Review ideas from previous lecture
Module vs. Script
print statement

Running a Script

- From the command line, type:
  python <script filename>
- Example:
  C:\> python my_module.py
  C:\> looks like nothing happened
- Actually, something did happen
  * Python executed all of my_module.py

Running my_module.py as a script

<table>
<thead>
<tr>
<th>my_module.py</th>
</tr>
</thead>
<tbody>
<tr>
<td># my_module.py</td>
</tr>
<tr>
<td>&quot;&quot;&quot;This is a simple module. It shows how modules work&quot;&quot;&quot;</td>
</tr>
<tr>
<td>x = 1+2</td>
</tr>
<tr>
<td>x = 3*x</td>
</tr>
<tr>
<td>C:&gt; python module.py</td>
</tr>
<tr>
<td>Python does not execute (because of &quot;&quot;&quot;&quot;)</td>
</tr>
<tr>
<td>x 9</td>
</tr>
</tbody>
</table>

Running fah2cel.py as a script

<table>
<thead>
<tr>
<th>fah2cel.py</th>
</tr>
</thead>
<tbody>
<tr>
<td># fah2cel.py</td>
</tr>
<tr>
<td>&quot;&quot;&quot;Convert 32 degrees Fahrenheit to degrees Celsius&quot;&quot;&quot;</td>
</tr>
<tr>
<td>f= 32.0</td>
</tr>
<tr>
<td>c= (f-32)*5/9</td>
</tr>
<tr>
<td>C:&gt; python fah2cel.py</td>
</tr>
<tr>
<td>C:&gt; fah2cel.c</td>
</tr>
<tr>
<td>After you hit &quot;Return&quot; here what will be printed next?</td>
</tr>
<tr>
<td>(A) &gt;&gt;&gt;</td>
</tr>
<tr>
<td>(B) 0.0 &gt;&gt;&gt;</td>
</tr>
<tr>
<td>(C) an error message</td>
</tr>
<tr>
<td>(D) The text of fah2cel.py</td>
</tr>
<tr>
<td>(E) Sorry, no clue.</td>
</tr>
</tbody>
</table>

Clicker Question

fah2cel.py

- f= 32.0
- c= (f-32)*5/9

Command Line

<table>
<thead>
<tr>
<th>Command Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>C:&gt; python fah2cel.py</td>
</tr>
<tr>
<td>C:&gt; fah2cel.c</td>
</tr>
<tr>
<td>After you hit &quot;Return&quot; here what will be printed next?</td>
</tr>
<tr>
<td>(A) &gt;&gt;&gt;</td>
</tr>
<tr>
<td>(B) 0.0 &gt;&gt;&gt;</td>
</tr>
<tr>
<td>(C) an error message</td>
</tr>
<tr>
<td>(D) The text of fah2cel.py</td>
</tr>
<tr>
<td>(E) Sorry, no clue.</td>
</tr>
</tbody>
</table>
Modules vs. Scripts

<table>
<thead>
<tr>
<th>Module</th>
<th>Script</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Provides functions, variables</td>
<td>• Behaves like an application</td>
</tr>
<tr>
<td>• import it into Python shell</td>
<td>• Run it from command line</td>
</tr>
<tr>
<td>• Within Python shell you have access to the functions and variables of the imported module</td>
<td>• After running the app you’re back at the command line (not in Python shell)</td>
</tr>
</tbody>
</table>

Files could look the same. Difference is how you use them.

From last time: Function Calls

• Function expressions have the form `fun(x,y,...)`

• Examples (math functions that work in Python):
  * `round(2.34)`
  * `max(a+3,24)`

Let’s define our own functions!

Anatomy of a Function Definition

- **Function Header**: `def increment(n):`
- **Docstring**: """Returns: the value of n+1"""
- **Function 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 executing `return` are ignored
• Optional (if absent, special value `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

Function definition
• Defines what the function will do
• Declaration of parameters, n in this case
• Parameter: the variable that is listed within the parentheses of a function header.

Function call
• Command to do the function
• Argument to assign to function parameter, n in this case
• Argument: a value to assign to the function parameter when it is called

# simple_math.py
def increment(n):
    return n+1

increment(2)

Example: get_feet in height.py module

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

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

Example: get_feet(68)

PHASE 2: Execute function body

def get_feet(ht_in_inches):
    return ht_in_inches // 12

C://> python simple_math.py
Example: get_feet(68)

PHASE 2:
Execute function body

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

PHASE 3: Delete (cross out) call frame

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

Local Variables (1)

- Call frames can make “local” variables
  - A variable created in the function

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

Local Variables (2)

- Call frames can make “local” variables
  - A variable created in the function

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

Local Variables (3)

- Call frames can make “local” variables
  - A variable created in the function

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

Local Variables (4)

- Call frames can make “local” variables
  - A variable created in the function

```python
>>> import height2
>>> height2.get_feet(68)
```
**Exercise Time**

<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</td>
<td><code>x = a</code></td>
</tr>
<tr>
<td>2</td>
<td><code>y = b</code></td>
</tr>
<tr>
<td>3</td>
<td><code>return x*y+y</code></td>
</tr>
</tbody>
</table>

**Function Access to Global Space**

- Top-most location in memory called *global space*
- Functions can access anything in that global space

```
INCHES_PER_FT = 12

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

get_feet(68)
```

**What about this??**

- What if you choose a local variable inside a function that happens to also be a global variable?

```
INCHES_PER_FT = 12
feet = "plural of foot"
def get_feet(ht_in_inches):
    feet = ht_in_inches // INCHES_PER_FT
    return feet
get_feet(68)
```

**Look, but don’t touch!**

- Can’t change global variables
- In a function, “assignment to a global” makes a new local variable!

```
INCHES_PER_FT = 12
feet = "plural of foot"
... def get_feet(ht_in_inches):
    feet = ht_in_inches // INCHES_PER_FT
    return feet
get_feet(68)
```

**Use “Python Tutor” to help visualize**

```
# height2.py
INCHES_PER_FT = 12
feet = "plural of foot"
def get_feet(ht_in_inches):
    feet = ht_in_inches // INCHES_PER_FT
    return feet
get_feet(68)
```

1. Visualize code as is
2. Change code to introduce an error, e.g. misspell `ht_in_inches`.
   Visualize again.

**Call Frames and Global Variables**

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

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

Question: What exactly gets swapped with function `swap`?
More Exercises (1)

Module Text

```python
# my_module.py
def foo(x):
    return x+1
```

Interactive Python

```python
>>> import my_module
>>> my_module.x
...  # What does Python give me?
A: 9
B: 10
C: 1
D: Nothing
E: Error
```

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

More Exercises (2)

Function Definition

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

Function Call

```python
>>> 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
```

Interactive Python

```python
>>> import module
>>> module.x
...  # What does Python give me?
A: 9
B: 10
C: 1
D: Nothing
E: Error
```

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

More Exercises (4)

Module Text

```python
# module.py
def foo(x):
    x = 1+2
    x = 3*x
```

Interactive Python

```python
>>> import module
>>> module.x
...  # What does Python give me?
A: 9
B: 10
C: 1
D: Nothing
E: Error
```

```python
x = foo(0)
```

More Exercises (5)

Module Text

```python
# module.py
def foo(x):
    x = 1+2
    x = 3*x
    return x+1
```

Interactive Python

```python
>>> import module
>>> module.x
...  # What does Python give me?
A: 9
B: 10
C: 1
D: Nothing
E: Error
```

```python
x = foo(0)
```