COMP1730/COMP6730
Programming for Scientists

Testing and Debugging.

Overview

* Testing
* Debugging
* Defensive Programming

Overview of testing

* There are many different types of testing - load testing, integration testing, user experience testing, etc.
* We are concerned with *unit-testing* or functional testing.
* Usually done at the function (or method level).
* Done by calling a function with specified parameters and checking that the return value is as expected.

Unit-testing in Python

* There are many ways to do unit-testing in Python. The simplest is Python's `unittest` module.

```
import unittest

class Test(unittest.TestCase):
    def test_function_a(self):
        self.assertEquals(function_a(
            x, y, z), expected_value)
```

Test methods

* `assertEquals(a, b)`: Test whether expression `a` and `b` are equal.
* `assertTrue(a)`: Test whether expression `a` is True.
* `assertIsNone(a)`: Test whether expression `a` evaluates to `None`.
* `assertIn(x, xs)`: Test whether `x` is an element in collection `xs`.

Tips for unit-testing

* Have your tests in a separate file.
* A small function is easier to test than a large function.
* A function that only does one thing is easier to test than a function that does many things.
* Unit-testing is only concerned with the outputs of a function (and occasionally side-effects). Don't try and test *how* a function does its thing.
* Especially true when testing class methods (not really covered in this course).

The Debugging Process

1. Detection - realising you have a bug.
2. Isolation - narrowing down the cause.
3. Comprehension - Working out what went wrong.
4. Correction - Fixing the problem.
5. Prevention - Making sure the bug can't happen again.
6. Go back to step *1*.

Syntax Errors

* The code is not valid python code.
* These are usually the easiest type to find because you can't run the code until you resolve them.
* Python usually tells you where they are (approximately).

Runtime Errors

* The code is valid Python code - but it's being used to do something Python doesn't know how to do.
* Causes an *exception* when run (possibly only under certain conditions).
* Learn to read (and understand) Python's error messages. `ZeroDivisionError` is largely self-explanatory, but understand what causes Python to raise an `AttributeError`.

Semantic Errors (Logic Errors)

* The code is valid Python code and runs without error, it just does the wrong thing (possibly only sometimes).
* To detect these type of bugs, you must have a good idea of what the code is *supposed* to do.
* These are usually the hardest type of bug to detect and fix, particularly if they only occur under certain conditions.

Working out what went wrong

* Work back from where a bug appears (e.g. the line number for an error message).
* Run the code with simpler inputs that still exhibit the bug, e.g. only use the first few records in a data set.
* Add print statements to view the state of variables.
* Use unit-tests to rule out functions that are working correctly. Be careful though, since if the bug only occurs under certain conditions, these conditions need to be tested in the unit-test suite.

Some Common Bugs

* Logical operations are not English.
* Loop condition is not modified.
* Off-by-one errors.
* Floating point precision.
* `is` vs `==`.

Defensive Programming

Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?

Brian Kernighan

Code Quality Matters!

* A function that is hard to read is hard to debug.

```
def AbC(ABc):
    ABC = len(ABc)
    ABc = ABc[ABC-1:-ABC-1:-1]
    if ABC == 0:
        return 0
    abC = AbC(ABc[-ABC:ABC-1:])
    if ABc[-ABC] < 0:
        abC += ABc[len(ABc)-ABC]
    return abC
```

Pre and Post Conditions

* Functions allow for breaking larger programs into small pieces which can be separately tested and debugged.
* `assert` statements allow us to ensure that only appropriate parameters are passed as arguments to functions.
  Example: `assert type(param_a) == int and param_a > 0`
* *Unit tests* allow us to verify that the function is returning the appropriate value for the given inputs.

Explicit vs Implicit

* Make things explicit if they are unclear or could be confusing. Even if they are working as intended.
* `return None` is better than no return statement.
* `- (2 ** 2)` instead of `- 2 ** 2`.
* `(a and b) or c` instead of `a and b or c`.
* `dict()` instead of `{ }`.

Avoid Language Tricks

* Don’t make use of language quirks in your code.
* Example: operator chaining.

```
>>> 1 == 2
False
>>> False is not True
True
>>> 1 == 2 is not True
???
```