Testing

UW CSE 160

Winter 2016

Testing

- Programming to analyze data is powerful
- It's useless (or worse!) if the results are not correct
- **Correctness is far more important than speed**

Famous examples

- Ariane 5 rocket
- Therac-25 radiation therapy machine

Testing does not *prove* correctness

- Edsger Dijkstra: “Program testing can be used to show the presence of bugs, but never to show their absence!”

- Testing can only increase our confidence in program correctness.

Testing your <u>program</u>

- How do you know your <u>**program**</u> is right?
  - Compare its output to a correct output
- How do you know a correct output?
  - Real data is big
  - You wrote a computer program because it is not convenient to compute it by hand
- Use small inputs so you can compute the expected output by hand
  - We did this in HW2 and HW3 with small data sets

Testing ≠ debugging

- **Testing**: determining whether your program is correct
  - Doesn't say where or how your program is incorrect
- **Debugging**: locating the specific defect in your program, and fixing it

  2 key ideas:
  - divide and conquer
  - the scientific method

Testing parts of your program

- Often called “unit testing”
- Testing that the output of individual functions is correct.

What is a test?

- A test consists of:
  - an input (sometimes called "test data")
  - expected output
- Example test for sum:
  - input: [1, 2, 3]
  - expected output: result is 6
  - write the test as: `sum([1, 2, 3]) == 6`
- Example test for sqrt:
  - input: 3.14
  - expected output: result is within 0.00001 of 1.772
  - ways to write the test:
    - `sqrt(3.14) – 1.772 < 0.00001 and sqrt(3.14) – 1.772 > -0.00001`
    - `-0.00001 < sqrt(3.14) – 1.772 < 0.00001`
    - `math.abs(sqrt(3.14) – 1.772) < 0.00001`

Test results

- The test passes if the boolean expression evaluates to **`True`**
- The test fails if the boolean expression evaluates to **`False`**
- Use the **`assert`** statement:

```
assert sum([1, 2, 3]) == 6
assert math.abs(sqrt(3.14) - 1.772) < 0.00001
```

- **`assert True`** does nothing
- **`assert False`** crashes the program
  - and prints a message

Where to write test cases

- At the **top level**: is run every time you load your program

```
def hypotenuse(a, b):
  … body of hypotenuse …
assert hypotenuse(3, 4) == 5
assert hypotenuse(5, 12) == 13
```

(As in HW 4)

- In a **test function**: is run when you invoke the function

```
def hypotenuse(a, b):
  … body of hypotenuse …
def test_hypotenuse():
  assert hypotenuse(3, 4) == 5
  assert hypotenuse(5, 12) == 1
```

(As in HW 3 and HW5)

Assertions are not just for test cases

- Use assertions throughout your code
- Documents what you think is true about your algorithm
- Lets you know immediately when something goes wrong
  - The longer between a code mistake and the programmer noticing, the harder it is to debug

Assertions make debugging easier

- Common, but unfortunate, course of events:
  - Code contains a mistake (incorrect assumption or algorithm)
  - Intermediate value (e.g., in local variable, or result of a function call) is incorrect
  - That value is used in other computations, or copied into other variables
  - Eventually, the user notices that the overall program produces a wrong result
  - Where is the mistake in the program? It could be anywhere.
- Suppose you had 10 assertions evenly distributed in your code
  - When one fails, you can localize the mistake to 1/10 of your code (the part between the last assertion that passes and the first one that fails)

Where to write assertions

- **Function entry**: are arguments of expected type/size/value/shape?
  - Place blame on the caller before the function fails
- **Function exit**: is result correct?
- Places with tricky or interesting code
- Assertions are ordinary statements; e.g., can appear within a loop:

```
for n in myNumbers:
  assert type(n) == int or type(n) == float
```

Where *not* to write assertions

- Don’t clutter the code
  - (Same rule as for comments)
- Don’t write assertions that are certain to succeed
  - The existence of an assertion tells a programmer that it might possibly fail
- Don’t need to write an assertion if the following code would fail informatively:

```
assert type(name) == str
print "Hello, " + name
```

- Write assertions where they may be useful for debugging

What to write assertions about

- Results of computations
- Correctly-formed data structures

```
assert 0 <= index < len(mylist)
assert len(list1) == len(list2)
```

When to write tests

- Two possibilities:
  - Write code first, then write tests
  - Write tests first, then write code
- It's best to write tests first

- If you write the code first, you remember the implementation while writing the tests
  - You are likely to make the same mistakes that you made in the implementation (e.g. assuming that negative values would never be present)
- If you write the tests first, you will think more about the functionality than about a particular implementation
  - You might notice some aspect of behavior that you would have made a mistake about, some special case of input that you would have forgotten to handle

Write the whole test

- A common **mistake**:
  1. Write the function
  2. Make up test **inputs**
  3. Run the function
  4. Use the result as the expected output – BAD!!
- You didn't write a full test: only half of a test!
  – Created the tests inputs, but not the expected output
- The test does not determine whether the function is correct
  – Only determines that it continues to be as correct (or incorrect) as it was before

Tests outside of function body are for behavior described in the specification

```
def roots(a, b, c):
  """Returns a list of the two roots of ax**2 + bx + c."""
  ... Body of roots S…
```

Tests *implementation-specific* behavior outside of function body: (BAD)

```
assert roots(1, 0, -1) == [-1, 1]
```

- Does the **specification** imply that this should be the *order* these two roots are returned?
- Assertions inside a routine can be used for implementation-specific behavior

Tests prevent you from introducing errors when you modify a function body

- **Abstraction**: the implementation details do not matter
- As long as the specification of the function remains the same, tests of the external behavior of the function should still apply.

- Preventing introducing errors when you make a change is called "regression testing"

Coming up with good test cases

- Think about and test “corner cases”
  - Numbers:
  - Lists:

Coming up with good test cases

- Think about and test "corner cases"
  - Numbers:
    - int vs. float values (remember not to test for equality with floats)
    - Zero
    - Negative values
  - Lists:
    - Empty list
    - Lists containing duplicate values (including all the same value)
    - Lists in ascending order/descending order
    - Mix of types in list (if specification does not rule out)

Testing Approaches

- **Black box testing** - Choose test data ***without*** looking at implementation, just test behavior mentioned in the specification

- **Glass box** (white box, clear box) **testing** - Choose test data ***with*** knowledge of implementation (e.g. test that all paths through your code are exercised and correct)

```
def isPrime(x):
    """Assumes x is a nonnegative int
    Returns True if x is prime; False otherwise"""
if x <= 2:
    return False
for i in range(2, x):
    if x % i == 0:
        return False
return True
```

Tests might not reveal an error

```
def mean(numbers):
  """Returns the average of the argument list.
     The argument must be a non-empty list of numbers."""
  return sum(numbers)/len(numbers)
# Tests
assert mean([1, 2, 3, 4, 5]) == 3
assert mean([1, 2.1, 3.2]) == 2.1
```

This implementation is elegant, but wrong!

```
mean([1,2,3,4])
```

Don’t write meaningless tests

```
def mean(numbers):
  """Returns the average of the argument list.
     The argument must be a non-empty list of numbers."""
  return sum(numbers)/len(numbers)
```

Unnecessary tests. Don’t write these:

```
mean([1, 2, "hello"])
mean("hello")
mean([])
```