COMP1730/COMP6730
Programming for Scientists

Testing and Defensive Programming.

Overview

* Testing
* Defensive Programming

Overview of testing

* There are many different types of testing - load testing, integration testing, user experience testing, etc.
* Different software systems have different testing requirements, based on:
  - Consequences of failure
  - Complexity of software
  - Frequency of use
  - Hardware and user interactions
* Even for critical, commercially developed software, testing gives no guarantees - e.g. Boing Max crashes and Mars Climate Orbiter.

Unit-Testing

* 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.
* We usually want to focus on *edge-cases*.

The `assert` Statement

* Basic usage:

```
assert boolean expression, message
```

* If the expression is `True` execution continues.
* If the expression is `False` an `AssertionError` is raised, execution stops and the message is printed.
* Can be used to intentially cause a run-time error if assumptions are violated.

Unit-testing in Python

* There are many ways to do unit-testing in Python. We are using the `pytest` module, which makes use of `assert` statements.

```
import pytest

def test_is_factor():
    assert is_factor(8, 4) == True
    assert is_factor(7, 4) == False
```

Identifying Edge-Cases

* A lot of the hardest to find bugs only occur under certain conditions or inputs, we often call these *edge-cases*.
* Typical numerical edge-cases
  - 0, very close to 0, very large or very small numbers, largest valid input.
  - Inputs that cause intermediate values to be 0
* Other examples: empty sequences, repeated values, x and y swapped around, etc.
* Don’t write unit tests for invalid inputs unless testing error handling.

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).

Other Testing Considerations

* Floating point precision
* Random numbers (use a *seed* to get reproducable results).
* User input (isolate the user input to a function and simulate input).
* Only use your code to generate tests for refactoring purposes, not for testing correctness.
* **Testing only guarantees your code works for the test cases!**

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
???
```

Mutable Default Arguments

* Syntactically valid but lead to hard to find bugs.

```
def fun_A(x, new_list = []):
    new_list.append(x)
    return [element * x for element
        in new_list]


a = [1, 2, 3]
print(fun_A(5))
print(fun_A(3, a))
print(fun_A(5))
```