We Write Programs to Do Things

• Functions are the **key doers**

Function Call | Function Definition
---|---
• Command to *do* the function | • Defines what function *does*

```plaintext
>>> 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 plus(n):</code></td>
<td>Function Header</td>
</tr>
<tr>
<td>&quot;&quot;&quot;Returns the number n+1&quot;&quot;&quot;</td>
<td>Docstring Specification</td>
</tr>
<tr>
<td>Parameter n: number to add to</td>
<td>Statements to execute when called</td>
</tr>
<tr>
<td>Precondition: n is a number&quot;&quot;&quot;</td>
<td></td>
</tr>
<tr>
<td><code>x = n+1</code></td>
<td></td>
</tr>
<tr>
<td><code>return x</code></td>
<td></td>
</tr>
</tbody>
</table>

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

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

```plaintext
>>> x = 2
```  

What is in the box?

A: 2  
B: 3  
C: 16  
D: Nothing!  
E: I do not know

Understanding How Functions Work

• **Function Frame**: Representation of function call

  A conceptual model of Python

  | function name | instruction counter |
  | parameters | local variables (later in lecture) |

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 -&gt; 80.0</code></td>
<td><code>1</code></td>
</tr>
<tr>
<td><code>x: 50.0</code></td>
<td><code>to_centigrade(80.0)</code></td>
</tr>
</tbody>
</table>

Definition:  

```python
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

```python
def to_centigrade(x):
    return 5*(x-32)/9.0
to_centigrade(50.0)
```

Call Frames vs. Global Variables

The specification is a lie:
```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
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

Exercise Time

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

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

What does the frame look like at the start?