Prerequisites

1. A Github account with an SSH key

   https://help.github.com/articles/generating-ssh-keys/

2. If you want to develop on your machine:
   3. pip and virtualenv
   4. git
   5. (Optional) Fork and clone our repo!

      https://github.com/keeppythonweird/catinabox

Intro to Unit Testing in Python with PyTest

Michael Tom-Wing @mtomwing
Christie Wilson @bobcatwilson

Obligatory Plug

DEMONWARE

- Software Engineers @ Demonware
- Video Game Industry
- Owned by Activision
- Online services for games

CALL OF DUTY

DESTINY

SKYLANDERS SUPERCHARGERS

NOW ON TO TESTING!

Welcome to our tutorial!!!!

Let's find out a bit about why we're all here

What's your role?

http://www.strawpoll.me/10337223

Have you written a test before?

http://www.strawpoll.me/10337208

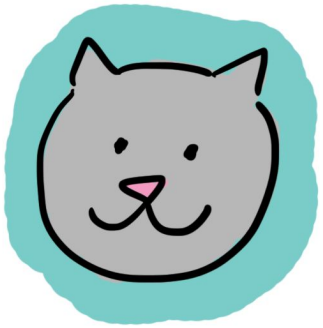
How much Python experience do you have?

http://www.strawpoll.me/10337237

Schedule

- What is a test?
- **Initial environment setup**
- What are unit tests?
- **Write some tests.**
- What is test automation?
- **Run your tests through our automation.**
- What are some advanced testing techniques?
- **Write some tests using those techniques.**
- Q & A

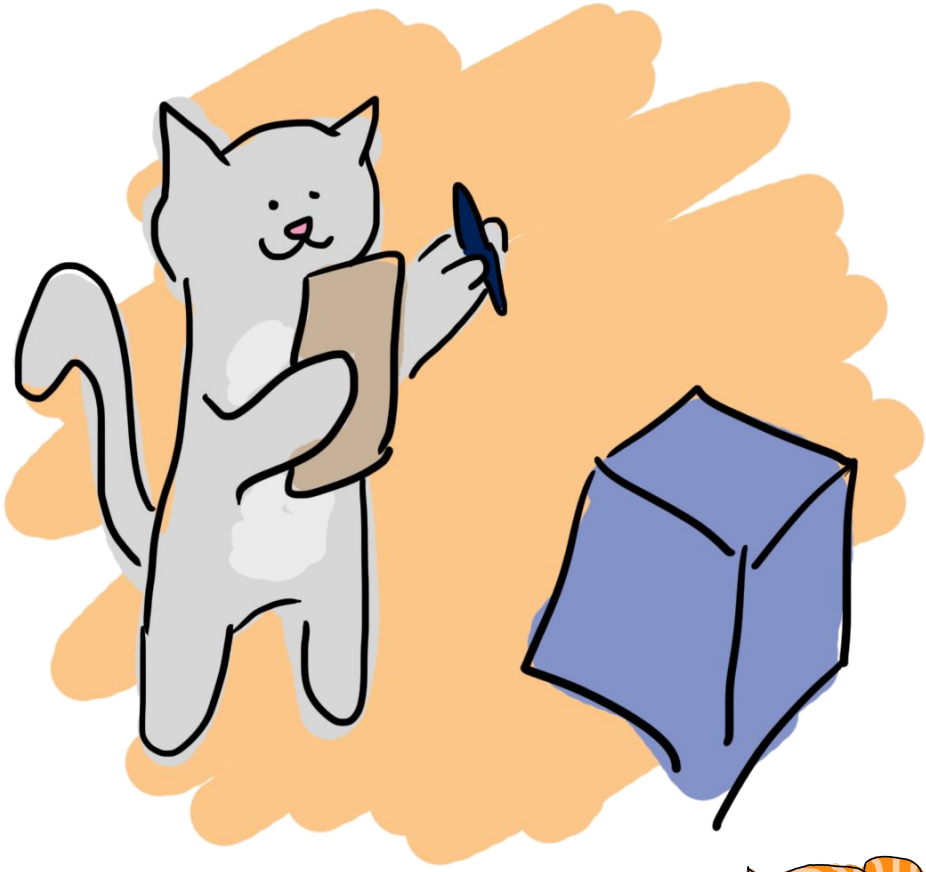
Learning Outcomes



- What tests are and why they are important
- What unit tests are and why you should write them
- How to approach writing unit tests
- Why you need test automation and some options
- Some ways to measure code / test quality
- Mocking, fixtures, and parametrization - oh my!
- Refactoring for unit testability
- Hopefully none of our bad habits :)

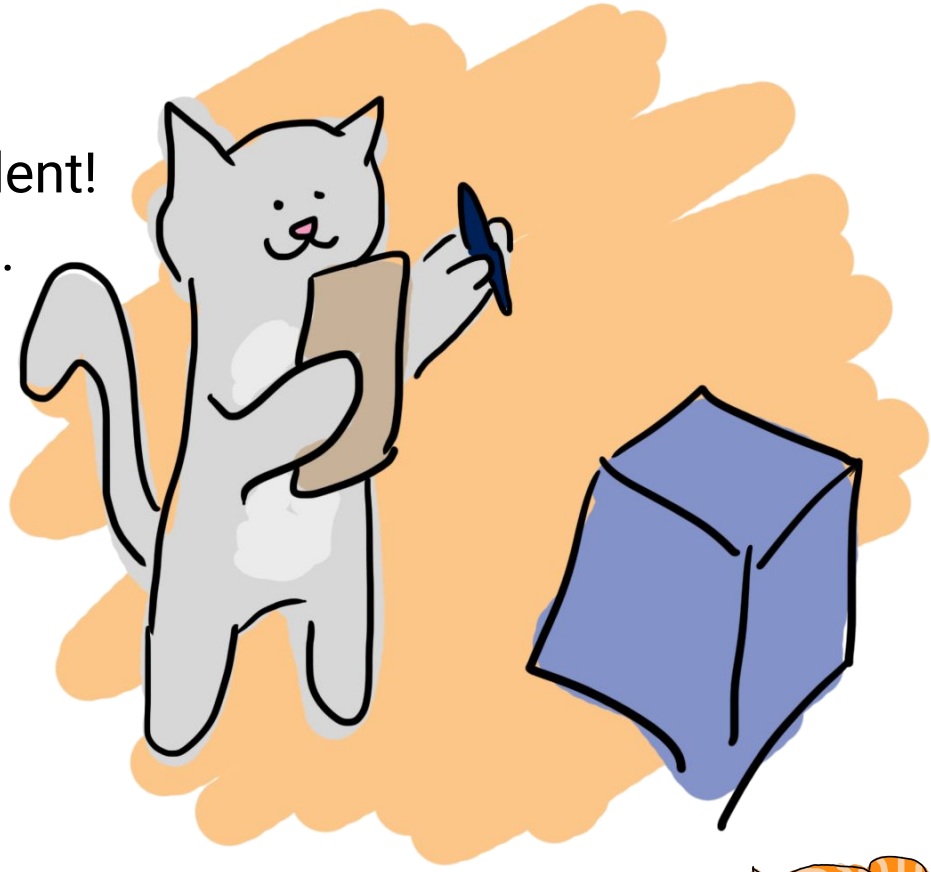
What is a test?

- Specifies how your software is intended to work
- Can be run against your software to verify it



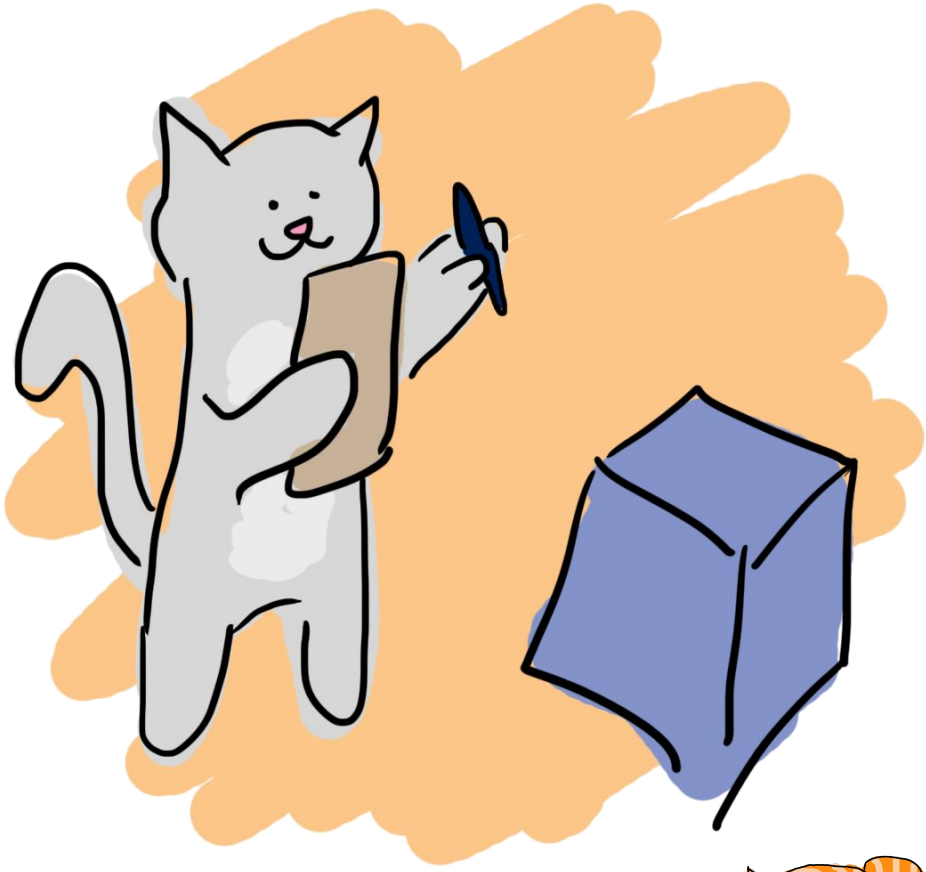
Why test?

- Increase:
  - Trust
  - Confidence
- You will never be 100% confident!
- But you can be 60% confident.



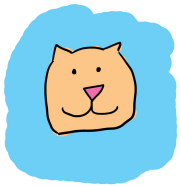
Types of tests

- Specifying and running tests for everything is:
  - Hard to maintain
  - Slow
  - Hard to write

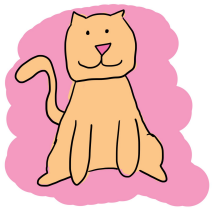


Types of tests

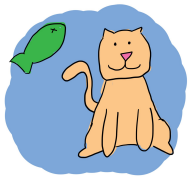
- Unit tests
  - Test ‘isolated units’
    - e.g. a method or function
  - Super high coverage
  - Most of the tests



- Integration tests
  - Combine units and test them together
  - Fill in the cracks between the tests

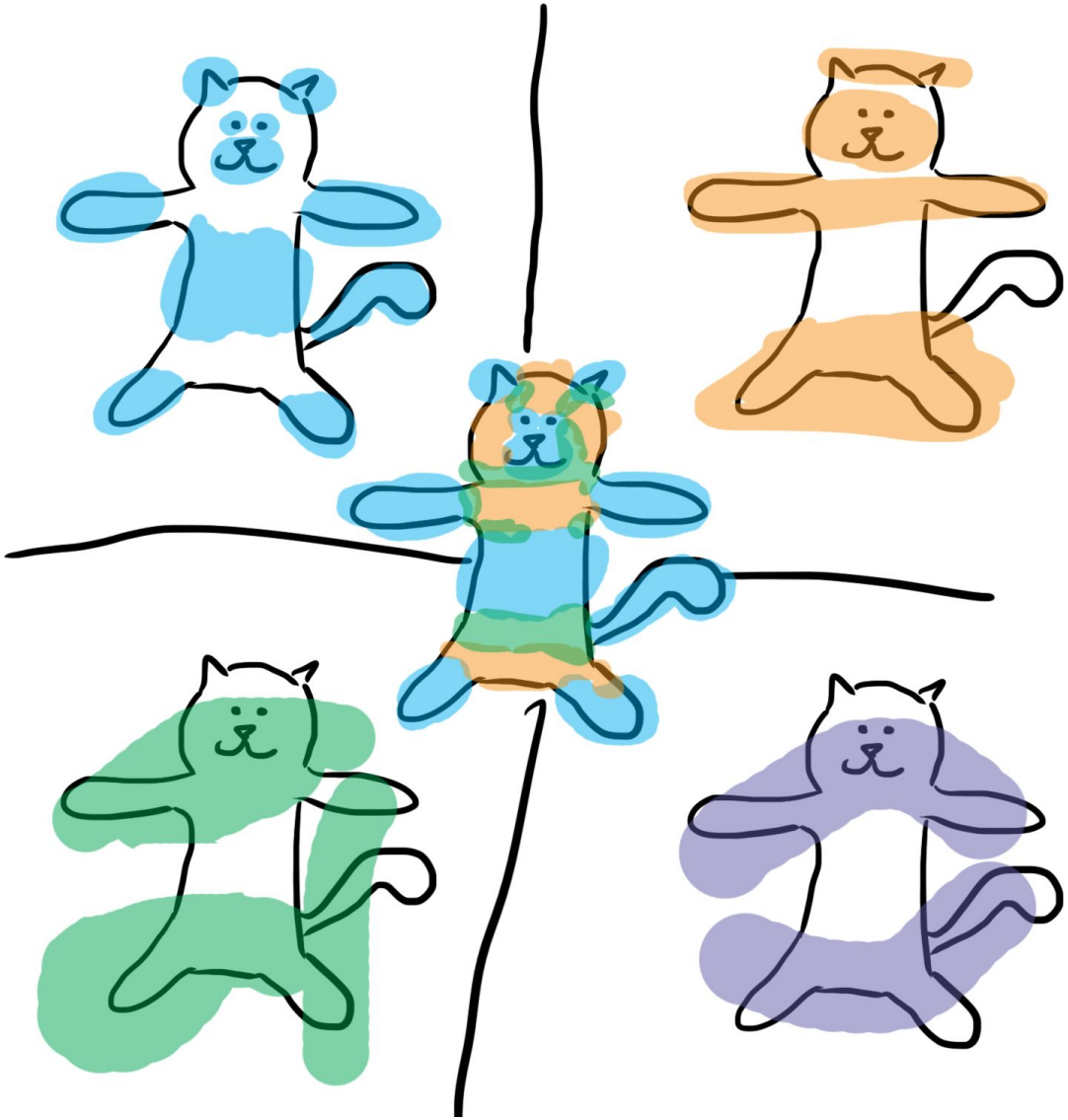


- System tests
  - Test with everything plugged together and configured as expected
  - From the end user’s perspective



- Acceptance tests
  - Test the customer’s use cases

Types of tests

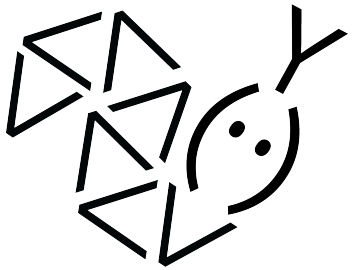


Tutorial: Setup and run existing tests

- https://github.com/keeppythonweird/catinabox/
- Follow along with /steps/1-run_tests.md
  - Setup a virtualenv
  - Run the existing tests

Optional - Use PythonAnywhere

- Sign up for an account (the beginner tier is free!)
- Start a bash session
- Create an SSH key and upload it to Github
  - **$ ssh-keygen**
  - < hit enter a bunch of times >
  - **$ cat .ssh/id_rsa.pub**
  - < copy the output to Github>
- Continue with the originally instructions at the “clone our repo” step



Coverage

Statement coverage == “Was this line executed?”

```
1 do_stuff()
2
3 if (a == 1) or (b == 2):
4     more_stuff()
```

statement

Decision coverage == “Was this code path executed?”

```
1 do_stuff()
2
3 if (a == 1) or (b == 2):
4     more_stuff()
```

decision

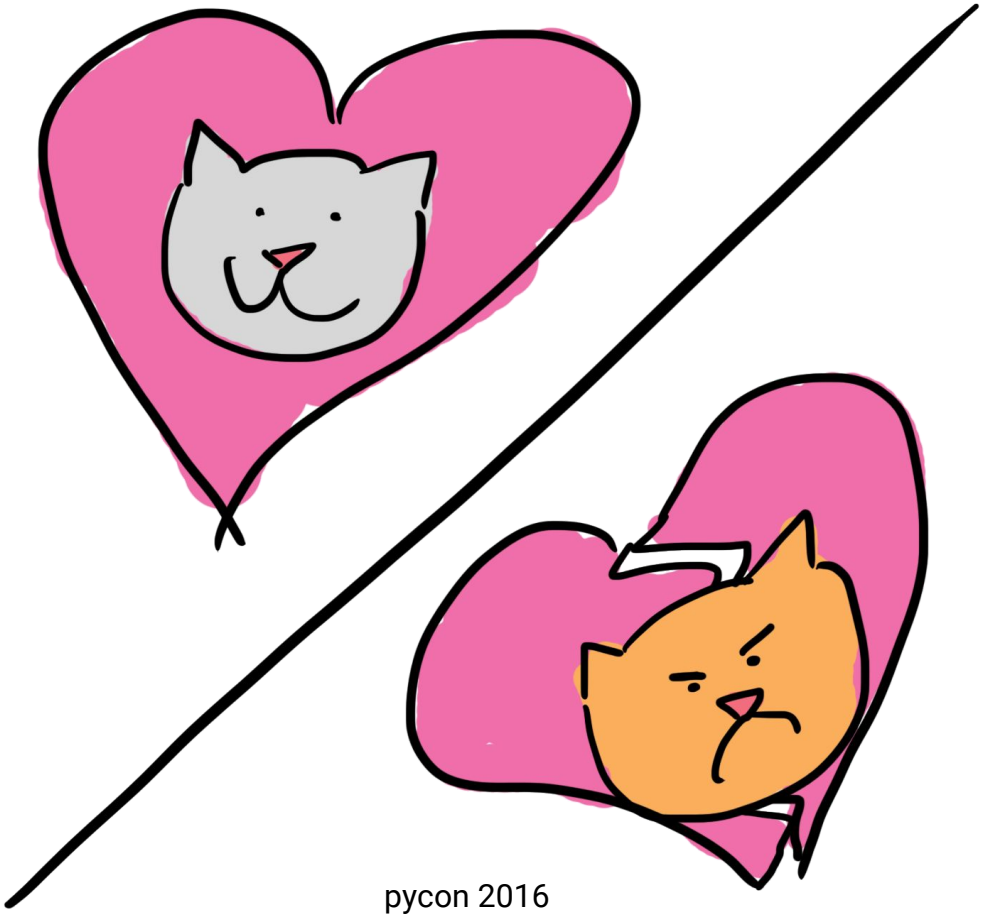
Condition coverage == “Was every part of the decision executed?”

```
1 do_stuff()
2
3 if (a == 1) or (b == 2):
4     more_stuff()
```

condition

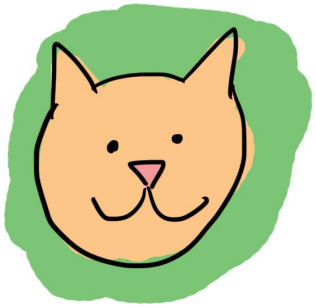
Unit Tests

- People often love or hate unit tests.
- But they are neutral, like brushing your teeth



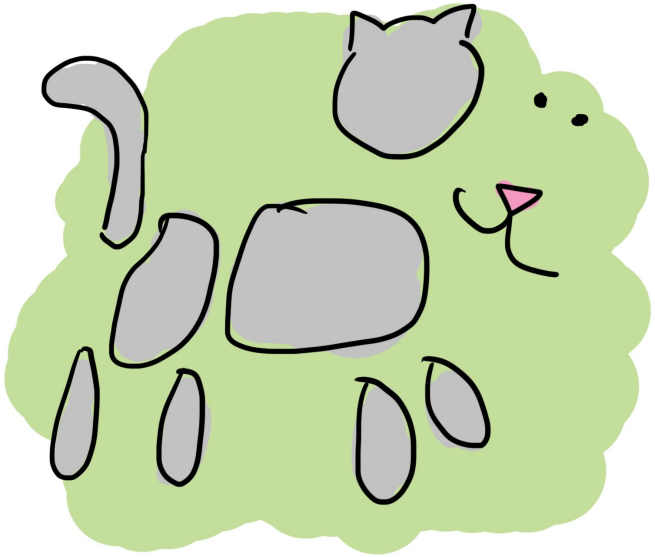
What are unit tests good for?

- Finding bugs DURING development
- A design tool
- Writing maintainable code
- Documenting a developer’s intentions
- Running quickly



What are unit tests not good for?

- Finding bugs
- Indicating that your application is functioning correctly
- Testing glue code
- Testing every possible permutation



Tests Pass!

Unit tests ARE NOT for preventing bugs

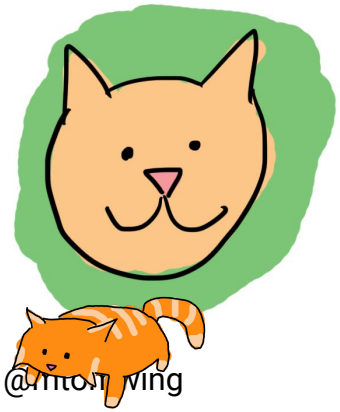
Unit tests ARE for writing clean maintainable code with confidence

Generating test cases

- Think about possible input
- Categorize the input into special cases
- One test per special case

How would we test this? - #1

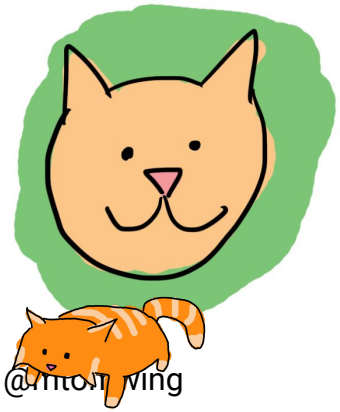
```
def is_palindrome(sequence):
    """Returns True iff. the sequence is a palindrome."""
    return sequence == sequence[::-1]
```



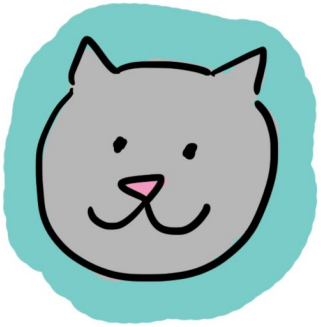
How would we test this?

```
def is_palindrome(sequence):
    """Returns True iff. the sequence is a palindrome."""
    return sequence == sequence[::-1]
```

- Input which IS a palindrome
- Input which is NOT a palindrome



Trusting sources of input

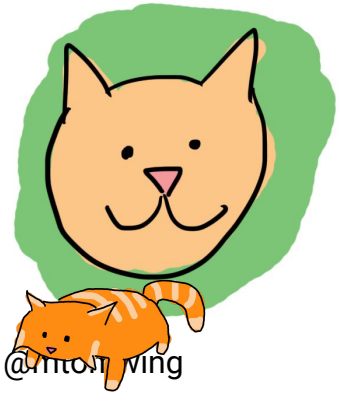


```
def is_palindrome(sequence):
    """Returns True iff. the sequence is a palindrome."""
    return sequence == sequence[::-1]
```

- What if the wrong type of data is passed in?
- What if the sequence is extremely large?
- Depends:
  - Where the input is coming from
  - Where you implement validation

How would we test this? - #2

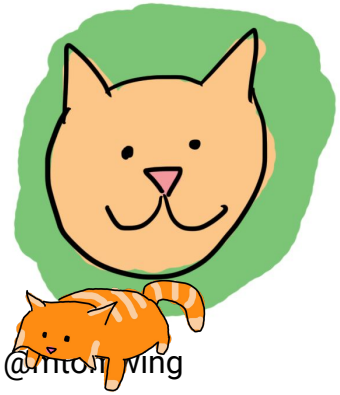
```
def is_leap_year(year):
    """Returns True iff. year is a leap year.

    This algorithm was shamelessly copied from Wikipedia.
    """
    if year % 4 != 0:
        return False
    elif year % 100 != 0:
        return True
    elif year % 400 != 0:
        return False
    else:
        return True
```



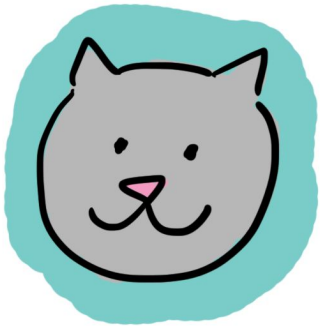
How would we test this? - #2

```
def is_leap_year(year):
    """Returns True iff. year is a leap year.

    This algorithm was shamelessly copied from Wikipedia.
    """
    if year % 4 != 0:
        return False
    elif year % 100 != 0:
        return True
    elif year % 400 != 0:
        return False
    else:
        return True
```

```
is_leap_year(1757) == False

is_leap_year(2004) == True

is_leap_year(1900) == False

is_leap_year(2000) == True
```



Python Unit Testing



unittest module

- Comes with the standard library
- Typically will do basically everything you need
- self.assertEqual(result, “cats”)

pytest module

- $ pip install pytest
- Provides everything that **unittest** does but with more batteries included!
- Less boilerplate thanks to magical fixtures.
- Assertions are more natural and do not require custom invocation.
- **assert result == “cats”**

We’ll be using **pytest** in this tutorial.

pytest - how to

1. **$ pip install pytest**
2. Create a module to hold your test (e.g. test_cool_stuff.py).
3. Write the test.

```
1 def test_cat_is_not_potato():
2     assert 'cat' != 'potato'
```

4. Run the test.

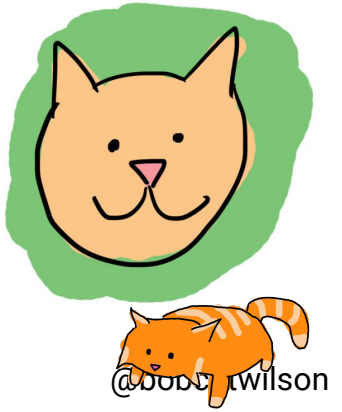
```
$ py.test test_cool_stuff.py
======================= test session starts ========================
platform linux -- Python 3.4.3 -- py-1.4.30 -- pytest-2.7.2
rootdir: /home/mtomwing/Projects/catinabox-pres, inifile:
plugins: cov, pep8, cache
collected 1 items

test_cool_stuff.py .

==================== 1 passed in 0.01 seconds ====================
```

pytest - how to continued

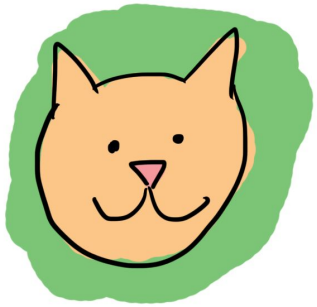
pytest will treat any function whose name starts with **test_** a test.
Same goes for test modules.

We can use plain old Python **assert** to test that things are as we expected them to be.

```
1 assert 'foo' in 'foobar'
2 assert len('cat') == 3
3 assert {'hello': 'world'} != {'world': 'hello'}
```



Unit Test Structure

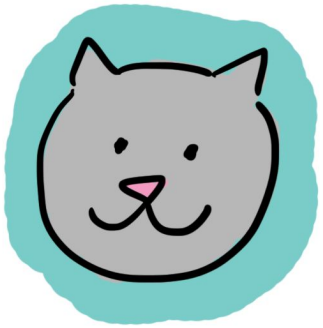


1. Define your inputs and any preconditions.
2. Invoke the thing.
3. Verify that it did what you expected.

TL;DR a test is an easy way for you to quantify what it means for your thing to "work".

```
1 def reverse(items):
2     return items[::-1]
3
4 def test_reverse_works_with_string():
5     assert reverse('abc') == 'cba'
6
7 def test_reverse_works_with_list():
8     assert reverse([1, 2, 3]) == [3, 2, 1]
```

PEP8



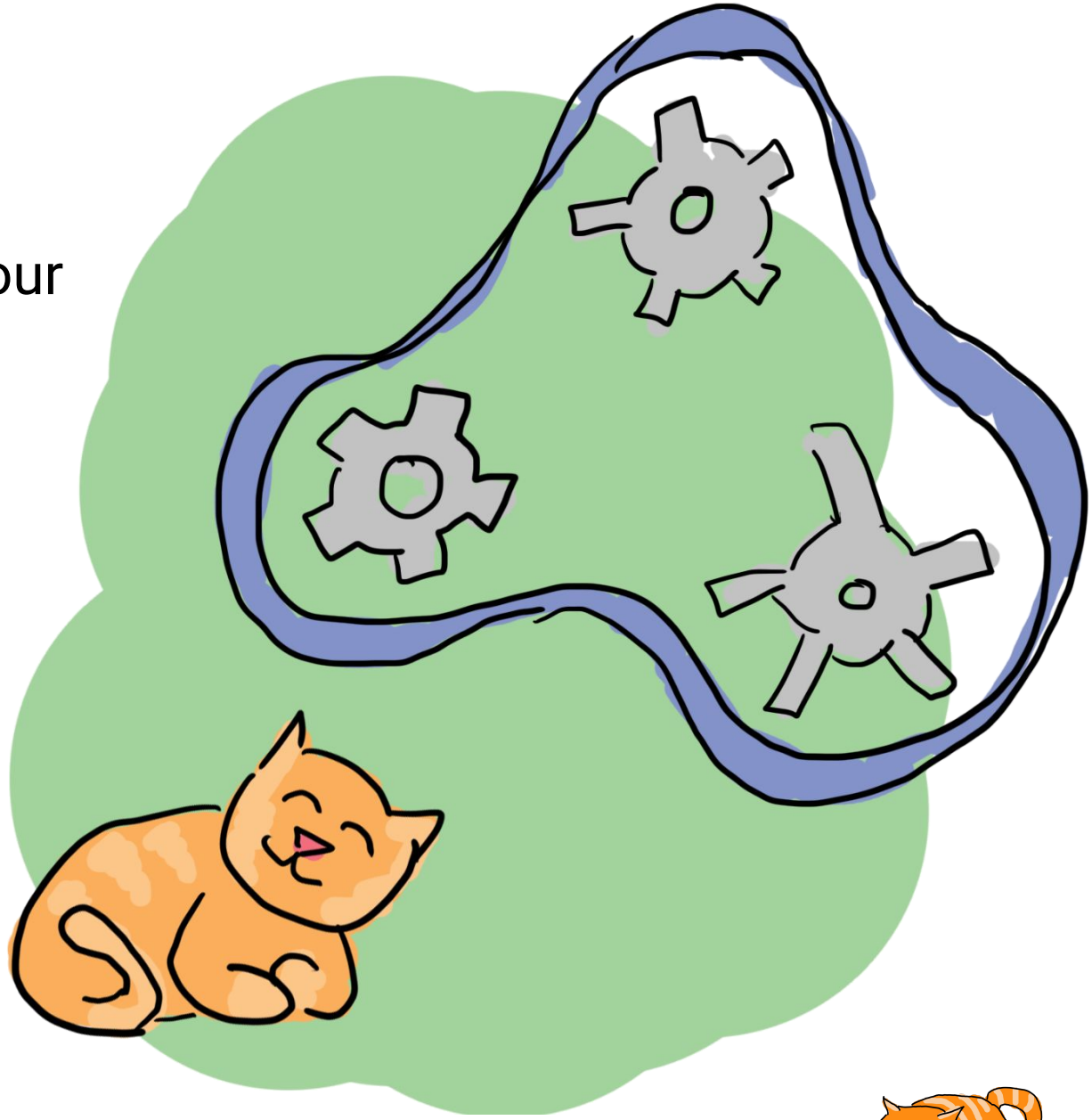
- It’s a coding standard
- Prescribes things like:
  - < 80 character lines
  - 2 new lines between functions in a module
  - 1 new line between methods in a class
  - Visual indentation rules
  - … and more!
- PEP8 isn’t the only standard out there! (see Google’s Python Style Guide)
- Main thing is to be consistent with the codebase
- Our tests will fail if **py.test** finds any PEP8 violations :)
- https://www.python.org/dev/peps/pep-0008/

Tutorial: Write your first test

- https://github.com/keeppythonweird/catinabox
- Follow along with steps/2-simple_function.md
  - Finish writing the tests in test_catmath.py

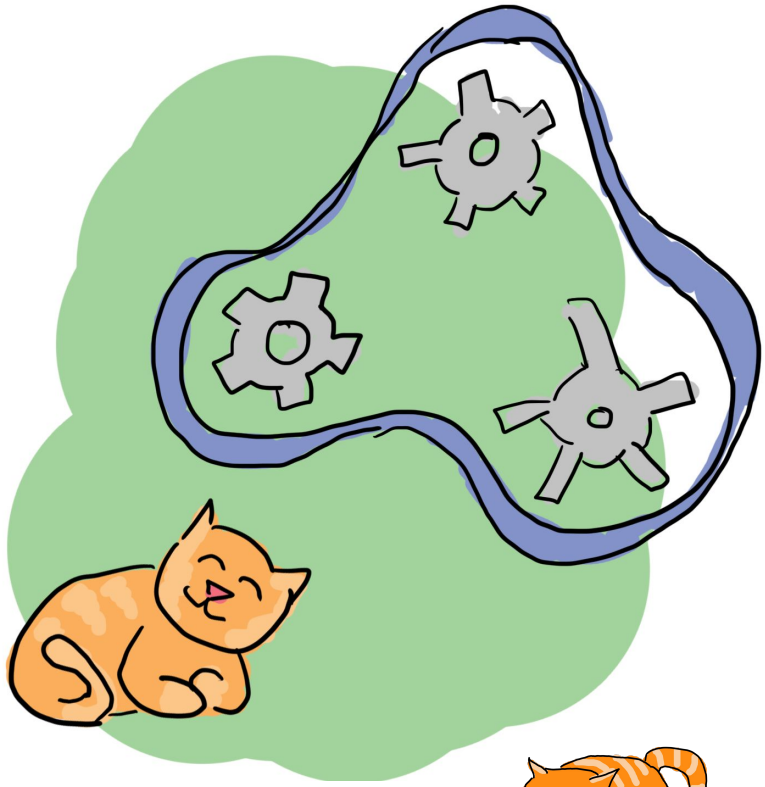
Test Automation

- You should run your tests regularly!



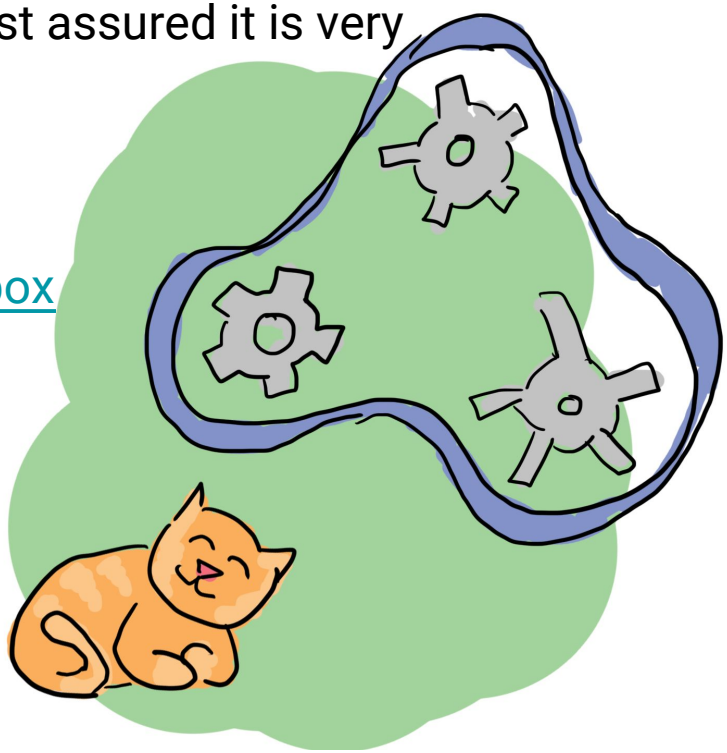
Test Automation - System tests

- Reduce developer burden
  - Slower
  - More difficult to set up



Travis CI

- CI = Continuous Integration
- Third party service that will “build” your Github projects
  - “build” = “run the tests” in our case
- Free for open source projects
- We won’t be covering setting up Travis, but rest assured it is very simple!
- Other CI services are available (e.g. Atlassian’s Bamboo)
- https://travis-ci.org/keeppythonweird/catinabox

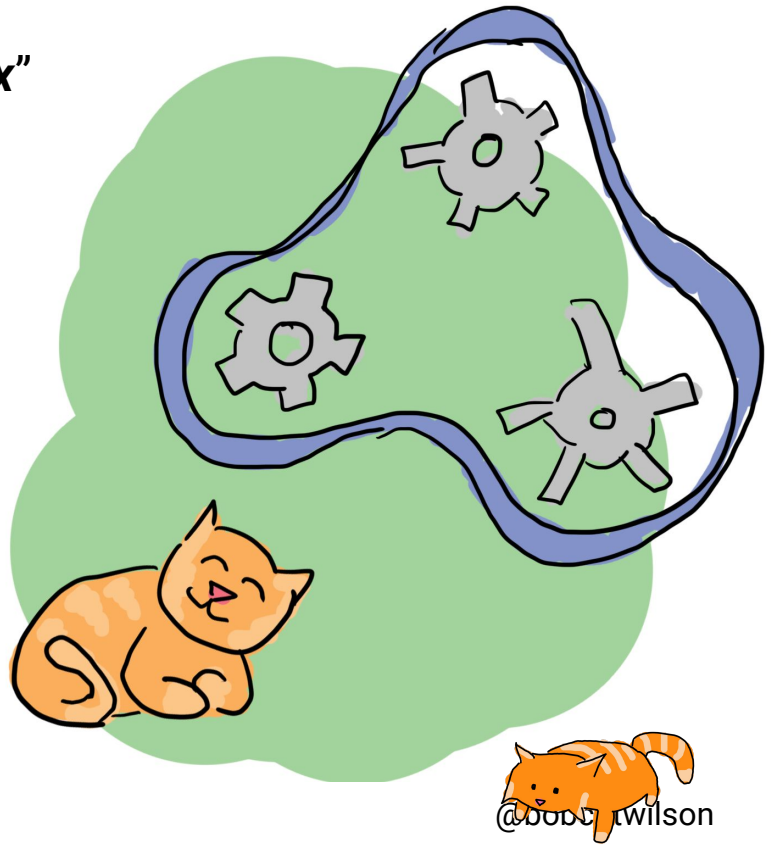


Coveralls

- Third-party service for measuring statement coverage of your Github project
- Free for open source projects
- Track changes in coverage over time
- https://coveralls.io/github/keeppythonweird/catinabox

Other Testable Aspects

- Sometimes it's also worth adding other checks to your testing pipeline.
- Static Analysis: Done entirely offline - without running your code
- Cyclomatic Complexity
  - A measure of how complex a function is
  - Checks that functions "***aren't too complex***"
  - $ pip install pytest-mccabe
- PEP8
  - Checks for PEP8 compliance
  - $ pip install pytest-pep8
- Pyflakes
  - Checks for syntax errors
  - $ pip install pytest-flakes
- You can have these run before your tests in order to fail fast!



Tutorial: Create a pull request

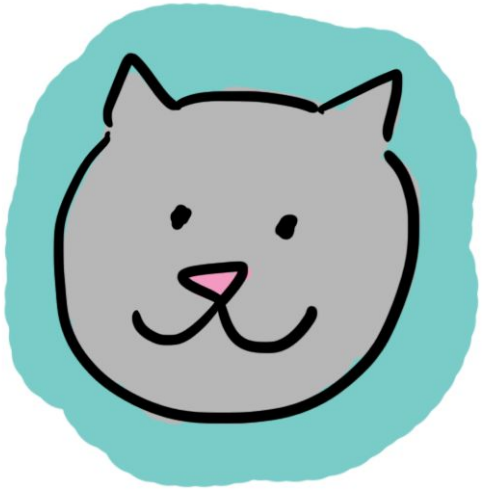
- https://github.com/keeppythonweird/catinabox
- Follow along with steps/3-pull.md
  - Commit your new tests
  - Create a pull request from your fork

BONUS!

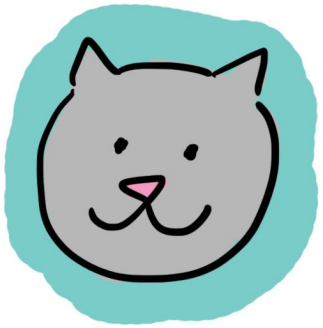
- If you finish early, review the other pull requests
  - Be respectful and positive
  - This presentation has great tips for effective code reviews:
    - http://confreaks.tv/videos/railsconf2015-implementing-a-strong-code-review-culture

Trusting sources of input

- What if we didn’t trust the input?
- What other test cases might we have for cat_years_to_hooman_years?



Generating test cases



- < 0
- 0
- Fraction of a year
- Most ages
- > 1000
- Wrong data type
- NaN

pytest - Testing for exceptions

- pytest.raises

```
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
>>> import pytest
>>> with pytest.raises(TypeError):
...     '2' + 2
...
```


Advanced cat hooman

- **catinabox/safecatmath.py**
- Now checks that **age_in_cat_years** is an **int** or **float**.
- Also makes sure the cat is not too young or too old.

```
25      if not isinstance(age_in_cat_years, (int, float)):
26          raise InvalidAge(age_in_cat_years)
27
28      if not 0 <= age_in_cat_years <= MAX_CAT_AGE:
29          raise InvalidAge(age_in_cat_years)
```


Tutorial: Testing incorrect input

- https://github.com/keeppythonweird/catinabox
- Follow along with steps/4-input.md

pytest - fixtures

Fixtures are a way to define reusable components that are required by your tests. **Pytest** will automagically hook up your fixtures to your tests (or other fixtures!) that require them.

```
1 import pytest
2
3 @pytest.fixture
4 def cool_stuff():
5     return CoolStuff()
6
7 def test_the_things(cool_stuff):
8     assert cool_stuff.is_cool == True
```

See https://pytest.org/latest/builtin.html for more information on the built-in fixtures provided by **pytest**.

pytest - fixtures continued

By default, fixtures are recreated for every test that requires them.

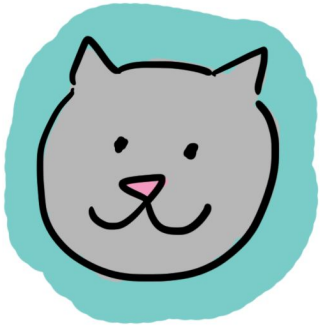
```
 1 import pytest
 2
 3 @pytest.fixture
 4 def cool_stuff():
 5     return [1, 2]
 6
 7 def test_the_things(cool_stuff):
 8     del cool_stuff[0]
 9     assert cool_stuff == [2]
10
11 def test_the_things_again(cool_stuff):
12     del cool_stuff[1]
13     assert cool_stuff == [1]
```

It is possible to control the lifetime of a fixture (e.g. create it once for all the tests), but that is out of scope for today! See https://pytest.org/latest/fixture.html.

Tutorial: Testing classes with fixtures

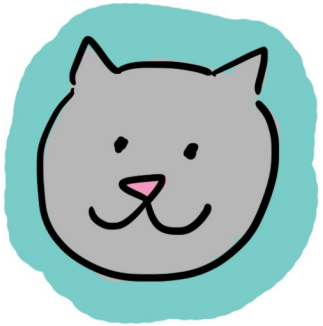
- https://github.com/keeppythonweird/catinabox
- Follow along with steps/5-classes.md

Unit testing and state of the outside world



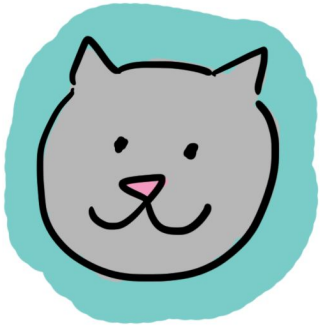
- What if you want to test functionality that:
  - Uses the current time/sleeps
  - Depends on an external service (e.g. an HTTP server or DB)
  - Uses random
- Super easy in Python!!!

Mocking



- Create “mock” objects that mimic the external objects/functions
- You can control their behaviour completely!
  - Return whatever time you want
  - Pretend to sleep
  - Return fake DB or HTTP results
  - Return deterministic results instead of random
- Verify arguments used
- Verify that everything is plugged together correctly
  - Test the true behaviour later with system tests

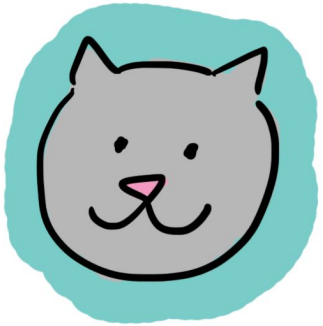
Mocking



- mock
  - pip install mock
- Included in the Python 3 standard lib

```
In [1]: import mock

In [2]: obj = mock.MagicMock()

In [3]: value = obj[5]

In [4]: value = obj.foo()

In [5]: obj.foo.call_count
Out[5]: 1
```


Mocking



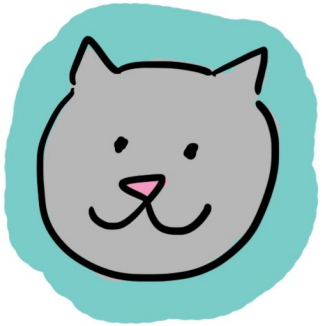
- mock

```
In [6]: value = obj.bar(6)

In [7]: obj.bar.assert_called_with(6)

In [8]: obj.bar.assert_called_with(7)
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-8-c790567ec790> in <module>()
----> 1 obj.bar.assert_called_with(7)

/Library/Python/2.7/site-packages/mock/mock.pyc in assert_called_with(_mock_self, *args, **kwargs)
    935         if expected != actual:
    936             cause = expected if isinstance(expected, Exception) else None
--> 937             six.raise_from(AssertionError(_error_message(cause)), cause)
    938
    939

/Library/Python/2.7/site-packages/six.pyc in raise_from(value, from_value)
    716 else:
    717     def raise_from(value, from_value):
--> 718         raise value
    719
    720

AssertionError: Expected call: bar(7)
Actual call: bar(6)
```


Patching

- Replace methods/classes/modules with mock objects
- Clean up automatically at the end of a test

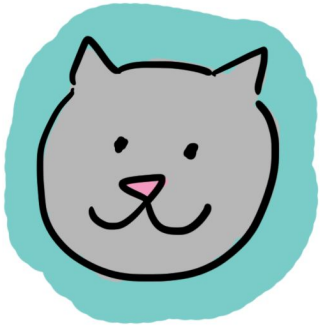
Patching with pytest



- pytest-mock
  - pip install pytest-mock
  - Wrapper around the mock library the works well with pytest

```
import time


def time_message():
    return "Time is {}".format(time.time())


def test_time_message(mocker):
    mocked_time = mocker.patch.object(time, 'time', autospec=True)
    mocked_time.return_value = 7
    assert time_message() == "Time is 7"
```

Mock and Patch - autospec



- Make sure that the expected interface is being
  - Raises if methods or attributes are used that don't exist on the mocked object
- Always use autospec!

```
mocked_time = mocker.patch.object(time, 'time', autospec=True)
```


Tutorial: Control time with mock

- https://github.com/keeppythonweird/catinabox
- Follow along with steps/6-mock.md

Parameterization - condensing tests

- cat_years_to_hooman_years
- What if we wanted to test for more bad input?
  - So many more tests to write!

```
@pytest.mark.parametrize("age", [
    "five",
    [3, 4],
    {2: 3},
    (),
    1000.1,
    -4
])
def test__cat_years_to_hooman_years__bad_input__raises(age):
    with pytest.raises(safecatmath.InvalidAge):
        safecatmath.cat_years_to_hooman_years(age)
```

Parameterization of fixtures

- Fixtures can be parametrized too!
- **py.test** will automatically run every permutation of tests and fixtures

```
 1 import pytest
 2
 3 @pytest.fixture(params=[
 4     "sqlite:////tmp/foobardb",
 5     "mysql://foo@bar/database",
 6 ])
 7 def db(request):
 8     return DatabaseConnector(request.param)
 9
10 def test_select_works(db):
11     assert db.select('foo').from_('bar')
```

Tutorial: Testing with parameterization

- https://github.com/keeppythonweird/catinabox
- Follow along with steps/7-params.md

Unit testability and well factored code

- Lots of code is hard to unit test
- Usually not well factored
- Refactoring for unit testability = higher quality code

Example: Poorly factored code

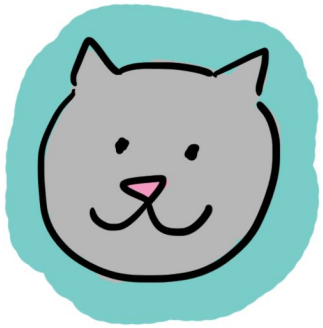
catinabox/examples/complected/cats.py

```
def setup_cats(num_cats):
    cat_names = ["Fluffles", "Enzo", "Lisa", "Berto", "Jillian", "Amy",
                 "Bella", "Moe", "Tibby"]

    foods = ["vinegar", "vegemite", "vanilla", "acorn squash",
             "Canadian bacon", "alligator", "cayenne pepper", "adobo",
             "almond butter",
             "garlic"]

    cats = []
    for _ in range(num_cats):
        new_cat = {
            "name": random.choice(cat_names),
            "last_ate": None
        }
        cats.append(new_cat)

    for cat in cats:
        cat["last_ate"] = random.choice(foods)

    return cats
```

Example: Test for poorly factored code

catinabox/examples/test_complected.py

- Hard to write
- Hard to read
- Hard to maintain
- Adds little
- Copy pasta



```python
def test__setup_cats__many_cats(mocker):
    random_choice = mocker.patch('random.choice')

    random_choice.side_effect = [
        # Mock the cat names
        "Frazzle", "Dazzle", "Razzle",
        # Mock the foods the cats will be fed
        "cheese", "cucumber", "papaya"
    ]

    result_cats = cats.setup_cats(3)

    assert result_cats[0] == {"name": "Frazzle",
                              "last_ate": "cheese"}

    assert result_cats[1] == {"name": "Dazzle",
                              "last_ate": "cucumber"}

    assert result_cats[2] == {"name": "Razzle",
                              "last_ate": "papaya"}
```

Well factored code

- Highly cohesive
- Loosely coupled
- Does one thing
- Isolate glue code (avoid complecting)*

* Rich Hickey: https://www.infoq.com/presentations/Simple-Made-Easy

Example: Refactor the code for testability

catinabox/examples/uncomplected/cats.py

```
def get_cat_name():
    cat_names = ["Fluffles", "Enzo", "Lisa", "Berto", "Jillian", "Amy",
                 "Bella", "Moe", "Tibby"]
    return random.choice(cat_names)


def get_food():
    return random.choice(
        ["vinegar", "vegemite", "vanilla", "acorn squash",
         "Canadian bacon", "alligator", "cayenne pepper", "adobo",
         "almond butter",
         "garlic"])


def setup_cats(num_cats):
    cats = [Cat(name=get_cat_name()) for _ in range(num_cats)]

    for cat in cats:
        cat.feed(get_food())

    return cats
```

Example: Refactor the code for testability

catinabox/examples/test_uncomplected.py

```
def test__get_cat_name(mocker):
    mocker.patch('random.choice', return_value="Snookums")
    cat_name = cats.get_cat_name()
    assert cat_name == "Snookums"


def test__get_food(mocker):
    mocker.patch('random.choice', return_value="carrot")
    food = cats.get_food()
    assert food == "carrot"


def test__setup_cats__many_cats(mocker):
    mocker.patch.object(cats, "get_cat_name", side_effect=["Jess", "Larry",
                                                           "Sue"])
    mocker.patch.object(cats, "get_food", side_effect=["berries", "milk",
                                                       "soda"])
    result_cats = cats.setup_cats(3)
    assert result_cats == [
        cats.Cat("Jess", "berries"),
        cats.Cat("Larry", "milk"),
        cats.Cat("Sue", "soda")
    ]
```

Tutorial: Refactoring for unit testability

- https://github.com/keeppythonweird/catinabox
- Follow along with steps/8-refactor.md

Refactor for testability: Group code review

As a group review solution: refactored catgenerator and its tests

- Is the code better or worse?
  - Which parts are better?
  - Which parts are worse?
- Is the code well tested?
- How readable is the test?

Summary

- MOAR TRUST!
- MOAR CONFIDENCE!

QUESTIONS