Anatomy of a Specification

```
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+'!'
```

One line description, followed by blank line

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

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
>>> 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.

```
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:

```
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')`
  - `last_name_first('Walker      White')`

Look at precondition when choosing tests

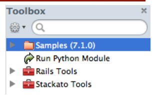
1

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 unittest module** (provided by us)
  - 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 unittest function

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

Running a Module as an Application



- Do not need the interactive shell
  - Can run python <filename> at start
  - Or use Komodo "Run Button"
  - Executes all statements in module and quits
  - Call this "running as an application"
- Applications often have "application code"
  - Code not executed if imported; only if run as app
  - Indented under line **if** __name__ == "__main__":
  - **Example**: application.py

Modules 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

```
# 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__':
    print 'Provide a temp. in Fahrenheit:'
    f = float(raw_input())
    c = round(to_centigrade(f),2)
    print 'The temperature is '+`c`+' C'
```

Testing last_name_first(n)

```
# 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()
    print 'Module name is working correctly'
```

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

Quits Python if not equal

Message will print out only if no errors.

Finding the Error

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

```
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
```

Print variable after each assignment

Optional: Annotate value to make it easier to identify

Types of Testing

Black Box Testing	White Box Testing
Function is "opaque"	Function is "transparent"

Black Box Testing

- Function is "opaque"
  - Test looks at what it does
  - **Fruitful**: what it returns
  - **Procedure**: what changes
- **Example**: Unit tests
- **Problems**:
  - Are the tests everything?
  - What caused the error?

White Box Testing

- Function is "transparent"
  - Tests/debugging takes place inside of function
  - Focuses on where error is
- **Example**: Use of print
- **Problems**:
  - Much harder to do
  - Must remove when done