Lecture 5

Specifications & Testing
Announcements For This Lecture

**Readings**
- See link on website:
  - Docstrings in Python
  - Material is not in Text

**Assignment 1**
- Posted on web page
  - Due Thu, Sep. 17th
  - Revise until correct

**Today’s Lab**
- Practice today’s lecture
- **Preparation for the first Assignment**
- Can work in pairs
  - One submission for pair
  - Meet people in your lab
- **Consultants can help**
One-on-One Sessions

• Starting tomorrow: 1/2-hour one-on-one sessions
  ▪ Bring computer to work with instructor, TA or consultant
  ▪ Hands on, dedicated help with Lab 2 and/or Lab 3
  ▪ To prepare for assignment, **not for help on assignment**

• **Limited availability:** we cannot get to everyone
  ▪ Students with experience or confidence should hold back

• Sign up online in CMS: first come, first served
  ▪ Choose assignment One-on-One
  ▪ Pick a time that works for you; will add slots as possible
  ▪ Can sign up starting at 1pm **TODAY**
Recall: The Python API

math.ceil(x)

Return the ceiling of x as a float, the smallest integer value greater than or equal to x.
Recall: The Python API

Function name: `math.ceil(x)`
Number of arguments: 1
What the function evaluates to:
Return the ceiling of `x` as a float, the smallest integer value greater than or equal to `x`.

- **This is a specification**
  - Enough info to use `func`.
  - But not how to implement.
- **Write them as docstrings**
**Anatomy of a Specification**

```python
def greet(n):
    """Prints a greeting to the name n
    Greeting has format 'Hello <n>!'
    Followed by conversation starter.
    Parameter n: person to greet
    Precondition: n is a string""
    print 'Hello '+n+'!'  
    print 'How are you?'
```

One line description, followed by blank line

---

9/8/15 Specifications & Testing 6
def greet(n):

    """Prints a greeting to the name n
    Greeting has format 'Hello <n>!'
    Followed by conversation starter.

    Parameter n: person to greet
    Precondition: n is a string"

    print 'Hello '+n+'!'
    print 'How are you?'
def greet(n):
    """Prints a greeting to the name n
    Greeting has format 'Hello <n>!'
    Followed by conversation starter.
    Parameter n: person to greet
    Precondition: n is a string"
    print 'Hello '+n+'!
    print 'How are you?"""
def greet(n):

"""Prints a greeting to the name n
Greeting has format 'Hello <n>!' 
Followed by conversation starter.

Parameter n: person to greet
Precondition: n is a string"
print 'Hello '+n+'!
print 'How are you?'
Anatomy of a Specification

```python
def to_centigrade(x):
    """Returns: x converted to centigrade
    Value returned has type float.""

    Parameter x: temp in fahrenheit
    Precondition: x is a float

    return 5*(x-32)/9.0
```

One line description, followed by blank line

More detail about the function. It may be many paragraphs.

Parameter description

Precondition specifies assumptions we make about the arguments
def to_centigrade(x):
    """Returns: x converted to centigrade
    Value returned has type float."""
    Parameter x: temp in fahrenheit
    Precondition: x is a float
    return 5*(x-32)/9.0
Anatomy of a Specification

```python
def to_centigrade(x):
    """Returns: x converted to centigrade
    Value returned has type float.
    Parameter x: temp in fahrenheit
    Precondition: x is a float"""
    return 5*(x-32)/9.0
```

“Returns” indicates a fruitful function

More detail about the function. It may be many paragraphs.

Parameter description

Precondition specifies assumptions we make about the arguments
Preconditions

• Precondition is a promise
  ▪ If precondition is true, the function works
  ▪ If precondition is false, no guarantees at all

• Get software bugs when
  ▪ Function precondition is not documented properly
  ▪ Function is used in ways that violates precondition

>>> to_centigrade(32)
0.0

>>> to_centigrade(212)
100.0
Preconditions

- **Precondition is a promise**
  - If precondition is true, the function works
  - If precondition is false, no guarantees at all
- **Get software bugs when**
  - Function precondition is not documented properly
  - Function is used in ways that violates precondition

```python
>>> to_centigrade(32)
0.0
>>> to_centigrade(212)
100.0
>>> to_centigrade('32')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "temperature.py", line 19 ...
TypeError: unsupported operand type(s) for -: 'str' and 'int'
```

Precondition violated
Global Variables and Specifications

- Python *does not support* docstrings for variables
  - Only functions and modules (e.g. first docstring)
  - `help()` shows “data”, but does not describe it
- But we still need to document them
  - Use a single line comment with #
  - Describe what the variable means
- **Example:**
  - `FREEZING_C = 0.0  # temp. water freezes in C`
  - `BOILING_C = 100.0 # temp. water boils in C`
Test Cases: Finding Errors

- **Bug**: Error in a program. (Always expect them!)
- **Debugging**: Process of finding bugs and removing them.
- **Testing**: Process of analyzing, running program, looking for bugs.
- **Test case**: A set of input values, together with the expected output.

Get in the habit of writing test cases for a function from the function’s specification — even *before* writing the function’s body.

```python
def number_vowels(w):
    """Returns: number of vowels in word w.

    Precondition: w string w/ at least one letter and only letters""
    pass  # nothing here yet!
```
Test Cases: Finding Errors

- **Bug**: Error in a program. (Always expect them!)
- **Debugging**: Process of finding bugs and removing them.
- **Testing**: Process of analyzing, running program, looking for bugs.
- **Test case**: A set of input values, together with the expected output.

Get in the habit of writing test cases for a function from the function’s specification—even before writing the function’s body.

Some Test Cases

- `number_vowels('Bob')`
  Answer should be 1
- `number_vowels('Aeiuo')`
  Answer should be 5
- `number_vowels('Grrr')`
  Answer should be 0

```python
def number_vowels(w):
    """Returns: number of vowels in word w.
    Precondition: w string w/ at least one letter and only letters""
    pass  # nothing here yet!
```
Representative Tests

- Cannot test all inputs
  - “Infinite” possibilities
- Limit ourselves to tests that are representative
  - Each test is a significantly different input
  - Every possible input is similar to one chosen
- An art, not a science
  - If easy, never have bugs
  - Learn with much practice

Representative Tests for number_vowels(w)

- Word with just one vowel
  - For each possible vowel!
- Word with multiple vowels
  - Of the same vowel
  - Of different vowels
- Word with only vowels
- Word with no vowels
Running Example

• The following function has a bug:

```python
def last_name_first(n):

    """Returns: copy of <n> but in the form <last-name>, <first-name>

    Precondition: <n> is in the form <first-name> <last-name>
    with one or more blanks between the two names""

    end_first = n.find(' ')  # Find index of space before last name
    first = n[:end_first]    
    last = n[end_first+1:]   
    return last+', '+first  
```

• Representative Tests:
  - `last_name_first('Walker White')` give 'White, Walker'
  - `last_name_first('Walker White')` gives 'White, Walker'
**Running Example**

- The following function has a bug:

  ```python
def last_name_first(n):
    """Returns: copy of <n> but in the form <last-name>, <first-name>
    
    Precondition: <n> is in the form <first-name> <last-name> 
    with one or more blanks between the two names""
    
    end_first = n.find(' ') 
    first = n[:end_first] 
    last = n[end_first+1:] 
    return last+', '+first 
  ```

- Representative Tests:
  - `last_name_first('Walker White')` give 'White, Walker'
  - `last_name_first('Walker White')` gives 'White, Walker'

Look at precondition when choosing tests
Unit Test: A Special Kind of Module

• A unit test is a module that tests another module
  ▪ It imports the other module (so it can access it)
  ▪ It imports the cornelltest module (for testing)
  ▪ It defines one or more test procedures
    • Evaluate the function(s) on the test cases
    • Compare the result to the expected value
  ▪ It has special code that calls the test procedures

• The test procedures use the cornelltest function

```python
def assert_equals(expected, received):
    """Quit program if expected and received differ""
```

9/8/15 Specifications & Testing 21
# test procedure

def test_last_name_first():

    """Test procedure for last_name_first(n)"""
    result = name.last_name_first('Walker White')
    cornelltest.assert_equals('White, Walker', result)
    result = name.last_name_first('Walker White')
    cornelltest.assert_equals('White, Walker', result)

# Execution of the testing code

test_last_name_first()

print 'Module name is working correctly'
# test procedure

def test_last_name_first():
    """Test procedure for last_name_first(n)"""
    result = name.last_name_first('Walker White')
    cornelltest.assert_equals('White, Walker', result)
    result = name.last_name_first('Walker White')
    cornelltest.assert_equals('White, Walker', result)

# Execution of the testing code

test_last_name_first()
print 'Module name is working correctly'
Testing `last_name_first(n)`

```python
# test procedure
def test_last_name_first():
    """Test procedure for last_name_first(n)"""
    result = name.last_name_first('Walker White')
    cornelltest.assert_equals('White, Walker', result)
    result = name.last_name_first('Walker White')
    cornelltest.assert_equals('White, Walker', result)

# Execution of the testing code
test_last_name_first()
print 'Module name is working correctly'
```

- **Call function on test input**
- **Compare to expected output**
- **Quits Python if not equal**
- **Message will print out only if no errors.**
### Modules vs. Scripts

<table>
<thead>
<tr>
<th><strong>Module</strong></th>
<th><strong>Script</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>• Provides functions, constants</td>
<td>• Behaves like an application</td>
</tr>
<tr>
<td>▪ <strong>Example</strong>: temp.py</td>
<td>▪ <strong>Example</strong>: helloApp.py</td>
</tr>
<tr>
<td>• import it into Python</td>
<td>▪ Run it from command line</td>
</tr>
<tr>
<td>▪ In interactive shell…</td>
<td>▪ <code>python helloApp.py</code></td>
</tr>
<tr>
<td>▪ or other module</td>
<td>▪ No interactive shell</td>
</tr>
<tr>
<td>• All code is either</td>
<td>▪ <code>import</code> acts “weird”</td>
</tr>
<tr>
<td>▪ In a function definition, or</td>
<td>▪ Commands <em>outside</em> functions</td>
</tr>
<tr>
<td>▪ A variable assignment</td>
<td>▪ Does each one in order</td>
</tr>
</tbody>
</table>
Combining Modules and Scripts

- Scripts often have functions in them
  - Can we import them without “running” script?
  - Want to separate script part from module part
- New feature: `if __name__ == '__main__':`
  - Put all “script code” underneath this line
  - Also, indent all the code underneath
  - Prevents code from running if imported
- **Example**: bettertemp.py
Testing last_name_first(n)

# test procedure

def test_last_name_first():
    """Test procedure for last_name_first(n)""
    result = name.last_name_first('Walker White')
    cornelltest.assert_equals('White, Walker', result)
    result = name.last_name_first('Walker            White')
    cornelltest.assert_equals('White, Walker', result)

# Application code

if __name__ == '__main__':
    test_last_name_first()
    print 'Module name is working correctly'

Test code is properly formatted as script
Our modules consist of:

- Function definitions
- “Constants” (global vars)
- Optional script code to call/test the functions

All statements must:

- be inside of a function or
- assign a constant or
- be in the application code

import will only use the definitions, not app code

```python
# temperature.py
...
# Functions
def to_centigrade(x):
    """Returns: x converted to C"""
...
# Constants
FREEZING_C = 0.0  # temp. water freezes
...
# Application code
if __name__ == '__main__':
    assert_floats_equal(0.0, to_centigrade(32.0))
    assert_floats_equal(100, to_centigrade(212))
    assert_floats_equal(32.0, to_fahrenheit(0.0))
    assert_floats_equal(212.0, to_fahrenheit(100.0))
```
## Types of Testing

<table>
<thead>
<tr>
<th>Black Box Testing</th>
<th>White Box Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Function is “opaque”</strong></td>
<td><strong>Function is “transparent”</strong></td>
</tr>
<tr>
<td>- Test looks at what it does</td>
<td>- Tests/debugging takes place inside of function</td>
</tr>
<tr>
<td>- <strong>Fruitful</strong>: what it returns</td>
<td>- Focuses on where error is</td>
</tr>
<tr>
<td>- <strong>Procedure</strong>: what changes</td>
<td></td>
</tr>
<tr>
<td><strong>Example</strong>: Unit tests</td>
<td><strong>Example</strong>: Use of print</td>
</tr>
<tr>
<td><strong>Problems:</strong></td>
<td><strong>Problems:</strong></td>
</tr>
<tr>
<td>- Are the tests everything?</td>
<td>- Much harder to do</td>
</tr>
<tr>
<td>- What caused the error?</td>
<td>- Must remove when done</td>
</tr>
</tbody>
</table>
Finding the Error

- Unit tests cannot find the source of an error
- Idea: “Visualize” the program with print statements

```python
def last_name_first(n):
    """Returns: copy of <n> in form <last>, <first>"""
    end_first = n.find(' ')  
    print end_first  
    first = n[:end_first]  
    print 'first is '+str(first)  
    last = n[end_first+1:]  
    print 'last is '+str(last)  
    return last+', '+first
```

Print variable after each assignment

Optional: Annotate value to make it easier to identify