Lecture 4: 
Defining Functions 
(Ch. 3.4-3.11)

CS 1110
Introduction to Computing Using Python

http://www.cs.cornell.edu/courses/cs1110/2022sp

[E. Andersen, A. Bracy, D. Gries, L. Lee, S. Marschner, C. Van Loan, W. White]
• We added a new slide (#10) to address the question of print vs return. See also this discussion on Ed: https://edstem.org/us/courses/19140/discussion/1084754?comment=2472733
• The lecture concluded with slide 42
• We will cover slides 43-45 at the beginning of the next lecture.
• We strongly suggest you check out the Python Tutor!
Announcements

• Zoom polls not appearing, and not using browser?
  ▪ "a little icon shows up on the bottom … sometimes you have to click it to see the poll." (Thanks, CS1110 student for the tip!)
From Last Time: Function Calls

- Function calls have the form:

  `best_function_ever(x,y,...)`

- Arguments: values given as inputs
  - Separated by commas
  - Can be any expression

A function might have 0, 1, ... or many arguments

Let’s define our own functions!
Anatomy of a Function Definition

Python keyword: def

function name: increment

function parameters: (variables for storing input)

function header:

"""Returns: the value of n+1"""

function body:

"return n+1"

Docstring specification
The `return` Statement

- Passes a value from the function to the caller
- Format: \texttt{return <expression>}
- Any function body statements placed after a `return` statement will be ignored
- Optional (if absent, special value \texttt{None} will be sent back)
Organization of a Module

- Function definition goes before any code that calls that function
- There can be multiple function definitions
- Can organize function definitions in any order
Function Definitions vs. Calls

---

Simple Function Example:

```python
# simple_math.py

def increment(n):
    return n+1

increment(2)
```

**Function definition**
- Defines what function will do
- Declaration of **parameters** (n in this case)
- **Parameter**: variable where input to function is stored

**Function call**
- Command to do the function
- **Argument** to assign to function parameter (Argument 2 to be assigned to parameter n in this case)
- **Argument**: an input value to assign to the function parameter when it is called
Executing the script simple_math.py

```
# simple_math.py

# Script that defines and calls one simple math function

def increment(n):
    # Returns: n+1
    return n+1

x = increment(2)
```

Python skips

Python skips

Python learns about the function

Python skips everything inside the function until the function is called

Python executes this statement

Now, python executes the function body
return vs. print

https://edstem.org/us/courses/19140/discussion/1084754?comment=2472733

```
# simple_math.py

"""script that defines and calls one simple math function""

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

x = increment(2)
```

```
C:/> python simple_math.py
C:/>
```

Notice that this script does not print anything!

The function `returns` the value (it gets saved in `x`) but does not print it.

If you want the function to also print to the screen, it needs a print statement.
Understanding How Functions Work

- We draw pictures to show what is in memory
- **Call Frame:** representation of function call

Draw parameters as variables (named boxes)

- Line number of the next statement in the function body to execute
- Starts with 1\textsuperscript{st} statement in function body

Not just a pretty picture!
The information in this picture depicts exactly what is stored in memory on your computer.

Note: slightly different than in the book (3.9) Please do it this way.
Example: `get_feet in height.py` module

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

```python
# height.py

def get_feet(ht_in_inches):
    return ht_in_inches // 12
```

height.py
Example: `get_feet(68)` (slide 1)

```python
>>> import height
>>> height.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
# height.py

def get_feet(ht_in_inches):
    return ht_in_inches // 12
```

```
get_feet
ht_in_inches
68
```

```python
>>> import height
>>> height.get_feet(68)
```
Example: get_feet(68)  (slide 2)

```python
# height.py

1  def get_feet(ht_in_inches):
2   return ht_in_inches // 12

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

PHASE 2:
Execute function body

Return statement creates a special variable for result

The return terminates; no next line to execute
Example: get_feet(68) (slide 3)

```python
>>> import height
>>> height.get_feet(68)
5
>>> 
```

PHASE 3: Delete (cross out) call frame

```
# height.py
1  def get_feet(ht_in_inches):
2    return ht_in_inches // 12
```
Local Variables (1)

Call frames can contain “local” variables

- A variable created in the function

```python
>>> import height2
>>> height2.get_feet(68)
```

```python
# height2.py

def get_feet(ht_in_inches):
    feet = ht_in_inches // 12
    return feet
```

```python
get_feet

ht_in_inches 68
```

Local Variables (1)

Call frames can contain “local” variables

- A variable created in the function

```python
>>> import height2
>>> height2.get_feet(68)
```

```python
# height2.py

def get_feet(ht_in_inches):
    feet = ht_in_inches // 12
    return feet
```

```python
get_feet

ht_in_inches 68
```
Local Variables (2)

Call frames can contain “local” variables

```python
# height2.py

def get_feet(ht_in_inches):
    feet = ht_in_inches // 12
    return feet

>>> import height2
>>> height2.get_feet(68)
```

```
get_feet

ht_in_inches 68
feet 5
```
Local Variables (3)

Call frames can contain “local” variables

```python
# height2.py

def get_feet(ht_in_inches):
    feet = ht_in_inches // 12
    return feet

>>> import height2
>>> height2.get_feet(68)
```

```
get_feet
ht_in_inches  68
feet  5
RETURN  5
```
Local Variables (4)

Call frames can contain “local” variables

```python
>>> import height2
>>> height2.get_feet(68)
5
```

```
# height22.py
def get_feet(ht_in_inches):
    feet = ht_in_inches // 12
    return feet

Variables are gone!
This function is over.
```
Exercise #1

Function Definition

def foo(a, b):
1    x = a
2    y = b
3    return x*y+y

Function Call

>>> foo(3, 4)

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
```
Exercise #2

Function Definition

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

Function Call

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

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

A:
- foo
- a 3
- b 4
- x 3
- 1 2

B:
- foo
- a 3
- b 4
- x 3
- 1

C:
- foo
- a 3
- b 4
- x 3
- 1 2

D:
- foo
- a 3
- b 4
- x 3
- y
- 1 2
Exercise Time *(no poll, just discuss)*

<table>
<thead>
<tr>
<th>Function Definition</th>
<th>Function Call</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>def foo(a, b):</code></td>
<td><code>&gt;&gt;&gt; foo(3, 4)</code></td>
</tr>
<tr>
<td>1 (x = a)</td>
<td></td>
</tr>
<tr>
<td>2 (y = b)</td>
<td></td>
</tr>
<tr>
<td>3 <code>return x*y+y</code></td>
<td></td>
</tr>
</tbody>
</table>

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

```
foo

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
</tr>
</tbody>
</table>
```

What is the **next step**?
Exercise #3

**Function Definition**

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

**Function Call**

```python
>>> foo(3, 4)
27
```

What is the next step?
Exercise Time (no poll, just discuss)

Function Definition

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

Function Call

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

```
foo
a  3  b  4
x  3  y  4
RETURN  16
```

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
```
Global Space

= the purple box we previously labeled “What Python can access directly”

• Top-most location in memory
• Variables in Global Space called Global Variables
• Functions can access anything global space (see next slides)

C:\> python
>>> x = 7
>>>
Call Stack

= the place in memory where the Call Frames live

Functions can only access the variables in their Call Frame or the Global Space.

This is the Call Frame for the function `foo`. It is created in response to a function call and lives on the Call Stack, distinct from the Global Space.

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

```python
Global Space
print()
...
x 7
```

```python
Call Stack
foo
a 3 b 4
1
```
Python just started. It has all the built-in functions. It hasn’t read any of the module yet.
Python just read line 1 of the module.
A variable has been added to the Global Space.
Python just read line 2 of the module. A new function has been added to the Global Space. Note: python has not yet looked inside the function.
To execute the assignment statement on line 5, Python needs to evaluate the RHS. Python creates a call frame for the function, which lives on the Call Stack.
Function Access to Global Space (5)

# height3.py

1. INCHES_PER_FT = 12
2. def get_feet(ht_in_inches):
3.     feet = ht_in_inches // INCHES_PER_FT
4.     return feet
5. answer = get_feet(68)
6. print(answer)

Python has just executed line 3. A new local variable feet has been created inside get_feet’s Call Frame.
Function Access to Global Space (6)

```python
# height3.py

INCHES_PER_FT = 12

def get_feet(ht_in_inches):
    feet = ht_in_inches // INCHES_PER_FT
    return feet

answer = get_feet(68)
print(answer)

Python has just executed line 4. A return value has been created.
```
INCHES_PER_FT = 12

def get_feet(ht_in_inches):
    feet = ht_in_inches // INCHES_PER_FT
    return feet

answer = get_feet(68)
print(answer)

Python has just executed line 5.
A new global variable answer has been created.
The call frame for get_feet has been deleted.
# height3.py

```python
INCHES_PER_FT = 12

def get_feet(ht_in_inches):
    feet = ht_in_inches // INCHES_PER_FT
    return feet

answer = get_feet(68)
print(answer)
```

Python has just executed line 6.

C:\> python height3.py
5
Function Access to Global Space (9)

```python
# height3.py

INCHES_PER_FT = 12

def get_feet(ht_in_inches):
    feet = ht_in_inches // INCHES_PER_FT
    return feet

answer = get_feet(68)
print(answer)
```

Python has completed executing all lines of the module. Python is no longer running, so the global space is gone. You can type a new command at the command line now.

C:\> python height3.py
5
C:\>
Q: what about this??

What if a local variable inside a function has the same name as a global variable?

```python
# height5.py

def get_feet(ht_in_inches):
    feet = ht_in_inches // 12
    return feet

C:\> python
>>> feet = "plural of foot"
>>> import height5
>>> height5.get_feet(68)
```

Global Space

```plaintext
feet "plural of foot"
height5
```

Call Stack (w/ 1 frame)

```plaintext
get_feet 1
ht_in_inches 68
```
A: Look, but don’t touch!

Can’t change global variables in a function! Assignment to a global makes a new local variable!

```python
# height5.py

def get_feet(ht_in_inches):
    feet = ht_in_inches // 12
    return feet

C:\> python
>>> feet = "plural of foot"
>>> import height5
>>> height5.get_feet(68)
```
Use **Python Tutor** to help visualize

Lots of code for today:


Paste it into the Python Tutor ([http://cs1110.cs.cornell.edu/tutor/#mode=edit](http://cs1110.cs.cornell.edu/tutor/#mode=edit))

- Visualize the code as is
- Change the code
  - Try something new!
  - Insert an error! (misspell `ht_in_inches` or `feet`)
- Visualize again and see what is different


```python
# bad_swap.py

def swap(a,b):
    """Bad attempt at swapping globals a & b""
    tmp = a
    a = b
    b = tmp

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

Question: Does this work?

What exactly gets swapped with function `swap`?

Paste this into the Python Tutor and see for yourself!
More Exercises (1)

Module Text

```python
# my_module.py

def foo(x):
    return x+1

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

Python Interactive Mode

```python
>>> import my_module
>>> my_module.x
```

What does Python give me?

A: 9
B: 10
C: 1
D: Nothing
E: Error
More Exercises (2)

**Function Definition**

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

**Function Call**

```python
>>> import silly
>>> x = 2
>>> foo(3,4)
>>> x
...
```

What does Python give me?

- A: 2
- B: 3
- C: 16
- D: Nothing
- E: I do not know
More Exercises (3)

Module Text

```python
# module.py

def foo(x):
    x = 1+2
    x = 3*x
```

Python Interactive Mode

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

A: 9
B: 10
C: 1
D: Nothing
E: Error
```
Module Text

```python
# module.py

def foo(x):
    x = 1+2
    x = 3*x
x = foo(0)
```

Python Interactive Mode

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

A: 9  
B: 10  
C: 1  
D: Nothing  
E: Error
```python
# module.py

def foo(x):
    x = 1+2
    x = 3*x
    return x+1

x = foo(0)
```

Python Interactive Mode:

```python
>>> import module
>>> module.x
...
```

What does Python give me?

- A: 9
- B: 10
- C: 1
- D: Nothing
- E: Error