

# Testing

BBM 101 - Introduction to Programming I

Hacettepe University  
Fall 2015

Fuat Akal, Aykut Erdem, Erkut Erdem, Vahid Garousi

# Testing

- Programming to analyze data is powerful
- It is useless if the results are not correct
- **Correctness is far more important than speed**

# Famous Examples

- Ariane 5 rocket
  - On June 4, 1996, the maiden flight of the European Ariane 5 launcher crashed about 40 seconds after takeoff.
  - Media reports indicated that the amount lost was half a billion dollars
  - The explosion was the result of a software error
- Therac-25 radiation therapy machine
  - In 1985 a Canadian-built radiation-treatment device began blasting holes through patients' bodies.

# 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 = Double-Checking Results

- How do you know your program 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 by hand
- Example: standard deviation
  - What are good tests for `std_dev`?

$$s = \sqrt{\frac{1}{N-1} \sum_{i=1}^N (x_i - \bar{x})^2},$$

# 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

# What is a Test?

- A test consists of:
  - an **input** (sometimes called “test data”)
  - an **oracle** (a predicate (boolean expression) of the output)
- Example test for **sum**:
  - input: [1, 2, 3]
  - oracle: result is 6
  - write the test as: `sum([1, 2, 3]) == 6`
- Example test for **sqrt**:
  - input: 3.14
  - oracle: 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 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):  
    ...  
    assert hypotenuse(3, 4) == 5  
    assert hypotenuse(5, 12) == 13
```

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

```
def hypotenuse(a, b):  
    ...  
def test_hypotenuse():  
    assert hypotenuse(3, 4) == 5  
    assert hypotenuse(5, 12) == 13
```

# 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., 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 legal?
  - 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 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
- If you write the **code first**, you remember the implementation while writing the tests
  - You are likely to make the same mistakes in the implementation
- 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
  - This is the better choice

# 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 oracle
- You didn't write a test, but only half of a test
  - Created the tests inputs, but not the oracle
- 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



# Testing Approaches

- **Black box testing** - Choose test data *without* looking at implementation
- **Glass box (white box, clear box) testing** - Choose test data *with* knowledge of implementation

# Inside Knowledge might be Nice

- Assume the code below:

```
c = a + b
if c > 100
    print "Tested"
print "Passed"
```

- Creating a test case with a=40 and b=70 is not enough
  - Although every line of the code will be executed
- Another test case with a=40 and b=30 would complete the test

# Tests might not Reveal an Error

## Sometimes

```
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])` → **would return 2!!!**

# Last but not Least, 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([])
```