

Lecture 3

Functions & Modules

Announcements

Reminders

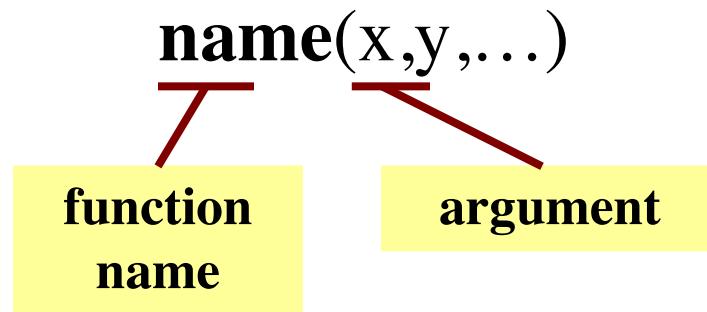
- Grading AI quiz **today**
 - Take now if have not
 - If make 9/10, are okay
 - Else must retake
- **Survey 0** is still open
 - For participation score
 - Must complete them
- Must access in CMS

Optional Videos

- **Today**
 - **Lesson 3:** Function Calls
 - **Lesson 4:** Modules
 - Videos 4.1-4.5
- Next Time
 - Video 4.6 of **Lesson 4**
 - **Lesson 5:** Function Defs
- Also *skim* Python API

Function Calls

- Python supports expressions with math-like functions
 - A function in an expression is a *function call*
- **Function calls** have the form



- **Arguments** are
 - **Expressions**, not values
 - Separated by commas

Built-In Functions

- Python has several math functions
 - `round(2.34)`
 - `max(a+3,24)`
- You have seen many functions already
 - Type casting functions: `int()`, `float()`, `bool()`
- Documentation of all of these are online
 - <https://docs.python.org/3/library/functions.html>
 - Most of these are too advanced for us right now

Functions as Commands/Statements

- Most functions are expressions.
 - You can use them in assignment statements
 - **Example:** `x = round(2.34)`
- But some functions are **commands**.
 - They instruct Python to do something
 - Help function: `help()`
 - Quit function: `quit()`
- How know which one? Read documentation.

These take no
arguments

Built-in Functions vs Modules

- The number of built-in functions is small
 - <http://docs.python.org/3/library/functions.html>
- Missing a lot of functions you would expect
 - **Example:** cos(), sqrt()
- **Module:** file that contains Python code
 - A way for Python to provide optional functions
 - To access a module, the import command
 - Access the functions using module as a *prefix*

Example: Module math

```
>>> import math
```

```
>>> math.cos(0)
```

```
1.0
```

```
>>> cos(0)
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
NameError: name 'cos' is not defined
```

```
>>> math.pi
```

```
3.141592653589793
```

```
>>> math.cos(math.pi)
```

```
-1.0
```

Example: Module math

```
>>> import math
```

To access math
functions

```
>>> math.cos(0)
```

```
1.0
```

```
>>> cos(0)
```

Functions
require math
prefix!

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

NameError: name 'cos' is not defined

```
>>> math.pi
```

```
3.141592653589793
```

```
>>> math.cos(math.pi)
```

```
-1.0
```

Example: Module math

```
>>> import math
```

To access math
functions

```
>>> math.cos(0)
```

```
1.0
```

```
>>> cos(0)
```

Functions
require math
prefix!

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

NameError: name 'cos' is not defined

```
>>> math.pi
```

Module has
variables too!

```
3.141592653589793
```

```
>>> math.cos(math.pi)
```

```
-1.0
```

Example: Module math

```
>>> import math
```

To access math functions

```
>>> math.cos(0)
```

Functions require math prefix!

```
1.0
```

```
>>> cos(0)
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

NameError: name 'cos' is not defined

```
>>> math.pi
```

Module has variables too!

```
3.141592653589793
```

```
>>> math.cos(math.pi)
```

```
-1.0
```

Other Modules

- **os**
 - Information about your OS
 - Cross-platform features
- **random**
 - Generate random numbers
 - Can pick any distribution
- **introscs**
 - Custom module for the course
 - Will be used a lot at start

Using the from Keyword

```
>>> import math
```

```
>>> math.pi
```

```
3.141592653589793
```

```
>>> from math import pi
```

```
>>> pi
```

No prefix needed
for variable pi

```
3.141592653589793
```

```
>>> from math import *
```

```
>>> cos(pi)
```

```
-1.0
```

No prefix needed
for anything in math

- Be careful using from!
- Using import is *safer*
 - Modules might conflict (functions w/ same name)
 - What if import both?
- **Example:** Turtles
 - Used in Assignment 4
 - 2 modules: turtle, introcs
 - Both have func. Turtle()

Reading the Python Documentation

The screenshot shows a web browser window displaying the Python Software Foundation documentation at docs.python.org/3/library/math.html. The page title is "9.2. math — Mathematical functions". The left sidebar contains a "Table Of Contents" with sections like "9.2.1. Number-theoretic and representation functions", "9.2.2. Power and logarithmic functions", "9.2.3. Trigonometric functions", "9.2.4. Angular conversion", and "9.2.5. Hyperbolic". The main content area starts with a brief introduction: "This module is always available. It provides access to the mathematical functions defined by the C standard. These functions cannot be used with complex numbers; use the functions of the same name from the `cmath` module if you require support for complex numbers. The distinction between functions which support complex numbers and those which don't is made since most users do not want to learn quite as much mathematics as required to understand complex numbers. Receiving an exception instead of a complex result allows earlier detection of the unexpected complex number used as a parameter, so that the programmer can determine how and why it was generated in the first place." It then states: "The following functions are provided by this module. Except when explicitly noted otherwise, all return values are floats."

math.`ceil`(*x*)

Return the ceiling of *x*, the smallest integer greater than or equal to *x*.

Next topic

9.3. `cmath` — Mathematical functions for complex numbers

This Page

[Report a Bug](#)
[Show Source](#)

Return a float with the magnitude (absolute value) of *x* but the sign of *y*. On platforms that support signed zeros, `copysign(1.0, -0.0)` returns `-1.0`.

`math.fabs(x)`

Return the absolute value of *x*.

`math.factorial(x)`

Return *x* factorial. Raises `ValueError` if *x* is not integral or is negative.

`math.floor(x)`

Return the floor of *x*, the largest integer less than or equal to *x*. If *x* is not a float, delegates to `x.floor()`, which should return an `Integral` value.

`math.fmod(x, y)`

Return `fmod(x, y)`, as defined by the platform C library. Note that the C standard is that `fmod(x, y)` be exactly (mathematically; to infinite precision) equal to $x - ny$ for some integer n such that the result has the same sign as *x* and magnitude less than `abs(y)`. Python's `x % y` returns a result with the sign of *y* instead, and may not be exactly computable for float arguments. For example, `fmod(-1e-100, 1e100)` is `-1e-100`, but the result of Python's `-1e-100 % 1e100` is `1e100-1e-100`, which cannot be

<http://docs.python.org/3/library>

Reading the Python Documentation

The screenshot shows a web browser displaying the Python Software Foundation documentation at docs.python.org/3/library/math.html. The page title is "9.2. math — Mathematical functions — Python 3.6.2 documentation". A green callout box labeled "Function name" points to the word "ceil" in the function signature `math.ceil(x)`. Another green callout box labeled "Possible arguments" points to the parameter `x` in the same signature. A third green callout box labeled "Module" points to the "math" module name. A fourth green callout box labeled "What the function evaluates to" points to the text "Return the ceiling of `x`, the smallest integer greater than or equal to `x`". The page content includes a brief description of the module, information about complex numbers, and a note about returning floats. It also lists other functions like `floor`, `fmod`, and `factorial`.

Table Of Contents
9.2. math — Mathematical functions

9.2. math — Mathematical functions

This module is always available. It provides access to the mathematical functions defined by the C standard.

These functions cannot be used with complex numbers; use the functions of the same name from the `cmath` module if you require support for complex numbers. The distinction between functions which support complex numbers and those which don't is made since most users do not want to learn quite as much mathematics as required to understand complex numbers. Receiving an exception instead of a complex result allows earlier detection of the unexpected complex number used as a parameter, so that the programmer can determine how and why it was generated in the first place.

The following functions are provided by this module. Except when explicitly noted otherwise, all return values are floats.

math.ceil**(x)**

Return the ceiling of `x`, the smallest integer greater than or equal to `x`.

Module

Possible arguments

What the function evaluates to

Return `x` factorial. Raises `ValueError` if `x` is not integral or is negative.

`math.floor(x)`

Return the floor of `x`, the largest integer less than or equal to `x`. If `x` is not a float, delegates to `x.**floor**()`, which should return an `Integral` value.

`math.fmod(x, y)`

Return `fmod(x, y)`, as defined by the platform C library. Note that the C standard is that `fmod(x, y)` be exactly (mathematically; to infinite precision) equal to `x - n*y` for some integer `n` such that the result has the same sign as `x` and magnitude less than `abs(y)`. Python's `x % y` returns a result with the sign of `y` instead, and may not be exactly computable for float arguments. For example, `fmod(-1e-100, 1e100)` is `-1e-100`, but the result of Python's `-1e-100 % 1e100` is `1e100-1e-100`, which cannot be

<http://docs.python.org/3/library>

Interactive Shell vs. Modules

A screenshot of a terminal window titled "wmwhite — python — 52x25". The window shows the Python 3.6.5 interactive shell. It starts with the Python banner, followed by a few arithmetic operations: `x = 1+2`, `x = 3*x`, and `x`. The output is `9`. The prompt `>>>` is visible at the bottom.

```
[wmwhite@Rlyeh]:~ > python
Python 3.6.5 |Anaconda, Inc.| (default, Apr 26 2018,
08:42:37)
[GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE_401/
final)] on darwin
Type "help", "copyright", "credits" or "license" for
more information.
>>> x = 1+2
>>> x = 3*x
>>> x
9
>>> 
```

A screenshot of a code editor window titled "module.py". The file path is "Users > wmmauthor > Documents > Professional > Courses > CS-1110 > Lectures > Lecture 3". The code in the file is a simple module named "module.py". It contains a docstring, a comment, and two arithmetic operations. The code editor has a sidebar with various icons.

```
1  """
2  A simple module.
3
4  This file shows how modules work
5
6  Author: Walker M. White (wmw2)
7  Date: August 25, 2017 (Python 3 Version)
8  """
9
10 x = 1+2    # I am a comment
11 x = 3*x
12 x 
```

- Launch in command line
- Type each line separately
- Python executes as you type

- Write in a code editor
 - We use VS Code
 - But anything will work
- Load module with import

Using a Module

Module Contents

```
""" A simple module.
```

```
This file shows how modules work
```

```
"""
```

```
# This is a comment
```

```
x = 1+2
```

```
x = 3*x
```

```
x
```

Using a Module

Module Contents

```
""" A simple module.
```

This file shows how modules work

```
"""
```

```
# This is a comment
```

```
x = 1+2
```

```
x = 3*x
```

```
x
```

Single line comment

(not executed)

Using a Module

Module Contents

```
""" A simple module.
```

```
This file shows how modules work
```

```
"""
```

```
# This is a comment
```

```
x = 1+2
```

```
x = 3*x
```

```
x
```

Docstring (note the Triple Quotes)
Acts as a multiple-line comment
Useful for *code documentation*

Single line comment
(not executed)

Using a Module

Module Contents

```
""" A simple module.
```

```
This file shows how modules work
```

```
"""
```

```
# This is a comment
```

```
x = 1+2
```

```
x = 3*x
```

```
x
```

Docstring (note the Triple Quotes)
Acts as a multiple-line comment
Useful for *code documentation*

Single line comment
(not executed)

Commands

Executed on import

Using a Module

Module Contents

```
""" A simple module.
```

```
This file shows how modules work
```

```
"""
```

```
# This is a comment
```

```
x = 1+2
```

```
x = 3*x
```

```
x
```

Docstring (note the Triple Quotes)
Acts as a multiple-line comment
Useful for *code documentation*

Single line comment
(not executed)

Commands

Executed on import

Not a command.
import **ignores this**

Using a Module

Module Contents

```
""" A simple module.
```

```
This file shows how modules work
```

```
"""
```

```
# This is a comment
```

```
x = 1+2
```

```
x = 3*x
```

```
x
```

Python Shell

```
>>> import module
```

```
>>> x
```

Using a Module

Module Contents

```
""" A simple module.
```

```
This file shows how modules work
```

```
"""
```

```
# This is a comment
```

```
x = 1+2
```

```
x = 3*x
```

```
x
```

Python Shell

```
>>> import module
```

```
>>> x
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
NameError: name 'x' is not defined
```

Using a Module

Module Contents

```
""" A simple module.
```

```
This file shows how modules work
```

```
"""
```

```
# This is a comment
```

```
x = 1+2
```

```
x = 3*x
```

```
x
```

“**Module data**” must be
prefixed by module name

Python Shell

```
>>> import module
```

```
>>> x
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
NameError: name 'x' is not defined
```

```
>>> module.x
```

```
9
```

Using a Module

Module Contents

```
""" A simple module.
```

```
This file shows how modules work
```

```
"""
```

```
# This is a comment
```

```
x = 1+2
```

```
x = 3*x
```

```
x
```

“**Module data**” must be
prefixed by module name

Prints **docstring** and
module contents

Python Shell

```
>>> import module
```

```
>>> x
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

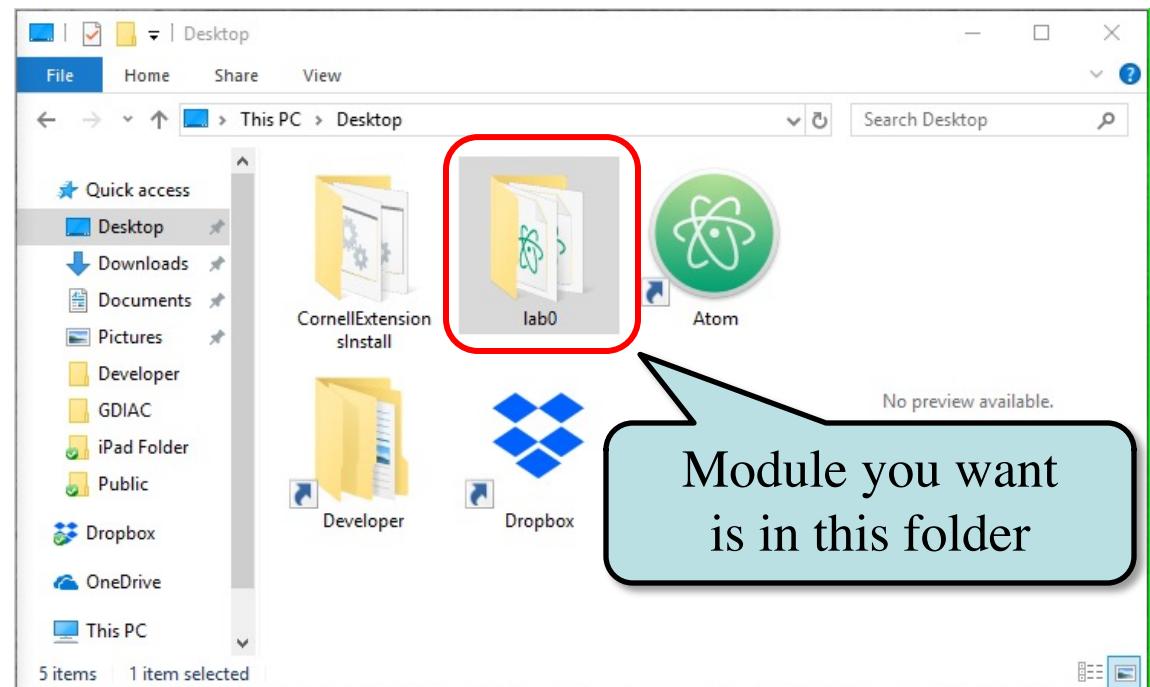
```
NameError: name 'x' is not defined
```

```
>>> module.x
```

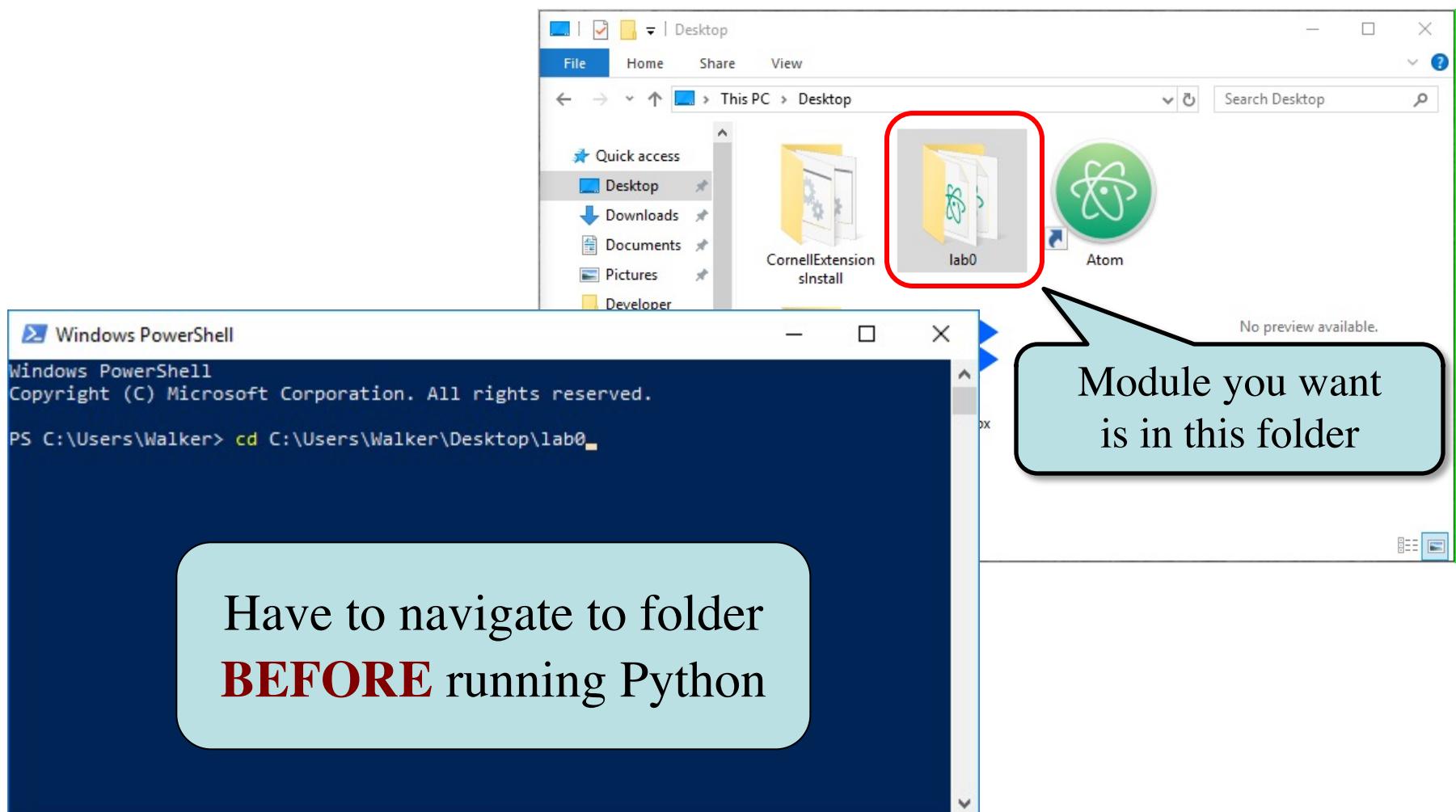
```
9
```

```
>>> help(module)
```

Modules Must be in Working Directory!



Modules Must be in Working Directory!



Modules vs. Scripts

Module

- Provides functions, variables
 - **Example:** temp.py

- import it into Python shell

```
>>> import temp
```

```
>>> temp.to_fahrenheit(100)
```

```
212.0
```

```
>>>
```

Script

- Behaves like an application
 - **Example:** helloApp.py
- Run it from command line:
`python helloApp.py`



Modules vs. Scripts

Module

- Provides functions, variables
 - **Example:** temp.py
- import it into Python shell

```
>>> import temp  
>>> temp.to_fahrenheit(100)  
212.0  
>>>
```

Script

- Behaves like an application
 - **Example:** helloApp.py
- Run it from command line:
`python helloApp.py`



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

Scripts and Print Statements

module.py

```
""" A simple module.
```

This file shows how modules work

```
"""
```

```
# This is a comment  
x = 1+2  
x = 3*x  
x
```

script.py

```
""" A simple script.
```

This file shows why we use print

```
"""
```

```
# This is a comment  
x = 1+2  
x = 3*x  
print(x)
```

Scripts and Print Statements

module.py

""" A simple module.

This file shows how modules work

"""

```
# This is a comment
```

```
x = 1+2
```

```
x = 3*x
```

x

script.py

""" A simple script.

This file shows why we use print

"""

```
# This is a comment
```

```
x = 1+2
```

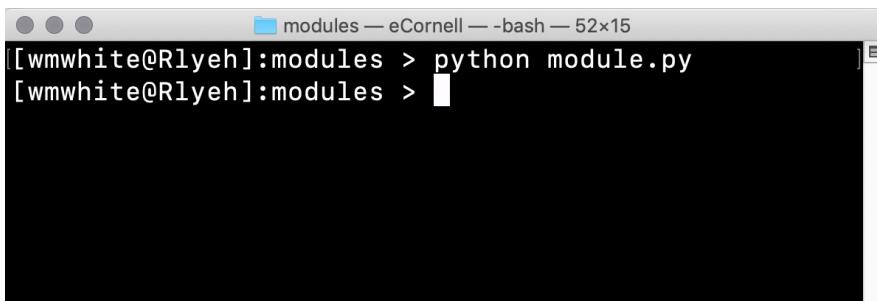
```
x = 3*x
```

print(x)



Scripts and Print Statements

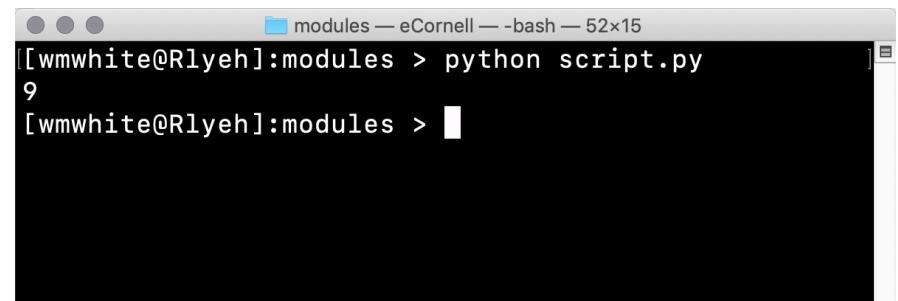
module.py



```
[wmwhite@Rlyeh]:modules > python module.py
[wmwhite@Rlyeh]:modules > 
```

- Looks like nothing happens
- Python did the following:
 - Executed the **assignments**
 - Skipped the last line
(‘x’ is not a statement)

script.py



```
[wmwhite@Rlyeh]:modules > python script.py
9
[wmwhite@Rlyeh]:modules > 
```

- We see something this time!
- Python did the following:
 - Executed the **assignments**
 - Executed the last line
(Prints the contents of x)

Scripts and Print Statements

module.py

```
[wmwhite@Rlyeh]:modules > python module.py  
[wmwhite@Rlyeh]:modules > 
```

script.py

```
[wmwhite@Rlyeh]:modules > python script.py  
9  
[wmwhite@Rlyeh]:modules > 
```

- Looks like a module
- Python ignores the last line
- Python executes the assignments
- Skipped the last line ('x' is not a statement)

When you run a script,
only statements are executed

User Input

```
>>> input('Type something')
```

```
Type somethingabc
```

```
'abc'
```

No space after the prompt.

```
>>> input('Type something: ')
```

```
Type something: abc
```

```
'abc'
```

Proper space after prompt.

```
>>> x = input('Type something: ')
```

```
Type something: abc
```

```
>>> x
```

Assign result to variable.

```
'abc'
```

8/30/22

Making a Script Interactive

====

A script showing off input.

This file shows how to make a script interactive.

====

```
x = input("Give me something: ")  
print("You said: "+x)
```

```
[wmw2] folder> python script.py
```

```
Give me something: Hello
```

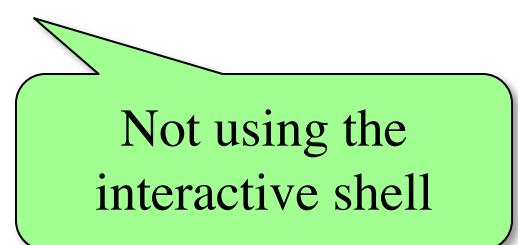
```
You said: Hello
```

```
[wmw2] folder> python script.py
```

```
Give me something: Goodbye
```

```
You said: Goodbye
```

```
[wmw2] folder>
```



Not using the
interactive shell

Numeric Input

- `input` returns a string
 - Even if looks like int
 - It cannot know better
 - You must convert values
 - `int()`, `float()`, `bool()`, etc.
 - Error if cannot convert
 - One way to program
 - But it is a *bad* way
 - Cannot be automated
- ```
>>> x = input('Number: ')
Number: 3

>>> x
'3'

>>> x + 1

TypeError: must be str, not int

>>> x = int(x)

>>> x+1
4
```
- Value is a string.
- Must convert to  
int.

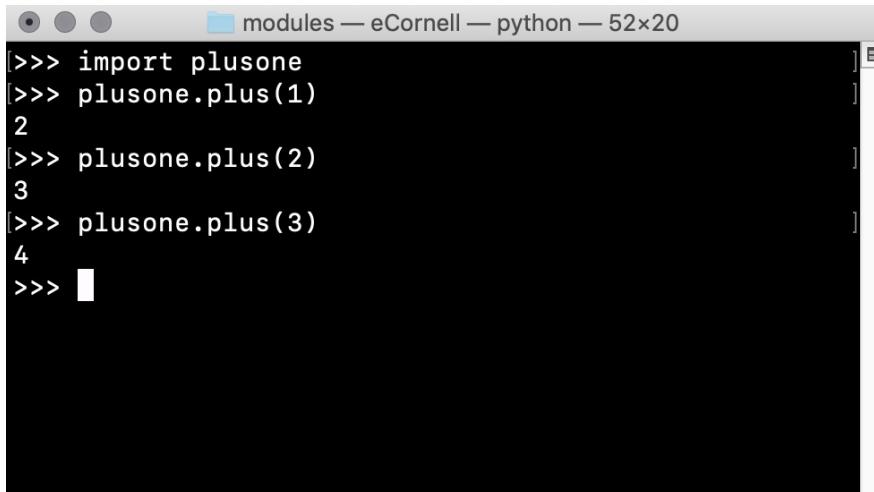
# Next Time: Defining Functions

---

## Function Call

---

- Command to **do** the function
- Can put it anywhere
  - In the Python shell
  - Inside another module



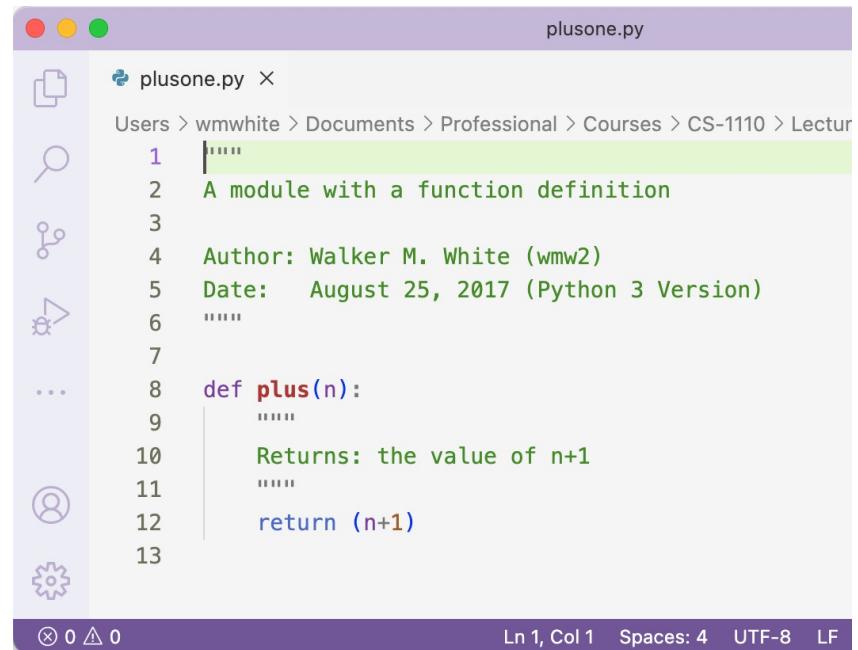
```
modules — eCornell — python — 52x20
>>> import plusone
>>> plusone.plus(1)
2
>>> plusone.plus(2)
3
>>> plusone.plus(3)
4
>>>
```

A screenshot of a terminal window titled "modules — eCornell — python — 52x20". It shows four lines of Python code being run at the prompt (>>>). The first three lines are function calls to "plusone.plus" with arguments 1, 2, and 3 respectively, resulting in outputs 2, 3, and 4. The fourth line is an empty line starting with a greater than sign.

## Function Definition

---

- Command to **do** the function
- Belongs inside a module



```
plusone.py
Users > wmmwhite > Documents > Professional > Courses > CS-1110 > Lectur
1 """
2 A module with a function definition
3
4 Author: Walker M. White (wmw2)
5 Date: August 25, 2017 (Python 3 Version)
6 """
7
8 def plus(n):
9 """
10 Returns: the value of n+1
11 """
12 return (n+1)
13
```

A screenshot of a code editor window titled "plusone.py". The file contains a Python module with a single function definition. The function is named "plus" and takes one argument "n". It returns the value of "n+1". The code is annotated with docstrings and a multi-line comment at the top. The status bar at the bottom indicates the file is 1 line long, 1 column wide, with 4 spaces per line, in UTF-8 encoding, and ends with a LF.

# Next Time: Defining Functions

## Function Call

- Command to **do** the function
- Can put it anywhere
  - In the Python shell
  - Inside another module

A screenshot of a terminal window titled "modules — eCornell — python — 52x20". The window contains the following Python code:

```
>>> import plusone
>>> plusone.plus(1)
2
>>> plusone.plus(2)
3
>>> plusone.plus(3)
4
>>> █
```

A red curly brace on the left side of the window groups the three calls to `plusone.plus()`. To its right, the text "arguments inside ()" is written in red. A large blue callout bubble at the bottom right contains the text "Can **call** as many times as you want".

## Function Definition

- Command to **do** the function
- Belongs inside a module

A screenshot of a code editor window titled "plusone.py". The file path is shown as "Users > wmmwhite > Documents > Professional > Courses > CS-1110 > Lectur". The code defines a function `plus`:

```
def plus(n):
 """
 Returns: the value of n+1
 """
 return (n+1)
```

A large blue callout bubble in the center of the code editor contains the text "But only define function **ONCE**".

## **Clickers (If Time)**

# Reading Documentation

---

## Weird Module

`weird.isclose(a, b, [tolerance])`

Returns True if the float `a` is close enough to `b`, and False otherwise.

The function determines the absolute difference between `a` and `b`. If this value is less than the optional `tolerance` argument, this function returns `True`, and otherwise it returns `False`. If `tolerance` is not specified, the function returns `True` only if the difference is less than `0.000001` (`1e-6`).

For example:

```
>>> weird.isclose(1.0,1.00005)
False
>>> weird.isclose(1.0,1.00005,0.001)
True
```

- Parameters:**
- `a` (`float`) – The first number to compare
  - `b` (`float`) – The second number to compare
  - `tolerance` (`float > 0`) – The maximum allowed distance between `a` and `b`

**Returns:** True if the float `a` is close enough to `b`, and False otherwise.

**Return type:** `bool`

# Reading `isclose`

---

- Assume that we type

```
>>> import weird
```

```
>>> isclose(2.000005,2.0)
```

- What is the result (value)?

- A: True
- B: False
- C: An error!
- D: Nothing!
- E: I do not know

# Reading `isclose`

---

- Assume that we type

```
>>> import weird
```

```
>>> isclose(2.000005,2.0)
```

- What is the result (value)?

- A: True
- B: False
- C: An error!
- D: Nothing!
- E: I do not know

**CORRECT**

# Reading `isclose`

---

- Assume that we type

```
>>> import weird
```

```
>>> weird.isclose(2.000005,2.0)
```

- What is the result (value)?

- A: True
- B: False
- C: An error!
- D: Nothing!
- E: I do not know

# Reading `isclose`

---

- Assume that we type

```
>>> import weird
```

```
>>> weird.isclose(2.000005,2.0)
```

- What is the result (value)?

A: True

B: False

C: An error!

D: Nothing!

E: I do not know

**CORRECT**

# Reading `isclose`

---

- Assume that we type

```
>>> import weird
```

```
>>> weird.isclose(2.0,3.0,4.0)
```

- What is the result (value)?

- A: True
- B: False
- C: An error!
- D: Nothing!
- E: I do not know

# Reading `isclose`

---

- Assume that we type

```
>>> import weird
```

```
>>> weird.isclose(2.0,3.0,4.0)
```

- What is the result (value)?

A: True

**CORRECT**

B: False

C: An error!

D: Nothing!

E: I do not know