Lecture 5

Visualizing Functions
Announcements For This Lecture

Readings

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

Assignment 1

- Posted on web page
  - Due Wed, Sep. 18th
  - Revise until correct

Today’s Lab

- Practice today’s lecture
- Highly recommend doing optional part

- Can work in pairs
  - One submission for pair
  - Link up on Piazza
- 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
def greet(n):
    """Prints a greeting to the name n
    Greeting has format 'Hello <n>!' 
    Followed by a conversation starter."
    print 'Hello ' + n + '!
    print 'How are you?'

Precondition: n is a string representing a person’s name"

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.
    Precondition: x is a float measuring temperature in fahrenheit""
    return 5*(x-32)/9.0
```

"""Returns""" indicates a fruitful function

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

```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
How Do Functions Work?

- **Function Frame**: Representation of function call
- A **conceptual model** of Python

Draw parameters as variables (named boxes)

- Number of statement in the function body to execute next
- **Starts with 1**

Draw template on a piece of paper

- function name
- instruction counter
- parameters
- local variables (later in lecture)
Text (Section 3.10) vs. Class

**Textbook**

```
to_centigrade
```

**This Class**

```
def to_centigrade(x):
    return 5*(x-32)/9.0
```

**Definition**:  
```
def to_centigrade(x):
    return 5*(x-32)/9.0
```

**Call**: to_centigrade(50.0)
Example: to_centigrade(50.0)

1. Draw a frame for the call
2. Assign the argument value to the parameter (in frame)
3. Execute the function body
   - Look for variables in the frame
   - If not there, look for global variables with that name
4. Erase the frame for the call

```
def to_centigrade(x):
    return 5*(x-32)/9.0
```
**Example:** \texttt{to\_centigrade(50.0)}

1. Draw a frame for the call
2. Assign the argument value to the parameter (in frame)
3. Execute the function body
   - Look for variables in the frame
   - If not there, look for global variables with that name
4. Erase the frame for the call

\begin{verbatim}
def to_centigrade(x):
    return 5*(x-32)/9.0
\end{verbatim}

----

Executing the return statement

Return statement creates a special variable for result
**Example:** to_centigrade(50.0)

1. Draw a frame for the call
2. Assign the argument value to the parameter (in frame)
3. Execute the function body
   - Look for variables in the frame
   - If not there, look for global variables with that name
4. Erase the frame for the call

```python
def to_centigrade(x):
    return 5*(x-32)/9.0
```

Executing the return statement

The return terminates; no next line to execute
**Example:** `to_centigrade(50.0)`

1. Draw a frame for the call
2. Assign the argument value to the parameter (in frame)
3. Execute the function body
   - Look for variables in the frame
   - If not there, look for global variables with that name
4. Erase the frame for the call

```python
def to_centigrade(x):
    return 5*(x-32)/9.0
```

But don’t actually erase on an exam
Call Frames vs. Global Variables

- This does not work:

```python
def swap(a, b):
    """Swap vars a & b"""
    tmp = a
    a = b
    b = tmp
```

```python
>>> a = 1
>>> b = 2
>>> swap(a, b)
```

Global Variables

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Call Frame

<table>
<thead>
<tr>
<th>swap</th>
</tr>
</thead>
<tbody>
<tr>
<td>a 1</td>
</tr>
<tr>
<td>b 2</td>
</tr>
</tbody>
</table>

9/9/14
Call Frames vs. Global Variables

- This does not work:

```python
def swap(a,b):
    """Swap vars a & b""
    tmp = a
    a = b
    b = tmp

>>> a = 1
>>> b = 2
>>> swap(a,b)
```

Global Variables

Call Frame

```
>>> a = 1
>>> b = 2
```
Call Frames vs. Global Variables

• This does not work:

```python
def swap(a,b):
    '''Swap vars a & b'''
    tmp = a
    a = b
    b = tmp
>>> a = 1
>>> b = 2
>>> swap(a,b)
```

Global Variables

```
1 2
```

Call Frame

```
1 2
```

```python
>>> a = 1
>>> b = 2
>>> swap(a,b)
```
Call Frames vs. Global Variables

- This does not work:

```python
def swap(a, b):
    """Swap vars a & b"""
    tmp = a
    a = b
    b = tmp

>>> a = 1
>>> b = 2
>>> swap(a, b)
```

Global Variables

Call Frame

```
\[
\begin{array}{c}
\text{swap} \\
\text{a} & \times 2 \text{ b} & \times 1 \\
\text{a} & 1 \text{ b} & 2 \\
\text{tmp} & 1
\end{array}
\]
```
Call Frames vs. Global Variables

- This does not work:

```python
def swap(a, b):
    """Swap vars a & b""
    tmp = a
    a = b
    b = tmp

>>> a = 1
>>> b = 2
>>> swap(a, b)
```

Global Variables

```
a 1
```

Call Frame

```
a 1
b 2
```
Visualizing Frames: The Python Tutor

```
1 def max(x,y):
2     if x > y:
3         return x
4     return y
5
6 a = 1
7 b = 2
8 max(a,b)
```

Frames

Objects

Global frame

function max(x, y)

max

| x | 1 |
| y | 2 |
Visualizing Frames: The Python Tutor

```python
1 def max(x, y):
2     if x > y:
3         return x
4     return y
5
6 a = 1
7 b = 2
8 max(a, b)
```

Global Space

Call Frame
Limitations of the Python Tutor

• The Python Tutor is extremely useful
  ▪ You can see exactly what Python is doing
  ▪ You could use it to find errors in your code!

• However, the Python tutor is very limited
  ▪ You can only import the most basic modules
  ▪ You cannot import user-defined modules

• We need some other way to search for errors
  ▪ This is the motivation for code testing
Limitations of the Python Tutor

- The Python Tutor is extremely useful
  - You can see exactly what Python is doing
  - You could use it to find errors in your code!
- However, the Python tutor is very limited
  - You can only import the most basic modules
  - You cannot import user-defined modules
- We need some other way to search for errors
  - This is the motivation for code testing

Many professional software development tools do this too.
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.
    """
    # nothing here yet!
    pass
```

9/9/14
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
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')` gives '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""
```
# Modules vs. Scripts

## Module
- Provides functions, constants
  - **Example:** temperature.py
- **import** it into Python
  - In interactive shell…
  - or other module
- All code is either
  - In a function definition, or
  - A variable assignment

## Script
- Behaves like an application
  - **Example:** helloApp.py
- **Run it from command line**
  - python helloApp.py
  - No interactive shell
  - import acts “weird”
- Commands *outside* functions
  - Does each one in order
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
• 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** should only pull in 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))
```
# test procedure
def test_last_name_first():
    
    """Test procedure for last_name_first(n)"""
    cornelltest.assert_equals('White, Walker',
                               last_name_first('Walker White'))
    cornelltest.assert_equals('White, Walker',
                               last_name_first('Walker     White'))

# Application code
if __name__ == '__main__':
    test_last_name_first()
    print 'Module name is working correctly'
# test procedure
def test_last_name_first():
    
    """Test procedure for last_name_first(n)""
    cornelltest.assert_equals('White, Walker',
    last_name_first('Walker White'))
    cornelltest.assert_equals('White, Walker',
    last_name_first('Walker     White'))

# Application code
if __name__ == '__main__':
    test_last_name_first()
    print 'Module name is working correctly'
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(' ')  # Optional: Annotate value to make it easier to identify
    print end_first
    first = n[:end_first]
    print 'first is ' + `first`  # Print variable after each assignment
    print 'first is ' + `first`
    last = n[end_first+1:]
    print 'last is ' + `last`
    return last+', ' + first
```

9/9/14
# Types of Testing

## 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