Coverage testing

https://www.udacity.com/course/software-testing--cs258

1. Introduction

- Usually problems in released software are coming from things that people forgot to test, i.e. testing was inadequate and the developers were unaware of that fact.
- In the following we present a collection of techniques called **code coverage** where automated tools can tell us places where our testing strategy is not doing a good job.

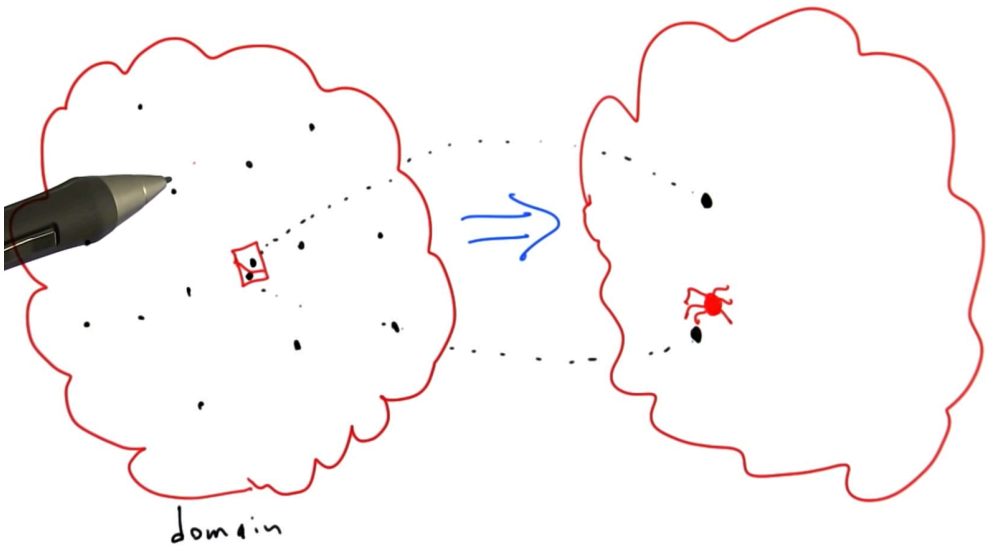
2. How much testing is enough?

- One of the trickiest things about testing software is that it's hard to know when you've done enough testing and the fact is, it is really easy to spend a lot of time testing and to start to believe that you did a good job and then to have some really nasty bug show up that are triggered by parts of the input space that we just didn't think to test.
- We consider some test cases of the domain, i.e. testing is being compliant to some small part of the input domain and the problem, is that even the small part of the domain may contain an infinite number of test cases.
- Next we consider test cases for other parts of the domain we didn't think to test, putting the results and outputs that are not okay (bad range).
- Assume we have a small Python program that cause the Python runtime to crash.
  - The distinguishing feature of it seem to be a large number of cascaded if statements.
  - It's easy if we're testing to remain in the part of the space where, for example, we have < 5 nested ifs.
  - Another region contains >= 5 nested ifs that cause the Python virtual machine to crash.

- Let's say that we're testing some software that somebody has inserted a back door so that can’t be triggered it accidentally (extremely bad range). We didn't tested inputs triggering the back door because we just didn't know it was there.
- We need some sort of a tool (automated scoring system) or methodology that if we are in fact testing only a small part of the input domain for a system to look at our testing effort and to say that the score is, for example, 14 out of 100. Reasons:
  - Our testing efforts will improve by helping us find the input domain that need more testing.
  - We can argue that we've done enough testing.
  - We can identify parts of the test suite that are completely redundant.

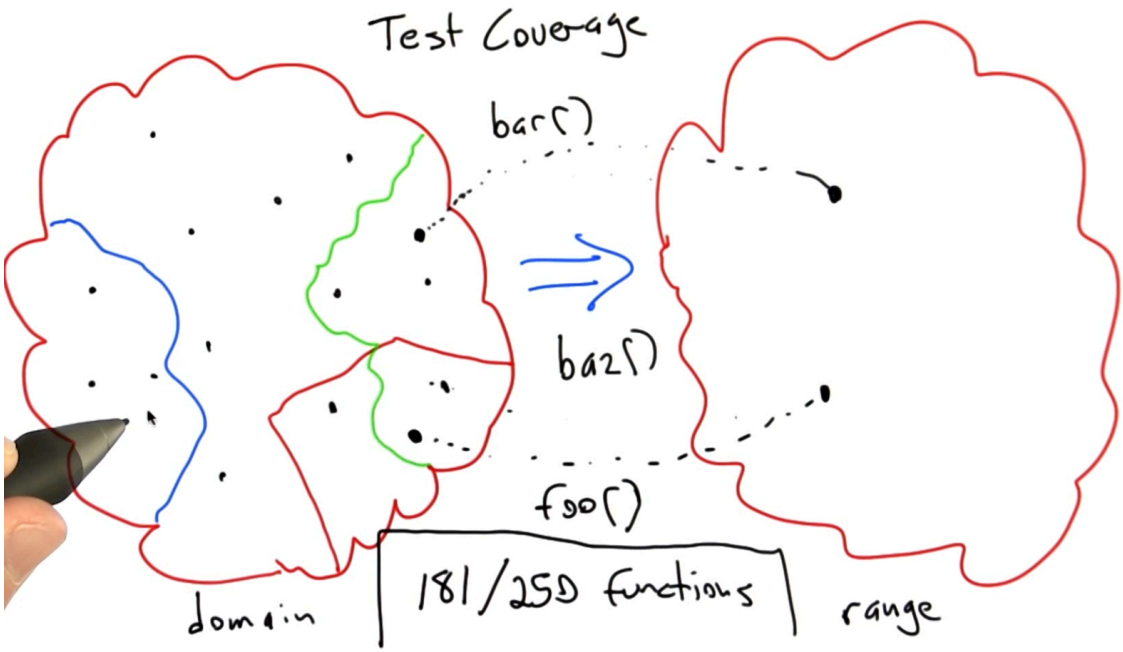
3. Partitioning the input domain

- We start with some SUT and it’s going to have a set of possible inputs, i.e. an input domain, and usually it consists of many possible test cases, that there is no way we can possibly test them all.
- People were often interested in ways to partition the input domain into a no. of different classes so that all of the points within each class are treated the same by the SUT.
- Let’s consider a subset of the input domain. For purposes of finding defects in the SUT, we pick an arbitrary point and execute the system on it. We look at the output, and if it is acceptable, then we’re done testing that class of inputs.
- In practice sometimes what we thought was a class of inputs that are all equivalent might be in different classes. We can blame the partitioning and the unfortunate fact is the original definition of this partitioning scheme didn’t give us good guidance in how to actually do the partitioning.



4. Coverage

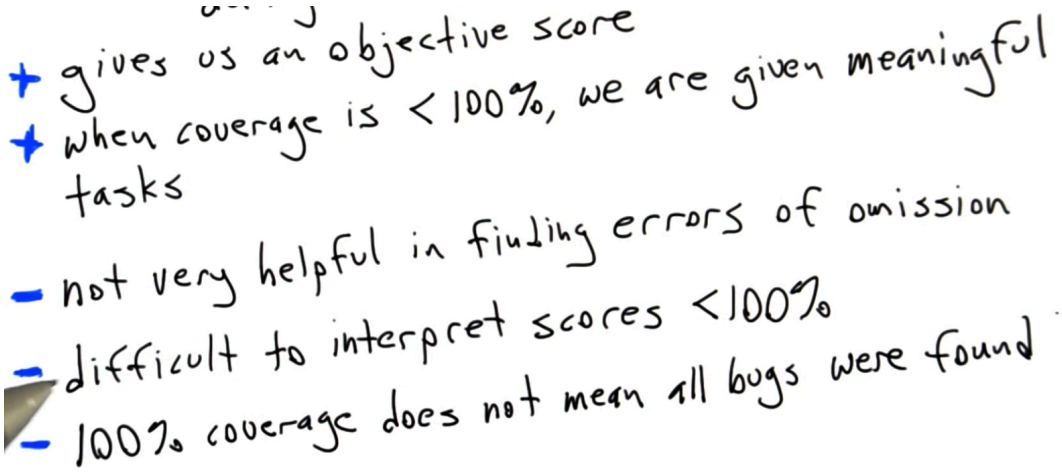
- In practice we ended up with the notion of test coverage instead of a good partitioning of the input domain.
- **Test coverage** is trying to accomplish exact the same thing that partitioning was accomplishing, but it goes about in a different way.
- Test coverage is an **automatic way of partitioning the input domain with some observed features of the source code.**
- One particular kind of test coverage is called **function coverage** and is achieved when every function in our source code is executed during testing.
- We subdivide the input domain for the SUT until we have split it into parts that results in every function being called.



- In practice, we start with a set of test of cases, and we run them all through the SUT. We see which functions are called and then we end up with some sort of a **score** called a **test coverage metric**.
- The score assigned to a collection of test cases is very useful.
  - For each of the functions that wasn't covered, we can go and look at it and we can try to come up with the test input that causes that function to execute.
  - If there is some function baz() for which we can't seem to devise an input that causes it to execute, then there are a couple of possibilities. One possibility is that it can't be called at all. It's **dead code**. Another possibility is that we simply don't understand our system well enough to be able to trigger it.

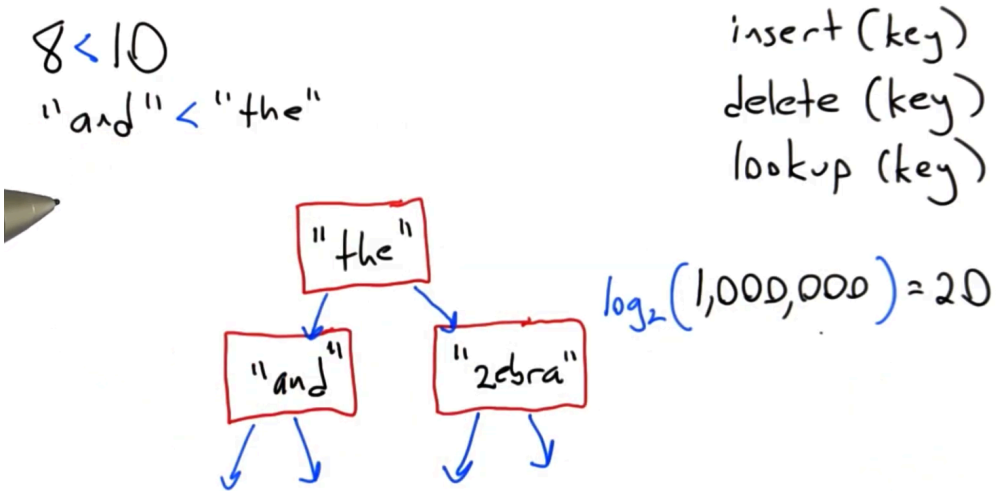
5. Test coverage

- Test coverage is **a measure of the proportion of a program exercised during testing**.



6. Splay tree

- Let's take a concrete look of what a coverage can do for us in practice.
- We'll look at some random open source Python codes that implements a splay tree, a kind of binary search tree.
- A binary search tree is a tree where every node has 2 leaves and it supports operations such as insert, delete, and lookup. The main important thing is the keys have to support an order in relation.
  - If we're using integers for keys then we can use < for order in relation.
  - If we're using words as our keys, then we can use dictionary order.



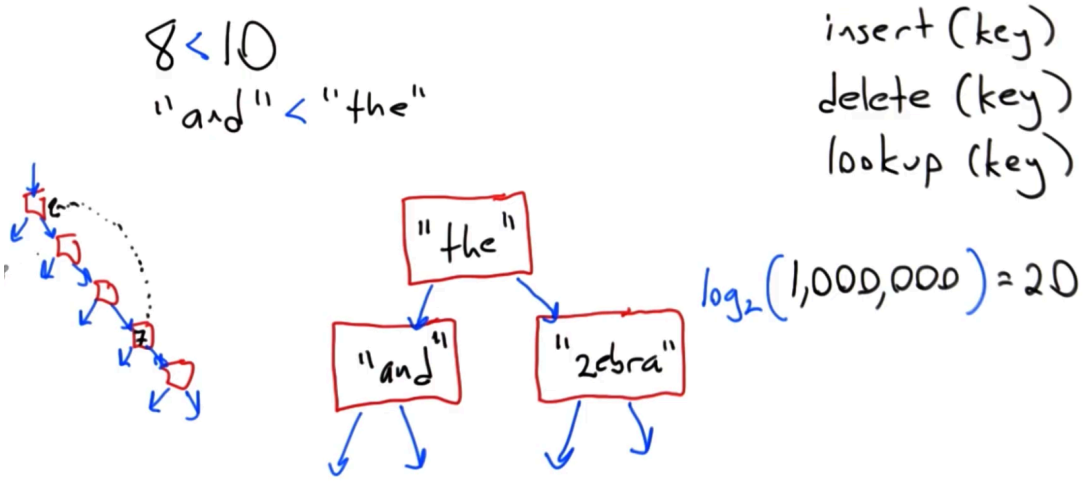
- The way the binary search tree is going to work is, we're going to build up a tree under the invariant that the left child of any node always has a key that's ordered before the

key of the parent node and the right child is always ordered after the parent node using the ordering.

- For a large tree with this kind of shape we have a procedure for fast lookup.
- The way that these trees are set up means that, on average, each comparison allows the operations to skip about half of the tree, so that each lookup, insertion or deletion takes time proportional to the logarithm of the number of items stored in the tree, O(log n).

7. Splay tree issues

- Splay tree is a the simplest example of self-balancing binary search tree. As we add elements a procedure keeps the tree balanced so that tree operations remain very fast.
- It has a really cool property that when we access the nodes, let's say we do a lookup of this node which contains 7, what's going to happen is as a side-effect of the lookup that node is going to get migrated up to the root and then whatever was previously at the root is going to be pushed down and possibly some sort of a balancing operation is going to happen.
  - The point is, that frequently accessed elements end up being pushed towards the root of a tree and therefore, future accesses to these elements become even faster.



- In the following we will look at an open source splay tree implemented in Python, found on the web, that comes with its own test suite and we're going to look at what kind of code coverage this test suite gets on the splay tree.

8. Splay tree example

https://codereview.stackexchange.com/questions/209904/unit-testing-for-splay-tree-in-python

```
class Node:
      def __init__(self, key):
            self.key = key
            self.left = self.right = None

      def equals(self, node):
            return self.key == node.key

class SplayTree:
      def __init__(self):
            self.root = None
            self.header = Node(None)  # for splay()

      def insert(self, key):
            if (self.root == None):
                  self.root = Node(key)
            return

            self.splay(key)
            if self.root.key == key:
                  # If the key is already there in the tree, don't do
anything.
                  return

            n = Node(key)
            if key < self.root.key:
                  n.left = self.root.left
                  n.right = self.root
                  self.root.left = None
            else:
                  n.right = self.root.right
                  n.left = self.root
                  self.root.right = None
            self.root = n

      def remove(self, key):
            self.splay(key)
            if key != self.root.key:
                  raise 'key not found in tree'

            # Now delete the root.
            if self.root.left == None:
                  self.root = self.root.right
            else:
                  x = self.root.right
                  self.root = self.root.left
                  self.splay(key)
                  self.root.right = x
```

```
    def findMin(self):
        if self.root == None:
            return None
        x = self.root
        while x.left != None:
            x = x.left
        self.splay(x.key)
        return x.key

    def findMax(self):
        if self.root == None:
            return None
        x = self.root
        while x.right != None:
            x = x.right
        self.splay(x.key)
        return x.key

    def find(self, key):
        if self.root == None:
            return None
        self.splay(key)
        if self.root.key != key:
            return None
        return self.root.key

    def isEmpty(self):
        return self.root == None
```

- The splay operation moves a particular key up to the root of the binary search tree. This serves as both the balancing operation and also the look up:

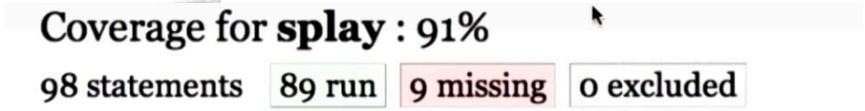
```
    def splay(self, key):
        l = r = self.header
        t = self.root
        self.header.left = self.header.right = None
        while True:
            if key < t.key:
                if t.left == None:
                    break
                if key < t.left.key:
                    y = t.left
                    t.left = y.right
                    r.right = t
                    t = y
                    if t.left == None:
                        break
                r.left = t
                r = t
                t = t.left
            elif key > t.key:
                if t.right == None:
                    break
                if key > t.right.key:
                    y = t.right
                    t.right = y.left
                    r.left = t
                    t = y
                    if t.right == None:
                        break
                l.right = t
                l = t
                t = t.right
            else:
                break
        l.right = t.left
        r.left = t.right
        t.left = self.header.right
        t.right = self.header.left
        self.root = t
```

- Now, let's look at the test suite:

```
import unittest  # module for unit testing
from splay_tree import SplayTree



class TestCase(unittest.TestCase):
    def setUp(self):
        self.keys = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
        self.t = SplayTree()
        for key in self.keys:
            self.t.insert(key)

    def testInsert(self):
        for key in self.keys:
            self.assertEquals(key, self.t.find(key))

    def testRemove(self):
        for key in self.keys:
            self.t.remove(key)
            self.assertEquals(self.t.find(key), None)

    def testLargeInserts(self):
        t = SplayTree()
        nums = 40000
        gap = 307  # stress testing
        i = gap
        while i != 0:
            t.insert(i)
            i = (i + gap) % nums

    def testIsEmpty(self):
        self.assertFalse(self.t.isEmpty())
        t = SplayTree()
        self.assertTrue(t.isEmpty())

    def testMinMax(self):
        self.assertEquals(self.t.findMin(), 0)
        self.assertEquals(self.t.findMax(), 9)


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

- After running the tests we can use a **code coverage tool** to measure coverage. We can generate a HTML report. In our case, it's telling us that when we run the splay tree on its own unit test suite, out of the 98 statements in the file, 89 of them got run, 9 of them failed to run.
  `coverage erase ; coverage run splay_tree_test.py ; coverage html -i`

```
udacitys-imac:files udacity$ emacs splay.py
udacitys-imac:files udacity$ python splay_test.py
.....
----------------------------------------------------------------------
Ran 5 tests in 1.686s

OK
udacitys-imac:files udacity$ coverage erase ; coverage run splay_test.py ; coverage html -i
```




```
1  class Node:
2      def __init__(self, key):
3          self.key = key
4          self.left = self.right = None
5
6      def equals(self, node):
7          return self.key == node.key
8
9  class SplayTree:
10     def __init__(self):
11         self.root = None
12         self.header = Node(None) #For splay()
13
14     def insert(self, key):
15         if (self.root == None):
16             self.root = Node(key)
17             return
```

- The test suite failed to test the case where we inserted an element into the tree and it was already there.
- In the splay tree's removing function the first thing this function does is, splays the tree based on the key to be removed and so this is intended to draw that key up to the root note of the tree. If the root node of the tree does not have the key that we're looking for, then we're going to raise an exception saying that this key wasn't found. But this wasn't tested.
- If we look in the body of the delete function, we see a pretty significant chunk of code that wasn't tested, so we have to go back and revisit this.

- In conclusion, the tool is showing us what we didn't think to test with the unit test suite that we wrote so far.

9. Improving coverage

- We will modify in the test suite, testRemove function:

```
    def testRemove(self):
        for key in self.keys:
            self.t.remove(key)
            self.assertEquals(self.t.find(key), None)
        self.t.remove(-999)
```

- After running the tool the line that we just added causes an exception re-thrown in the splay function. So we have found a bug not anticipated by the developer of the splay tree.
- It often turns out that the stuff that we thought was going to run might be running only some of it. So the coverage tool told us something interesting.
- On the other hand, if the coverage tool hasn't told us anything interesting, i.e. everything we hoped was executed well when we run the unit test suite then that's good, too.
- Another thing we noticed is that the bug was somewhere completely different buried in the splay routine and if we go back and look at the coverage information, it turns out that the splay routine is entirely covered, i.e. every line of code was executed during the execution of the unit test for the splay tree.
- We deduce that just because some code was covered, especially at the statement level, this does not mean that it doesn’t contain a bug in it.
  - We have to ask the question, "What do we want to really read into the fact that we failed to cover something?” The coverage tool has given an example suggesting that our test suite is poorly thought out.
  - So, when coverage fails its better to try to think about why went wrong rather than just blindly writing a test case and just exercise the code which wasn't covered.

10. Problems with coverage

- We looked to an example where measuring coverage was useful in finding a bug in a piece of code. The coverage is not particularly useful in spotting the bug.
- In the following we will discuss about another piece of code, a broken function whose job is to determine whether a number is prime.

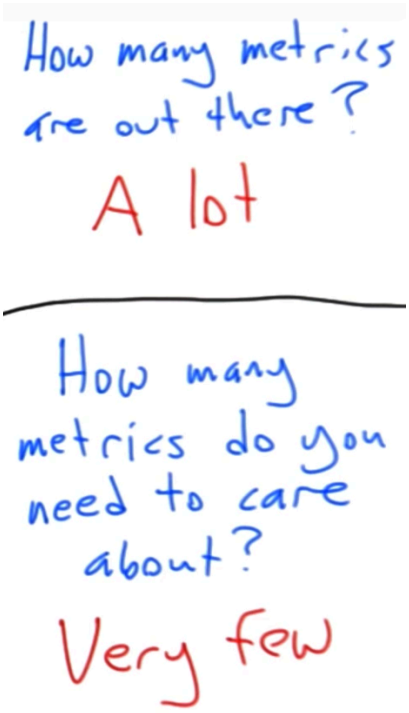
```
# CORRECT SPECIFICATION:
#
# isPrime checks if a positive integer is prime.
#
# A positive integer is prime if it is greater than
# 1, and its only divisors are 1 and itself.


import math

def isPrime(number):
    if number <= 1 or (number % 2) == 0:
        return False
    for check in range(3, int(math.sqrt(number))):
        if (number % check) == 0:
            return False
    return True

def check(n):
    print("isPrime(" + str(n) + ") = " + str(isPrime(n)))

check(1)
check(2)
check(3)
check(4)
check(5)
check(20)
check(21)
check(22)
check(23)
check(24)

def test():
    assert isPrime(1) == False
    assert isPrime(2) == False
    assert isPrime(3) == True
    assert isPrime(4) == False
    assert isPrime(5) == True
    assert isPrime(20) == False
    assert isPrime(21) == False
    assert isPrime(22) == False
    assert isPrime(23) == True
    assert isPrime(24) == False
```

- isPrime function has successfully identified whether the input is prime or not for the 10 numbers in the assertions.
- When we run the code coverage tool we can see out of the 20 statements in the file, all of them run, and none of them failed to be covered. Statement coverage gives us a perfect result for this particular code and yet another set is wrong.
- `coverage erase ; coverage run --branch prime.py ; coverage html -i`

```
# TASKS:
#
# 1) Add an assertion to test() that shows
#    isPrime(number) to be incorrect for
#    some input.
#
# 2) Write isPrime2(number) to correctly
#    check if a positive integer is prime.

# Note that there is an error where isPrime() incorrectly returns False for
an input of 2, despite 2 being a prime number.
# This has been fixed in the following.
# In isPrime function the range loops between 3 and the square of the input-1


import math

def isPrime2(number):
    if number == 2:
        return True
    if number <= 1 or (number % 2)==0:
        return False
    for check in range(3, int(math.sqrt(number)) + 1):
        if (number % check) == 0:
            return False
    return True

def test():
    assert isPrime(6) == False
    assert isPrime(7) == True
    assert isPrime(8) == False
    assert isPrime(9) == True
    assert isPrime(10) == False
    assert isPrime(25) == True
    assert isPrime(26) == False
    assert isPrime(27) == False
    assert isPrime(28) == False
    assert isPrime(29) == True

def test2():
    assert isPrime2(6) == False
    assert isPrime2(7) == True
    assert isPrime2(8) == False
    assert isPrime2(9) == False
    assert isPrime2(10) == False
    assert isPrime2(25) == False
    assert isPrime2(26) == False
    assert isPrime2(27) == False
    assert isPrime2(28) == False
    assert isPrime2(29) == True
```

- Why did test coverage fail to identify the bug? Statement coverage is a rather crude metric that only checks whether each statement executes once. Each statement executes at least once that lets a lot of bugs slip through.
- The lesson here is we should not let complete coverage plus a number of successful test cases fool as into thinking that a piece of code of right. It's often the case that deeper analysis is necessary.

11. Coverage metrics

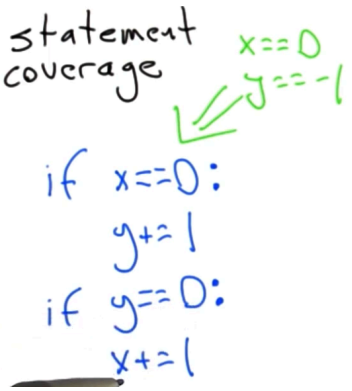


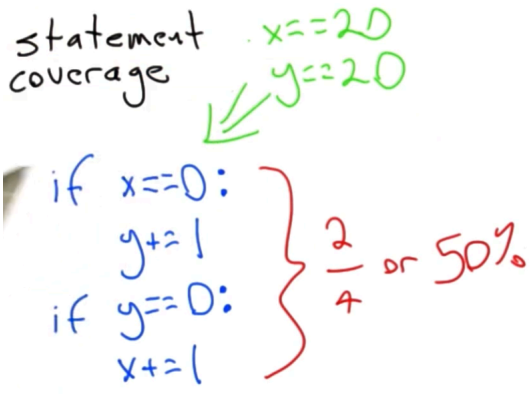
•There is a large number of test coverage metrics. An article lists **101 test coverage metrics**.

•Here we talk about a fairly small number of coverage metrics that matter for everyday programming life.

•The first is statement coverage measured by the Python test coverage tool where we looked at.

•Let's use a very simple 4 line codes and try to measure the statement coverage. Let's say we call this code with x=0 and y=-1. All 4 will be executed and this will give us a statement coverage of **100%**.





•Let's call this code with x=20 and y=20. Both tests will fail, and so, we will end up with a code coverage of 2/4 statements or **50%**.

- **Line coverage** is very similar to statement coverage but the metric is tied to actual physical lines in the source code. In this case, there is only one statement for each line so statement coverage and line coverage would be exactly identical.

- But on the other hand, if we decided to write some code that had multiple statements per line, line coverage would conflate them whereas statement coverage would consider them individual statements. For most practical purposes, these are very similar. So, statement coverage has a slightly finer granularity.

12. Testing coverage

- The function stats is defined to take a list of numbers as input and what the function does is computes the smallest element, the largest element, the median element and the mode, i.e. the element which occurs the most frequently in the list.

```
def stats(lst):
    min = None
    max = None
    freq = {}
    for i in lst:
        if min is None or i < min:
            min = i
        if max is None or i > max:
            max = i
        if i in freq:
            freq[i] += 1
        else:
            freq[i] = 1
    lst_sorted = sorted(lst)
    if len(lst_sorted) % 2 == 0:
        middle = len(lst_sorted)/2
        median = (lst_sorted[middle] + lst_sorted[middle-1]) / 2
    else:
        median = lst_sorted[len(lst_sorted)/2]
    mode_times = None
    for i in freq.values():
        if mode_times is None or i > mode_times:
            mode_times = i
    mode = []
    for (num, count) in freq.items():
        if count == mode_times:
            mode.append (num)
    print "list = " + str(lst)
    print "min = " + str(min)
    print "max = " + str(max)
    print "median = " + str(median)
    print "mode(s) = " + str(mode)

def test():
    # Change l to something that manages full coverage. You may
    # need to call stats twice with different input in order
    # to achieve full coverage.
    l = [31]
    stats(l)

test()
```

- We can see above that the bad test managed to cover 29 statements in the stats function.
- In the following we'll write a collection of test cases in order to get 100% statement coverage.

```
def test():
    l = [31,32,33,34]. # even number of elements
    stats(l)
    l = [31,32,33,33]. # odd number of elements
    stats(l)
test()
```

13. Fooling coverage

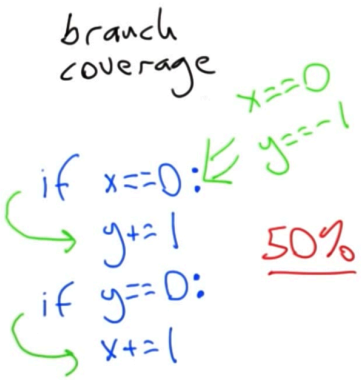
- Below we insert a bug into the stats module, i.e. make it wrong but in a way that's undetectable by test cases that get full statement coverage.
-

```
# TASK:

# Achieve full statement coverage on the stats function.
# All you should have to do is modify the test function
# to call stats with different lists of values.

# You will need to:
# 1) Insert a bug into the stats function.
# 2) Modify test1 so that it still achieves full test
#    coverage, but does not trigger your bug. Depending
#    on the bug you insert, you may not need to modify
#    test1 at all.
# 3) Write test2 so that it also achieves full test
#    coverage, but does trigger your bug.
```

```
import math

def stats(lst):
    min = None
    max = None
    freq = {}
    for i in lst:
        # i = abs(i)  => in test2 we consider l = [-33,-34]
        if min is None or i < min:
            min = i
        if max is None or i > max:
            max = 31  # bug
        if i in freq:
            freq[i] += 1
        else:
            freq[i] = 1
    lst_sorted = sorted(lst)
    if len(lst_sorted) % 2 == 0:
        middle = len(lst_sorted)/2
        median = (lst_sorted[middle] + lst_sorted[middle-1]) / 2
    else:
        median = lst_sorted[len(lst_sorted)/2]
    mode_times = None
    for i in freq.values():
        if mode_times is None or i > mode_times:
            mode_times = i
    mode = []
    for (num, count) in freq.items():
        if count == mode_times:
            mode.append (num)
    print "list = " + str(lst)
    print "min = " + str(min)
    print "max = " + str(max)
    print "median = " + str(median)
    print "mode(s) = " + str(mode)

# test1 should achieve full statement coverage of the stats function
# without triggering the bug you've inserted into the stats function.
def test1():
    ###Your test1 code here. Depending on what
    # bug you choose to put in the stats function,
    # you may or may not need to modify test1.
    l = [31, 31, 1, 2, 2, 1]
    stats(l)
    l = [31]
    stats(l)

# test2 should also achieve full statement coverage of the stats function,
# but should trigger the bug you've inserted into the stats function.
def test2():
    l = [31, 31, 1, 2, 2, 1]
    stats(l)
    l = [32]
    stats(l)

test1()
test2()
```

14. Branch coverage

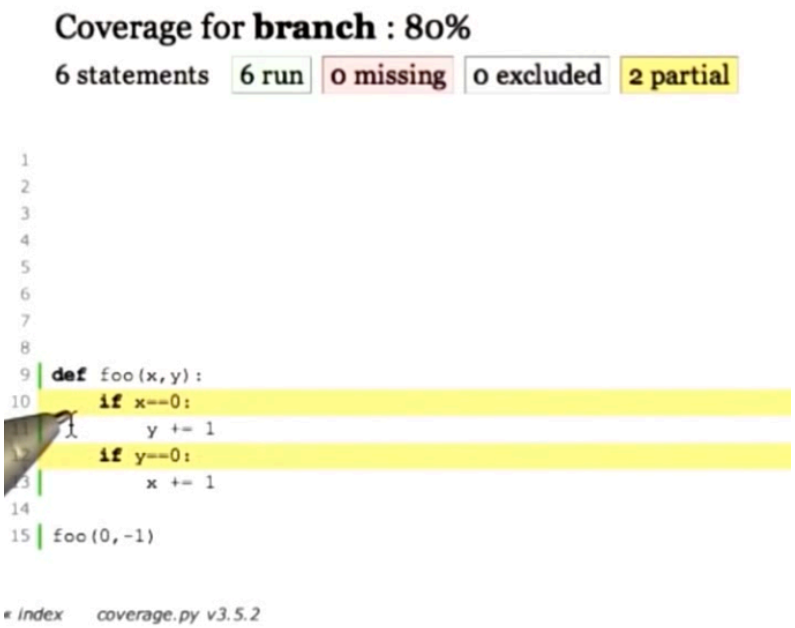
- We just talked about statement coverage, which is closely related to line coverage, but it's a bit more fine-grained, and now let's talk about what is probably the only other test coverage metric that will matter in your day-to-day life unless you go build avionics software.
- Branch coverage is a metric where a branch in a code is covered, if it executes both ways.
- In many cases, branch coverage and statement coverage have the same effect. For example, if our code only contained if-then-else loops, the metrics would be equivalent. On the other hand for code like on the right that's missing the else branches they're not quite equivalent.



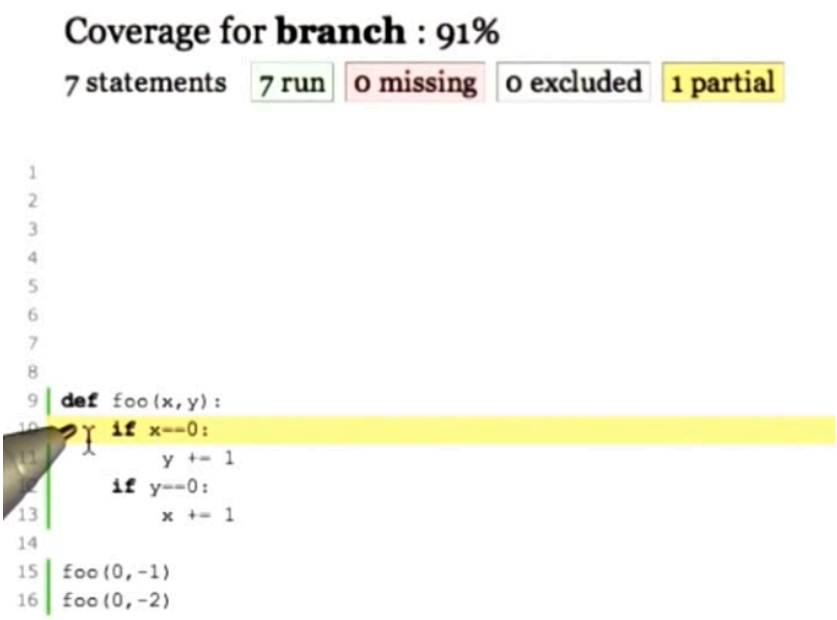
- These inputs are sufficient to get 100% statement coverage and 50% branch coverage:
- We're going run this under the coverage tool, but this time we're going to give the coverage run command an argument --branch:

```
coverage erase ; coverage run --branch foo.py ; coverage html -i
```

- That simply tells it to measure branch coverage instead of just measuring statement coverage.

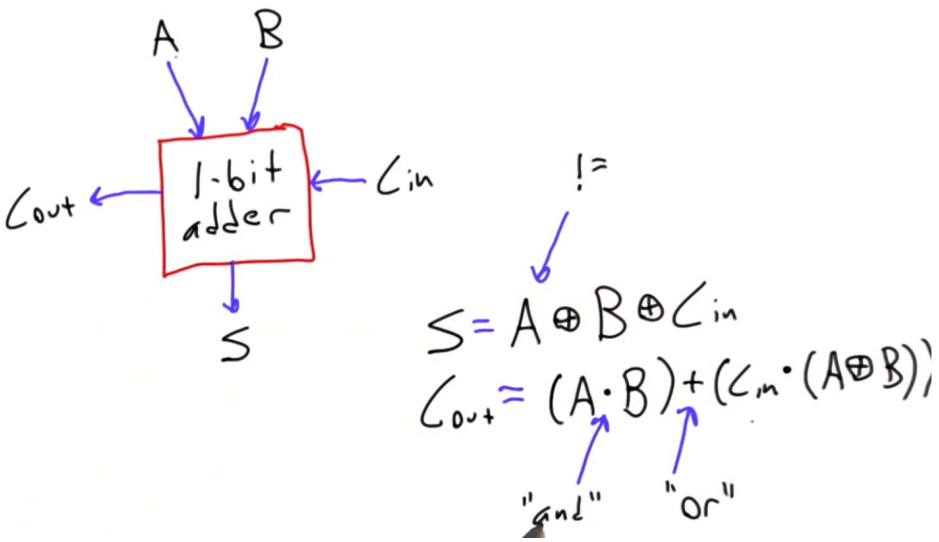


- If we call foo a second time with 0 and -2, we get:

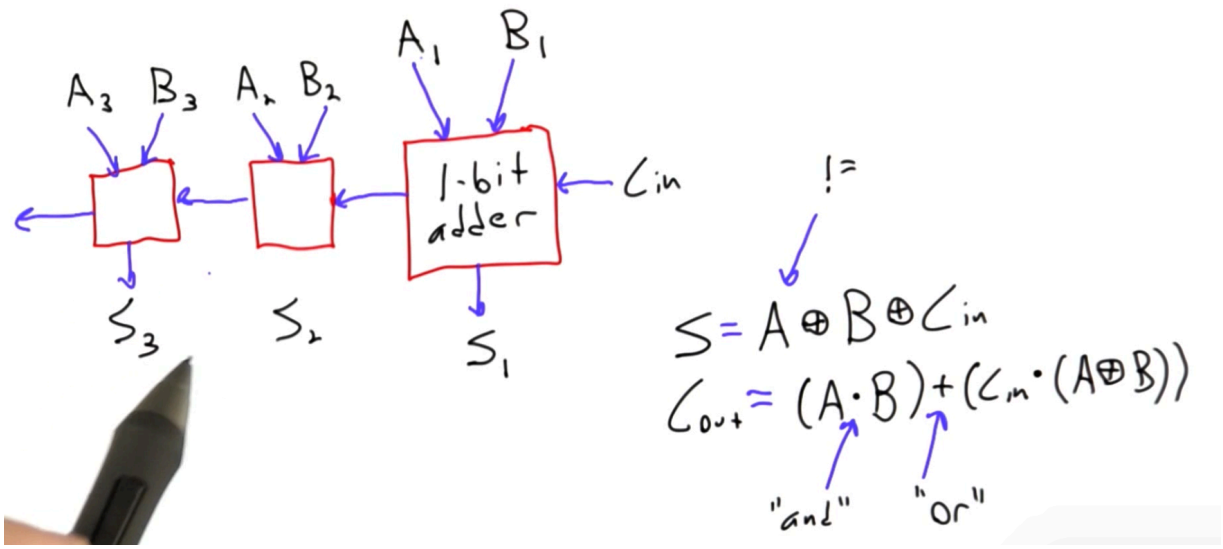


15. 8 Bit adder

- Let's consider an example of Python code that simulates some adders, i.e. simple hardware modules that perform addition.
- The way that 1-bit adder works, where A and B are 2 input bits, C in is the carry into the column, C out is the carry out of the column, S is the sum, ⊕ is XOR operator implemented in Python using the ! Operator:



- An actual adder that corresponds to the add instruction that you would find in an instruction set for a real computer looks like in the following:



- Python code implementing an 8-bit adder:

```
## SPECIFICATION:
#
# add8 emulates an 8-bit hardware adder.
# it takes 17 bits, representing two 8-bit
# numbers and a carry bit.
#
# TASK:
#
# Write test() such that it achieves 100%
# branch coverage of the add8 function.

def add8(a0,a1,a2,a3,a4,a5,a6,a7,b0,b1,b2,b3,b4,b5,b6,b7,c0):
    s1 = False
    if (a0 != b0) != c0:
        s1 = True
    c1 = False
    if (a0 and b0) != (c0 and (a0 != b0)):
        c1 = True
    s2 = False
    if (a1 != b1) != c1:
        s2 = True
    c2 = False
    if (a1 and b1) != (c1 and (a1 != b1)):
        c2 = True
    s3 = False
    if (a2 != b2) != c2:
        s3 = True
    c3 = False
    if (a2 and b2) != (c2 and (a2 != b2)):
        c3 = True
    s4 = False
    if (a3 != b3) != c3:
        s4 = True
    c4 = False
    if (a3 and b3) != (c3 and (a3 != b3)):
        c4 = True
```

```
    s5 = False
    if (a4 != b4) != c4:
        s5 = True
    c5 = False
    if (a4 and b4) != (c4 and (a4 != b4)):
        c5 = True
    s6 = False
    if (a5 != b5) != c5:
        s6 = True
    c6 = False
    if (a5 and b5) != (c5 and (a5 != b5)):
        c6 = True
    s7 = False
    if (a6 != b6) != c6:
        s7 = True
    c7 = False
    if (a6 and b6) != (c6 and (a6 != b6)):
        c7 = True
    s8 = False
    if (a7 != b7) != c7:
        s8 = True
    c8 = False
    if (a7 and b7) != (c7 and (a7 != b7)):
        c8 = True
    return (s1,s2,s3,s4,s5,s6,s7,s8,c8)
```

- Let's say this code is part of some sort of a circuit simulation, and we want to test the validity of our circuit, therefore we pass it actual numbers. We can write some little support functions that take integers and convert them into the bit format.
- In the following there is a solution using exhaustive testing:

```
def split (n):
    return (n&0x1,n&0x2,n&0x4,n&0x8,n&0x10,n&0x20,n&0x40,n&0x80)

def glue (b0,b1,b2,b3,b4,b5,b6,b7,c):
    t = 0
    if b0:
        t += 1
    if b1:
        t += 2
    if b2:
        t += 4
    if b3:
        t += 8
    if b4:
        t += 16
    if b5:
        t += 32
    if b6:
        t += 64
    if b7:
        t += 128
    if c:
        t += 256
    return t

def myadd (a,b):
    (a0,a1,a2,a3,a4,a5,a6,a7) = split(a)
    (b0,b1,b2,b3,b4,b5,b6,b7) = split(b)
    (s0,s1,s2,s3,s4,s5,s6,s7,c) =
add8(a0,a1,a2,a3,a4,a5,a6,a7,b0,b1,b2,b3,b4,b5,b6,b7,false)
    return glue (s0,s1,s2,s3,s4,s5,s6,s7,c)

def testExhaustive():
    for i in range(256):
        for j in range(256):
            res = myadd(i,j)
            print str(i) + " " + str(j) + " " + str(res)
            assert res == (i+j)

testExhaustive()
```

- When running

  `coverage erase ; coverage run --branch 8bits-adder.py ; coverage html -i`

  the coverage output is:

  Coverage for **adder** : 100%

  85 statements | 85 run | 0 missing | 0 excluded | 0 partial

- Now let's look at an alternative solution. Instead of our exhaustive test, we could have written a much smaller test that gets 100% branch coverage.

```
def smallTest():
    myadd(0,0);
    myadd(0,1);
    myadd(255,255);

smallTest()
```

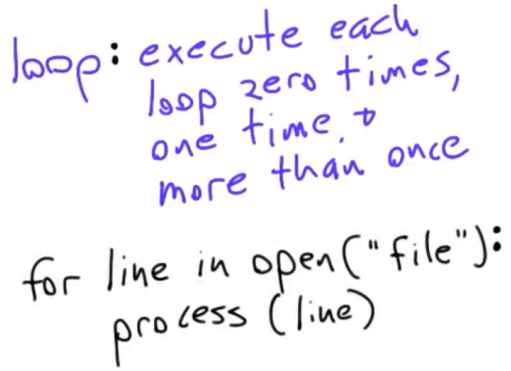
Coverage for **adder** : 100%

81 statements	81 run	0 missing	0 excluded	0 partial

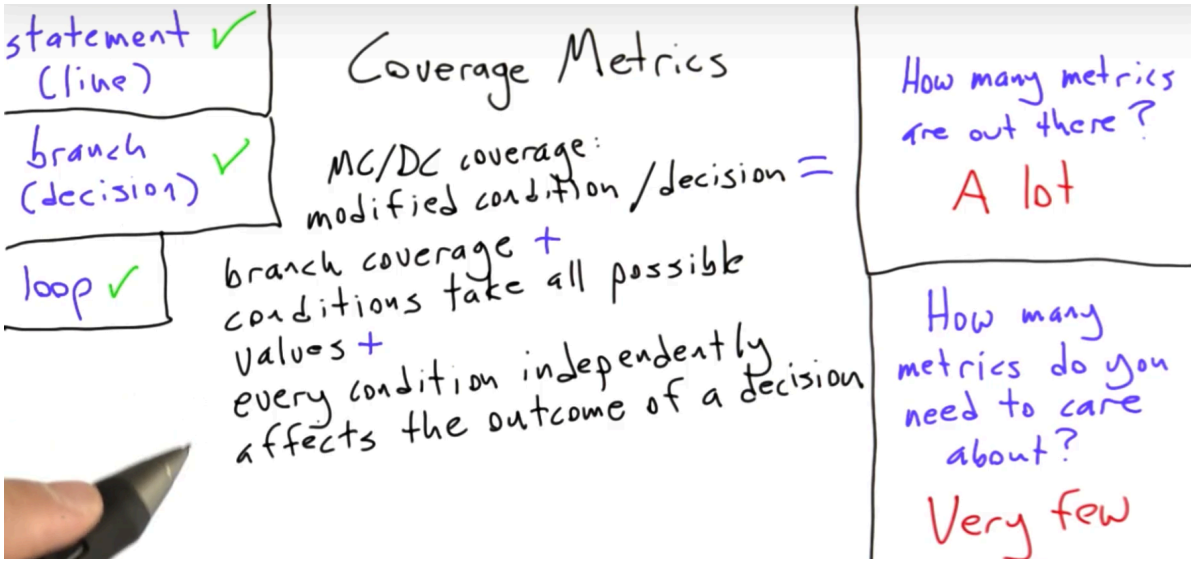
- So, as you can see, the coverage tool believes that just these 3 really simple test cases were sufficient to get 100% branch coverage of our adder.

16. Other metrics

- We looked at a couple common coverage metrics that come up in practice
  - **Statement coverage**, which is close relative to line coverage,
  - **Branch coverage**, also called decision coverage.
- These are the coverage metrics that are going to matter for everyday life.
- There are many other coverage metrics, and we're going to just look at a few of them not because we're going to go out and obsessively get 100% coverage on our code on all these metrics, but rather because they form part of the answer to the question **how shall we come up with good test inputs in order to effectively find bugs in our software**?
  - **Loop coverage** specifies that we execute each loop 0 times, once, and more than once. In the example on the right, to get full loop coverage we would need to test this code using a file that contains no lines, using a file that contains just one line, and using a file that contains multiple lines.

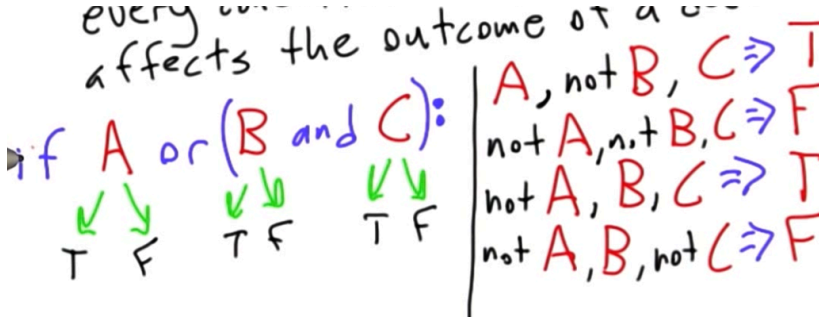


  - **Modified condition decision coverage** or MC/DC starts off with a branch coverage, it additionally states that every condition involved in a decision takes on every possible outcome:



17. MC/DC coverage

- To get full MC/DC coverage of the next Python conditional statement, we need to test each of the variables so that every condition independently affects the outcome:

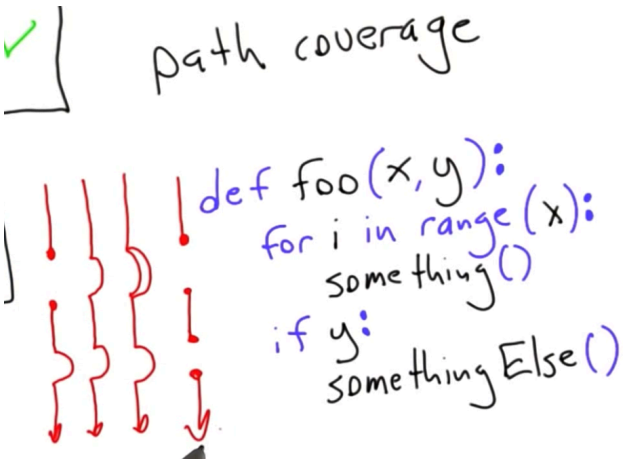


- We have above a minimal - or if not minimal, at least fairly small - set of test cases that together gets 100% MC/DC coverage.
- We could've written a conditional much more complicated, and if we had, we probably would've had a fairly hard time reasoning this stuff out by hand, and what we would've needed to do in that case is probably draw out a full truth table.
- The domain of interest for MC/DC coverage, i.e. embedded control systems that happen to be embedded in avionics systems end up having generally lots of complicated conditionals.
- We lack good tool support for MC/DC coverage in Python.

18. Path coverage

- It cares about how you got to a certain piece of code.

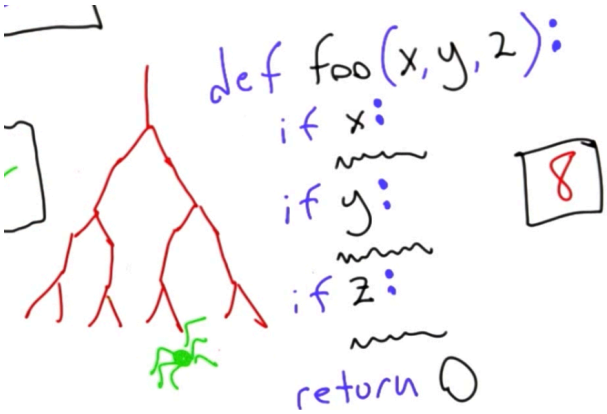
- Statement coverage and branch coverage, and even to a large extent MC/DC coverage and loop coverage, don't really care how you got somewhere as long as you executed the code in such a way that you met the conditions.



•A **path through a program** is a sequence of decisions made by operators in the program.

•Suppose we have a function foo and we will try to visualize the decisions made by the Python language. Let's consider x = 0, y = true, then (1, true). As x increases in value, we get more and more and more paths. There's going to be a similar family of paths for y is false.

- By changing x how many paths can we get through the code? The answer is that it's unlimited, so **achieving path coverage is going to be impossible for all real code**.
- Path coverage is basically an ideal that we'd like to approach if we want to do a good job testing. It's not going to be something that we can actually achieve.
- Suppose we have another function foo. How many paths through this code are?

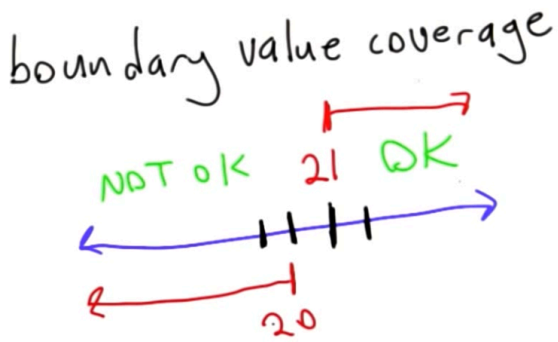


- We need path coverage to find a certain bug. This is a valuable weapon to have in our testing arsenal, even if we're not going to be achieving path coverage in practice.

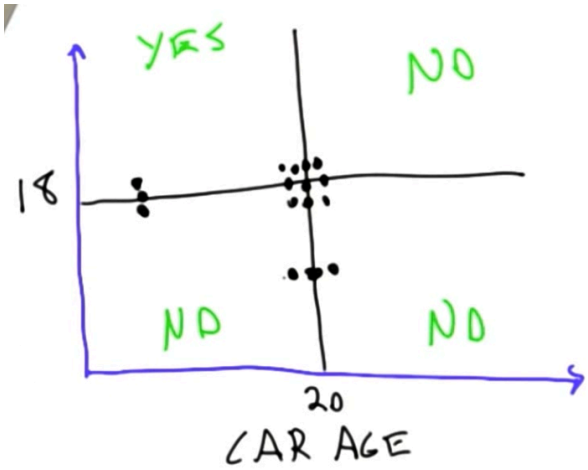
19. Boundary value coverage

- Unlike some of the other coverage measures we discussed in this lesson, doesn't have any specially take technical definition.
- What boundary value coverage basically says is when a program depends on some numerical range and when the program has different behaviors based on numbers within that range then we should test numbers close to the boundary.

- Let's take the example where we're writing a program to determine whether somebody who lives in the USA is permitted to drink alcohol.
- We want to get the boundary value coverage on this program, so we want to include the ages of 20 and 21 in our test input and possibly also of 19 and 22 close enough to the boundary values that there may be interesting behaviors looking at.



- We framed the boundary value coverage as a function of only one variable in terms of the program specification not in terms of the implementation.
- Let's consider a program with 2 inputs. Assume that these inputs are treated independently by the software that we are running. The first input is going to be the age of a car. An insurance company declines to insure cars more than 20 years old. The other parameter is the age of the driver. Drivers who are less than 18 years old will be declined to insure.
- If we have specific knowledge of our implementation we consider these variables together therefore we probably also need to test combinations of inputs



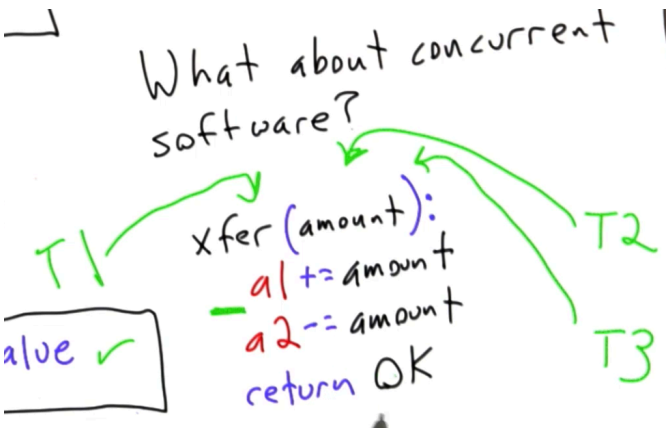
- As the number of inputs of the program goes up the number of test cases can grow very large because we have to consider the interaction between all possible

combinations of variables that are dependent. On the other hand, if our variables are independent then we can test this separately.

- Issue: we are doing **boundary value coverage with respect to the requirements** of the specification purpose of software or whether we are doing it with respect to the **implementation**.
- Recall the stats function described before where we inserted the bug i = abs(i) which causes it to misbehave for some inputs and not for others.
- We find a collection of test cases to get good coverage for the absolute value of the inputs when we pass numbers into the function which contain negative values.
- Considering the implementation not just the specification, what will happen is a function like absolute value would change its behavior around 0 and so what we need to call it with at least one negative number.
- In the stats function we have a lot of different operators with different behaviors around certain boundaries and so to get good boundary value coverage over all would be probably extremely difficult.
- There aren't good tools automating boundary value coverage in Python. There are techniques such as **mutation testing** that can automate boundary value coverage in some forms.

20. Concurrent software

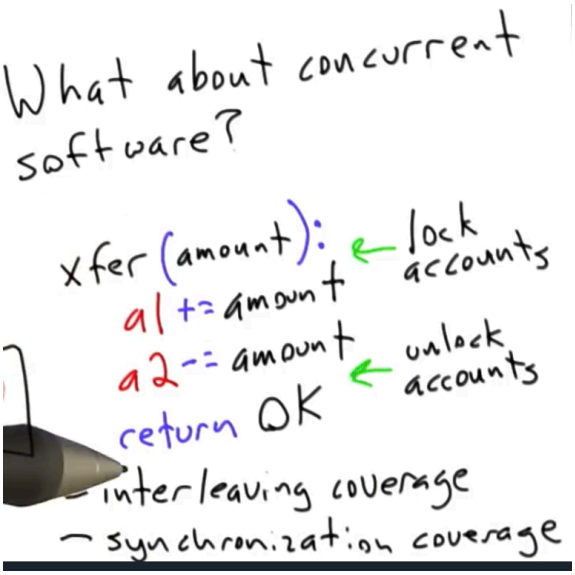
- We haven't been dealing at all with testing of concurrent code and this mainly it is a difficult and specialized skill.
- It's clear that applying sequential code coverage metrics to concurrent software is a fine idea, but these aren't going to give us any confidence with the code lacks concurrency such as race condition and the deadlocks.
- Let's take, for example, a function xfer, which transfer some amount of money between bank account one and bank account two.
- This particular function is designed to be called from different threads.
- The transfer function does not synchronise, i.e. it hasn't taken any sort of a lock while it manipulates the accounts. If these threads are operating on the same accounts concurrently, then it's going to be a problem.



- What sort of coverage what we are looking for while testing this function in order to detect this kind of bug? We want to make sure we tested the case where the transfer account is concurrently call.

21. Synchronization coverage

- The likely fix for the bug that we had in xfer function is to lock both of the bank accounts that we're processing, transfer the balance between them, and then unlock the accounts.
- We can delete all of the locks out of a code, run it through some tests, and often it passes. This is in spite of the test coverage metric called **synchronization coverage,** which ensures that during testing this lock actually does something.



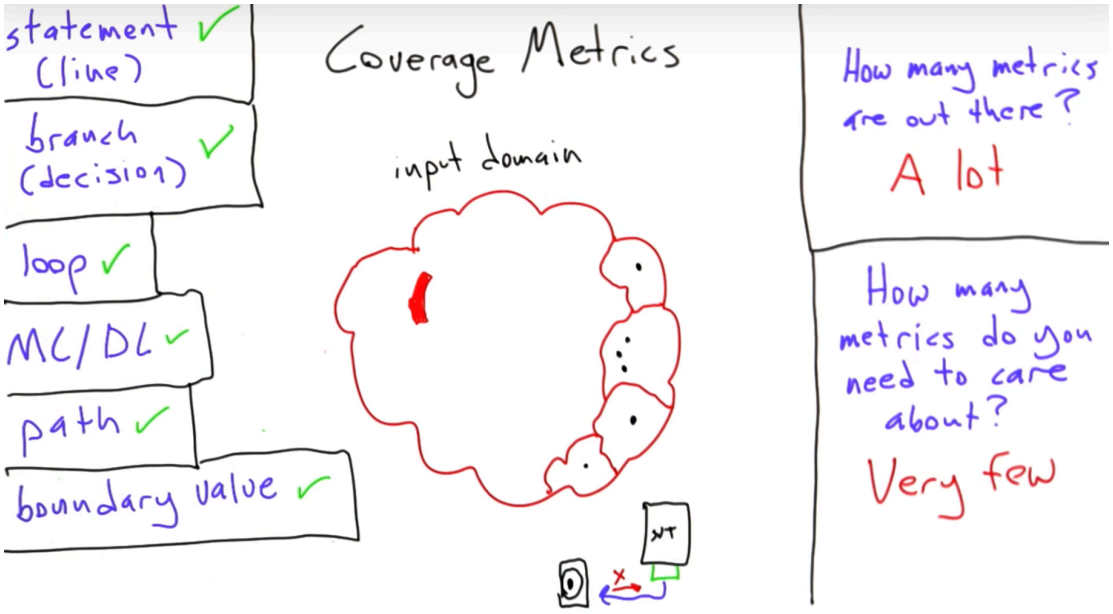
- In other words, during testing, the xfer function is going to be called to transfer money between accounts when the accounts are already locked and this ensures that we're stressing the system to a level that the synchronization code is actually firing.
- **Interleaving coverage** means if we recall functions which accessed shared data are actually called and in a truly concurrent fashion, i.e. by multiple threads at the same time.

22. When coverage doesn’t work

- What coverage is letting us do is divide up the input domain into regions in such a way, for any given region, any test input in that region, will accomplish some specific coverage task.
- Any input that we select within this particular region of the input domain will cause a particular statement or line to execute, cause the branch to execute in some specific direction, execute a loop zero times, one time, or more, etc.
- The obvious problem is, if we partition the input domain this way and we go ahead and test, it is easier to reach the region in the input domain, and get good coverage, but **we won’t be able to find errors of omission.**
- For example, the SUT is creating files on the hard disc. One extremely possible kind of bug that we would put into the SUT is failing to check error codes that could be

returned from file creation operations that happen when the disc is full or when there is a hard disc failure or something like that.

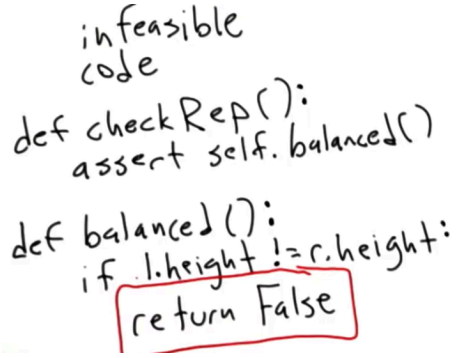
- Coverage metrics are not particularly good at discovering errors of omission like missing error checks.
- We discussed fault injection where we make the disc fail, we will make it send something bad up to the system and we see what happens, and in that case, if an error check is missing, then the system should actually do the wrong thing and we will be able to discover this by watching the system misbehave.



- Another thing we could have done is partition the input domain in a different way, i.e. not partition the input domain using an automated coverage metrics rather using the specification.
- In conclusion, there are multiple ways of partitioning the input domain for purposes of testing.

23. Infeasible code

- What does it mean when we have code that doesn't get covered, for example, if we're using statement coverage, **what happens when we have some statements that we haven't been able to cover?**
- There're basically **3 possibilities**. **The first possibility is infeasible code.**
- Let's say that we're writing the checkRep function for a balanced tree data structure. See open source code in Python for AVL tree at https://github.com/



majek/dump/blob/master/avl/avl.py-orig

- Assuming that the code is not bugged and assuming that we're testing a correct tree, we'll never going to be able to return false from our balanced function.
- A coverage tool is going to tell that we failed to achieve code coverage for this particular statement in the code. We have to ask ourselves, is that a bad thing? It's not bad because that code only can execute if we made a mistake somewhere. So, the proper response to this kind of situation is, we need to t**ell our coverage tool that we believe this line to be infeasible** and then the tool won't count this line when we're measuring code coverage.
- The code comes with its own test suite so let's run it under the coverage monitor using `coverage erase ; coverage run --branch avl_tree.py ; coverage html`

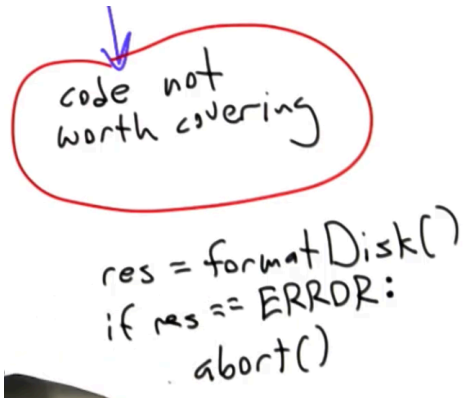
Coverage for **avltree** : 86%

389 statements	328 run	61 missing	0 excluded	20 partial

- Every statement that raises an exception has failed to have been covered. So superficially, we haven't gotten very good coverage of this function, but actually, what we hope is that this AVL tree code is correct and these are truly infeasible. Therefore, we can tell the coverage tool to ignore them using a comment that has a special form:
  `# pragma: no cover`
- When we run the coverage tool again things look a little bit better.

24. Code not worth covering

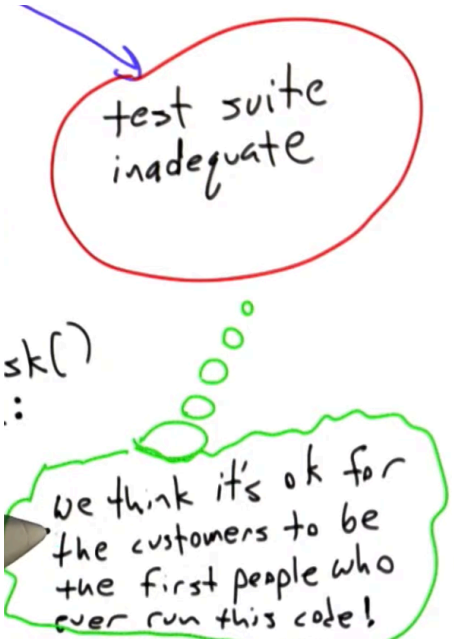
- **The second case** is the code that we believe to be feasible but which isn't worth covering because it's very hard to trigger and it's very simple.



•As an example, let's consider res variable the result of the command to format a disc:

•Might be the case that we lack an appropriate fault injection tool that will let us easily simulate the failure of a format disc call. Furthermore, the abort code, which is going to terminate the entire application, is presumably something that was tested elsewhere.

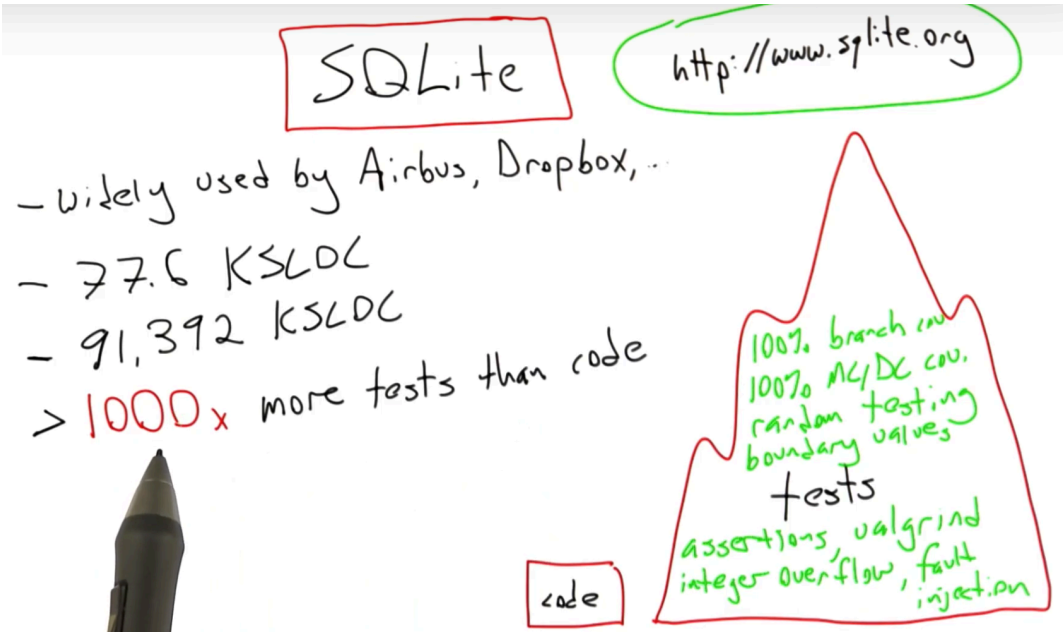
25. Inadequate test suite



- **The third reason t**he code might not have been covered is that the **test suite** might simply just be **inadequate**, and in that case we have to decide what to do about it.
- One option of course is to improve the test suites so that the code gets covered. Since it can be difficult, we might decide to ship our software without achieving a 100% code coverage, because like all forms of testing, is basically just a cost-benefit tradeoff.

26. Sqlite

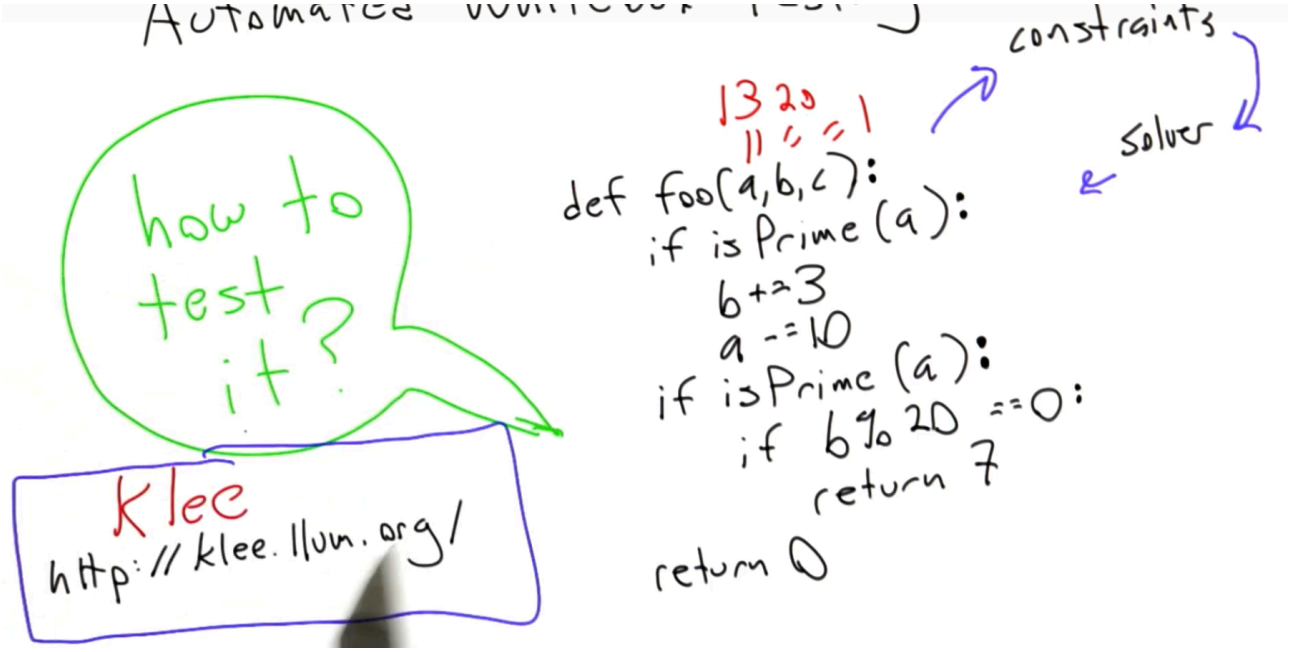
- An example of testing done right is an open source database called SQLite. This small open source database is designed to be easily embeddable in other applications.
- It's very widely used by companies like Airbus, Dropbox, in various Apple products, Android phones, etc.



- Almost every single testing technique presented in this lesson and many of the coverage metrics have been applied to SQLite, and the result is generally, a really solid piece of software.

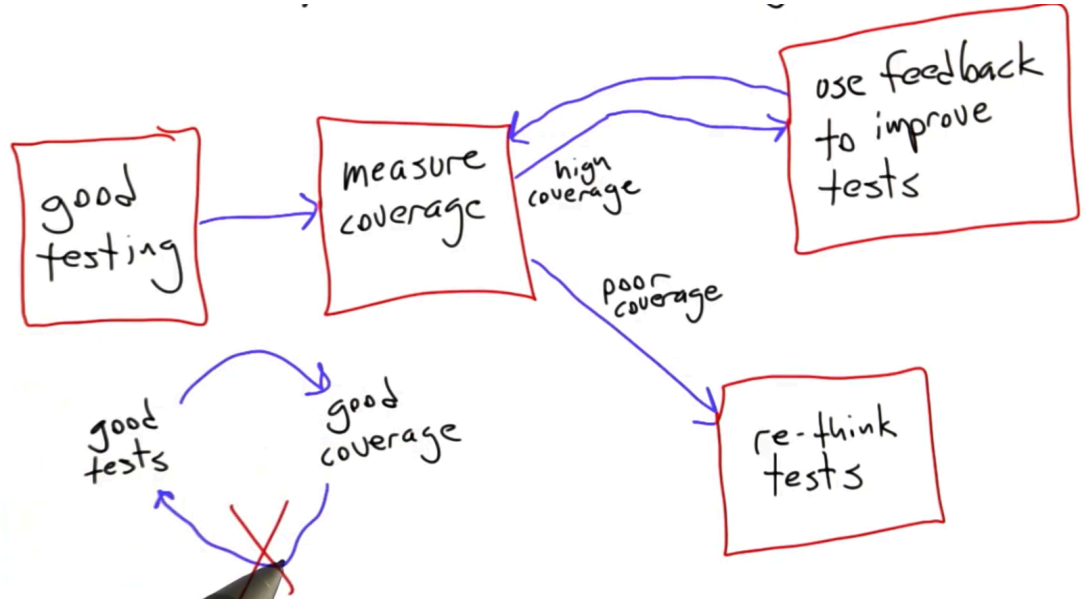
27. Automated white box testing

- Is not a form of code coverage but what rather a way to get software tools to automatically generate test for our code.



- The goal for this tool is to generate good path coverage for the code.
- Therefore, a tool will iterate the process of generating inputs that take different branches multiple times and then using what it learned about the code build up a set of constraints to explore different paths, pass it to the solver and the solver is either going to succeed in coming up with a new value or possibly it will fail.
- There are no automated whitebox testing tools for Python.
- For C programs there is a tool called Klee.
- Microsoft uses these techniques of automatically generating good test inputs for finding a very large number of bugs in real products.

28. How to use coverage



- We strongly believe that if we have a good test suite, and we measure it's coverage, the coverage will be good. We do not believe, on the other hand, that if we have a test suite which gets good coverage, it must be a good test suite.
- Used in the right way, coverage can be a relatively low cost way To improve the testing that we do for a piece of software.
- Used incorrectly, it can waste our time, and perhaps worst, lead to a false sense of security.