

## 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

# Announcement: Partner Finding Social

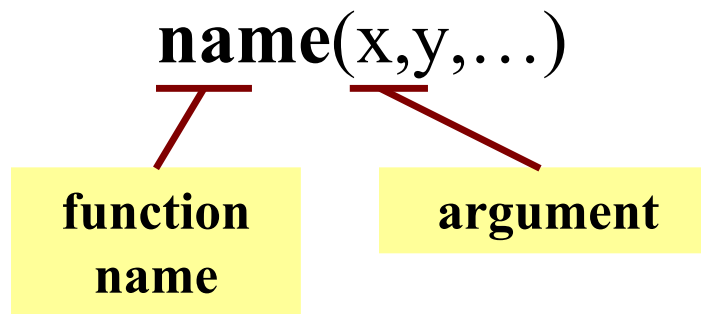
---

- **Reminder:** Assignments can be done in **pairs**
  - Can work with anyone in class (in any section)
  - Can change partners every assignment
- Having a mixer to help find a partner
  - 5-7pm Wednesday, September 11<sup>th</sup>
  - Duffield Atrium (all of it)
  - But make sure you find the 1110 group!
- No registration is necessary

# Function Calls

---

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

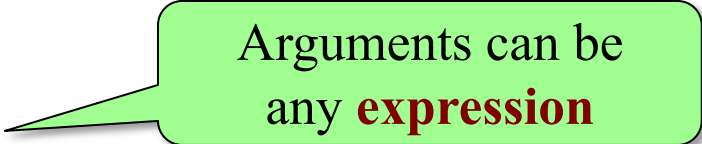


- **Arguments** are
  - **Expressions**, not values
  - Separated by commas

# Built-In Functions

---

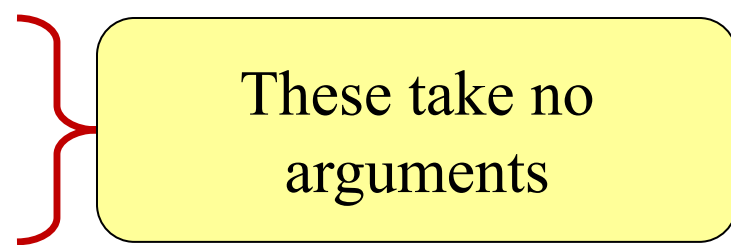
- Python has several math functions
  - `round(2.34)`
  - `max(a+3,24)`
- You have seen many functions already
  - Type casting functions: `int()`, `float()`, `bool()`
- Documentation of all of these are online
  - <https://docs.python.org/3/library/functions.html>
  - Most of these are two advanced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



Arguments can be  
any **expression**

# 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()`

These take no arguments
- How know which one? Read documentation.

# 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
```

Functions require math prefix!

```
>>> 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
```

# 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
```

## Other Modules

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

# Using the from Keyword

```
>>> import math
```

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

Must prefix with  
module name

```
3.141592653589793
```

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

```
>>> pi
```

No prefix needed  
for variable pi

```
3.141592653589793
```

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

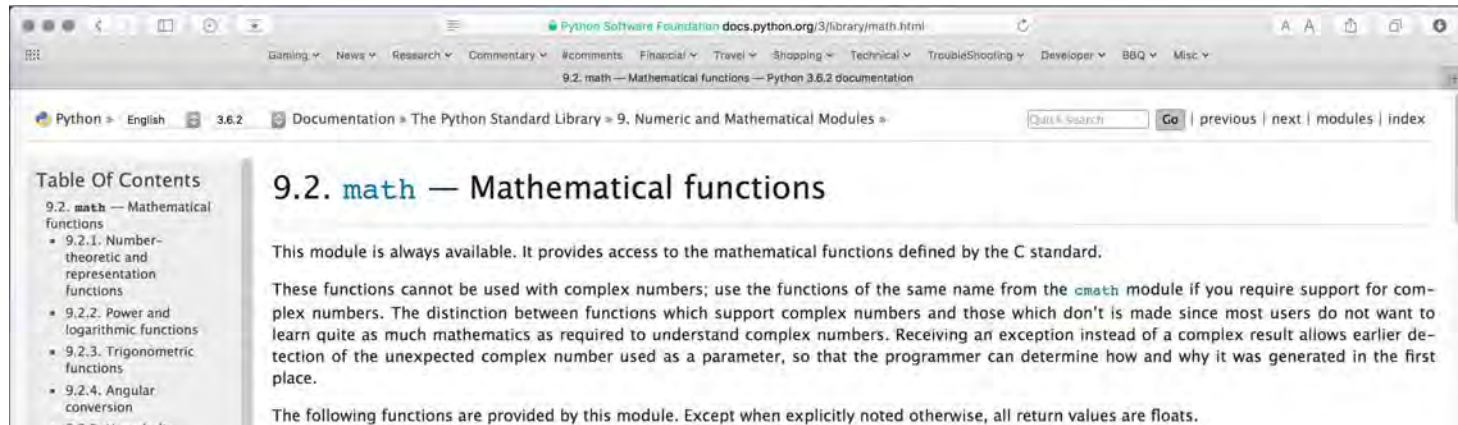
```
>>> cos(pi)
```

```
-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



**`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 - 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>

# Reading the Python Documentation

The screenshot shows the Python documentation page for the `math` module. The page title is "9.2. `math` — Mathematical functions". The page content includes a description of the module and a list of functions. The following functions are provided by this module. Except when explicitly noted otherwise, all return values are floats.

**Function name**: `math.ceil(x)`

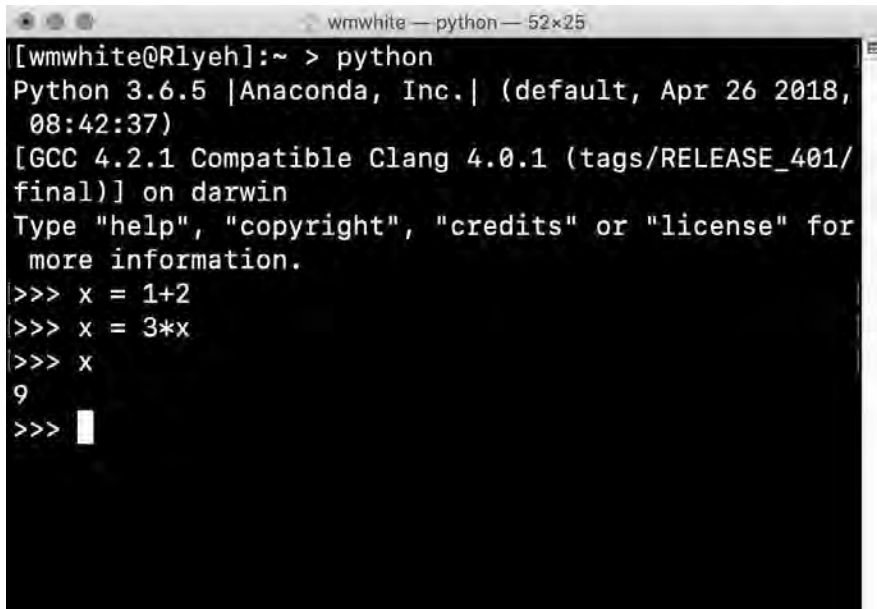
**Possible arguments**: `x`

**Module**: `math`

**What the function evaluates to**: Return the ceiling of `x`, the smallest integer greater than or equal to `x`.

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

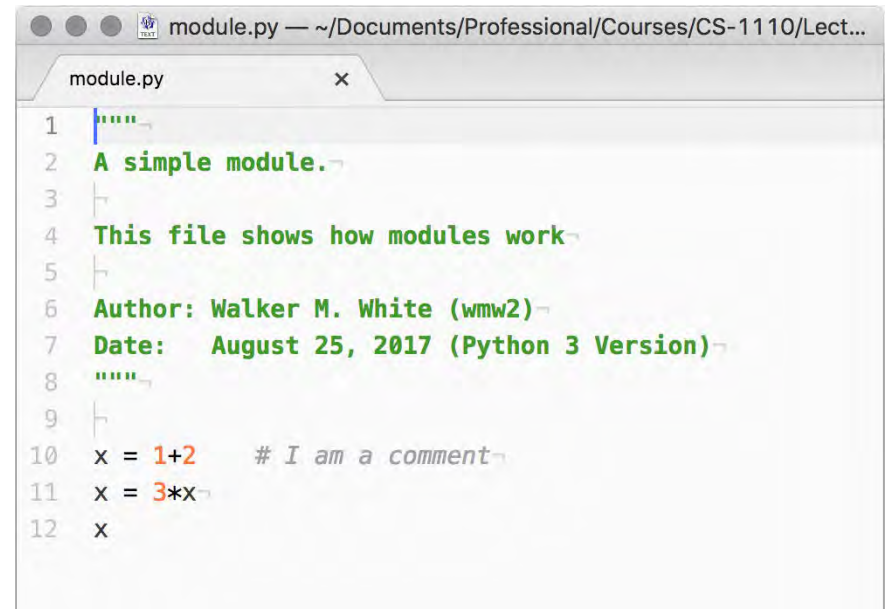
# Interactive Shell vs. Modules



A screenshot of a terminal window titled 'wmwhite — python — 52x25'. The prompt is '[wmwhite@Rlyeh]:~ >'. The user has entered 'python', which has launched Python 3.6.5. The output shows the Python version, the Anaconda environment, and the GCC/Clang compiler information. The user then enters a series of commands: 'x = 1+2', 'x = 3\*x', and 'x', which are executed line by line, resulting in the output '9'.

```
[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
>>> 
```

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



A screenshot of a code editor window titled 'module.py — ~/Documents/Professional/Courses/CS-1110/Lect...'. The file 'module.py' is open, showing a simple Python module. The code includes a docstring, a comment, and two lines of code that calculate 'x'.

```
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
```

- **Write in a code editor**
  - We use Pulsar
  - 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
```

**Single line comment**  
(not executed)

```
x = 1+2
```

```
x = 3*x
```

```
x
```

# Using a Module

---

## Module Contents

---

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

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

```
"""
```

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

**Single line comment**  
(not executed)

```
# 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
```

```
"""
```

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

```
# This is a comment
```

**Single line comment**  
(not executed)

```
x = 1+2
```

```
x = 3*x
```

```
x
```

**Commands**  
Executed on import

# Using a Module

## Module Contents

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

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

```
"""
```

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

```
# This is a comment
```

**Single line comment**  
(not executed)

```
x = 1+2
```

```
x = 3*x
```

**Commands**  
Executed on import

```
x
```

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>
```

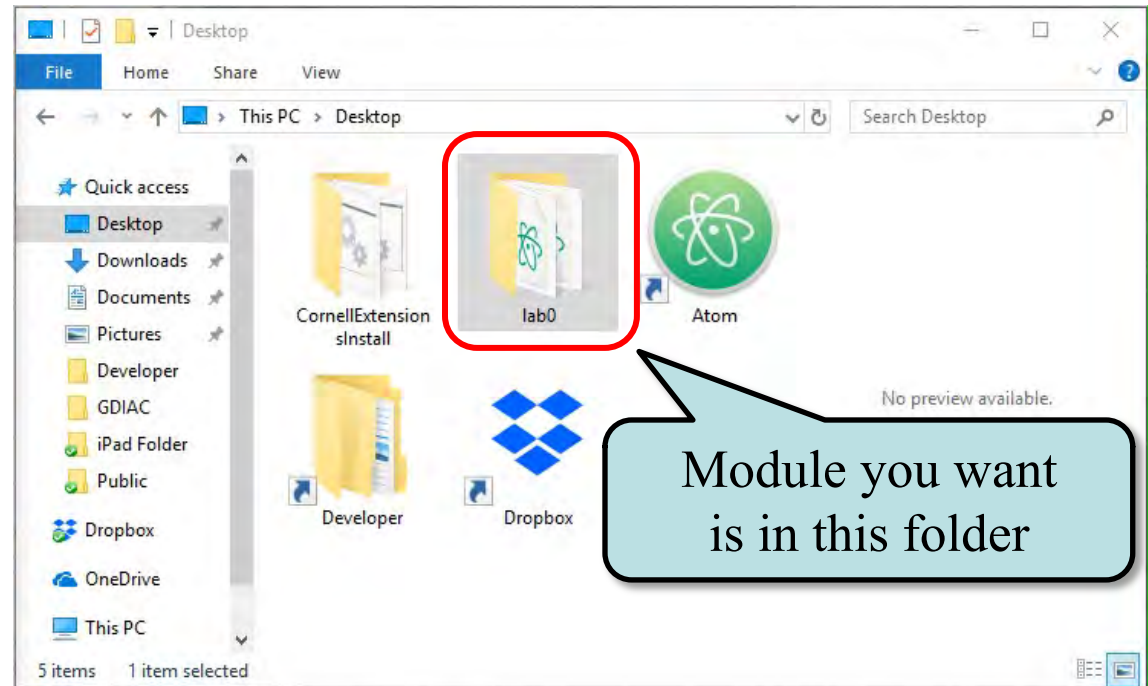
```
NameError: name 'x' is not defined
```

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

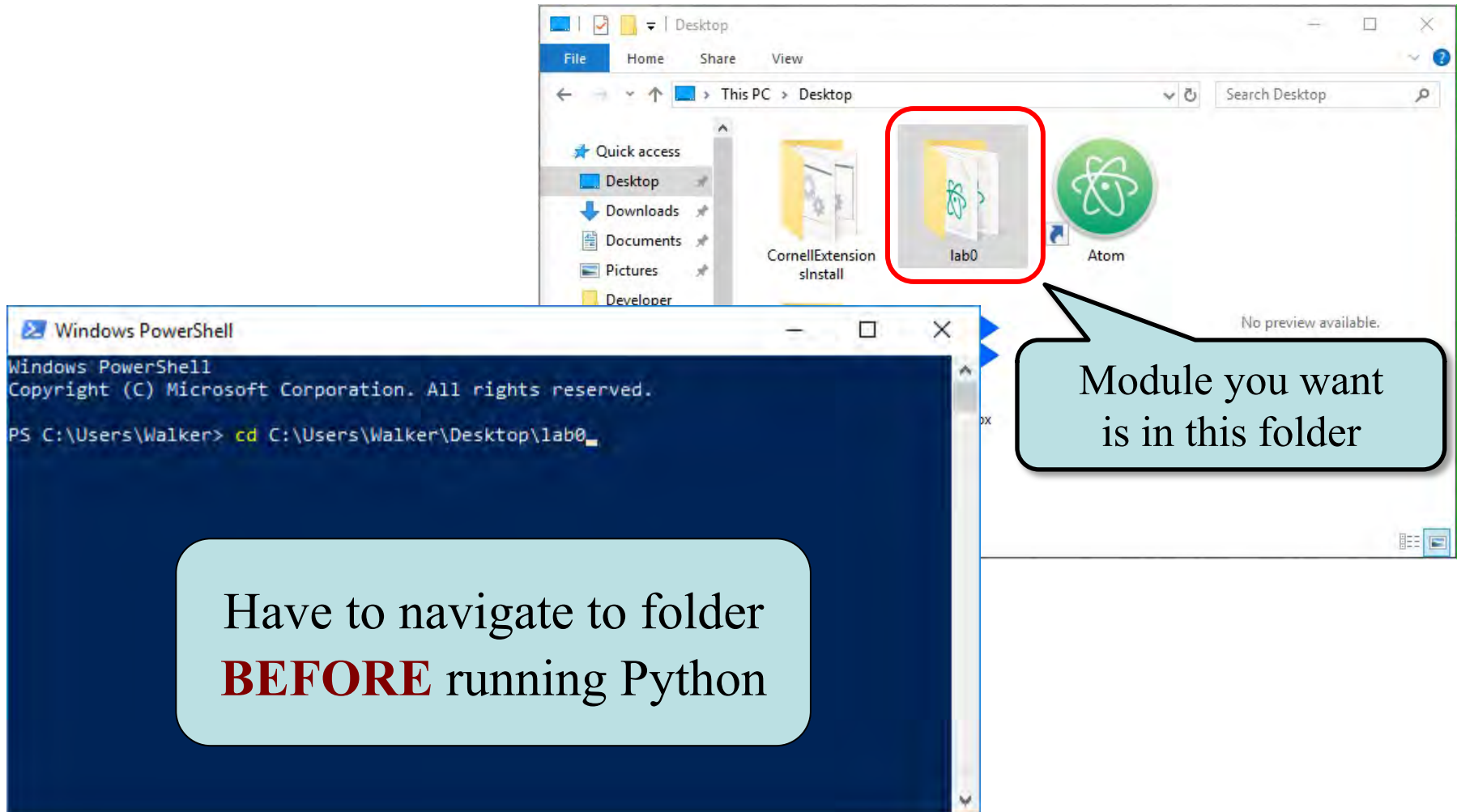
```
9
```

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

# Modules Must be in Working Directory!



# Modules Must be in Working Directory!



File Explorer window showing the Desktop. The 'lab0' folder is highlighted with a red box. A speech bubble points to it with the text "Module you want is in this folder".

Windows PowerShell window showing the command:

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

PS C:\Users\Walker> cd C:\Users\Walker\Desktop\lab0
```

Have to navigate to folder **BEFORE** running Python

# 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)
```



Only difference

# Scripts and Print Statements

## module.py

```
modules — eCornell — -bash — 52x15
[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

```
modules — eCornell — -bash — 52x15
[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

script.py

```
modules — eCornell — -bash — 52x15
[wmwhite@Rlyeh]:modules > python module.py
[wmwhite@Rlyeh]:modules > █
```

```
modules — eCornell — -bash — 52x15
[wmwhite@Rlyeh]:modules > python script.py
9
[wmwhite@Rlyeh]:modules > █
```

When you run a script,  
only statements are executed

- Looks like this time!
- Python executed the following:
  - Executed the assignments
  - Skipped the last line ('x' is not a statement)
- Executed the assignments
- Executed the last line (Prints the contents of x)

# 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'
```

# Making a Script Interactive

---

"""

A script showing off input.

This file shows how to make a script interactive.

"""

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

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

```
Give me something: Hello
```

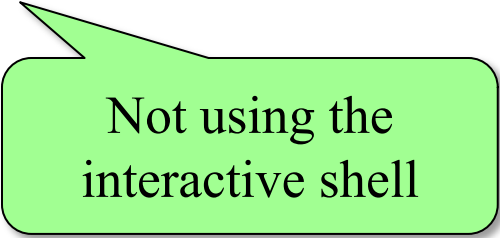
```
You said: Hello
```

```
[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'
```

Value is a string.

```
>>> x + 1
```

```
TypeError: must be str, not int
```

```
>>> x = int(x)
```

```
>>> x+1
```

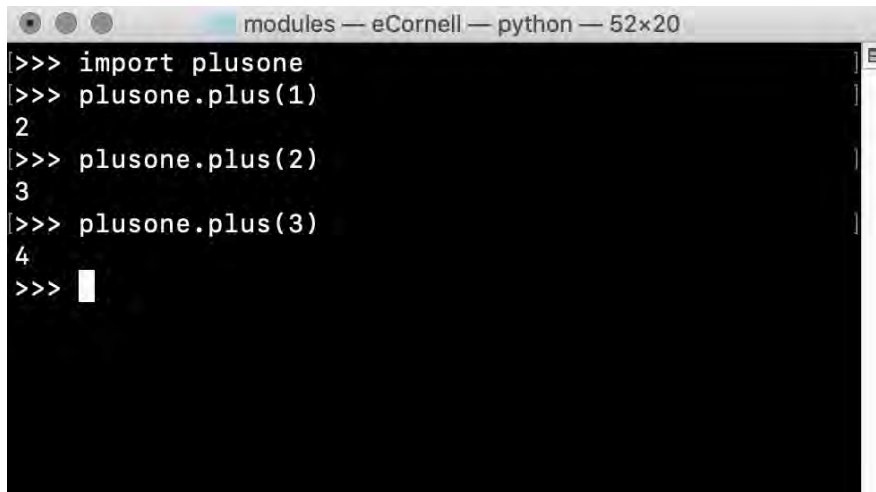
```
4
```

Must convert to  
int.

# Next Time: Defining Functions

## Function Call

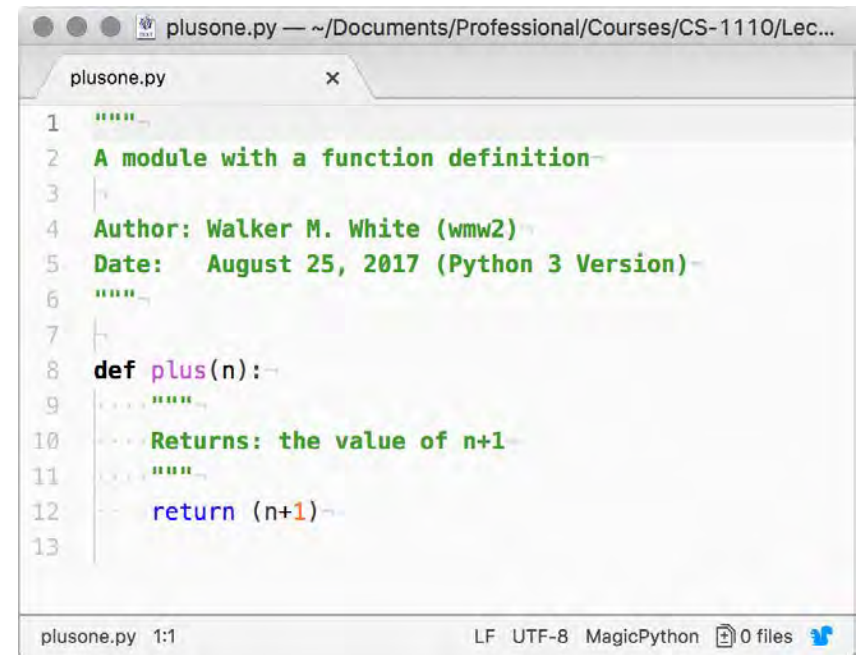
- Command to **do** the function
- Can put it anywhere
  - In the Python shell
  - Inside another module



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

## Function Definition

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

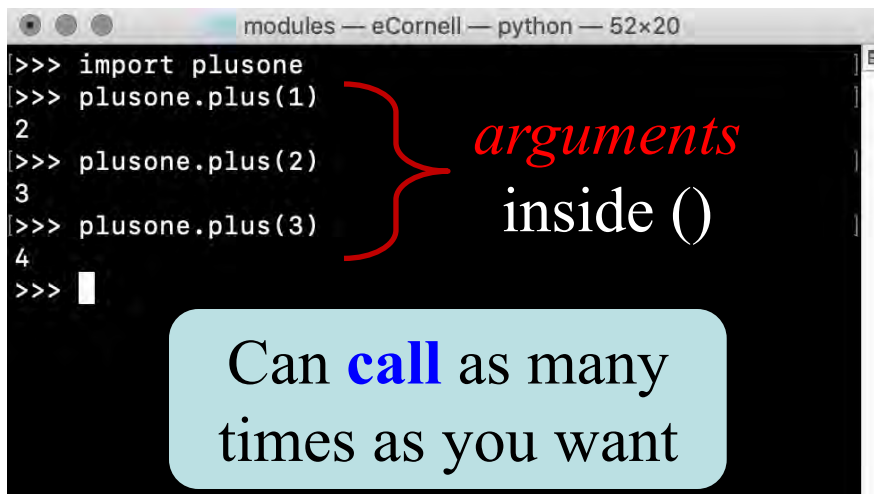


```
plusone.py — ~/Documents/Professional/Courses/CS-1110/Lec...
plusone.py x
1  """
2  A module with a function definition
3
4  Author: Walker M. White (wmw2)
5  Date:   August 25, 2017 (Python 3 Version)
6  """
7
8  def plus(n):
9      """
10     Returns: the value of n+1
11     """
12     return (n+1)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

# Next Time: Defining Functions

## Function Call

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



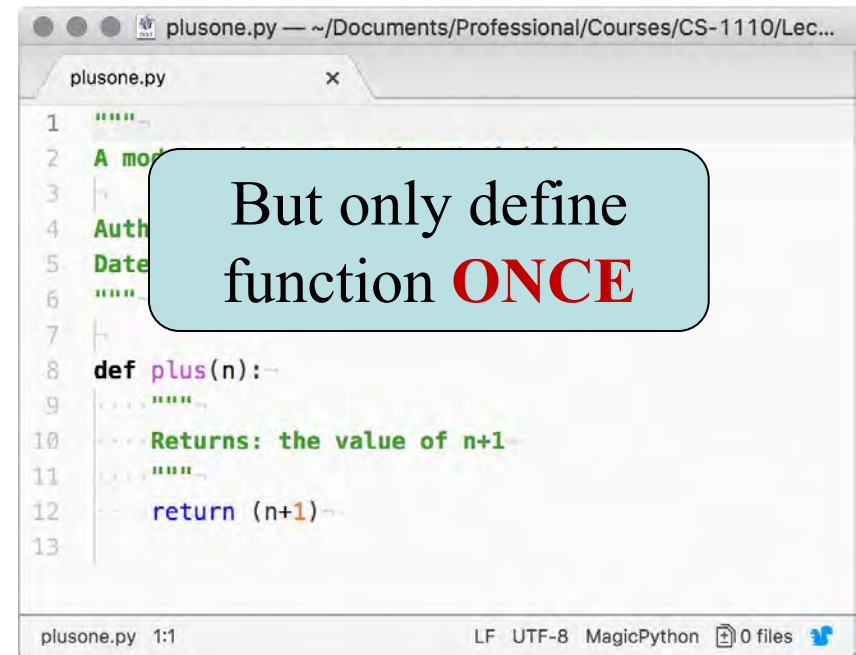
A screenshot of a Python shell window titled "modules — eCornell — python — 52x20". The shell shows the following commands and output:

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

A red curly bracket groups the arguments 1, 2, and 3, with the text *arguments* in red. To the right of the bracket, the text "inside ()" is written in white. A light blue rounded rectangle at the bottom 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 Python file editor window titled "plusone.py — ~/Documents/Professional/Courses/CS-1110/Lec...". The editor shows the following code:

```
1  """
2  A module
3
4  Auth
5  Date
6  """
7
8  def plus(n):
9      """
10     Returns: the value of n+1
11     """
12     return (n+1)
13
```

A light blue rounded rectangle is overlaid on the code, containing the text "But only define function **ONCE**". The status bar at the bottom shows "plusone.py 1:1", "LF UTF-8", "MagicPython", and "0 files".

## **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 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

<http://cs1110.cs.cornell.edu/weird/>

# 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!      **CORRECT**

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

# Reading `isclose`

---

- Assume that we type

```
>>> import weird
>>> weird.isclose(2.000005,2.0)
```
- What is the result (value)?

A: True

B: False

**CORRECT**

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

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          | <b>CORRECT</b> |
| B: False         |                |
| C: An error!     |                |
| D: Nothing!      |                |
| E: I do not know |                |