We Write Programs to Do Things

• Functions are the **key doers**

<table>
<thead>
<tr>
<th>Function Call</th>
<th>Function Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command to do the function</td>
<td>Defines what function does</td>
</tr>
</tbody>
</table>

```python
>>> plus(23)
24
```

• 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

<table>
<thead>
<tr>
<th>name</th>
<th>parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>def</code> plus(n):</td>
<td>Function Header</td>
</tr>
<tr>
<td><code>&gt;&gt;&gt; Returns the number n+1</code></td>
<td>Docstring</td>
</tr>
</tbody>
</table>

Parameter n: number to add to
Precondition: n is a number

```python
x = n+1
return x
```

The vertical line indicates indentation

Use vertical lines when you write Python on exams so we can see indentation

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

<table>
<thead>
<tr>
<th>Function Definition</th>
<th>Function Call</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>def</code> foo(a,b):</td>
<td><code>&gt;&gt;&gt; x = 2</code></td>
</tr>
<tr>
<td>&quot;&quot;&quot;Return something&quot;&quot;</td>
<td><code>&gt;&gt;&gt; foo(3,4)</code></td>
</tr>
<tr>
<td>Param a: number</td>
<td>x ?</td>
</tr>
<tr>
<td>Param b: number</td>
<td>A: 2</td>
</tr>
<tr>
<td>x = a</td>
<td>B: 3</td>
</tr>
<tr>
<td>y = b</td>
<td>C: 16</td>
</tr>
<tr>
<td>return x*y+y</td>
<td>D: Nothing!</td>
</tr>
<tr>
<td></td>
<td>E: I do not know</td>
</tr>
</tbody>
</table>

Understanding How Functions Work

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

<table>
<thead>
<tr>
<th>function name</th>
<th>instruction counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>parameters</td>
<td>local variables (later in lecture)</td>
</tr>
</tbody>
</table>

Text (Section 3.10) vs. Class

<table>
<thead>
<tr>
<th>Textbook</th>
<th>This Class</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>to_centigrade</code></td>
<td><code>to_centigrade</code></td>
</tr>
<tr>
<td><code>x = 80.0</code></td>
<td><code>1</code></td>
</tr>
<tr>
<td><code>x = 50.0</code></td>
<td><code>to_centigrade(50.0)</code></td>
</tr>
</tbody>
</table>

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

Global Variables
- `a` 1
- `b` 2

Call Frame
- `a` 2
- `b` 1
- `tmp` 1

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

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

```
# globals.py
"""Show how globals work""
# a = 4 # global space
def get_a():
    return a # returns global
```

Function Access to Global Space

```
def foo(a,b):
    """Return something"
    Param x: a number
    Param y: a number"
    a = 3.5 # local variable
    return a
```

Exercise Time

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

What does the frame look like at the start?