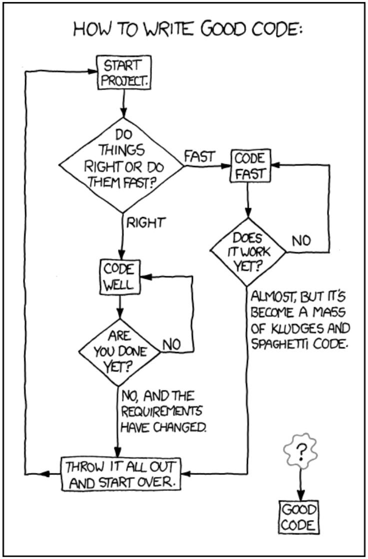
INFO1110 & COMP9001: Introduction to Programming

School of Information Technologies, University of Sydney

Copyright Warning

Lecture 13: Testing

Types of testing, what needs testing, assertions, unit tests


HOW TO WRITE GOOD CODE:
START PROJECT.
DO THINGS RIGHT OR DO THEM FAST?
FAST
CODE FAST
DOES IT WORK YET?
NO
ALMOST, BUT IT'S BECOME A MASS OF KLUDGES AND SPAGHETTI CODE.
RIGHT
CODE WELL.
ARE YOU DONE YET?
NO
NO, AND THE REQUIREMENTS HAVE CHANGED.
THROW IT ALL OUT AND START OVER.
?
GOOD CODE

Source: xkcd

It's not enough to be "confident" when you write your code that it just works: you have to *test* it.

Untested software is unreliable and often fails

Tested software can still be unreliable, but seldom fails

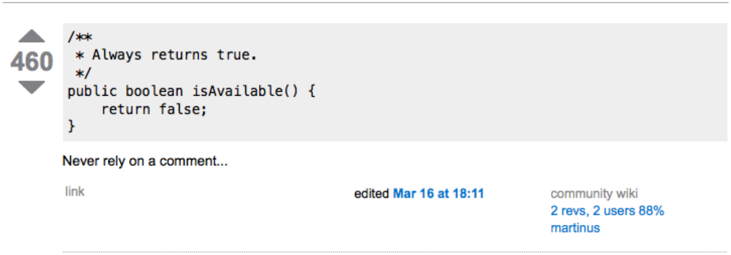
Thoroughly tested software is reliable and very rarely fails

Critical software systems such as (air-)traffic control, medical monitoring, construction, demand thorough testing. Testing is more significant than the code itself.

Documentation doesn't always help

```
# returns 1
return 1
```

or


460

```
/**
 * Always returns true.
 */
public boolean isAvailable() {
    return false;
}
```

Never rely on a comment...

link

edited Mar 16 at 18:11

community wiki
2 revs, 2 users 88%
martinus

And just in case you needed more motivation

Microsoft employs at least as many testers as developers.

Just because it compiles…

“I’ve only got three compile errors: I’ll be finished soon.”
“It compiled with no errors! I’m done’

Compilation isn’t any kind of guarantee of correctness.

To test, we need need to inspect the contents of the memory

To test, we need see if the values of what we compute matches with what we expect

Tracing Code

It is a very good exercise in working out, if something is wrong, what that problem is by *tracing through the code* itself: you can write out the state (value) of variables as you go through loops and if/elses [1]

This becomes time consuming with large amounts of code, however,

- the chance of identifying a problem is still very high
- the chance of making a mistake in code tracing (>1000 lines of code) also increases if not careful

[1] a desk check!

What is a test?

If testing isn't just compiling, what *is* it?

A test gets some input to (part of) a program and then investigates the output.

Testing "`HelloWorld.py`" is easy:

input: `python HelloWorld.py` (or in fact, nothing)

output: "Hello, World!"

What is a test?

If testing isn't just compiling, what *is* it?

A test gets some input to (part of) a program and then investigates the output.

Testing "`HelloWorld.py`" is easy:

input: `python HelloWorld.py` (or in fact, nothing)

output: "Hello, World!"

Testing other things is more involved: e.g., testing a *squareRoot* function:

input: 4

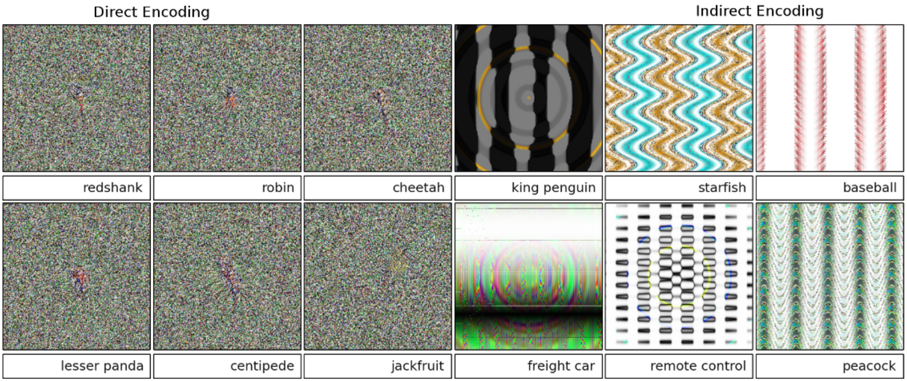
output: 2

Now how do you test that the answer given (2) is the (positive) square root of 4? Let's look at testing some simple functions next.

Oracles

An *oracle* is some external “thing” that we can ask for the *right* answer. We don’t always have this luxury.

When we do, we can check that our answer is the same as that of the oracle.



state-of-the-art DNNs trained on ImageNet believe with >= 99.6% certainty to be a familiar object.

Test-driven development

This is a way of software development that requires you to write the tests for a given function *first*, before you even write the function!

This has a number of advantages: it makes you think exactly what you want your function(s) to do in terms of

1. what the normal functionality should be
2. what should happen given “bad” input (e.g., passing a negative number to a function that expects positive numbers)

Failing a test is a *good thing*

It doesn't mean you've done a bad job coding (unless you never learn how to fix the code!)

1. It means you've written a test
2. It means you've found an error
3. It even helps you *fix* the error

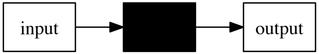
The test has brought your attention to a specific case where things should work.

The time spent on making the test, for a small part of the program, is much less than bug hunting the entire program.

Different flavours of testing

There are several approaches to testing.

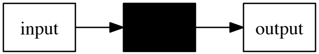
One is to treat the program as a kind of "black box" which simply processes input and gives output, with no reference to how it works. . . This "black box testing" is often quick and powerful.

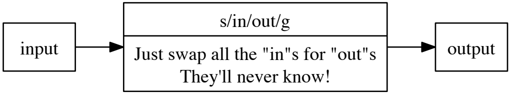


Different flavours of testing

There are several approaches to testing.

One is to treat the program as a kind of “black box” which simply processes input and gives output, with no reference to how it works… This “black box testing” is often quick and powerful.





Alternatively we can look inside the box to see what’s going on. This is known as “white box testing”

Black & White Box Testing

With black box testing you don't look inside each function: you just look at the input and output.

With white box testing you look inside the function in detail. This enables you to test different kinds of input and therefore different *execution paths*.

Testing by printing

This is a very common function and also has its (limited) use:

Print out a running log of what's going on, while you're developing code, and print out error messages if things go wrong.

This is fine up to a point but *don't do it all the time!*

After finding the problems, you have to go back and comment out all those print statements, essentially all your checking/testing is never used again.

It's *far,* ***far*** better to have your testing separate from the rest of the code!

Tests are better if they're automated

I once read a developer (of some commercial software) state that the way he tests his code is to write it and then try as hard as he could to "break" it, **manually**.

His view was that as he should know the code best, he'd also know what were the most likely problems.

In general tests should be **automated**. This is so they can be *repeated*, because if it can't be repeated easily, how can you tell that a problem has been fixed?

Regression testing

Every time you fix something, you should re-run all your *previous* tests.

Suppose you have some code that calculates, say, the determinant of a matrix.
You have a test that works for a very simple matrix, which is 2×2, like this:

$$M = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

The determinant of this matrix is given by $det(M) = ad - bc$.

When you expand your code to cope with bigger matrices, you should *definitely* test that finding the determinant of the simple 2×2 matrix still works!

Regression testing (cont.)

Regression Testing is running ALL your tests again whenever you make a significant change to your code, to make sure you haven't broken code that used to work.

Weird input

What if there are some *weird* inputs?

For example, what should happen when

- an empty list is given to a sorting function
- a negative number is given to a `log` function?
- a `None` is given to a print function?
- an index of `-1` is given as the index of an item in an array?

Your code should cope sensibly with such things, and your tests should test that it does!

Corners and Edges

In engineering terms, an *edge* case is where one dimension of a problem is at an extreme, for instance when the load on a bridge is maximum.

A *corner case* is where more than one dimension is at an extreme, such as when a bridge has maximum load *and* it's being hit with highest possible winds *and* it's in an earthquake.

Design your code to cope with corners and edges! ☺

Test every bit

It should be obvious that you should test every possible part of your code, but how will you do that?

If you have a test for every *line* of your code, then you have great *coverage* (coverage is just the proportion of the code that is tested) — but that doesn't necessarily test every way in which your code will work.

Code coverage in testing is the proportion of (lines of) code that you have tested.

You need to think about each possible *execution path*

Execution paths

An execution path is a route through the flow of the program.

For example, if you have an “if (a == 1)” statement, one execution path happens if a is 1, and another execution path happens if a is not 1.

For example, if you have a loop, the execution path is either 1) enter the loop, 2) once inside, perform another iteration of the loop, 3) stop the loop.

```
while conditionA: // ??

   if conditionB: //  ??
       break
```

Assertions

An *assertion* is a condition (Boolean expression) written at some point in a program

We assert that the condition must be true whenever the program reaches that point

```
import random
x = random.randint(0, 5)
if x < 2:
    x = 2

# at this point in the code, x is expected to be at least value 2
# in other words, it should not be less than 2
if x < 2:
    stop everything! something is seriously wrong

# more specifically, x should be in the range [2, 5)
```

Assertions are designed to **stop execution**.

The expectation of what programmer knows to be true at this point does not match with the state of the program!

Assertions (cont.)

Assume there was mismatch of expectations

```

def get_percentage( value, largest_value ):
    ''' guarantees to return a float type that is within the range c
        to represent 0% to 100%'''

    return value / largest_value


percent = get_percentage( 13, 71 )

assert percent >= 0 and percent <= 1
```

Assertions in Python are handled as Exceptions. You can catch them, and then decide what you could possibly do in that situation of unexpected values. (it may not make sense to use assertion)

Assertions (cont.)

What if this were on a different range scale? instead of $[0, max] \rightarrow [0, 1]$ it is $[min, max] \rightarrow [0, 1]$?

By keeping these high-risk foods under 5 degrees Celsius it stops them from entering the 'danger-zone' —temperatures between 5 degrees Celsius and 60 degrees Celsius.

```

def get_percentage( value, largest_value ):
    ''' guarantees to return a float type that is within the range c
       to represent 0% to 100%'''
    return value / largest_value

percent = get_percentage( -1, 5 )
# OR
# percent = get_percentage( -10, 5 )

assert percent >= 0 and percent <= 1
```

Precondition and postcondition

Preconditions and postconditions are *assertions* placed before and after some piece of code

```
# Pre : x >= 0
y = math.sqrt(x)
# Post : y >= 0
```

The precondition asserts that x is non-negative at that point, which is needed to ensure that the sqrt function will work properly;

The postcondition asserts that y must equal $\sqrt{x}$ afterwards, which is also true since otherwise something must be wrong with the *sqrt* function.

These communicate the idea that: this code will *always* produce a true postcondition if the precondition is true

Testing a returned String

In many cases, e.g., "Hello World", you return a String object. It's easy to test whether a function returns the right string: we can just use compare the output string with what expect:

```python
def greet():
      return "Hello, World!"

def test_greet():
      if greet() == "Hello, World!":
          return True
      return False
```

which is a little clunky but does the job. Here we are testing to see whether

the answer is correct and returning True if (and only if) it is.

Testing a returned String (cont.)

clean this up!

```python
def greet():
      return "Hello, World!"

def test_greet():
   # actual running of real code, storing the "output"
   output = greet()

   # what we expected running of real code
   expected = "Hello, World!"

   # figure out how to measure output vs expected
   # this could be very complicated, or very simple
   passed = (output == expected)

   # final outcome
   if passed:
      return True
   return False
```

Testing a returned String (cont.)

Another way is to throw an exception if the test is not passed.

```python
    ...
    # final outcome
    if not passed:
        raise SomeSpecificException("Greet is not working!")
```

An Error is like an Exception in that it is throwable and it is handled and dealt with in a similar way

```python
import traceback
try:
    test_greet()
except SomeSpecificException as e:
    print(e)
    traceback.print_exc()
```

Testing a returned String (cont.)

Use assertion as the exception

```
    ...
    # final outcome
    assert passed
```

```
import traceback
try:
    test_greet()
except AssertionError as e:
    print(e)
    traceback.print_exc()
```