Introduction to Unit Tests in Python
  - **What is a Unit Test**
  - Unit Tests Best Practices in Python
- Mock: What is it good for?
  - Examples Using Mock
  - Common Pitfalls to Avoid
- Conclusion

Types of Software Tests



What is Unit Testing?

- Tests short “units” of code
  - Could be a function, a class, a module, etc.
    - Preferably the smallest testable part of the code!
- Atomic: Each unit test is independent from every other unit test

Testing only small discrete bits of code is what distinguishes unit tests from other types of tests

Why Is Unit Testing Important?

- Allows you to more easily debug your code
- When you edit your code, unit tests give you confidence that you haven’t inadvertently broken something existing.
- Corollary: When you add features to your code, you should add new test(s) to cover the new functionality.

Unit Testing in Python: Mock Trials and Tribulations

- Introduction to Unit Tests in Python
  - What is a Unit Test
  - **Unit Tests Best Practices in Python**
- Mock: What is it good for?
  - Examples Using Mock
  - Common Pitfalls to Avoid
- Conclusion

Structuring Your Python Project For Unit Tests

```
project_root/
├── README.md           # Bonus Points for adding a README file.
├── chemical_module/
│   ├── __init__.py    # Empty file, needed to use your code in your tests.
│   └── chemical_property_lookup.py
└── test/
    ├── __init__.py    # Also add empty __init__.py to your test package.
    ├── test_chemical_property_lookup.py  # Begin file name with “test”
    └── test_chemical_property_lookup_with_mocks.py
```

Writing Your Tests: The Test Module

```python
import unittest  # You must import this package

# You must also import the code you want to test
# In this example, we are testing chemical_property_lookup() in the
# chemical_module.
from chemical_module.chemical_property_lookup import get_molecular_formula_by_cid


class TestChemicalPropertyLookup(unittest.TestCase):
    """
        This is a test class. Test classes group a series of related
        tests together.

        Some important things to note:
          1. The test class MUST begin with the word "Test". A good
             naming convention is Test<NameOfTheThingImTryingToTest>.
          2. The test class MUST inherit from unittest.TestCase.
    """

    def setUp(self):
        """
        An optional setUp method is executed BEFORE each test is run.
        """
        # set expected values for aspirin
        self.cid = 2244
        self.expected_mol_form = 'C9H8O4'
```

Writing Your Tests: The Test Module

```
    def tearDown(self):
        """
        An optional tearDown method is executed AFTER each test is run.
        """
        pass

    def test_get_molecular_formula(self):
        """ Verify the correct molecular formula is returned. """

        mol_form = get_molecular_formula_by_cid(self.cid)
        self.assertEqual(self.expected_mol_form, mol_form)

if __name__ == '__main__':
    # This will let you invoke your test from command line if you'd like.
    # We will use nose, so this is NOT strictly necessary.
    unittest.main()
```

What Makes a Good Unit Test?

A good unit test:

- Does not depend on other unit tests (*tests are independent!*)
- Asserts the results of your code
- Tests a single unit of work
- Covers all the paths of the code under test, including error handling and edge cases
- Runs fast
- Does not rely on the environment or external data
  (you can run it locally without a network) ⟸ *this is where mocking comes in!*

The Chemical Property Lookup Module

```python
# Helper module for looking up chemical property information in PubChem.


def get_molecular_formula_by_cid(cid):
    """
    Find the molecular formula of a compound.

    Args:
        cid: A PubChem CID (e.g., 5793 for glucose), which must be numeric.

    Returns:
        The conventional representation of the compound's molecular formula
        as a string.
    """

    properties = get_properties_by_cid(cid, [MOLECULAR_FORMULA])
    return properties[MOLECULAR_FORMULA]
```

```python
if __name__ == "__main__":
    # get the molecular formula of aspirin
    print '\nThe molecular formula of aspirin is', get_molecular_formula_by_cid(2244)
```

A unit test of `get_molecular_formula()`

```python
class TestChemicalPropertyLookup(unittest.TestCase):
    """ These unit tests do not test with mock. """

    def setUp(self):
        """
        The setUp method is executed BEFORE each test method in a class is run.
        Similiarly a tearDown method can be implmented, which will run AFTER
        each test method in a class is run.
        """

        # set expected values for aspirin
        self.aspirin_cid = 2244
        self.expected_mol_form = 'C9H8O4'
        self.expected_InChIKey = 'BSYNRYMUTXBXSQ-UHFFFAOYSA-N'
        self.expected_mol_wt = 180.159
        self.expected_exact_mass = 180.042
        self.expected_charge = 0
        self.expected_IUPACName = '2-acetyloxybenzoic acid'

    def test_get_molecular_formula(self):
        """Verify the correct molecular formula is returned for aspirin."""

        mol_form = get_molecular_formula_by_cid(self.aspirin_cid)  # external API call
        self.assertEqual(self.expected_mol_form, mol_form)
```

What Can You Assert in a Python Unit Test?

`assertEqual(a, b)`	**Checks:** `a == b`
`assertNotEqual(a, b)`	**Checks:** `a != b`
`assertTrue(x)`	**Checks:** `bool(x) is True`
`assertFalse(x)`	**Checks:** `bool(x) is False`
`assertIs(a, b)`	**Checks:** `a is b`
`assertIsNot(a, b)`	**Checks:** `a is not b`
`assertIsNone(x)`	**Checks:** `x is None`
`assertIsNotNone(x)`	**Checks:** `x is not None`
`assertIn(a, b)`	**Checks:** `a in b`
`assertNotIn(a, b)`	**Checks:** `a not in b`
`assertIsInstance(a, b)`	**Checks:** `isinstance(a, b)`
`assertIsNotInstance(a, b)`	**Checks:** `not isinstance(a, b)`

Full list: https://docs.python.org/2/library/unittest.html#unittest.TestCase

Unit Testing in Python: Mock Trials and Tribulations

- Introduction to Unit Tests in Python
  - What is a Unit Test
  - Unit Tests Best Practices
- **Mock: What is it good for?**
  - Examples Using Mock
  - Common Pitfalls to Avoid
- Conclusion

Mock: What Is It Good For?

Mocking allows us to use “fake” method return values and/or object creation when those methods and objects are NOT the things we are actually testing. A mock can return a predefined value immediately, without doing any computation.

Example: Assume a method we want to test makes a call to a service that makes a database lookup, which ultimately returns a value to to our method.

- In our method’s unit test, we do not need to test the service or the database!
- The service should have its own unit test.
- We only need to verify that our function passed the correct arguments to the service and then correctly processed the return value it received from the service.

Mock: What Is It Good For?

- Remember, a good unit test:
  - ***should be independent of other modules in the code***
  - ***should not rely on the environment or external data***

- Mocking will pretend to do these things for us, without actually doing them.
- Mocking allows us to test that our code does what it claims to do without requiring external services to be running.
  - Of course, you should also test with the actual service running if you can! That's called an integration test NOT a unit test.

The Python Mock Library

In Python, mocking is easily accomplished using the **`unittest.mock`** library.

- The mock library contains a number of useful classes and functions.
- Mocking in Python is often accomplished using **`patch`** or **`MagicMock`**.
- We have focused on the `patch` decorator in our example code because it is very powerful and also easy to use!

Unit Testing in Python: Mock Trials and Tribulations

- Introduction to Unit Tests in Python
  - What is a Unit Test
  - Unit Tests Best Practices
- Mock: What is it good for?
  - **Examples Using Mock**
  - Common Pitfalls to Avoid
- Conclusion

We can test `get_molecular_formula()` by mocking the external API call

Our code to test

```python
from requests import get
def get_molecular_formula_by_cid(cid):
    """
    Find the molecular formula of a compound.

    Args:
        cid: A PubChem CID (e.g., 5793 for glucose), which must be numeric.

    Returns:
        The conventional representation of the compound's molecular formula
        as a string.
    """

    properties = get_properties_by_cid(cid, [MOLECULAR_FORMULA])
    return properties[MOLECULAR_FORMULA]
```

```python
def get_properties_by_cid(cid, properties_list):
    """
    Lookup one or more chemical properites of a compound.

    Args:
        cid: A PubChem CID (e.g., 5793 for glucose), which must be numeric.
        properties_list: A list of property name strings to lookup.

    Returns:
        A dictionary of chemical properties where keys are property names
        and values are the associated property values for the passed CID.
    """

    url = build_url(cid, properties_list)
    return get_response_properties(url)
```

```python
def get_response_properties(url):
    """
    Make the request to PubChem and return the results as a dictionary.

    Args:
        url: A formatted PubChem query string.

    Returns:
        A dictionary of chemical properties where keys are property names
        and values are the associated property values for the passed CID.

    """
    response = get(url)
    results = response.json()
    return results["PropertyTable"]["Properties"][0]
```

We can test `get_molecular_formula()` by mocking the external API call

Our code to test

```python
from requests import get
def get_molecular_formula_by_cid(cid):
    """
    Find the molecular formula of a compound.

    Args:
        cid: A PubChem CID (e.g., 5793 for glucose), which must be numeric.

    Returns:
        The conventional representation of the compound's molecular formula
        as a string.
    """

    properties = get_properties_by_cid(cid, [MOLECULAR_FORMULA])
    return properties[MOLECULAR_FORMULA]
```

```python
def get_properties_by_cid(cid, properties_list):
    """
    Lookup one or more chemical properites of a compound.

    Args:
        cid: A PubChem CID (e.g., 5793 for glucose), which must be numeric.
        properties_list: A list of property name strings to lookup.

    Returns:
        A dictionary of chemical properties where keys are property names
        and values are the associated property values for the passed CID.
    """

    url = build_url(cid, properties_list)
    return get_response_properties(url)
```

```python
def get_response_properties(url):
    """
    Make the request to PubChem and return the results as a dictionary.

    Args:
        url: A formatted PubChem query string.

    Returns:
        A dictionary of chemical properties where keys are property names
        and values are the associated property values for the passed CID.

    """
    response = get(url)
    results = response.json()
    return results["PropertyTable"]["Properties"][0]
```

Our test with mocks

```python
@patch('chemical_module.chemical_property_lookup.get')
def test_get_molecular_formula_second_mock(self, mock_get):
    """
    Verify the correct molecular formula is returned for aspirin.

    The code we're trying to mock looks like:

        response = get(url)
        results = response.json()

    Thus, we need our mock_get function to return an object with a
    function that returns the json that our test expects to see.  We
    set the return_value of our mock object to be a stub of a class with
    the expected .json() function.
    """

    # Our mock_results class is a mock with a method, json, which returns
    # our expected json
    mock_results = Mock()
    mock_results.json.return_value = \
        {'PropertyTable': {'Properties': [{'MolecularFormula': 'C9H8O4'}]}}

    # When get() is called, our mock results are returned and no
    # external API call is made
    mock_get.return_value = mock_results

    # now call the function we are testing
    mol_form = get_molecular_formula_by_cid(self.aspirin_cid)

    # We can both check that our mock was called and that our function
    # took the result of the mock and returned the correct information
    mock_results.json.assert_called()
    mock_get.assert_called_with(MOL_FORM_QUERY_STR)
    self.assertEqual(self.expected_mol_form, mol_form)
```

Another way to mock `requests.get()`

```python
from mock import patch, Mock
import requests_mock
import unittest
```

```python
# And here's yet another way we mock the same function.
@requests_mock.mock()  # another way to mock a request!
def test_get_molecular_formula_third_mock(self, mock_req):
    """
    Verify the correct molecular formula is returned for aspirin.

    The code we're trying to mock looks like:

        response = get(url)
        results = response.json()

    Thus, we need our mock_get function to return an object with a
    function that returns the json that our test expects to see.  We
    set the return_value of our mock object to be a stub of a class with
    the expected .json() function.
    """

    # The requests object will be mocked. When get() is called, the request will
    # contain the information we set here
    mock_req.get(MOL_FORM_QUERY_STR,
                 text='{"PropertyTable": {"Properties": [{"MolecularFormula": '
                      '"C9H8O4", "CID": 2244}]}}')

    mol_form = get_molecular_formula_by_cid(self.aspirin_cid)
    self.assertEqual(True, mock_req.called)
    self.assertEqual(1, mock_req.call_count)
    self.assertEqual(self.expected_mol_form, mol_form)
```

Here is another attempt to use mocks. What do you think of this test?

Our code to test

```python
from requests import get

def get_molecular_formula_by_cid(cid):
    """
    Find the molecular formula of a compound.

    Args:
        cid: A PubChem CID (e.g., 5793 for glucose), which must be numeric.

    Returns:
        The conventional representation of the compound's molecular formula
        as a string.
    """

    properties = get_properties_by_cid(cid, [MOLECULAR_FORMULA])
    return properties[MOLECULAR_FORMULA]
```

```python
def get_properties_by_cid(cid, properties_list):
    """
    Lookup one or more chemical properites of a compound.

    Args:
        cid: A PubChem CID (e.g., 5793 for glucose), which must be numeric.
        properties_list: A list of property name strings to lookup.

    Returns:
        A dictionary of chemical properties where keys are property names
        and values are the associated property values for the passed CID.
    """

    url = build_url(cid, properties_list)
    return get_response_properties(url)
```

Our test with a mock

```python
# Questions:
# What do you think about this mock?
# What is it testing?
# Might this be an "over-mock"? Why or why not?
# Can you suggest another way to test this function?
@patch('chemical_module.chemical_property_lookup.get_properties_by_cid')
def test_get_molecular_formula_first_mock(self, get_properties_mock):
    """
        Verify the correct molecular formula is returned for aspirin.

        The code we're mocking looks like this, which is executed from
        within get_molecular_formula_by_cid:

            properties = get_properties_by_cid(cid, [MOLECULAR_FORMULA])
    """

    # mock the return value of get_properties_mock
    get_properties_mock.return_value = {'MolecularFormula': 'C9H8O4'}

    # did we get expected formula?
    mol_form = get_molecular_formula_by_cid(self.aspirin_cid)
    self.assertEqual(self.expected_mol_form, mol_form)

    # was the mock called as expected?
    get_properties_mock.assert_called()
    get_properties_mock.assert_called_with(self.aspirin_cid, ['MolecularFormula'])
```

Unit Testing in Python: Mock Trials and Tribulations

- Introduction to Unit Tests in Python
  - What is a Unit Test
  - Unit Tests Best Practices
- Mock: What is it good for?
  - Examples Using Mock
  - **Common Pitfalls to Avoid**
- Conclusion

Mock Trials and Tribulations: Mock from the correct namespace

Our code to test

```python
from requests import get
```

```python
def get_molecular_formula_by_cid(cid):
    """
    Find the molecular formula of a compound.

    Args:
        cid: A PubChem CID (e.g., 5793 for glucose), which must be numeric.

    Returns:
        The conventional representation of the compound's molecular formula
        as a string.
    """

    properties = get_properties_by_cid(cid, [MOLECULAR_FORMULA])
    return properties[MOLECULAR_FORMULA]
```

```python
def get_properties_by_cid(cid, properties_list):
    """
    Lookup one or more chemical properites of a compound.

    Args:
        cid: A PubChem CID (e.g., 5793 for glucose), which must be numeric.
        properties_list: A list of property name strings to lookup.

    Returns:
        A dictionary of chemical properties where keys are property names
        and values are the associated property values for the passed CID.
    """

    url = build_url(cid, properties_list)
    return get_response_properties(url)
```

Our test with mocks

```python
# @patch('requests.get')                                    # Note: this will not work
@patch('chemical_module.chemical_property_lookup.get')  # this will!
def test_get_molecular_formula_second_mock(self, mock_get):
    """
    Verify the correct molecular formula is returned for aspirin.

    The code we're trying to mock looks like:

        response = get(url)
        results = response.json()

    Thus, we need our mock_get function to return an object with a
    function that returns the json that our test expects to see.  We
    set the return_value of our mock object to be a stub of a class with
    the expected .json() function.
    """

    # Our mock_results class is a mock with a method, json, which returns
    # our expected json
    mock_results = Mock()
    mock_results.json.return_value = \
        {'PropertyTable': {'Properties': [{'MolecularFormula': 'C9H8O4'}]}}

    # When get() is called, our mock results are returned and no
    # external API call is made
    mock_get.return_value = mock_results

    # now call the function we are testing
    mol_form = get_molecular_formula_by_cid(self.aspirin_cid)

    # We can both check that our mock was called and that our function
    # took the result of the mock and returned the correct information
    mock_results.json.assert_called()
    mock_get.assert_called_with(MOL_FORM_QUERY_STR)
    self.assertEqual(self.expected_mol_form, mol_form)
```

Mock Trials and Tribulations: What to Mock

- Calls to an external REST endpoint
  - You can mock the response to be whatever you want, even if it’s not what the actual REST api would return. (BUT: If you’re not careful, your test could pass even though the code would fail in the real world).
- Calls to external hardware (printers, automation robots):
  - You can mock responses like rare errors that would be hard to generate in a real world environment.
- Calls to code that doesn’t exist yet
  - Useful for collaborating with someone coding in parallel with you.
  - You can replace the mock with real code once it’s finished if needed.
- WARNING: If code you mock changes its behavior and/or return value, you have to change your mock!

Mock Trials and Tribulations: Over-mocking

- If you have too many mocks, it can become difficult to understand what is actually being tested.
- It is possible that you will not test anything in your code, but only mocks.
  - Is that useful??
- You can mock local functions (e.g., not an external service your code depends on).
  - This can be useful if you have a long running piece of code and you want your unit tests to run fast, but it can also be dangerous
- If you change the underlying code that is mocked (and not actually executed by your test), your test will still pass, giving you a false sense of security.

Unit Testing in Python: Mock Trials and Tribulations

- Introduction to Unit Tests in Python
  - What is a Unit Test
  - Unit Tests Best Practices
- Mock: What is it good for?
  - Examples Using Mock
  - Common Pitfalls to Avoid
- **Conclusion**

Conclusions

- Unit testing is a critical part of good software engineering
- Ideally a unit test will test an atomic/independent unit of code
- In Python, use the unit test framework, ideally with nosetests or similar package for test discovery to ensure all tests are run and pass
- Mocking is a powerful tool to use in unit tests.
  - But beware the overmock!
- Some useful Python Mock references:
  - https://blog.fugue.co/2016-02-11-python-mocking-101.html
  - http://alexmarandon.com/articles/python_mock_gotchas
  - http://www.voidspace.org.uk/python/mock/patch.html#where-to-patch

Thank you for your attention!

Optional pre-setup

To get source code:

```
git clone https://github.com/mwynn1/intro_to_python_tests_and_mocks.git
```

```
(or we have a thumb drive with source code on it)
```

To run the module (and query PubChem for chemical attributes of aspirin)

```
cd intro_to_python_tests_and_mocks
python chemical_module/chemical_property_lookup.py
```

To run the unittests (with virtualenv)

```
virtualenv venv
source venv/bin/activate
pip install -r requirements.txt
nosetests
```

9 tests should run and pass