Lecture 5 - Testing

Our Sample Chunk of Code

main.py

```
# Step 5 - with function and a test.

import mi_to_km

print ( "Enter Miles" )

miles_str = input()
miles = int(miles_str)

km = mi_to_km.mi_to_km(miles)

print ( "km = {}".format(km) )
```

mi_to_km.py

```
# mi_to_km converts from miles as an integer or float to kilometers.
def mi_to_km ( mi ):
    conv = 1.60934
    km = mi * conv
    return (km)

# Automated Test
if __name__ == "__main__":
    n_err = 0
    x = mi_to_km ( 3 )
    if x !=  4.82802:
        n_err = n_err + 1
        print ( "Error: Test 1: conversion not working, expected {} got {}".format (  4
    x = mi_to_km ( 0 )
    if x != 0:
        n_err = n_err + 1
        print ( "Error: Test 2: conversion not working, expected {} got {}".format ( 0,

    if n_err == 0 :
        print ( "PASS" )
    else:
        print ( "FAILED" )
```

Importance of Testing

Deadly radiation therapy

The Therac-25 medical radiation therapy equipment was involved in several cases where massive overdoses, often 100x the expected amount, of radiation was administered to patients. This killed at least 3 people.

This was not a one-time event. IN 2000 in Parama City Florida a set of “planning” software had an error that resulted in a large set of patients receiving excessive rations. Al teat 5 died.

In both cases tests did not check for all possible inputs and these should have been prevented with unit tests.

Crashed into Mars

The Mars Climate Orbiter had a contractor that used English units in calculating the distance from the orbiter to the surface of mars. The other software on the system used metric. A mile is not a kilometer and the MRO mis-calculated how close to the surface it was. Boom. Years lost and $327 million incest.

The broad picture is that you need to test entire systems not just chunks individually.

Knightb $440 Million Oops...

Knight, a middle layer vendor in stock trading, attempted to upgrade the system - without testing the software upgrade process. They failed to remove the old software when they installed the new. So the old software kept re-issuing trades when the trade had already taken place and they lost $440 million in under 30 minutes. This killed the company.

That is exciting - Loosing $440 million in $\frac{1}{2}$ hour!

Airbus Flight A333, China airlines

“Incident: China Airlines A333 at Taipei on Jun 14th 2020, all primary computers, reversers and autobrakes failed on touchdown

“A China Airlines Airbus A330-300, registration B-18302 performing flight CI-202 from Shanghai Pudong (China) to Taipei Songshan (Taiwan) with 87 passengers and 11 crew, landed on Songshan’s wet runway 10, when upon touchdown all three primary flight computers, thrust reversers and autobrake systems failed affecting the stopping distance of the aircraft. The crew applied maximum manual braking and managed to stop the aircraft 10 meters/33 feet ahead of the runway end …”

From: https://avherald.com/h?article=4d97ca46&opt=0, The Aviation Herald, Sep 4, 2021

Unit Tests

Our Pattern is:

1. Call some set of code
2. Know an expected result
3. Compare what we “got” from the call with “expected” values.
4. If they don’t match - then report an error and count up the number of errors
5. Do more tests… (go back to step 1)
6. When all tests are complete, then report a cumulative result (PASS or FAIL)

Testing With State

Suppose you have a pair of functions:

```
it = 0

def set_it(newit):
    global it
    it = newit

def double_it():
    it = it * 2
    return it
```

And we want to test these functions.

First thin is we have to test them together. Individually makes no rational sense.

Second Our test cases have to create the “state” of the variable “it” first, then use the other function to see if it works.

```
it = 0

def set_it(newit):
    global it
    it = newit

def double_it():
    it = it * 2
    return it


# Automated Test
if __name__ == "__main__":
    n_err = 0
    x = mi_to_km ( 3 )
    set_it(5)
    x = double_it()
    if x !=  10:
        n_err = n_err + 1
        print ( "Error: Test 1: conversion not working, expected {} got {}".format (  1

    # Now 2nd test is dependent on 1st test
    x = double_it()
    if x != 20:
        n_err = n_err + 1
        print ( "Error: Test 2: conversion not working, expected {} got {}".format ( 20

    if n_err == 0 :
        print ( "PASS" )
    else:
        print ( "FAILED" )
```

This is going to be really important when we work with things that persist state formation like Classes and files.

Pure Functional Testing

As long as you build code that takes a set of input, transforms it and then returns it - this is a “pure functional” set of code. It is much easier to test and validate the code. Our “hypotenuse” function is like this. Our conversion functions are like this.

Integration Testing

This is when you test complete systems all at once. You are not testing a function - you are testing the entire system. People that work in quality assurance (QA) are doing Integration Testing. This is it's one entire field of work in computer science.

Code Review

SO... With testing being a high-stakes difficult thing. What can we do. Can we get computers to just do tall the testing for us?

The answer is no - they are not good at testing - the really can only test what we provide them with as tests.

What about People - are they any good at testing.

Microsoft did extensive research into this. The book that they published on the subject, "Code Complete" is now about 20 years old but they have subsequently published regular papers showing that in the world of testing - not much has changed.

Unit testing finds about 23% of retros.

Integration testing finds about 22% - but the problems overlap with unit testing.

Code Review - this is were you have a 2nd developer go and read through all of your code - line by line - asking questions, developing additional tests, asking why things are the way they are - this finds 67% of errors.

Beta user groups - this is sending your code to a small set of users to actually run it will find 89% of errors. But this is usually after you have already gotten the code ready to ship to customers - so it is very expensive.

The best, least expensive is code review. But this is usually after you have already gotten the code ready to ship to customers - so it is very expensive.

The best, least expensive is code review.

Copyright