Anatomy of a Specification

```python
def greet(n):
    """Prints a greeting to the name n
    Greeting has format 'Hello <n>!
    Precondition: n is a string representing a person's name"
    print 'Hello ' + n + '!
```

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

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.
- 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.

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"
    first = n.find(' ')
    last = n[end_first+1:]
    return last + ', ' + first
```
- Representative Tests:
  - last_name_first('Walker White')
  - last_name_first('Walker White')
  Look at precondition when choosing tests

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

Precondition violated

```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'
```
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

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

Modules vs. Scripts

<table>
<thead>
<tr>
<th>Module</th>
<th>Script</th>
</tr>
</thead>
<tbody>
<tr>
<td>Provides functions, constants</td>
<td>Behaves like an application</td>
</tr>
<tr>
<td>Example: temperature.py</td>
<td>Example: 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>python helloApp.py</td>
</tr>
<tr>
<td>or other module</td>
<td>No interactive shell</td>
</tr>
<tr>
<td>All code is either</td>
<td>import acts &quot;weird&quot;</td>
</tr>
<tr>
<td>In a function definition, or</td>
<td>Commands outside functions</td>
</tr>
<tr>
<td>A variable assignment</td>
<td>Does each one in order</td>
</tr>
</tbody>
</table>

Modules/Scripts in this Course

- Our modules consist of
  - Function definitions
  - “Constants” (global vars)
  - Optional application code to call the functions
  - All statements must
    - be inside of a function or
    - assign a constant or
    - be in the application code
  - import should only pull in definitions, not app code

```python
# temperature.py
# Functions
def to_centigrade(x):
    """Returns: x converted to C"""
    C = 9/5 * x + 32
    return C

# Constants
FREEZING_C = 0.0
# temp. water freezes

# Application code
if __name__ == '__main__':
    print('Provide a temp. in Fahrenheit:
    f = float(input())
    c = round(to_centigrade(f),2)
    print('The temperature is ' + `c` + ' C')
```

Testing last_name_first(n)

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

# Application code
if __name__ == '__main__':
    test_last_name_first()
```

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 ' + `first`
    last = n[end_first+1:]
    print 'last is ' + `last`
    return last + ', ' + first
```

Types of Testing

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

Expressions inside of () can be split over several lines.