W4156

Testing I

Agenda

(This is one of my favorite topics so brace for some enthusiasm)

We are going to cover *theory* in parallel to *practice/code*

- ❏ Why test?
- ❏ It is impossible to *prove* correctness*
- ❏ Define Testing
- ❏ Testing Techniques
- ❏ Test Suite Adequacy

* this is a bit of a 'whoa' moment

Why Test?

Reason 1: We want our software to work

Ariane 5 Maiden Flight

Corrupted Blood
Deals 263 to 337 damage every 2 sec.
Spreads to nearby allies.

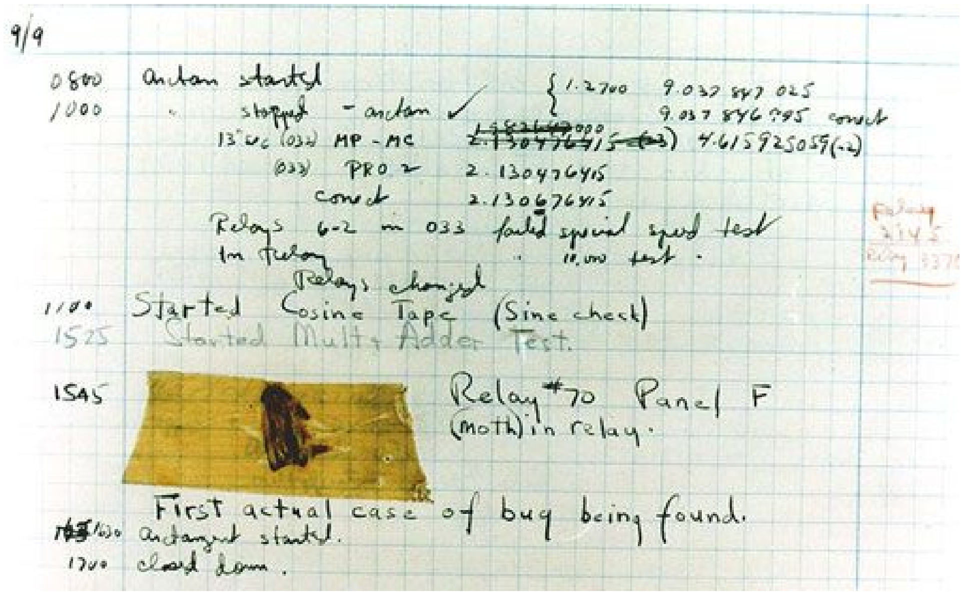
Accountant 1.3 Loaded
Patch 1.7 is live! Customers who are using custom user-interfaces may need to remove them in order to avoid potential in-game issues.
For more patch information visit www.worldofwarcraft.com.
Amork dies.
You create: [Major Healthstone].
[Trickclover] says: We should have a bear race lol
[Rog] says: WTF?

Warcraft ‘Corrupted Blood’ incident

Therac 25

Reason 2: Customer experience/quality ***not*** the only reason. We will look at software engineer perspective in a later lecture

Terminology I



9/9

0800 Arctan started

1000 " stopped - arctan ✓ {1.2700 9.037 847 025 / 9.037 846 995 correct

13" uc (032) MP - MC ~~1.982147000~~ ~~2.130476415 (-3)~~ 4.615925059(-2)

(033) PRO 2 2.130476415

correct 2.130676415

Relays 6-2 in 033 failed special speed test

In relay " 10,000 test.

Relays changed

1100 Started Cosine Tape (Sine check)

1525 Started Mult + Adder Test.

1545 Relay #70 Panel F (moth) in relay.

First actual case of bug being found.

1630 Arctangent started.

1700 closed down.

Relay 2145

Relay 3370

Terminology II

Mistake/Error: a human action which introduces a fault

Fault (Defect): Incorrect code that when exercised creates a error

Error: State of the system differs from specification (not yet externally visible)

Failure: Failure of program to meet any quality objective (visible to user)

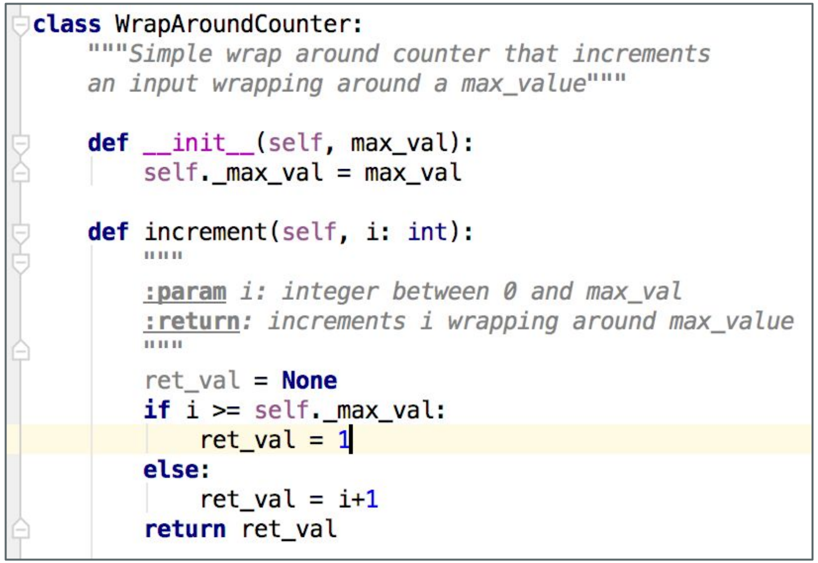
The program behavior is being *compared* to defined F & NFR

Defining Testing

Testing can be described as a process used for *revealing defects* in software, and for establishing that the software has attained a specified degree of quality with respect to selected attributes

- Practical Software Testing

Testing: WrapAroundCounter



```
class WrapAroundCounter:
    """Simple wrap around counter that increments
    an input wrapping around a max_value"""

    def __init__(self, max_val):
        self._max_val = max_val

    def increment(self, i: int):
        """
        :param i: integer between 0 and max_val
        :return: increments i wrapping around max_value
        """
        ret_val = None
        if i >= self._max_val:
            ret_val = 1
        else:
            ret_val = i+1
        return ret_val
```

1. How can we test this simple function?

2. What values would we push in?

(see corresponding lecture code)

Testing

Intuitively (for now), it is simple

1. Create a wrap around counter
2. Push in a set of values
3. Get the result
4. Assert result == expected

Testing Terminology

Code/System Under Test: code/system we are attempting to verify behavior

Test Case: <inputs, expected conditions, expected output>

Test Suite: Set of test cases

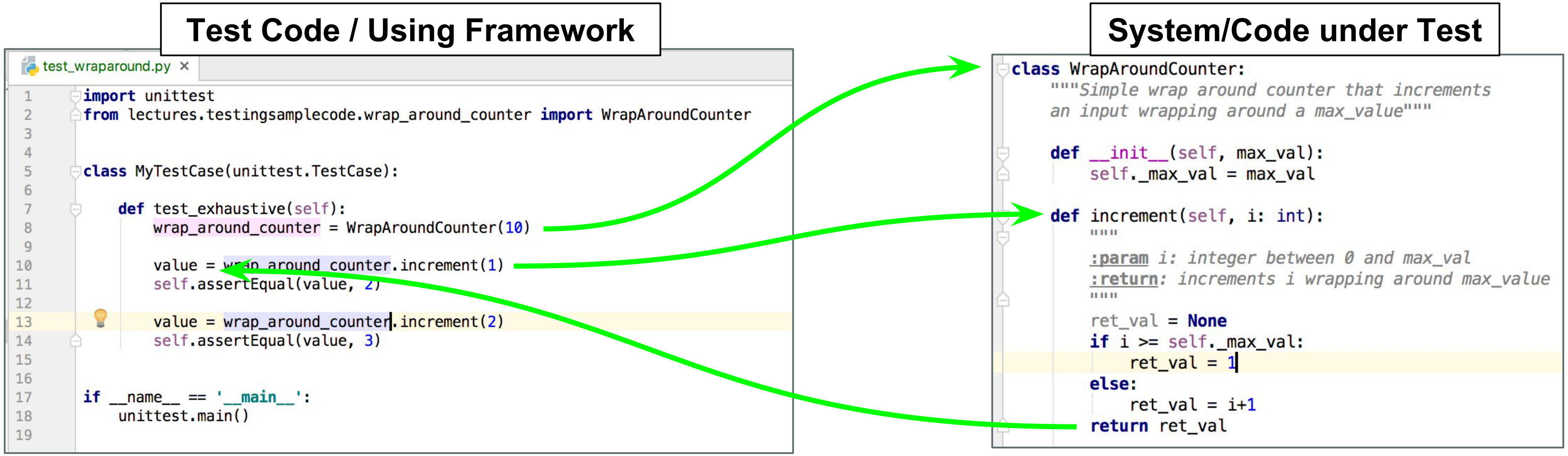
Oracle: Mechanism for determining pass/fail
(oracle generally produces expected output to compare to actual output. Hint: you are often the oracle)

Software Quality: extent to which system meets stated requirements

Test Suite Adequacy: confidence test suite establishes quality

Introducing Testing Frameworks

1. Testing frameworks help us implement how we decided to test our code
2. They allow us to write code (the tests) which execute our code under test
3. We push in values, get the results, then check/assert that the values are as we anticipate
   (Of course we now have the advantage we can run our tests again and again ….)



Test Code / Using Framework

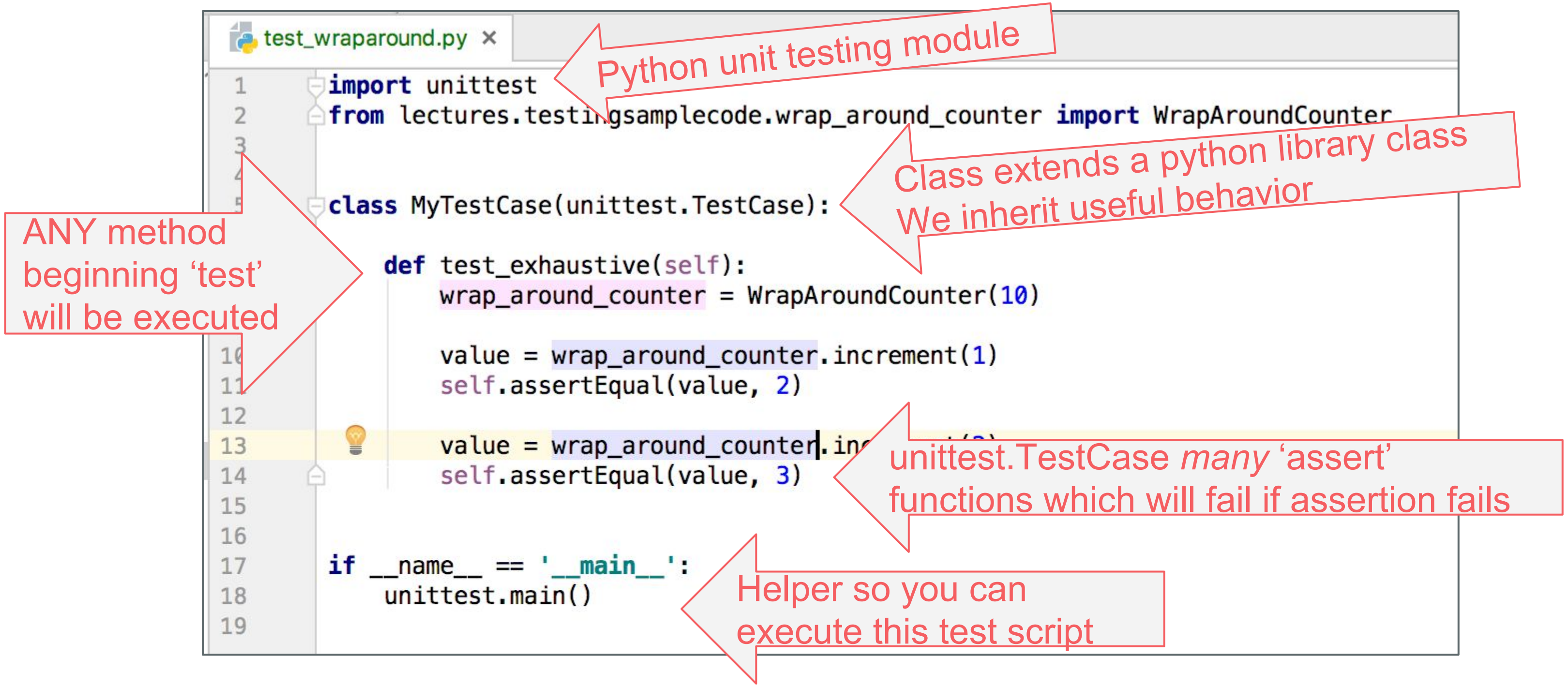
test_wraparound.py

```python
import unittest
from lectures.testingsamplecode.wrap_around_counter import WrapAroundCounter


class MyTestCase(unittest.TestCase):

    def test_exhaustive(self):
        wrap_around_counter = WrapAroundCounter(10)

        value = wrap_around_counter.increment(1)
        self.assertEqual(value, 2)

        value = wrap_around_counter.increment(2)
        self.assertEqual(value, 3)


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

System/Code under Test

```python
class WrapAroundCounter:
    """Simple wrap around counter that increments
    an input wrapping around a max_value"""

    def __init__(self, max_val):
        self._max_val = max_val

    def increment(self, i: int):
        """

        :param i: integer between 0 and max_val
        :return: increments i wrapping around max_value
        """

        ret_val = None
        if i >= self._max_val:
            ret_val = 1
        else:
            ret_val = i+1
        return ret_val
```

Anatomy of a unit test



test_wraparound.py

```
import unittest
from lectures.testingsamplecode.wrap_around_counter import WrapAroundCounter


class MyTestCase(unittest.TestCase):

    def test_exhaustive(self):
        wrap_around_counter = WrapAroundCounter(10)

        value = wrap_around_counter.increment(1)
        self.assertEqual(value, 2)

        value = wrap_around_counter.in
        self.assertEqual(value, 3)


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

Python unit testing module

Class extends a python library class
We inherit useful behavior

ANY method beginning 'test' will be executed

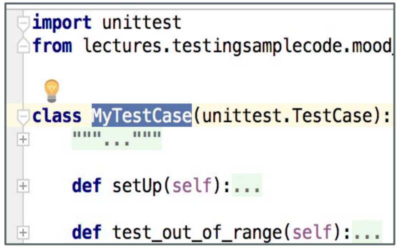
unittest.TestCase *many* 'assert' functions which will fail if assertion fails

Helper so you can execute this test script

https://docs.python.org/3/library/unittest.html

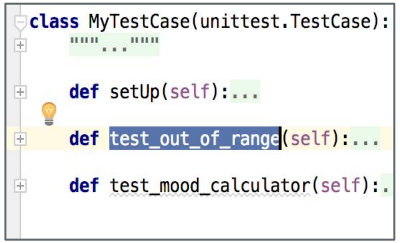
Running a Unit Test

1. Select ….



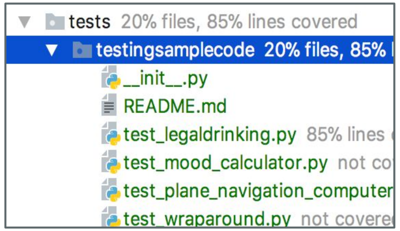
```
import unittest
from lectures.testingsamplecode.mood

class MyTestCase(unittest.TestCase):
    """..."""

    def setUp(self):...

    def test_out_of_range(self):...
```

A) Test Case



```
class MyTestCase(unittest.TestCase):
    """..."""

    def setUp(self):...

    def test_out_of_range(self):...

    def test_mood_calculator(self):.
```

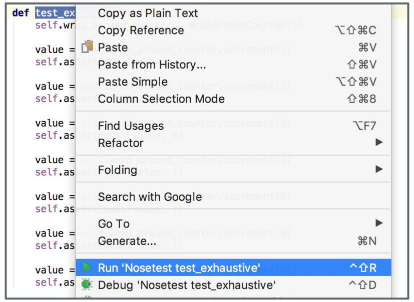
B) Test Case



```
▼ tests  20% files, 85% lines covered
  ▼ testingsamplecode  20% files, 85% l
      __init__.py
      README.md
      test_legaldrinking.py  85% lines
      test_mood_calculator.py  not co
      test_plane_navigation_computer
      test_wraparound.py  not covere
```

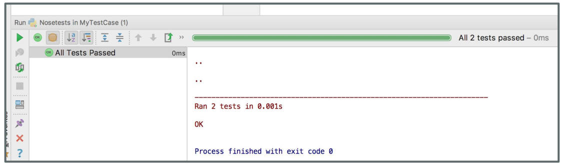
C) File/Folder

2. Right click
3. Run or Debug*



```
def test_ex
    self.wr
    value =
    self.as
    value =
    self.as
    value =
    self.as
    value =
    self.as
    value =
    self.as
    value =
    self.as
    value =
    self.as
```

4. See Result



```
..

..

----------------------------------------------------------------------
Ran 2 tests in 0.001s

OK


Process finished with exit code 0
```

* Will cover debugging in later lecture.
Test suite will also be run automatically in later lecture (CI/CD)

Exhaustive Testing: I

In our previous trivial example of wrap around counter of N=5

1. We *could exhaustively* test every possible input and output combination
2. However, even for trivial size it was time consuming (and tedious)
3. In this trivial example it was also easy to compute the correct result. For non-trivial functions (or programs) this is complex.

Exhaustive Testing: II

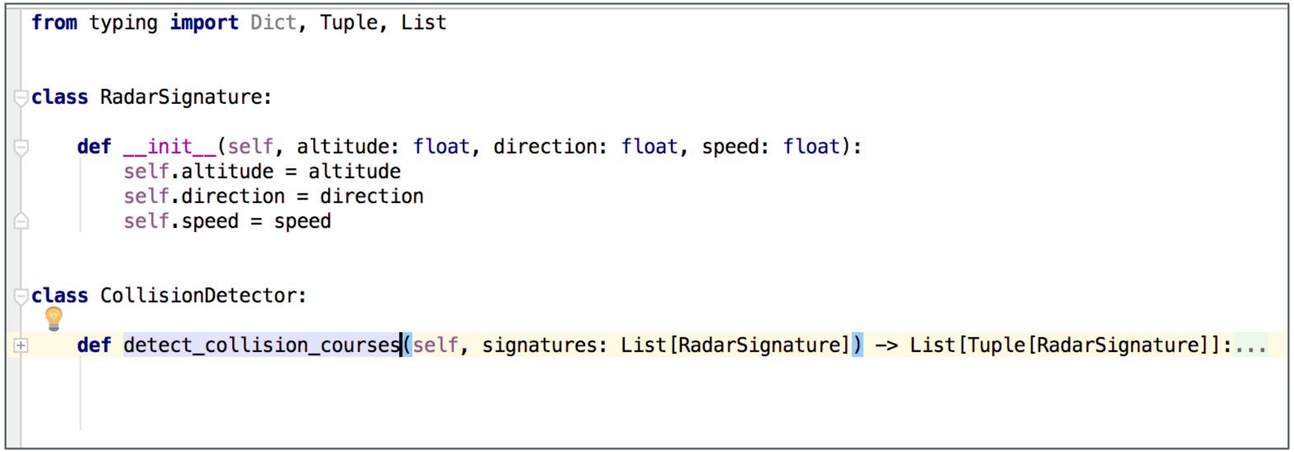
1. What if the wrap around counter limit was 1000 or 10000?

2. Or if a function took two integers as arguments?
    a. Each input is –2,147,483,648 to 2,147,483,67
    b. Input space is 1.8 x 10^{19}
    c. At 1bn test cases/sec still takes years

[Beginning Software Engineering]

Exhaustive Testing: II

1. Previous ‘two ints’ is still a trivial program
2. The input space of the below *class* is significantly larger
3. Let alone real software systems (multiple classes, modules, M LOC)



```
from typing import Dict, Tuple, List


class RadarSignature:

    def __init__(self, altitude: float, direction: float, speed: float):
        self.altitude = altitude
        self.direction = direction
        self.speed = speed


class CollisionDetector:

    def detect_collision_courses(self, signatures: List[RadarSignature]) -> List[Tuple[RadarSignature]]:...
```

In Practice it is impossible to Exhaustively Test

[1] Formal methods used in some safety critical systems but small percentage of software in development, incredibly expensive and still with limitations. We have not even discussed testing non-functionals

What does this mean for “Testing”

- Testing can only show *presence* of bugs, not the absence - Dijkstra
- Can not exhaustively test input domain (inc. finite resources: time, compute)

Therefore,

1. Test using a ***finite*** set of test cases
2. ***Find*** test cases with the ***highest probability of exposing defects***
3. After executing a *test suite* ask: “should we be confident?”

Our previous definition makes more sense

Testing can be described as a process used for *revealing defects* in software, and for establishing that the software has attained a specified degree of quality with respect to selected attributes

- Practical Software Testing

We are not ‘proving’ correctness. We are trying to find defects. If test suite fails we obviously have a bug. If it passes we need to ask whether we have established confidence we have met quality requirements

Testing: LPT

- There are a lot of testing techniques. Don’t get overwhelmed

- We have *established* we can not exhaustively test

- All the techniques are different ways of either
  - *Generating* test cases with high probability of exposing defects

  OR
  - *Assessing* how confident we should be after executing a test suite

Cheat Sheet

Can I see code internals?	No: Black Box		
	Yes: White Box		

Cheat Sheet

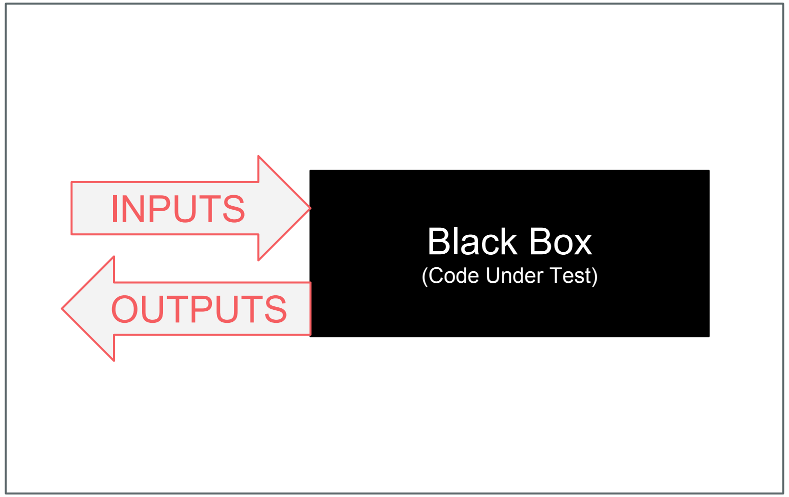
		Am I allowed to execute the code?	
		No: Static Testing	Yes: Dynamic Testing
Can I see code internals?	No:Black Box		
	Yes:White Box		

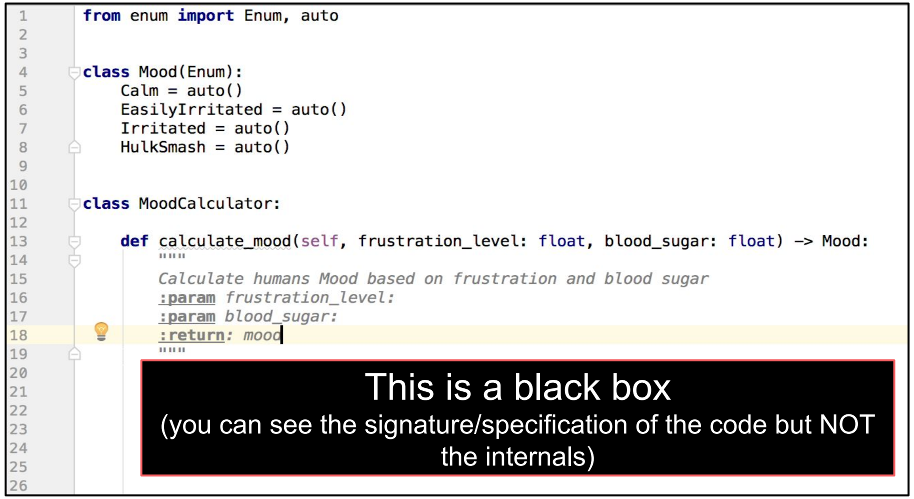
Cheat Sheet

		Am I allowed to execute the code?	
		Static	Dynamic
Box / Perspective	Black	● Specification walk-through	● Equivalence Partitioning ● Boundary Value Analysis
	White	● Code standards ● Code reviews ● Compilers	● Flow Graph ● Coverage (statement, branch, condition) ● Mutation Testing

We will dive into dynamic black and white. Remember the context/framing

Technique 1: Black Box



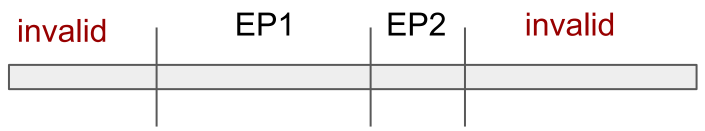

```python
from enum import Enum, auto


class Mood(Enum):
    Calm = auto()
    EasilyIrritated = auto()
    Irritated = auto()
    HulkSmash = auto()


class MoodCalculator:

    def calculate_mood(self, frustration_level: float, blood_sugar: float) -> Mood:
        """
        Calculate humans Mood based on frustration and blood sugar
        :param frustration_level:
        :param blood_sugar:
        :return: mood
        """
```

This is a black box
(you can see the signature/specification of the code but NOT the internals)

Black box testing: techniques to *derive test cases* based on the *interface/ specification* of the code/ system under test

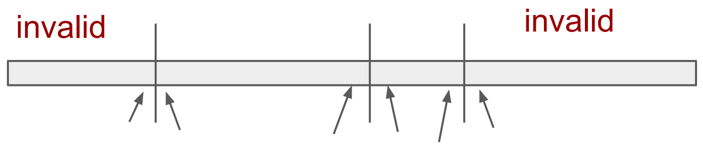
Technique: Black Box EP & BVA

Equivalence Partitioning



- Domain can not be exhaustively explored
- However, behavior is often the same for ranges within the input domain
- *Partition* input domain into ranges of value with similar behavior to reduce number of test cases

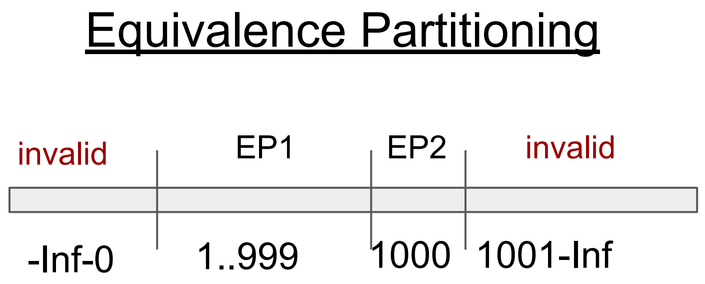
Boundary Value Analysis

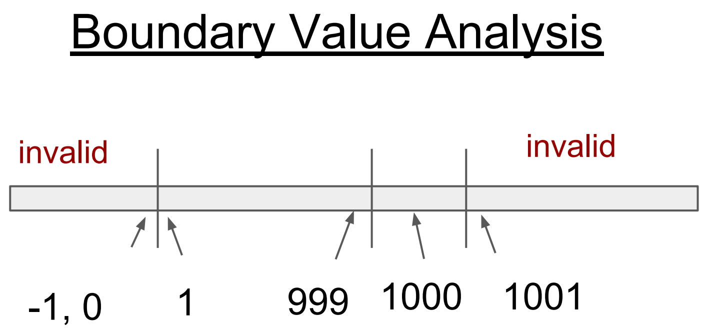


- Amplifying technique is BVA
- Write test cases on the boundaries of equivalence partitions
- Exposes logical and coding errors (classic example is < vs <=)

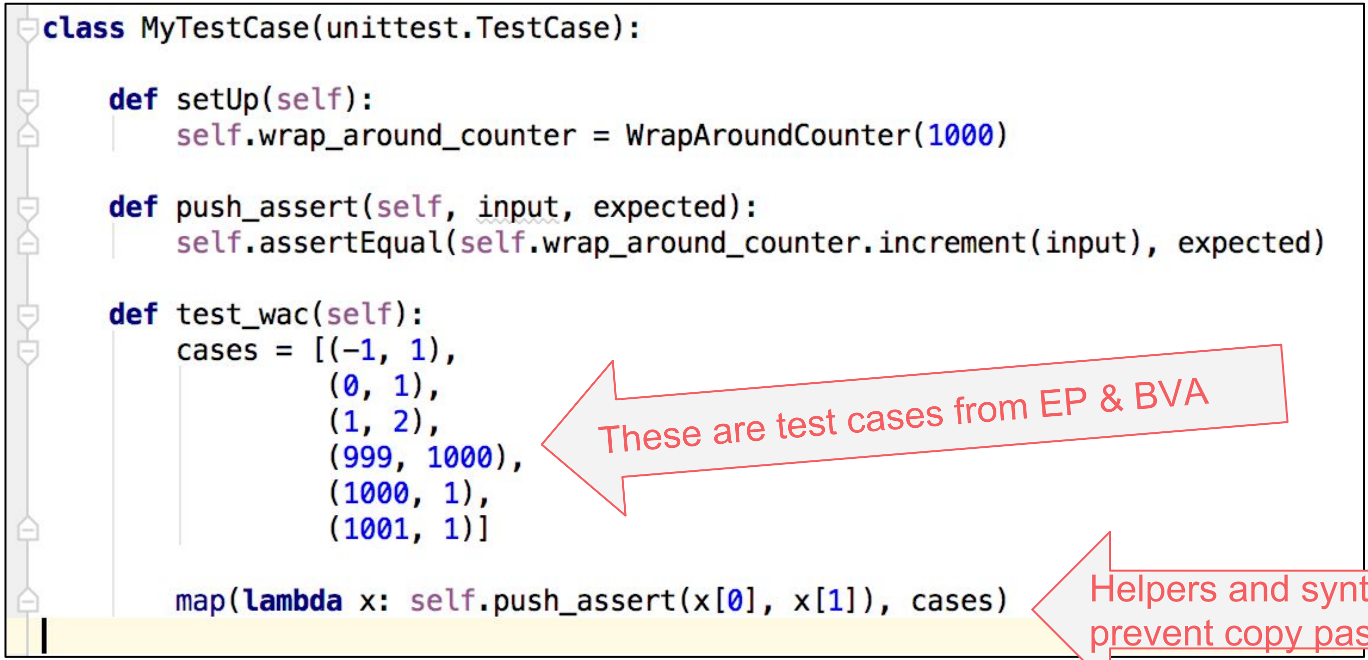
Applied Black Box: I

Applying the techniques to our wrap around counter (max_val 1000)





Applied Black Box: I



```
class MyTestCase(unittest.TestCase):

    def setUp(self):
        self.wrap_around_counter = WrapAroundCounter(1000)

    def push_assert(self, input, expected):
        self.assertEqual(self.wrap_around_counter.increment(input), expected)

    def test_wac(self):
        cases = [(-1, 1),
                 (0, 1),
                 (1, 2),
                 (999, 1000),
                 (1000, 1),
                 (1001, 1)]

        map(lambda x: self.push_assert(x[0], x[1]), cases)
```

These are test cases from EP & BVA

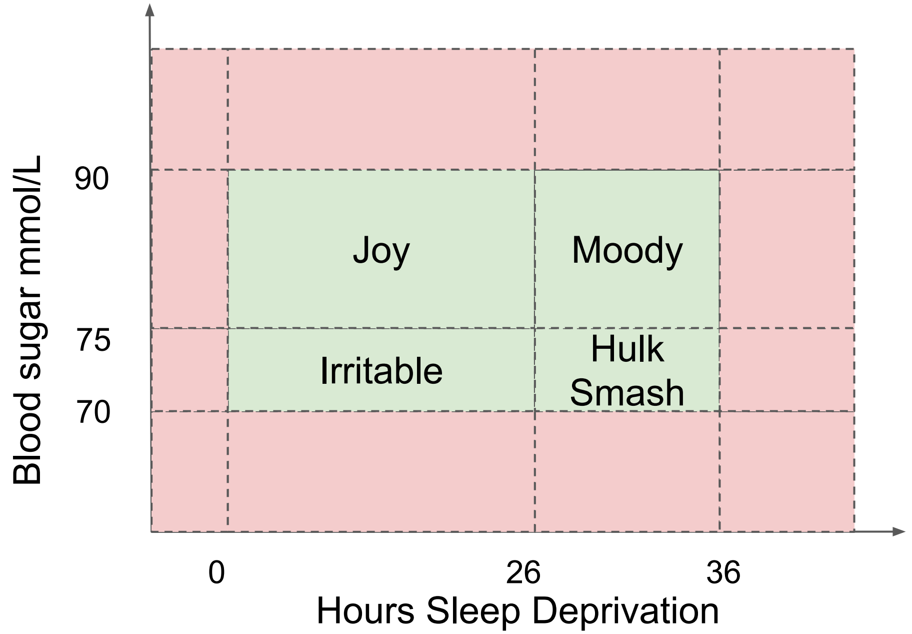
Helpers and syntactic sugar to prevent copy pasting code

Applied Black Box: II

Someone's wife suggested their mood is a simple function of sleep deprivation and blood sugar …..

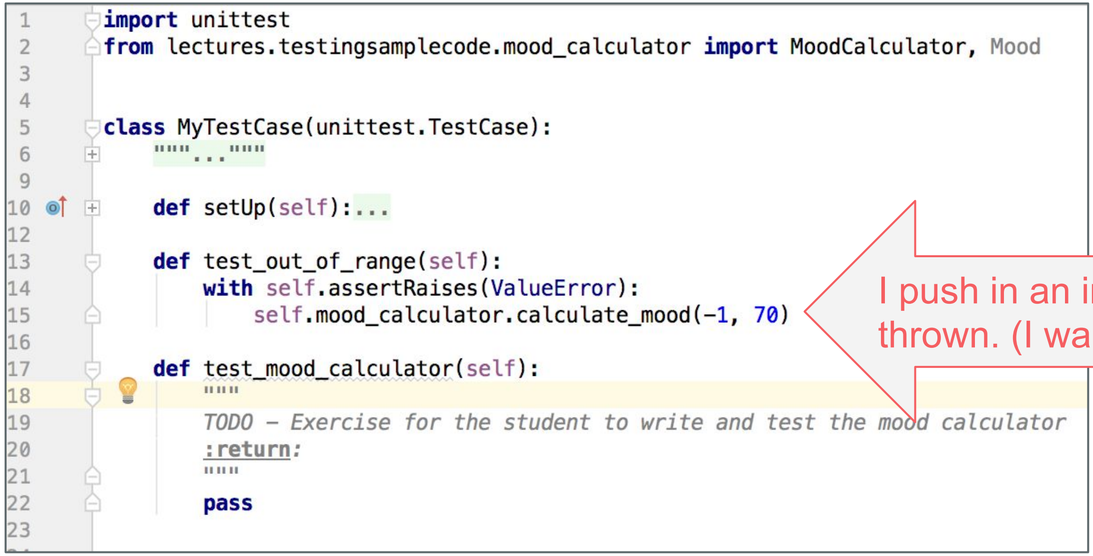
$$\text{m(bloodsugar, sleep\_deprivation)} = \begin{cases} (0 \le \text{SleepDep} \le 26) \ \& \ (70 \le \text{bloodsugar} \le 75) \Rightarrow \text{Irritable} \\ (0 \le \text{SleepDep} \le 26) \ \& \ (75 < \text{bloodsugar} \le 90) \Rightarrow \text{Joy} \\ (26 < \text{SleepDep} \le 36) \ \& \ (70 \le \text{bloodsugar} \le 75) \Rightarrow \text{Moody} \\ (26 \le \text{SleepDep} \le 36) \ \& \ (75 < \text{bloodsugar} \le 90) \Rightarrow \text{Hulk Smash} \\ \text{else: Error} \end{cases}$$

Black Box: EP and BVA



We start to see that the technique can apply in more dimensions

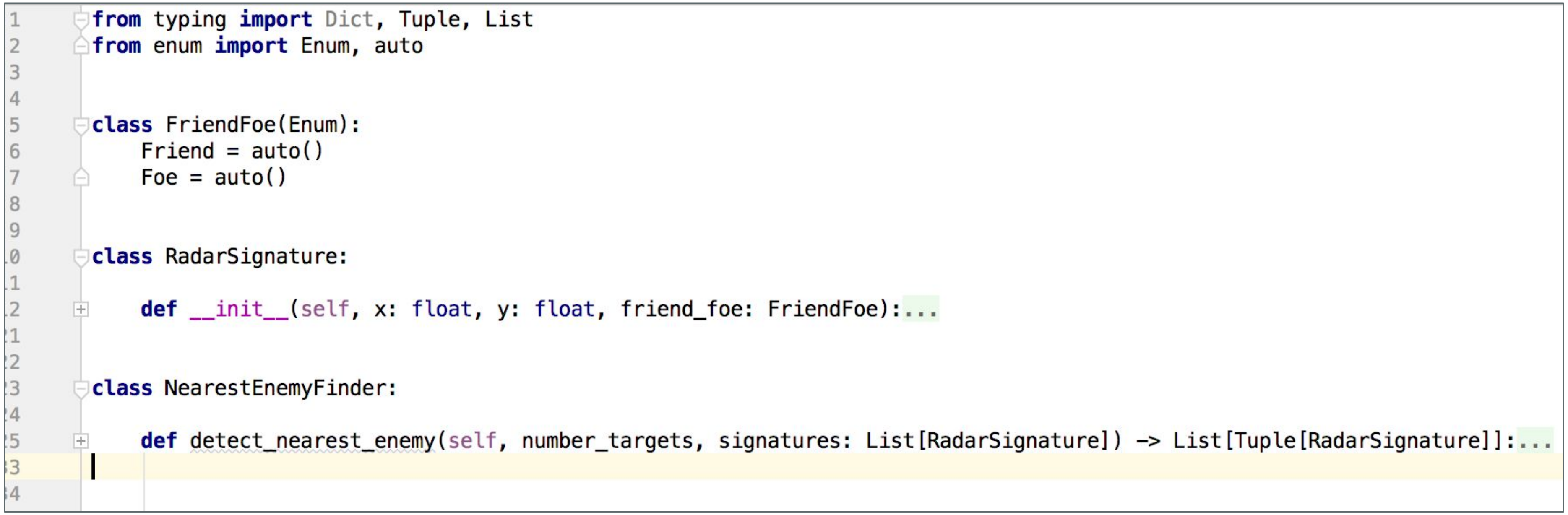
Black Box III: Asserting Exceptions



```
import unittest
from lectures.testingsamplecode.mood_calculator import MoodCalculator, Mood


class MyTestCase(unittest.TestCase):
    """..."""

    def setUp(self):...

    def test_out_of_range(self):
        with self.assertRaises(ValueError):
            self.mood_calculator.calculate_mood(-1, 70)

    def test_mood_calculator(self):
        """
        TODO - Exercise for the student to write and test the mood calculator
        :return:
        """
        pass

```

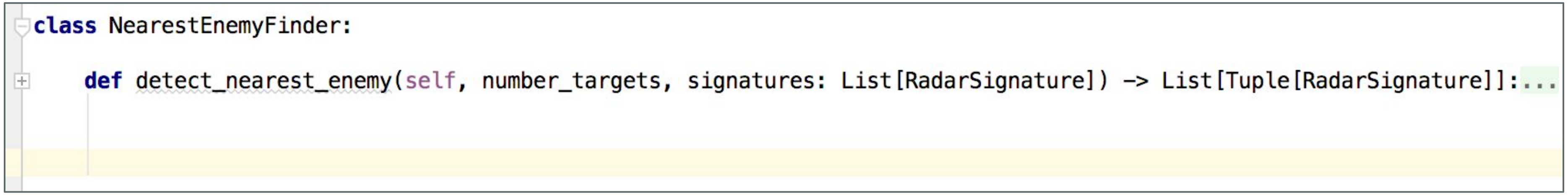
I push in an invalid value and I *assert* a ValueError is thrown. (I want/expect/hope a ValueError to be thrown)

Black Box III: More Complex



```python
from typing import Dict, Tuple, List
from enum import Enum, auto


class FriendFoe(Enum):
    Friend = auto()
    Foe = auto()


class RadarSignature:

    def __init__(self, x: float, y: float, friend_foe: FriendFoe):...


class NearestEnemyFinder:

    def detect_nearest_enemy(self, number_targets, signatures: List[RadarSignature]) -> List[Tuple[RadarSignature]]:...

```

Black Box III: More Complex



```
class NearestEnemyFinder:

    def detect_nearest_enemy(self, number_targets, signatures: List[RadarSignature]) -> List[Tuple[RadarSignature]]:...
```

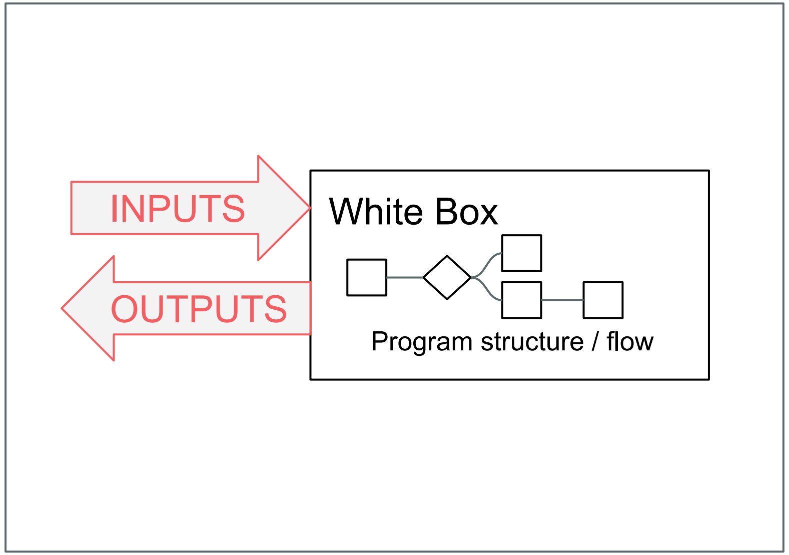
- There input space of the domain is huge (list of planes x coordinates x friend/foe)
- However, your mind should jump to think of EP and BVA
  - 0 planes, 1 plane enemy, 1 plane friend
  - …
  - Do individual values of x, y matter?
  - Does the angle matter? (0, 45, 60, 90, 180, 190, 245, 355, 360?)
  - Or is it the distance? (10, 20, 80, 1000 units away?)
  - Or is it the sequencing of distance (foeA closer, foeA and foeB equal distance)
  - What are the logical equivalence partitions?

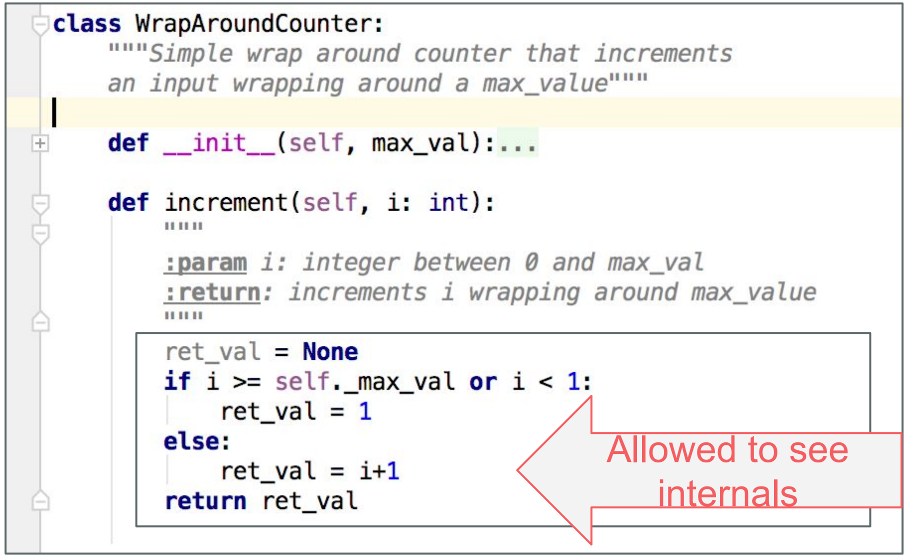
Black Box: IIIIIII

- Okay okay okay. Let’s move on
- Study and complete the code exercises
- Develop *analytical mindset* to see EP and BVA in problems

<Cough>I love this topic for exam questions ……</Cough>

Technique II: White Box

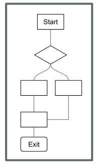




```python
class WrapAroundCounter:
    """Simple wrap around counter that increments
    an input wrapping around a max_value"""

    def __init__(self, max_val):...

    def increment(self, i: int):
        """
        :param i: integer between 0 and max_val
        :return: increments i wrapping around max_value
        """
        ret_val = None
        if i >= self._max_val or i < 1:
            ret_val = 1
        else:
            ret_val = i+1
        return ret_val
```

White Box: We are now allowed to see the internals of code.
Opening *internals* gives the opportunity for new ways of generating or assessing test suite

White Box: Program as a Graph

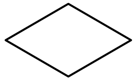


- Consider your program as a graph

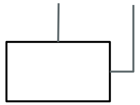
- Inputs arrive at the top

- Statement blocks contain functionality to execute

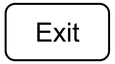
(no control stmts)



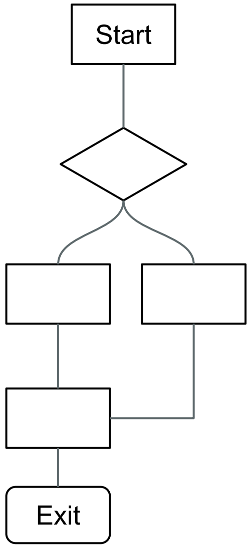
- Control flow statements (if, switch, try catch)



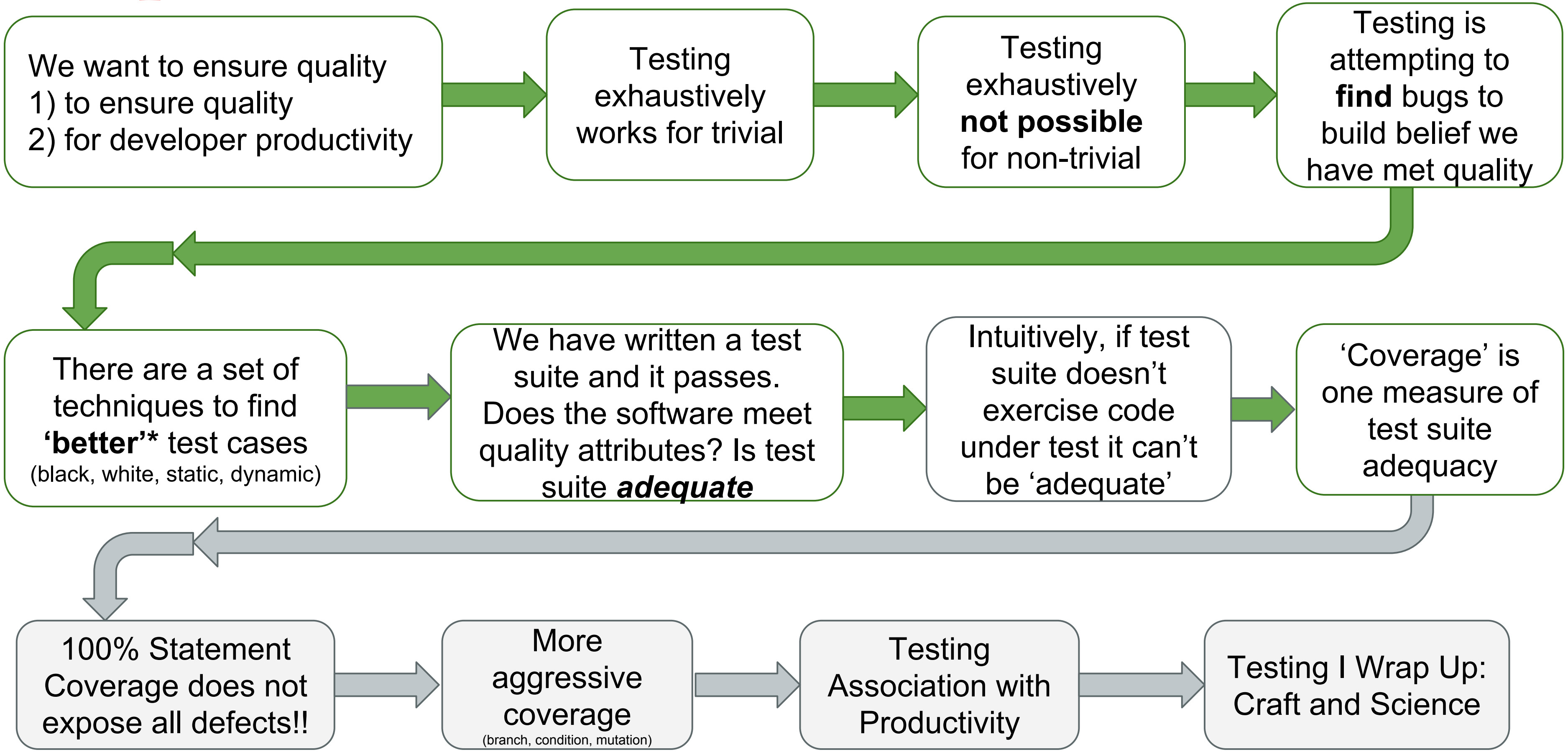
- Paths may merge



- Exit node returns value from code



Recap



We want to ensure quality
1) to ensure quality
2) for developer productivity

Testing exhaustively works for trivial

Testing exhaustively **not possible** for non-trivial

Testing is attempting to **find** bugs to build belief we have met quality

There are a set of techniques to find **‘better’*** test cases
(black, white, static, dynamic)

We have written a test suite and it passes. Does the software meet quality attributes? Is test suite ***adequate***

Intuitively, if test suite doesn’t exercise code under test it can’t be ‘adequate’

‘Coverage’ is one measure of test suite adequacy

100% Statement Coverage does not expose all defects!!

More aggressive coverage
(branch, condition, mutation)

Testing Association with Productivity

Testing I Wrap Up: Craft and Science

* better=higher probability of finding defects

Assessing Test Suite Effectiveness

1. Established we can not exhaustively test
2. We have defined a test suite
3. That test suite passed
4. Should we be confident?

NOT SURE IF QUALITY IS REALLY GOOD
OR TESTING WAS REALLY LOUSY
quickmeme.com

Test Suite Effectiveness: How effective is my test *suite* at finding faults?

The more effective our test suite the more confidence/quality we have assured

Technique III: Coverage

Question 1: Recall our test cases *exercise* our code under test

- What does it tell us if our test cases only exercised 20% of our code under test?
- (Would you fly on a plane where the test cases only exercised 20% of the code?)
  (Congrats! You now understand the basics of white box testing)

Question 2: If you knew how the code worked could you generate new test case?

Assessing Test Suite Effectiveness: Coverage

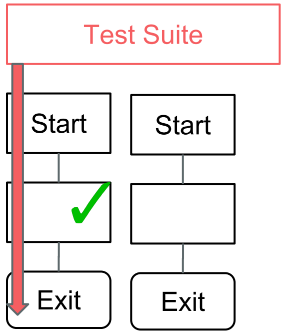
Coverage is the most common measure of test suite effectiveness in industry

Coverage measures how ‘much’ of the cost under test is exercised by the test suite

Coverage is a ‘white box’ technique as we are using knowledge of code internals

We can use control flow graphs to understand code coverage

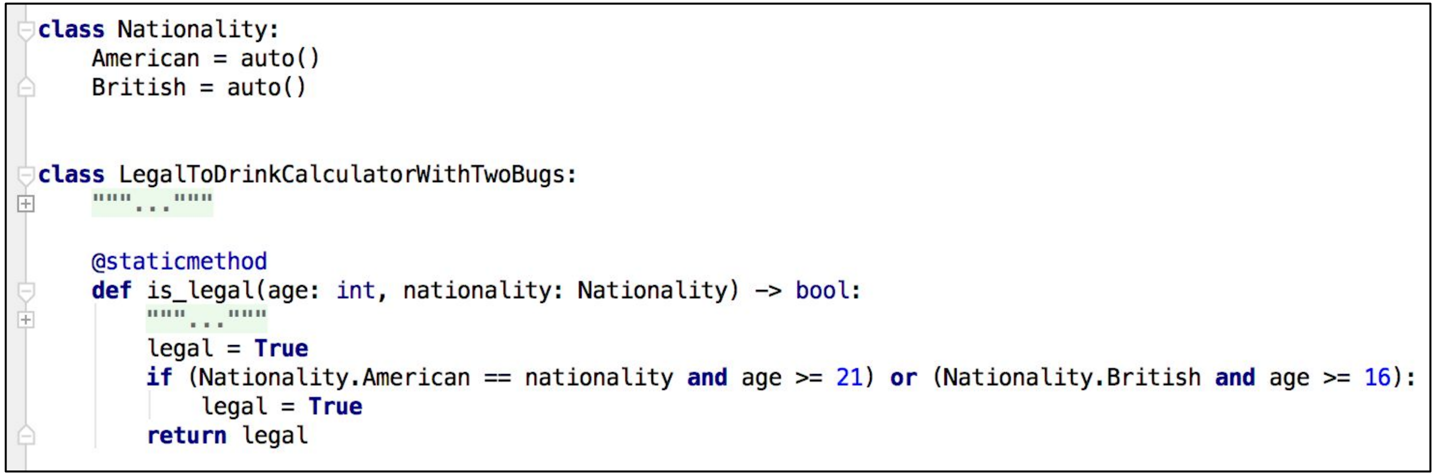
Coverage Measure I: Statement



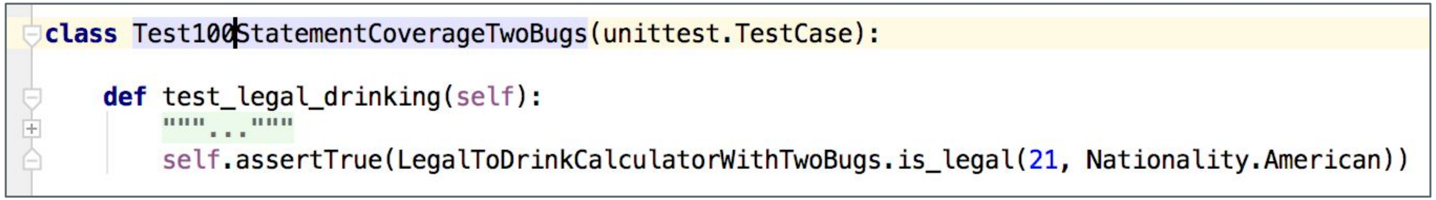
- **Statement coverage:** asks whether every *statement* in the code under test has been exercised by the test suite
    - Note: Within a codebase this is all classes/methods
    - As we build up theory we are discussing only methods

- Intuitively, this make sense. How can we hope to have proven that the code works if we have not even run it as part of the test suite?

Statement Coverage



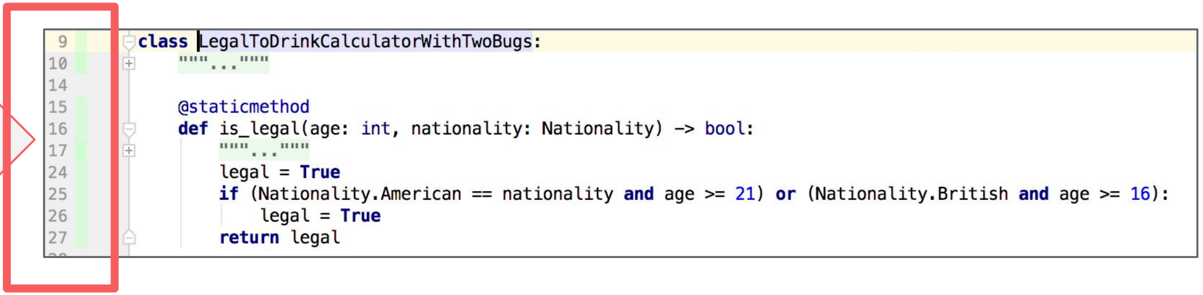
```python
class Nationality:
    American = auto()
    British = auto()


class LegalToDrinkCalculatorWithTwoBugs:
    """..."""

    @staticmethod
    def is_legal(age: int, nationality: Nationality) -> bool:
        """..."""
        legal = True
        if (Nationality.American == nationality and age >= 21) or (Nationality.British and age >= 16):
            legal = True
        return legal
```



```python
class Test100StatementCoverageTwoBugs(unittest.TestCase):

    def test_legal_drinking(self):
        """..."""
        self.assertTrue(LegalToDrinkCalculatorWithTwoBugs.is_legal(21, Nationality.American))
```

Statement Coverage

Coverage.py keeps track of which statements are executed during test suite (green shows stmt executed)

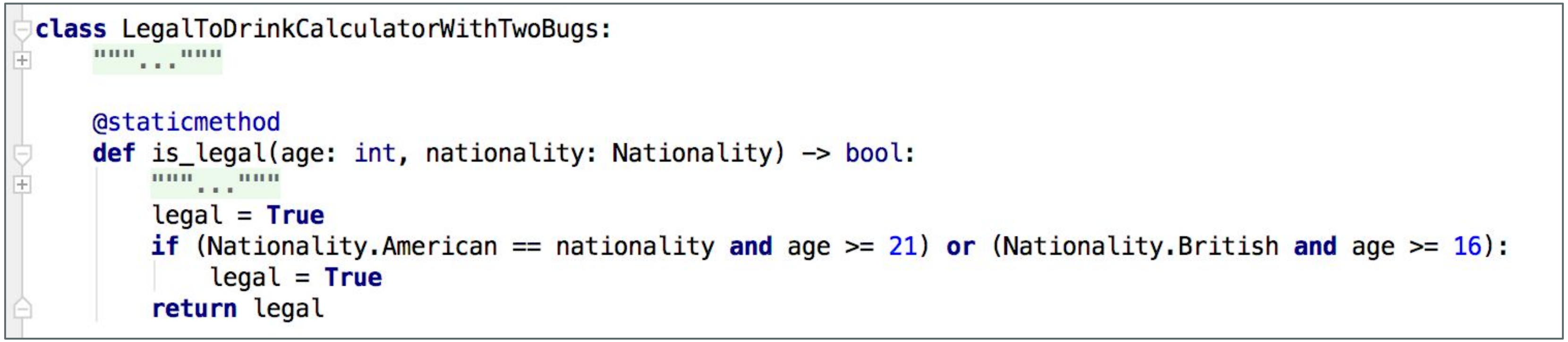


```
class LegalToDrinkCalculatorWithTwoBugs:
    """..."""

    @staticmethod
    def is_legal(age: int, nationality: Nationality) -> bool:
        """..."""
        legal = True
        if (Nationality.American == nationality and age >= 21) or (Nationality.British and age >= 16):
            legal = True
        return legal
```

100% Statement Coverage != Correct

- We created a simple program.
- Wrote a single test case
    - It passed
    - We achieved 100% statement coverage
    - (I wrote it in a way that even with an ‘if’ statement we did cover 100% of lines)

BUT there are ***STILL TWO*** bugs in the program which the unit test ***DID NOT EXPOSE***

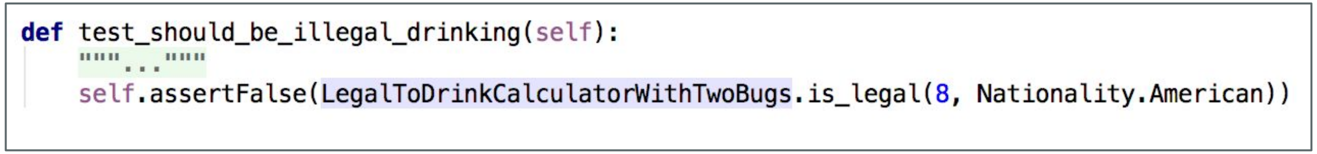


```
class LegalToDrinkCalculatorWithTwoBugs:
    """..."""

    @staticmethod
    def is_legal(age: int, nationality: Nationality) -> bool:
        """..."""
        legal = True
        if (Nationality.American == nationality and age >= 21) or (Nationality.British and age >= 16):
            legal = True
        return legal
```

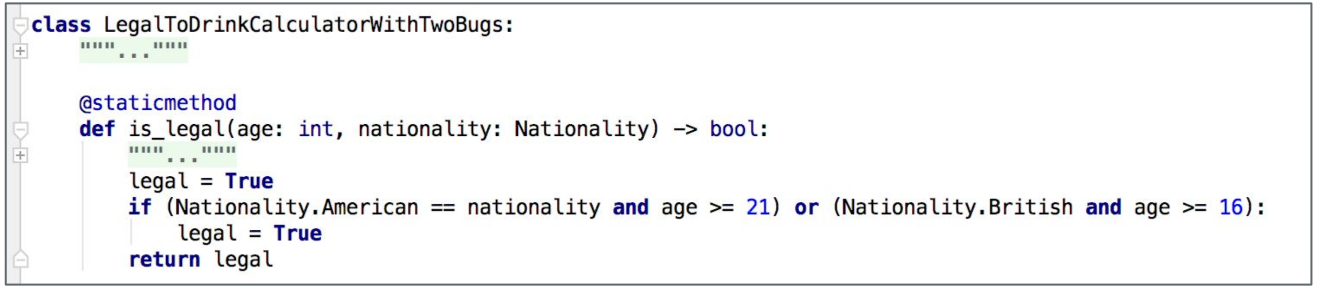
Statement Coverage Limitation

Remember: Before we added this test case we had achieved 100% statement coverage

Does this test case ‘pass’ (in the sense is_legal returns **False** and assertFalse(**False**) passes)?



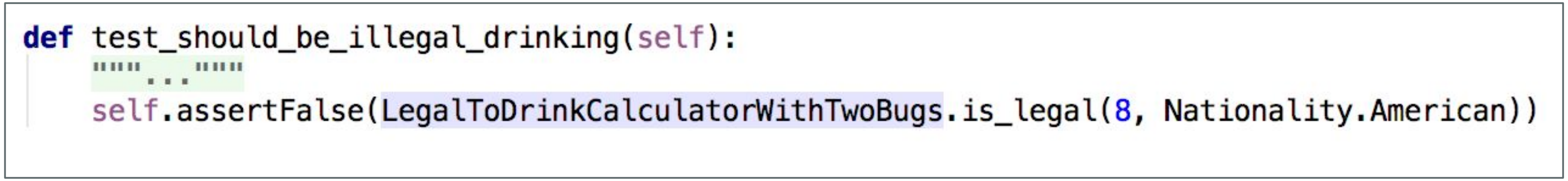
```
def test_should_be_illegal_drinking(self):
    """..."""
    self.assertFalse(LegalToDrinkCalculatorWithTwoBugs.is_legal(8, Nationality.American))
```



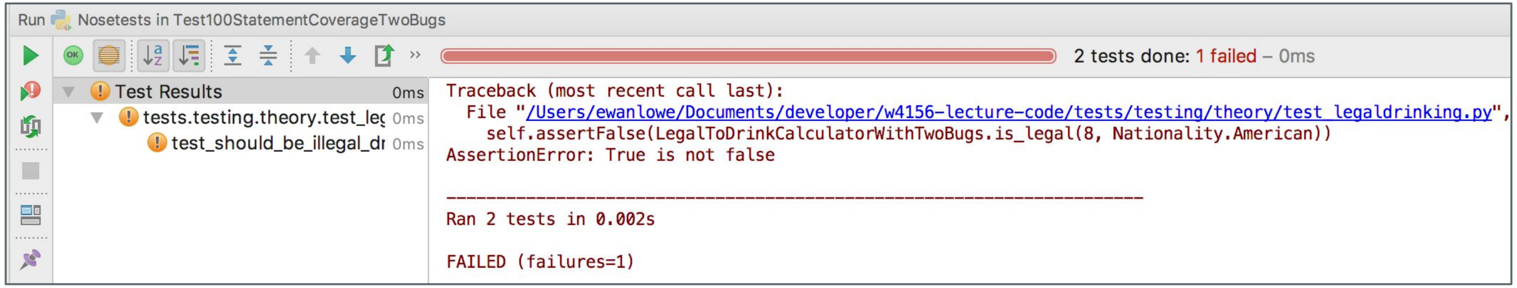
```
class LegalToDrinkCalculatorWithTwoBugs:
    """..."""

    @staticmethod
    def is_legal(age: int, nationality: Nationality) -> bool:
        """..."""
        legal = True
        if (Nationality.American == nationality and age >= 21) or (Nationality.British and age >= 16):
            legal = True
        return legal
```

Statement Coverage Limitation

We had previously ***achieved*** 100% coverage (and it passed)
And yet! There was an unexposed bug.
This test case exposes that bug.



```
def test_should_be_illegal_drinking(self):
    """..."""
    self.assertFalse(LegalToDrinkCalculatorWithTwoBugs.is_legal(8, Nationality.American))
```



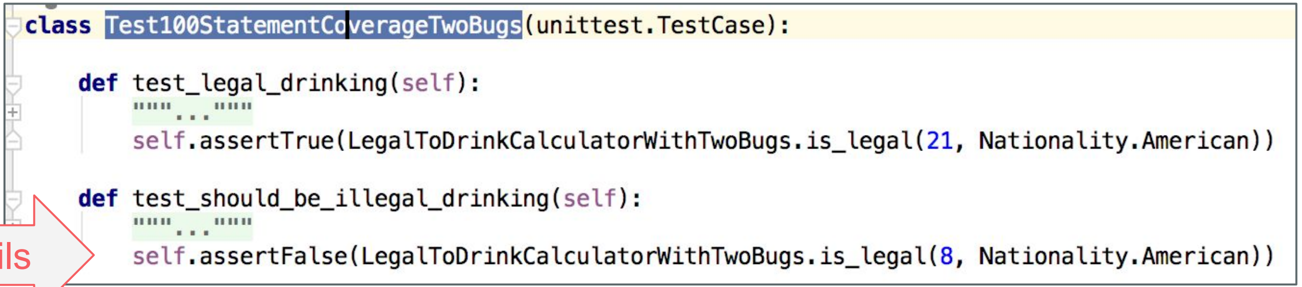
Run Nosetests in Test100StatementCoverageTwoBugs

2 tests done: 1 failed – 0ms

Test Results	0ms
tests.testing.theory.test_leg	0ms
test_should_be_illegal_dr	0ms

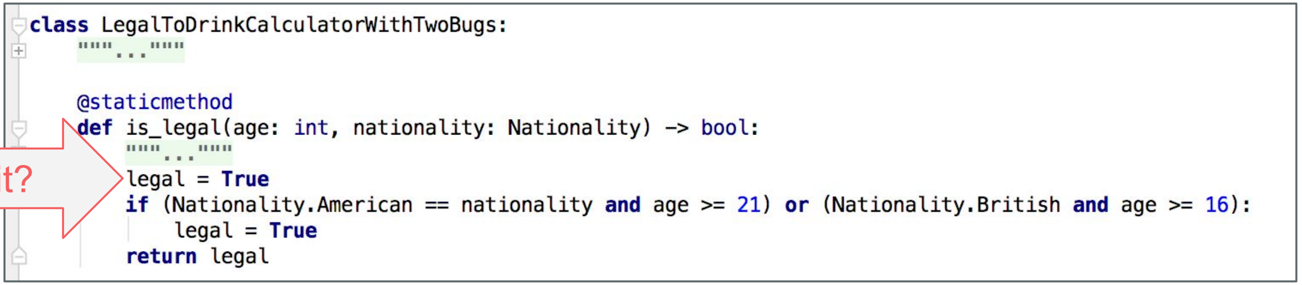
```
Traceback (most recent call last):
  File "/Users/ewanlowe/Documents/developer/w4156-lecture-code/tests/testing/theory/test_legaldrinking.py",
    self.assertFalse(LegalToDrinkCalculatorWithTwoBugs.is_legal(8, Nationality.American))
AssertionError: True is not false

----------------------------------------------------------------------
Ran 2 tests in 0.002s

FAILED (failures=1)
```

It Failed (our code has a bug)



```
class Test100StatementCoverageTwoBugs(unittest.TestCase):

    def test_legal_drinking(self):
        """..."""
        self.assertTrue(LegalToDrinkCalculatorWithTwoBugs.is_legal(21, Nationality.American))

    def test_should_be_illegal_drinking(self):
        """..."""
        self.assertFalse(LegalToDrinkCalculatorWithTwoBugs.is_legal(8, Nationality.American))
```

Test case fails

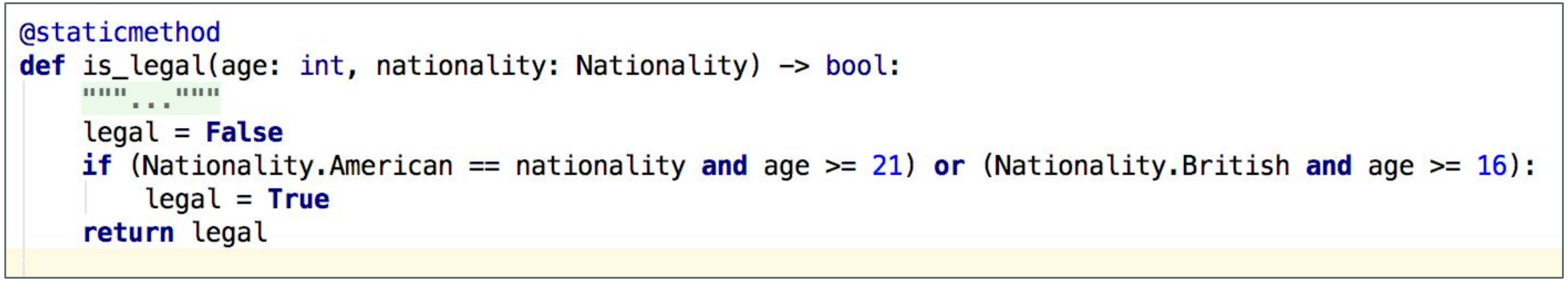


```
class LegalToDrinkCalculatorWithTwoBugs:
    """..."""

    @staticmethod
    def is_legal(age: int, nationality: Nationality) -> bool:
        """..."""
        legal = True
        if (Nationality.American == nationality and age >= 21) or (Nationality.British and age >= 16):
            legal = True
        return legal
```

Culprit?

Limitations of Statement Coverage

- Our initial single test case was is_legal(21, American)
- However, one of the bugs is the initial value of legal is set to **True**
- Uur initial test case we ***did not*** execute path where ‘if’ evaluated **False**



```python
@staticmethod
def is_legal(age: int, nationality: Nationality) -> bool:
    """..."""
    legal = False
    if (Nationality.American == nationality and age >= 21) or (Nationality.British and age >= 16):
        legal = True
    return legal
```

Walk through the example and run the tests until you accept with 100% statement coverage there was still a latent defect

100% Statement Coverage != Correct

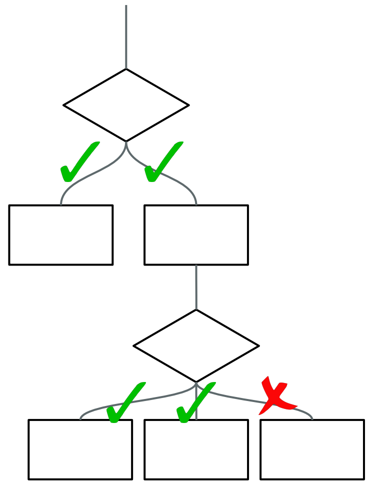
- The finding that 100% statement coverage does not guarantee correctness is important
- We need to understand why this is?
- Program is a state space of inputs and transitions
  - There are a set of possible paths through the program
  - 100% statement coverage ***does not mean*** we have exercised all transitions
  - Defects can remain undetected in other paths

(Even ***without*** an ‘if’ statement I could have introduced a / 0 error that would not work for certain inputs. Alternatively, I could have two values that only certain combinations caused an overflow etc. Two more scenarios where 100% statement coverage does not mean correctness)

Coverage Measure II: Branch Coverage

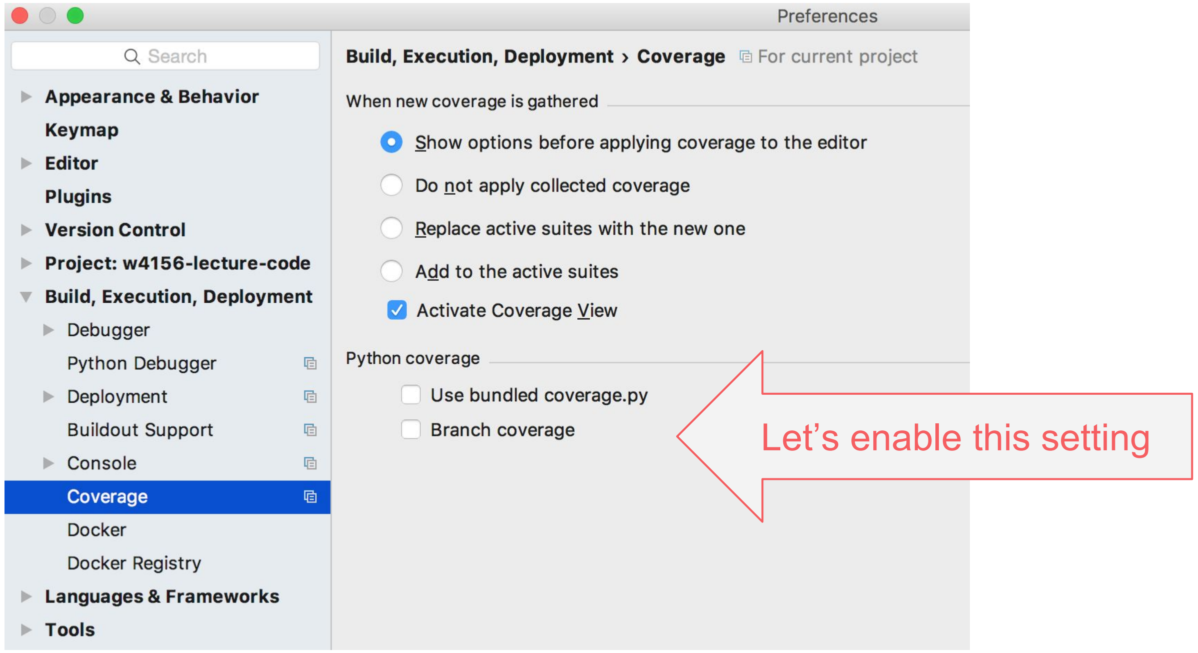
Branch coverage: asks what % of possible paths from control statement were exercised by the test suite.

i.e. : were all branch outcomes executed



Tooling: Enabling Branch Coverage

For some reason PyCharm does not ship with Branch Coverage enabled by default



Preferences

Search

- Appearance & Behavior
- Keymap
- Editor
- Plugins
- Version Control
- Project: w4156-lecture-code
- Build, Execution, Deployment
  - Debugger
  - Python Debugger
  - Deployment
  - Buildout Support
  - Console
  - Coverage
  - Docker
  - Docker Registry
- Languages & Frameworks
- Tools

Build, Execution, Deployment › Coverage   For current project

When new coverage is gathered

- [✓ Show options before applying coverage to the editor]
- Do not apply collected coverage
- Replace active suites with the new one
- Add to the active suites
- [✓] Activate Coverage View

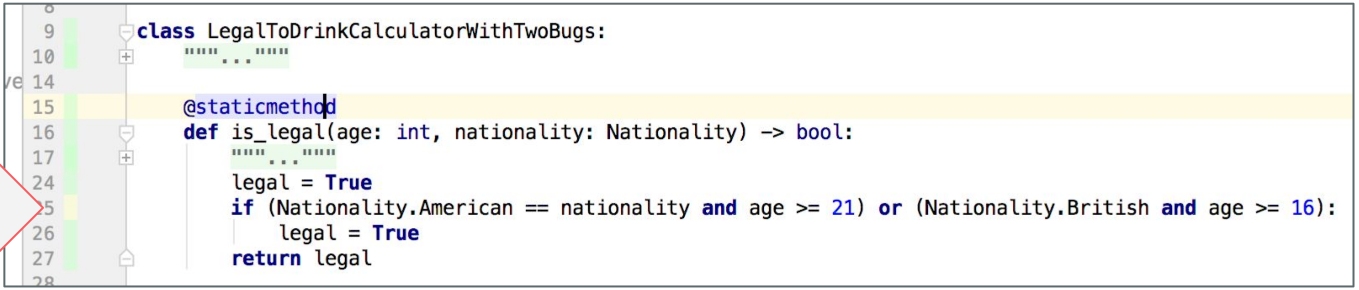
Python coverage

- [ ] Use bundled coverage.py
- [ ] Branch coverage

Let's enable this setting

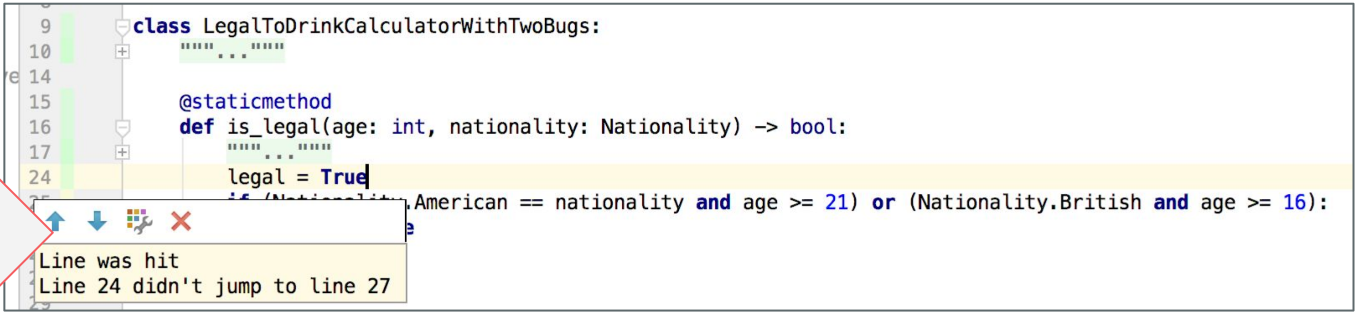
Rerunning with Branch Coverage

Rerunning the test_legal_drinking **single test case** (21, American), with branch coverage enabled
We see that while we hit stmt coverage we did ***not*** hit branch coverage



```
class LegalToDrinkCalculatorWithTwoBugs:
    """..."""

    @staticmethod
    def is_legal(age: int, nationality: Nationality) -> bool:
        """..."""
        legal = True
        if (Nationality.American == nationality and age >= 21) or (Nationality.British and age >= 16):
            legal = True
        return legal
```

This orange says we didn't hit branch coverage

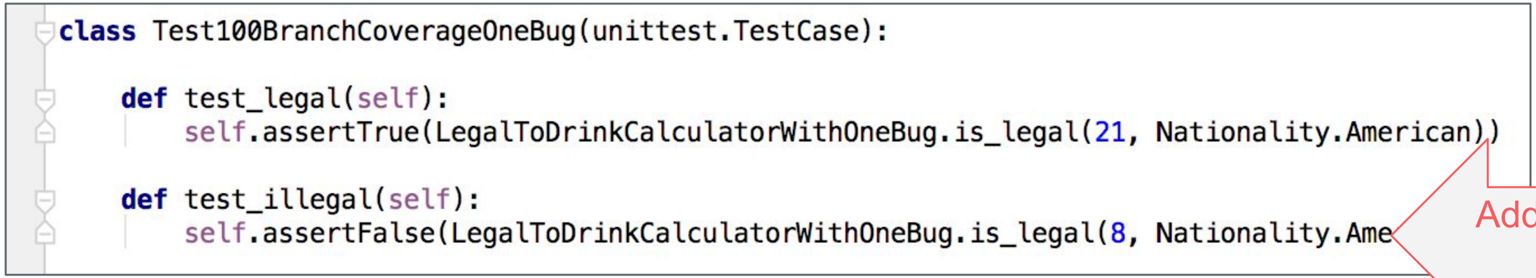


```
class LegalToDrinkCalculatorWithTwoBugs:
    """..."""

    @staticmethod
    def is_legal(age: int, nationality: Nationality) -> bool:
        """..."""
        legal = True
        if (Nationality.American == nationality and age >= 21) or (Nationality.British and age >= 16):
```

```
Line was hit
Line 24 didn't jump to line 27
```

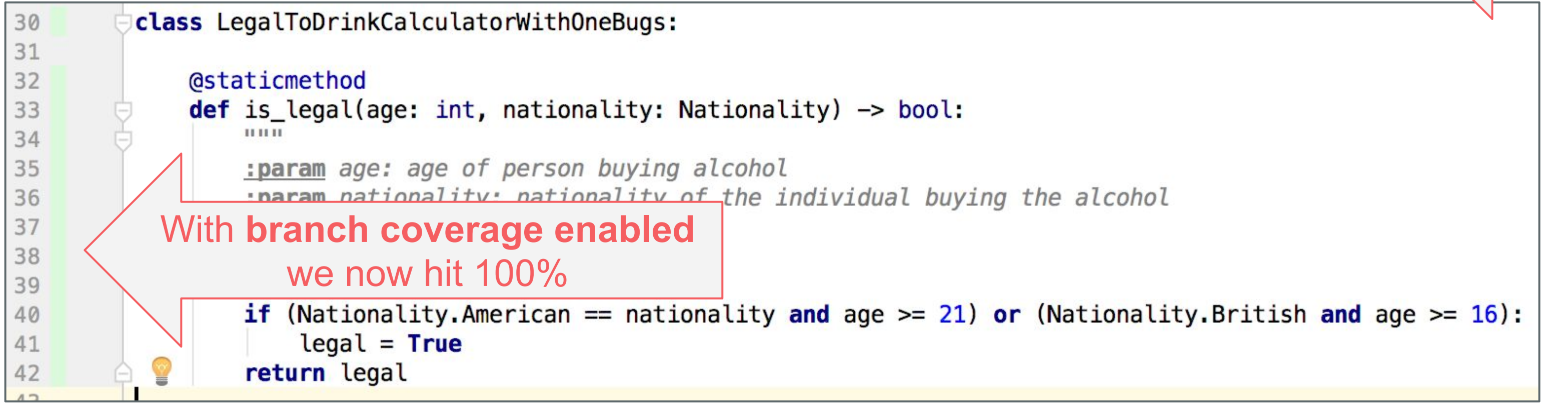
We even get told why!

100% Branch Coverage



```
class Test100BranchCoverageOneBug(unittest.TestCase):

    def test_legal(self):
        self.assertTrue(LegalToDrinkCalculatorWithOneBug.is_legal(21, Nationality.American))

    def test_illegal(self):
        self.assertFalse(LegalToDrinkCalculatorWithOneBug.is_legal(8, Nationality.Ame
```

Adding a second test case



```
class LegalToDrinkCalculatorWithOneBugs:

    @staticmethod
    def is_legal(age: int, nationality: Nationality) -> bool:
        """
        :param age: age of person buying alcohol
        :param nationality: nationality of the individual buying the alcohol




        if (Nationality.American == nationality and age >= 21) or (Nationality.British and age >= 16):
            legal = True
        return legal
```

With **branch coverage enabled** we now hit 100%

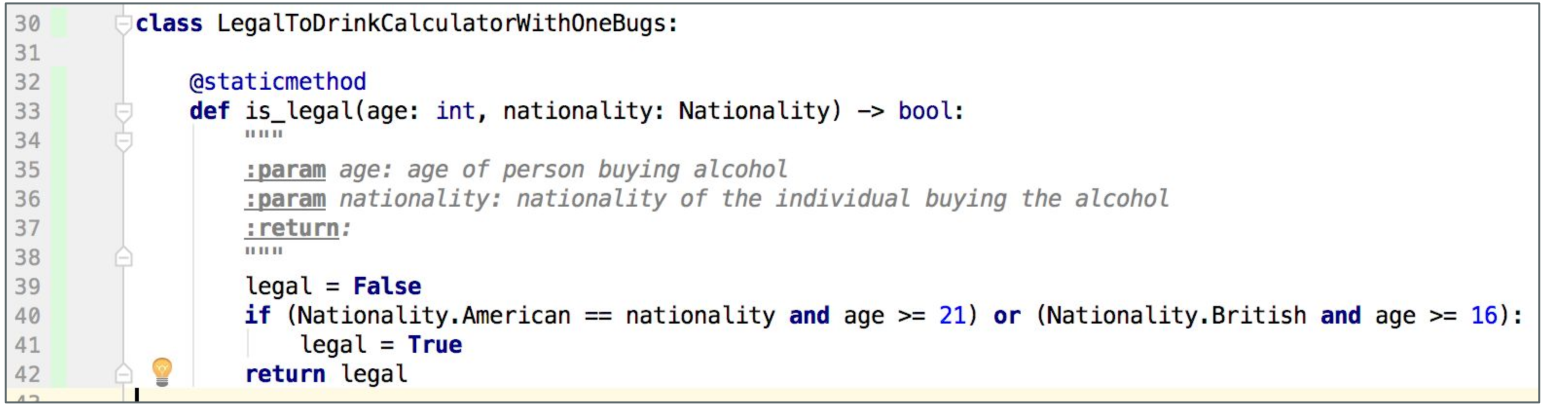
Recap

1. We wrote some code
2. We wrote a test and achieved 100% statement coverage
3. But there was still a bug
4. We then discovered branch coverage and wrote tests to achieve 100% branch coverage
5. That exposed one of the latent bugs

Are we confident in the code? Are there still latent bugs?

Final Bug

The legal drinking age in the UK is 18 not 16!!!

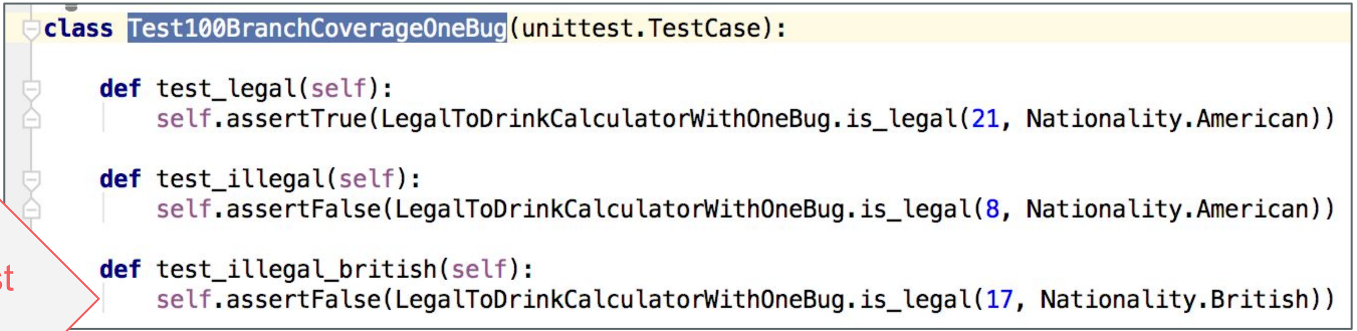


```
class LegalToDrinkCalculatorWithOneBugs:

    @staticmethod
    def is_legal(age: int, nationality: Nationality) -> bool:
        """
        :param age: age of person buying alcohol
        :param nationality: nationality of the individual buying the alcohol
        :return:
        """
        legal = False
        if (Nationality.American == nationality and age >= 21) or (Nationality.British and age >= 16):
            legal = True
        return legal
```

Remember, at this stage we have 100% statement and 100% branch coverage and we have still not exposed this bug

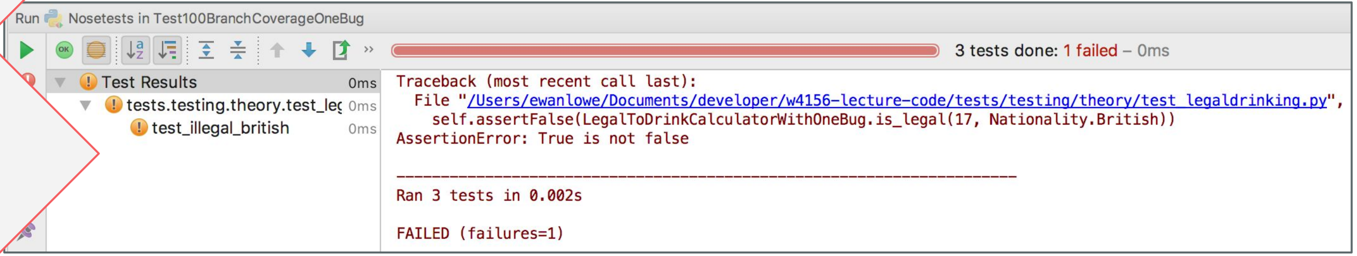
Limitations of Branch Coverage

However, although we have now hit 100% <u>branch</u> coverage there is still a bug ...



```
class Test100BranchCoverageOneBug(unittest.TestCase):

    def test_legal(self):
        self.assertTrue(LegalToDrinkCalculatorWithOneBug.is_legal(21, Nationality.American))

    def test_illegal(self):
        self.assertFalse(LegalToDrinkCalculatorWithOneBug.is_legal(8, Nationality.American))

    def test_illegal_british(self):
        self.assertFalse(LegalToDrinkCalculatorWithOneBug.is_legal(17, Nationality.British))
```

Added this test case

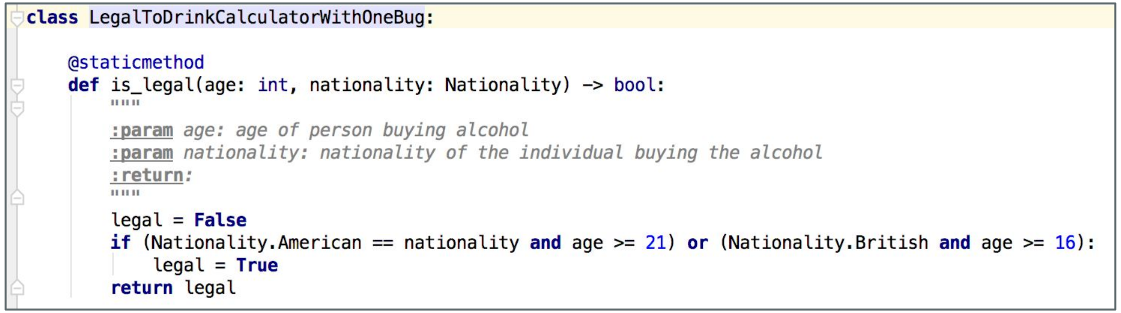

Run Nosetests in Test100BranchCoverageOneBug
3 tests done: 1 failed – 0ms
Test Results 0ms
tests.testing.theory.test_leg 0ms
test_illegal_british 0ms

```
Traceback (most recent call last):
  File "/Users/ewanlowe/Documents/developer/w4156-lecture-code/tests/testing/theory/test_legaldrinking.py",
    self.assertFalse(LegalToDrinkCalculatorWithOneBug.is_legal(17, Nationality.British))
AssertionError: True is not false

----------------------------------------------------------------------
Ran 3 tests in 0.002s

FAILED (failures=1)
```

And it fails

Limitations of Branch Coverage

Again, branch detection has a gap/flaw. We executed every statement. We also constructed test cases such that the branch evaluated to both **True** and **False.**



```
class LegalToDrinkCalculatorWithOneBug:

    @staticmethod
    def is_legal(age: int, nationality: Nationality) -> bool:
        """
        :param age: age of person buying alcohol
        :param nationality: nationality of the individual buying the alcohol
        :return:
        """
        legal = False
        if (Nationality.American == nationality and age >= 21) or (Nationality.British and age >= 16):
            legal = True
        return legal
```

- However, the ‘if’ has **two conditions (clauses)**
- If the first condition == True the whole branch is True (and second statement is not evaluated)
- If the first condition == False the second statement is evaluated. In our test cases it was always False
- The British legal drinking age is 18

Even with 100% branch coverage we can have undetected defects

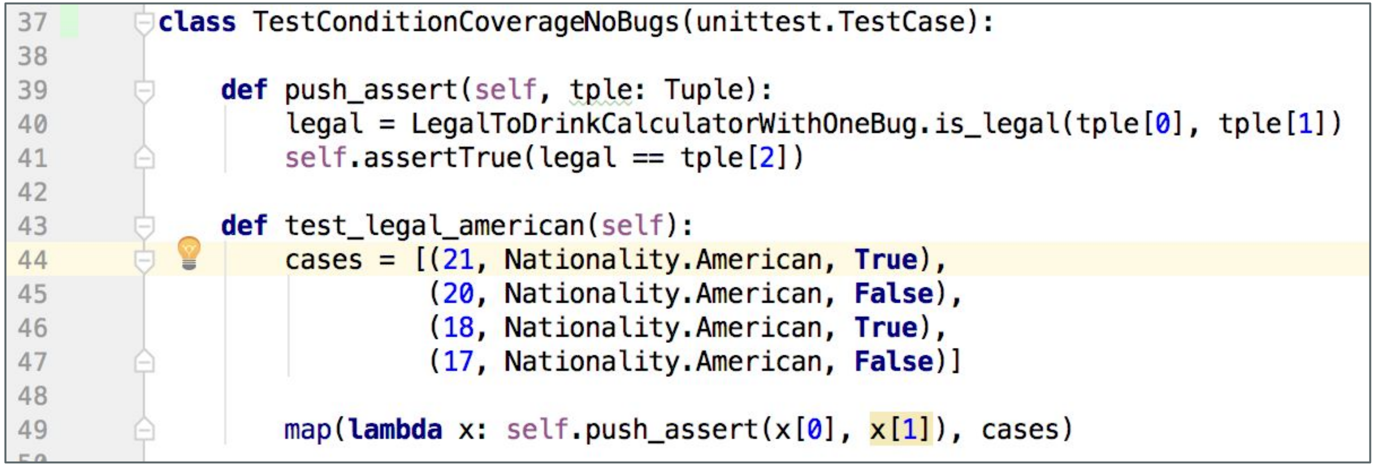
Condition Coverage

Even with 100% branch coverage we can have undetected defects

(You guessed it)

Condition coverage: asks what whether each *conditional* has evaluated to both **True** and **False**

Condition Coverage (+ Sugar)



```
class TestConditionCoverageNoBugs(unittest.TestCase):

    def push_assert(self, tple: Tuple):
        legal = LegalToDrinkCalculatorWithOneBug.is_legal(tple[0], tple[1])
        self.assertTrue(legal == tple[2])

    def test_legal_american(self):
        cases = [(21, Nationality.American, True),
                 (20, Nationality.American, False),
                 (18, Nationality.American, True),
                 (17, Nationality.American, False)]

        map(lambda x: self.push_assert(x[0], x[1]), cases)
```

These test cases hit statement, branch and condition coverage

Summary: Coverage Measures

Coverage Measure	Description
Statement	All statements executed
Branch	Every branch evaluated T and F
Condition	Every conditional evaluated T and F

Summary: Coverage Measures

We can now answer the original question we posed ourselves.
Fry was not sure because there was two options:

NOT SURE IF QUALITY IS REALLY GOOD
OR TESTING WAS REALLY LOUSY
quickmeme.com

		Test Suite Adequacy	
		Rigorous (how to define?)	Lousy (how to define?)
Test Suite	Passes	What do we think of quality and defects?	What do we think of quality and defects?
	Fails	At least one bug	

Summary: Coverage Measures

Rigour defined by higher % coverage and more aggressive measures (statement, branch, condition)

		Test Suite Adequacy	
		Rigorous **‘Good Coverage’**	Lousy **‘Poor Coverage’**
Test Suite	Passes	**Higher confidence in quality Lower defects likely**	**Lower confidence in quality. Higher defects likely**
	Fails	At least one bug	

Do not be confident in your code if your tests pass unless they are rigorous

Why Test II

On the train from NY to Philly and got into conversation with a entrepreneur who dropped of college build an app for Veterinarians* (he was reading the lean startup). He was super excited and talking about his company. But as we got deeper into the conversation he revealed a problem that had been stressing him.

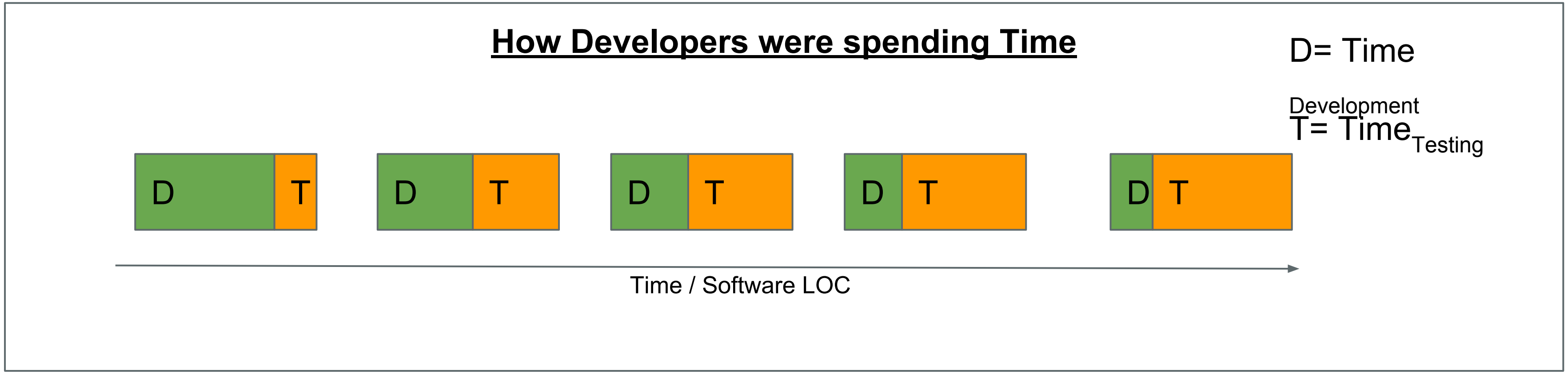
- As the company grew customers were asking for features but his team could not build and release new features without breaking existing functionality.
- This created a dilemma: customers wanted new features but he couldn’t release new features without breaking existing functionality (and upsetting the same customers!).
- The overall pace of development was also slowing!
- Customers were getting unhappy …...

Why was this happening?

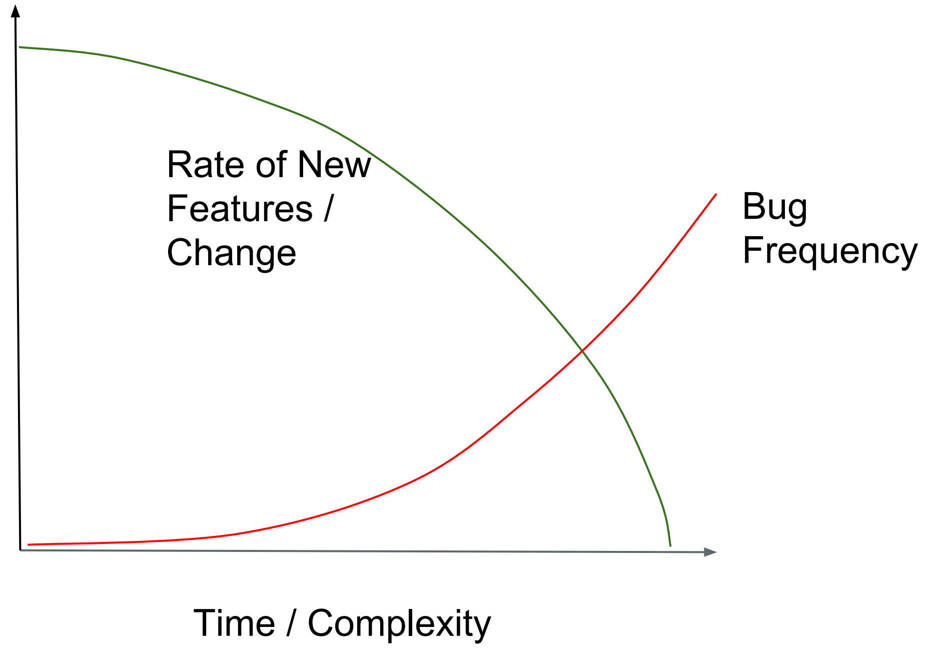
* industry changed to protect the innocent / company is google-able

What was going on?

1. As the complexity of the product grew there were more features / lines of code.
2. Developers were ***manually*** testing the software (click, click, check, etc)
3. Testing was ***repeated*** each release (they re-executed the same test cases each month)
4. As complexity grew they spent more time ***testing*** vs ***developing*** new features
5. Manually testing is also ***error prone*** so bugs leaked through
6. Furthermore, when bugs occurred they were ***hard to trace***



Poor Testing Practices vs Complexity (Ewww!)



Remember: software is the codification of a business (process, rules, etc).
If you can't change the software you can't change the business. Over time someone will do what you wanted to

Differential Diagnosis and Treatment

Diagnosis

- No *automated* testing / generally poor testing practices
- Reduced developer productivity
- Only going to get *worse* (sometimes called the ‘death spiral of IT’)

Treatment Plan

The hard fact is there is technical *debt*

(it will take time to build automated tests and generally you can’t stop for a month)

- Change team *culture* to value automated testing (and writing ‘good’ test cases)
- Introduce *incrementally* based on highest risk / where are bugs appearing and most damaging?
- Repeat
- Climb out of the hole slowly
- N months time this problem *could* be conquered

  (actually, this current problem will be conquered but there will be a newer problem which is testing more complex product and maintaining velocity)

LPT / Takeaway / Rant

Question 1: If we implemented the techniques does it give us significantly more confidence in correctness?
Question 2: How long did it take us to think of the scenarios and write the tests?

Rant: You will often hear “testing takes too long or is too difficult”.

- ***Some*** developers don’t know/care to write good quality/effective tests
  - They write ‘random’ test cases. These test cases ***take time to write.***
  - However, ***we know*** these test cases have a ***low probability*** of exposing defects
  - (They also write code that is difficult to test - we will cover that later)
  - They ***resist testing*** because <u>***they do***</u> get a poor return on time
  - The consequence is they will ship ***lower quality*** and spend ***more time*** fixing bugs
- However, <u>***we***</u> can ***apply techniques*** quickly:
  - We will then write tests with ***high probability*** of exposing defects
  - We will have an excellent return on our time/investment in testing
  - We will ship ***higher quality*** and spend ***more time writing new functionality*** vs fixing bugs
- But - as always - be pragmatic. Balance design impact, commercials and assess your test strategy

The wrap around counter example is based on an hiring test we gave to every single senior engineer.
<50% of engineers passed this question and this question was a veto

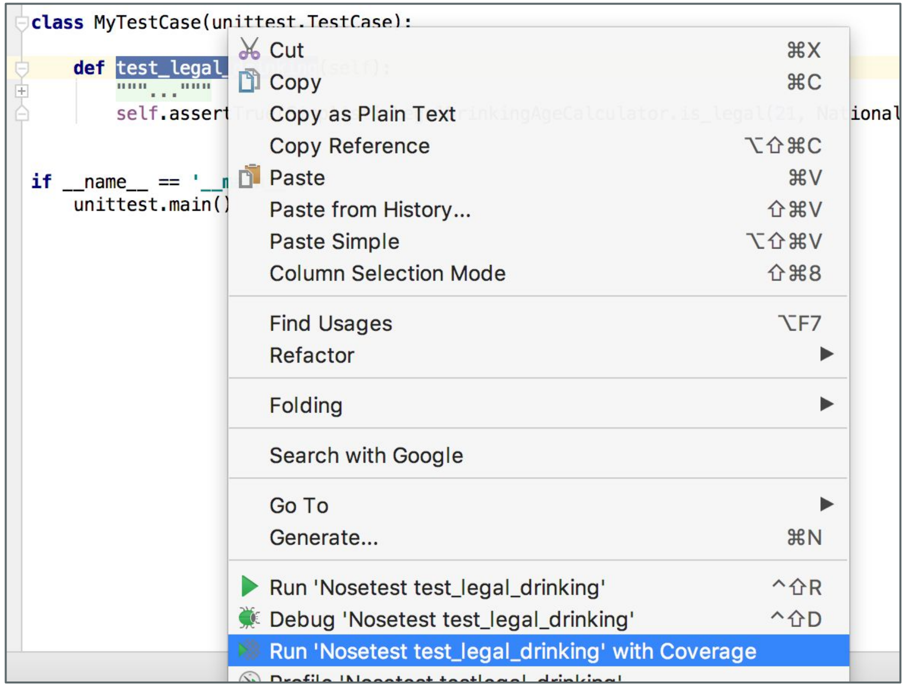
Pop Quiz

Question	Answer
Testing verified the {functional, non-functional, both} behavior of programs?	
Can we exhaustively test a non-trivial program?	
Under normal circumstance can we prove correctness of software?	
A good test case has a <high/low/not relevant> probability of exposing a defect?	
Within black box testing the tester can see <specification, internals, both>?	
Within white box testing the tester can see <specification, internals, both>?	
Once I have written a bunch of tests that pass I am <done, not done, depends>?	
Coverage is a <method of assessing test suite adequacy, way to identify test cases, american and canadian football defensive scheme>	
If I have 100% code coverage my code is correct?	

Reading

Material	Optionality
Finish Beginning S.E. Chapter 4 (across this and next lecture)	Required
Understanding Unit Testing	Required

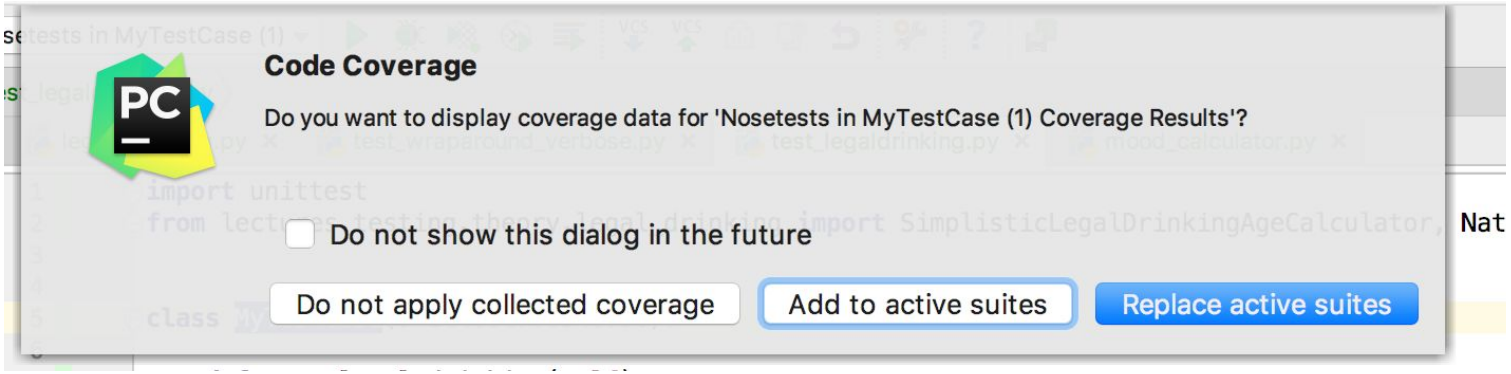
Coverage Tool



```
class MyTestCase(unittest.TestCase):

    def test_legal_
        """..."""
        self.assert                                                          ional

if __name__ == '__m
    unittest.main()
```

Cut ⌘X
Copy ⌘C
Copy as Plain Text
Copy Reference ⌥⇧⌘C
Paste ⌘V
Paste from History... ⇧⌘V
Paste Simple ⌥⇧⌘V
Column Selection Mode ⇧⌘8
Find Usages ⌥F7
Refactor
Folding
Search with Google
Go To
Generate... ⌘N
Run 'Nosetest test_legal_drinking' ^⇧R
Debug 'Nosetest test_legal_drinking' ^⇧D
Run 'Nosetest test_legal_drinking' with Coverage

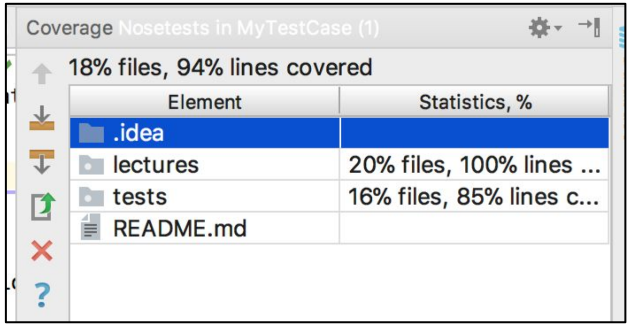
Coverage Report


Code Coverage

Do you want to display coverage data for 'Nosetests in MyTestCase (1) Coverage Results'?

Do not show this dialog in the future

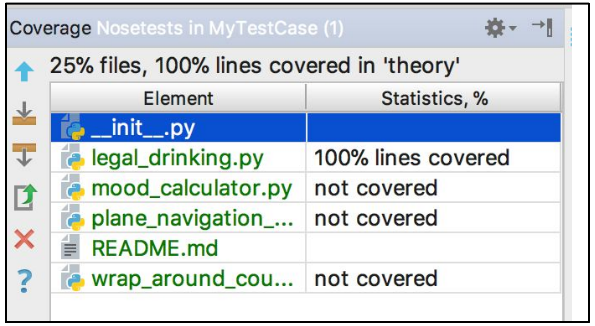
Coverage Report



Coverage Nosetests in MyTestCase (1)

18% files, 94% lines covered

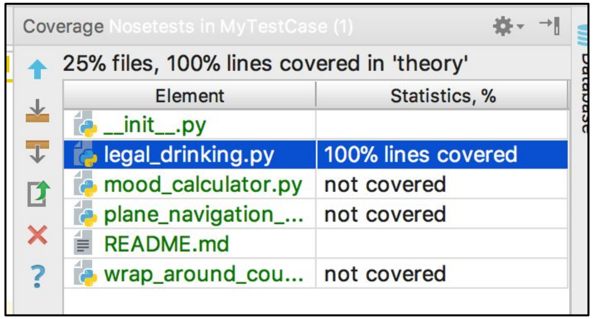
Element	Statistics, %
.idea	
lectures	20% files, 100% lines ...
tests	16% files, 85% lines c...
README.md	



Coverage Nosetests in MyTestCase (1)

25% files, 100% lines covered in 'theory'

Element	Statistics, %
__init__.py	
legal_drinking.py	100% lines covered
mood_calculator.py	not covered
plane_navigation_...	not covered
README.md	
wrap_around_cou...	not covered



Coverage Nosetests in MyTestCase (1)

25% files, 100% lines covered in 'theory'

Element	Statistics, %
__init__.py	
legal_drinking.py	100% lines covered
mood_calculator.py	not covered
plane_navigation_...	not covered
README.md	
wrap_around_cou...	not covered