We Write Programs to Do Things
- Functions are the **key doers**

### Function Call vs. Function Definition
- **Function Call**
  - Command to do the function
  - Example:
    ```python
def plus(n):
    return n + 1
```  
- **Function Definition**
  - Defines what function does
  - Example:
    ```python
def plus(n):
    return n + 1
```  
- **Parameter**: variable that is listed within the parentheses of a method header.
- **Argument**: a value to assign to the method parameter when it is called

### Anatomy of a Function Definition
- **Function Header**
  - Function name
  - Parameters
  - Docstring
  - Specification
  - Statements to execute when called

- **Function Body**
  - (indented)

### The return Statement
- **Format**: `return <expression>`
  - Used to evaluate **function call** (as an expression)
  - Also stops executing the function!
  - Any statements after a `return` are ignored
- **Example**: temperature converter function
  ```python
def to_centigrade(x):
    """Returns: x converted to centigrade"
    return 5*(x - 32)/9.0
```  

### A More Complex Example
- **Function Definition**
  ```python
def foo(a, b):
    """Return something"
    Param a: number
    Param b: number"
    x = a
    y = b
    return x*y+y
```  
- **Function Call**
  ```python
>>> x = 2
>>> foo(3,4)
```  

---

### Understanding How 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

### Text (Section 3.10) vs. Class
- **Textbook**
  ```python
def to_centigrade(x):
    return 5*(x - 32)/9.0
```  
- **This Class**
  ```python
def to_centigrade(x):
    return 5*(x - 32)/9.0
```  

---

### What is in the box?
A: 2
B: 3
C: 16
D: Nothing!
E: I do not know
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
```

```
to_centigrade
```

```
1
```

```
next line to execute
```

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

```
to_centigrade
```

```
1
```

```
x
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
```

```
to_centigrade
```

```
1
```

```
x
50.0
```

Call Frames vs. Global Variables

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
```

```
1
```

```
b
2
```

```
Call Frame
```

```
swap
```

```
a
2
```

```
b
1
```

```
tmp
1
```

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
  - temperature.to_centigrade uses global for temperature
- But cannot change values
  - Assignment to a global makes a new local variable!
  - Why we limit to constants

```
def get_a():
    return a
```

```
get_a
```

```
1
```

```
# globals.py
'''Show how globals work'''
```

```
a = 4 # global space
def get_a():
    return a # returns global
```

```
change_a
```

```
1
```

```
a
3.5
```

```
# globals.py
'''Show how globals work'''
```

```
a = 4 # global space
def change_a():
    a = 3.5 # local variable
    return a
```

```
change_a
```

```
1
```

```
a
3.5
```

Exercise Time

Function Definition
```
def foo(a,b):
    '''Return something'''
    Param x: a number
    Param y: a number"
    a = x
    y = b
    return x*y+y
```

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

Function Call
```
>>> x = foo(3,4)
```

What does the frame look like at the start?