Lecture 4: Defining Functions (Ch. 3.4-3.11)
Things to Do Before Next Class

Readings:
- Sections 8.1, 8.2, 8.4, 8.5, first paragraph of 8.9

Labs:
- Go to Lab! (Lab 2 is this week)
- Get Credit for Lab 1:
  - can be checked off during Tuesday's consulting hours 4:30-9:30 in the ACCEL lab
  - cannot be checked off after 3:45pm Wednesday
  - check online if you received credit:
modules vs. scripts, script vs interactive mode, printing vs. not [was: modules and scripts]

I don't really understand the difference between a module and a script and also what "creating evidence that a script ran" means. Also is the interactive mode for a module:

```python
>>> import my_module
>>> my_module.x
9
if so what is the interactive mode for a script?
```

#pin

~ An instructor (Prof. Lee) thinks this is a good question ~

https://piazza.com/class/jckqwmqflaz6i?cid=21
The student answer is great the for the difference between a module and a script!

The "creating evidence that a script ran", which is the title of slide 38 in the lecture 3 presentation slides, means the following.

If you don't have any print statements in your program, then when you run the program, you will see no output. So you don't really even know whether your computer actually executed the program.

Example: I've written a file bracy_height.py, with the following contents:

```
# bracy_height.py

"""A Python program that computes Prof. Bracy's height as feet and left-over inches, rather than just total inches """

bracy_total_inches = 68
bracy_feet = bracy_total_inches // 12
bracy_inches_left = bracy_total_inches % 12
```

Then, here's what happens if I type "python bracy_height.py" at the command shell:

```
[llee@platform9.75 ~/temp] python bracy_height.py
[llee@platform9.75 ~/temp]
```
Anway, NOTHING seems to have happened. I have no evidence that Python did the int division or did any of my commands.

Similarly, here's what happens if I enter Python interactive mode, and then import the module:

```
[lllee@platform9.75 ~/temp] python
Python 3.6.1 |Anaconda 4.4.0 (x86_64)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import bracy_height
```

Again, it looks like nothing happened.

So, if I want evidence that Python actually computed Prof. Bracy's height in feet and inches, I can force it to provide this evidence by asking it to print its results. So, here's a new program, bracy_height_verbose.py:

```
# bracy_height_verbose.py

"""A Python program that computes Prof. Bracy's height as feet and left-over inches, rather than just total inches, and prints the feet and left-over inches """

bracy_total_inches = 68
bracy_feet = bracy_total_inches // 12
bracy_inches_left = bracy_total_inches % 12

print("Prof. Bracy is " + str(bracy_feet) + " feet " + str(bracy_inches_left))
```
Now, if I run this as a script at the command shell, here's what happens:

```
[1lee@platform9.75 ~/temp] python bracy_heightVerbose.py
Prof. Bracy is 5 feet 8
```

So you can see evidence that the script ran: it gave an output. Hurrah!

You can also treat the file as a module, and import it in Python interactive mode, as so:

```
[1lee@platform9.75 ~/temp] python
Python 3.6.1 |Anaconda 4.4.0 (x86_64)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import bracy_heightVerbose
Prof. Bracy is 5 feet 8
```  

And again, you can see that the module does something, in this case, because it gave an output. Hurrah again!
From last time: Function Calls

• Function expressions have the form \texttt{fun}(x, y, \ldots)

• \textbf{Examples} (math functions that work in Python):
  - \texttt{round(2.34)}
  - \texttt{max(a+3, 24)}
From last time: Modules

- Modules provide extra functions, variables
  - Access them with the `import` command
- **Example**: module `math`

```python
>>> import math
>>> math.cos(2.0)
-0.4161468365471424
>>> math.pi
3.141592653589793
```
From last time: Modules

<table>
<thead>
<tr>
<th>Module Text</th>
<th>Interactive Python</th>
</tr>
</thead>
<tbody>
<tr>
<td># my_module.py</td>
<td>&gt;&gt;&gt; import my_module</td>
</tr>
<tr>
<td>&quot;&quot;&quot;This is a simple module. It shows how modules work&quot;&quot;&quot;&quot;</td>
<td>&gt;&gt;&gt; my_module.x</td>
</tr>
<tr>
<td>x = 1+2</td>
<td>9</td>
</tr>
<tr>
<td>x = 3*x</td>
<td></td>
</tr>
</tbody>
</table>

- We discussed how to make module **variables**
- Have not covered how to make **functions**
simple_math.py

```python
>>> import simple_math
>>> simple_math.increment(1)
2
>>> simple_math.increment(2)
3
```
Anatomy of a Function Definition

```
def increment(n):
    """Returns: the value of n+1"""
    return n+1
```

- **name**: increment
- **parameters**: n
- **Function Header**: def increment(n):
- **Docstring Specification**: """Returns: the value of n+1""
- **Body**: return n+1

The vertical line indicates indentation. Use vertical lines when you write Python on **exams** so we can see indentation.
The `return` Statement

• Passes a value from the function to the caller
• **Format**: `return <expression>`
• Any statements after `return` are ignored
• Optional (if absent, special value `None` will be sent back)
Function Calls vs. Definitions

Function Call

- Command to do the function

```python
>>> simple_math.increment(23)
24
>>>```

Function Definition

- Defines what function does

```python
def increment(n):
    return n+1
```

- **Parameter**: variable that is listed within the parentheses of a function header.
- **Argument**: a value to assign to the function parameter when it is called
## Using simple_math.py

<table>
<thead>
<tr>
<th>Module Text</th>
<th>Interactive Python</th>
</tr>
</thead>
<tbody>
<tr>
<td><code># simple_math.py</code></td>
<td><code>&gt;&gt;&gt; import simple_math</code></td>
</tr>
<tr>
<td>&quot;&quot;&quot;module with two simple math functions&quot;&quot;</td>
<td>Python skips</td>
</tr>
<tr>
<td><code>def increment(n):</code></td>
<td>Python skips</td>
</tr>
<tr>
<td>&quot;&quot;&quot;Returns: n+1&quot;&quot;`</td>
<td>Python learns the function definition</td>
</tr>
<tr>
<td><code>return n+1</code></td>
<td>Python skips everything inside the function</td>
</tr>
</tbody>
</table>

Repeat for all functions in module
Using simple_math.py

Module Text

```python
# simple_math.py

"""module with two simple math functions"""

def increment(n):
    """Returns: n+1"""
    return n+1
```

Interactive Python

```python
>>> import simple_math
>>> simple_math.increment(23)
15
```

Python knows what this is!

Now Python executes the function body
Using `simple_math.py`

<table>
<thead>
<tr>
<th>Module Text</th>
<th>Interactive Python</th>
</tr>
</thead>
<tbody>
<tr>
<td># simple_math.py</td>
<td><code>&gt;&gt;&gt; import simple_math</code></td>
</tr>
<tr>
<td>&quot;&quot;&quot;module with two simple math functions&quot;&quot;&quot;</td>
<td><code>&gt;&gt;&gt; simple_math.increment(23)</code></td>
</tr>
<tr>
<td>def increment(n):</td>
<td></td>
</tr>
<tr>
<td>&quot;&quot;&quot;Returns: n+1&quot;&quot;&quot;</td>
<td></td>
</tr>
<tr>
<td>return n+1</td>
<td></td>
</tr>
</tbody>
</table>
Using `simple_math.py`

Module Text

```python
# simple_math.py

"""module with two simple
math functions""

def increment(n):
    """Returns: n+1""
    return n+1
```

Interactive Python

```python
>>> import simple_math
>>> simple_math.increment(23)
24
```
Understanding How Functions Work

- We will draw pictures to show what is in memory
- **Function Frame**: Representation of function call

Note: slightly different than in the book (3.9) Please do it **this** way.
Example: get_feet

```python
def get_feet(height_in_inches):
    return height_in_inches // INCHES_PER_FOOT
```

```python
>>> import height
>>> height.get_feet(68)
```

19
**Example:** `get_feet(68)`

**PHASE 1: Set up call frame**

1. Draw a frame for the call
2. Assign the argument value to the parameter (in frame)
3. Indicate next line to execute

```python
def get_feet(height_in_inches):
    return height_in_inches // INCHES_PER FOOT
```

next line to execute
Example: `get_feet(68)`

**PHASE 2:**
Execute function body

Return statement creates a special variable for result

```python
def get_feet(height_in_inches):
    return height_in_inches // INCHES_PER_FOOT
```

```
def get_feet(height_in_inches):
    return height_in_inches // INCHES_PER_FOOT
```
Example: `get_feet(68)`

**PHASE 2:**
Execute function body

```
def get_feet(height_in_inches):
    return height_in_inches // INCHES_PER_FOOT
```

The return terminates; no next line to execute
Example: `get_feet(68)`

**PHASE 3:** Erase call frame

```python
def get_feet(height_in_inches):
    return height_in_inches // INCHES_PER_FOOT
```

```
def get_feet(height_in_inches):
    return height_in_inches // INCHES_PER_FOOT
```
Example: `get_feet(68)`

**PHASE 3: Erase call frame**

```python
def get_feet(height_in_inches):
    return height_in_inches // INCHES_PER_FOOT
```

But don’t actually erase on an exam.
Local Variables (1)

- Call frames can make “local” variables

```python
>>> import room_numbers

>>> room_numbers.lab_rooms()
```

```python
def lab_rooms():
    red_room = 235
    orange_room = 236
```

![Diagram of local variables]
Local Variables (2)

• Call frames can make “local” variables

```python
>>> import room_numbers

>>> room_numbers.lab_rooms()
26
```

```python
def lab_rooms():
    red_room = 235
    orange_room = 236
```

```
lab_rooms

<table>
<thead>
<tr>
<th>red_room</th>
<th>235</th>
</tr>
</thead>
</table>
```
Local Variables (3)

• Call frames can make “local” variables

```python
>>> import room_numbers
>>> room_numbers.lab_rooms()
```

```
def lab_rooms():
    red_room = 235
    orange_room = 236
    return None
```

```plaintext
lab_rooms

red_room 235
orange_room 236
```
Local Variables (4)

- Call frames can make “local” variables

```python
>>> import room_numbers
>>> room_numbers.lab_rooms()
```

```python
def lab_rooms():
    red_room = 235
    orange_room = 236
```

Variables are gone! This function is useless.
# Exercise Time

## Function Definition

```python
def foo(a,b):
    x = a
    y = b
    return x*y+y
```

## Function Call

```python
>>> foo(3,4)
29
```

What does the frame look like at the **start**?
Which One is Closest to Your Answer?

A:

```
foo
a  3  b  4
x  a
```

B:

```
foo
a  3  b  4

✓
```

C:

```
foo
a  3  b  4
x  3
```

D:

```
foo
a  3  b  4
x  y
```
def foo(a,b):
    x = a
    y = b
    return x*y+y

>>> foo(3,4)

B:

What is the next step?
Which One is Closest to Your Answer?

A:

```
  foo                                   2
  a  3  b  4
```

B:

```
  foo                                   1
  a  3  b  4
  x  3
```

C:

```
  foo                                   2
  a  3  b  4
  x  3
```

D:

```
  foo                                   2
  a  3  b  4
  x  3  y
```
Exercise Time

**Function Definition**

```python
def foo(a,b):
    x = a
    y = b
    return x*y+y
```

**Function Call**

```python
>>> foo(3,4)
```

**C:**

```
foo
   a     b
     3    4
   x     3
```

What is the **next step**?
### Exercise Time

#### Function Definition

```python
def foo(a,b):
    x = a
    y = b
    return x*y+y
```

#### Function Call

```python
>>> foo(3,4)
34
```

What is the next step?
Which One is Closest to Your Answer?

A: foo 3
   RETURN 16

B: foo 3
   a 3 b 4
   RETURN 16

C: foo
   a 3 b 4
   x 3 y 4
   RETURN 16

D: 
   ERASE THE FRAME
Exercise Time

Function Definition

```python
def foo(a,b):
    x = a
    y = b
    return x*y+y
```

Function Call

```python
>>> foo(3,4)
```

C:

What is the next step?
Exercise Time

Function Definition

```python
def foo(a,b):
    x = a
    y = b
    return x*y+y
```

Function Call

```python
>>> foo(3,4)
>>> 16
```
Function Access to Global Space

• All function definitions are in some module
• Call can access global space for that module
  - math.cos: global for math
  - height.get_feet uses global for height
• But cannot change values
  - “Assignment to a global” makes a new local variable!
Function Access to Global Space

- All function definitions are in some module
- Call can access global space for **that module**
  - `math.cos`: global for `math`
  - `height.get_feet` uses global for `height`
- But **cannot** change values
  - Assignment to a global makes a new local variable!

```python
# scope_example.py
"""Show how globals work"""
a = 4 # global space
def get_a():
    return a # returns global
```
Call Frames and Global Variables

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

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

Global Variables:

- a: 1
- b: 2

Call Frame:

1. (swap)
   - a: 1
   - b: 2
   - 1
Call Frames and Global Variables

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

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

Global Variables

```
a  b
1  2
```

Call Frame

```
swap
a  b
1  2
```

tmp

```
1
```
def swap(a, b):
    """Swap global a & b"""
    tmp = a
    a = b
    b = tmp

>>> a = 1
>>> b = 2
>>> swap(a, b)
def swap(a, b):
    """Swap global a & b""
    tmp = a
    a = b
    b = tmp

>>> a = 1
>>> b = 2
>>> swap(a, b)
The specification is a lie:

def swap(a,b):
    '''Swap global a & b'''
    tmp = a
    a = b
    b = tmp

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

Global Variables

Call Frame

ERASE THE FRAME
Call Frames and Global Variables

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

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

Global Variables

```
a 1  
\hline
b 2  
```

Call Frame

THIS FUNCTION DOES NOT SWAP the global a and global b
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)
```

**Frames**
- Global frame
  - `max`
  - `a 1`
  - `b 2`

**Objects**
- function `max(x, y)`
- `x 1`
- `y 2`
More Exercises

### Module Text

```python
# my_module.py

def foo(x):
    return x+1

x = 1+2
x = 3*x
```

### Interactive Python

```python
>>> import my_module
>>> my_module.x
... What does Python give me?
```

**A: 9  CORRECT**

**B: 10**

**C: 1**

**D: Nothing**

**E: Error**