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

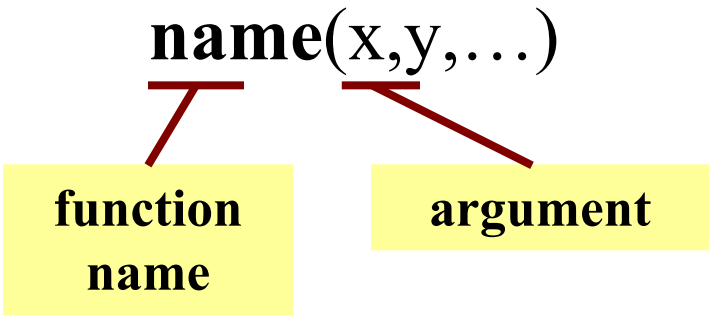
- **Today**
  - **Lesson 3:** Function Calls
  - **Lesson 4:** Modules
  - Videos 4.1-4.5
- Next Time
  - Video 4.6 of **Lesson 4**
  - **Lesson 5:** Function Defs
- Also *skim* Python API

# 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
  - 4:30-5:30pm Thursday, September 3rd
  - Gates 114 (Past Gimme Coffee)
  - 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)`

Arguments can be any **expression**

- 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

# 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
>>> 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
```

To access math functions

Functions require math 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
```

To access math functions

Functions require math prefix!

Module has variables too!

# 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
```

To access math functions

Functions require math prefix!

Module has variables too!

## 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
3.141592653589793
>>> from math import pi
>>> pi
3.141592653589793
>>> from math import *
>>> cos(pi)
-1.0
```

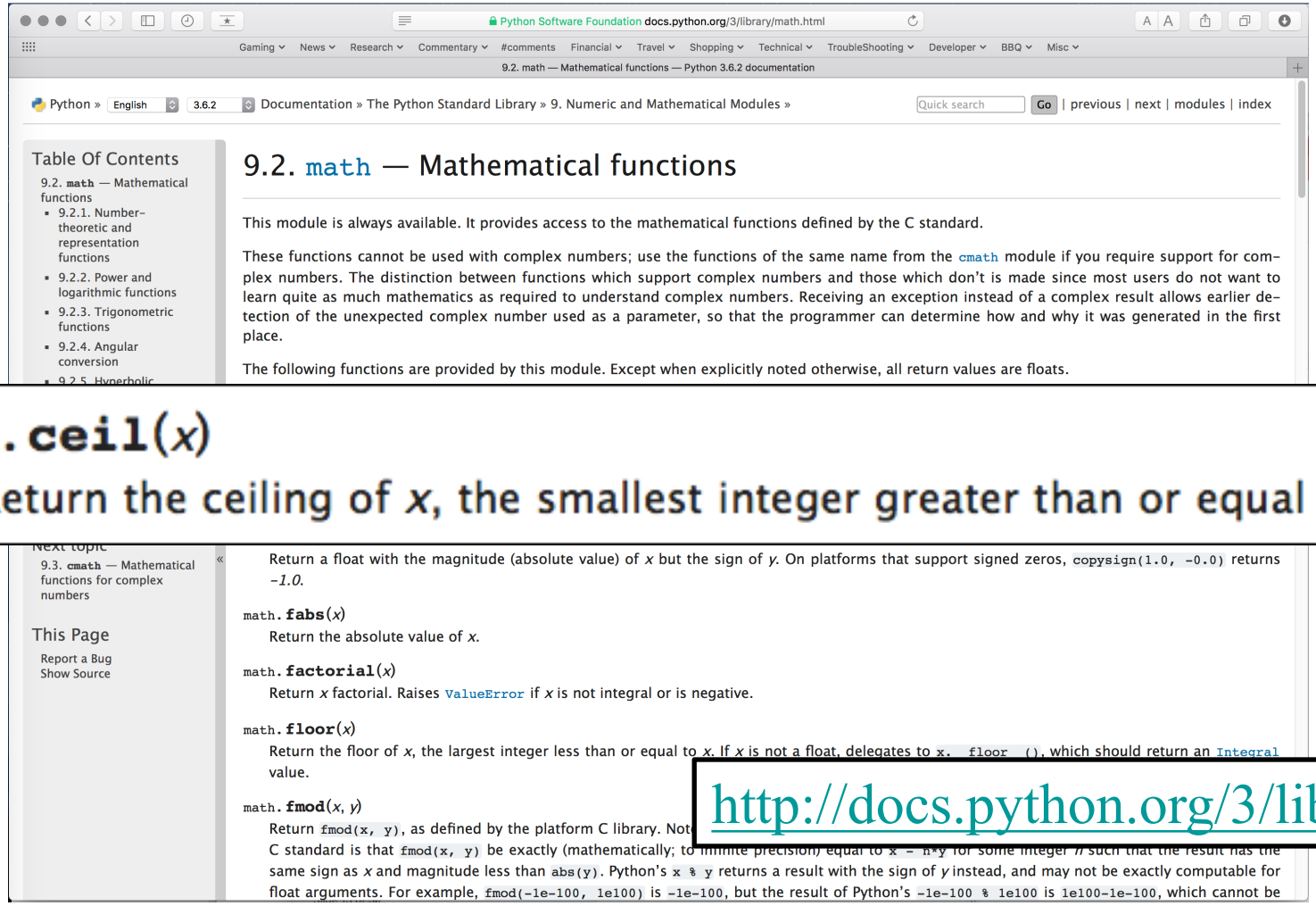
Must prefix with module name

No prefix needed for variable pi

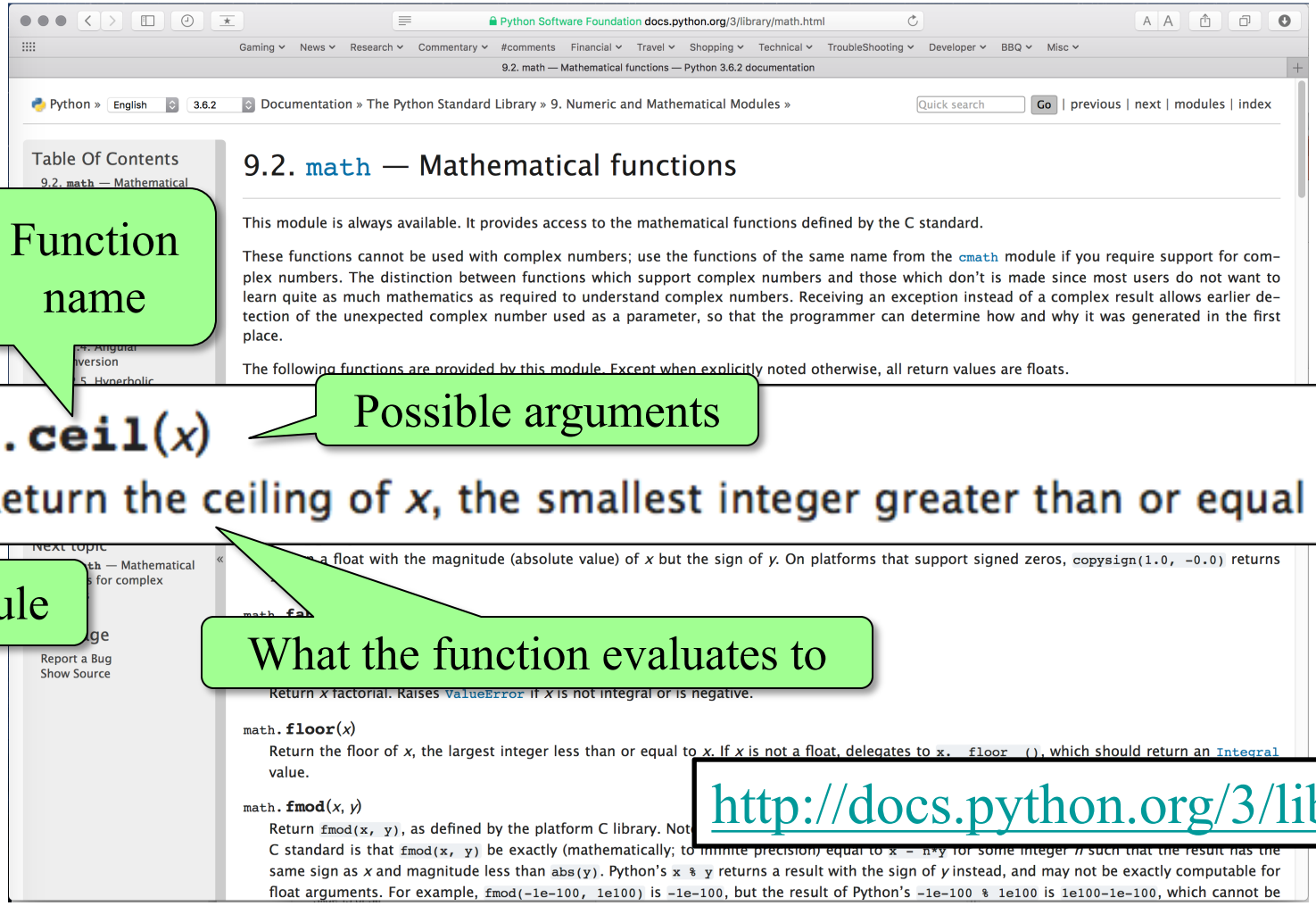
No prefix needed for anything in math

- 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

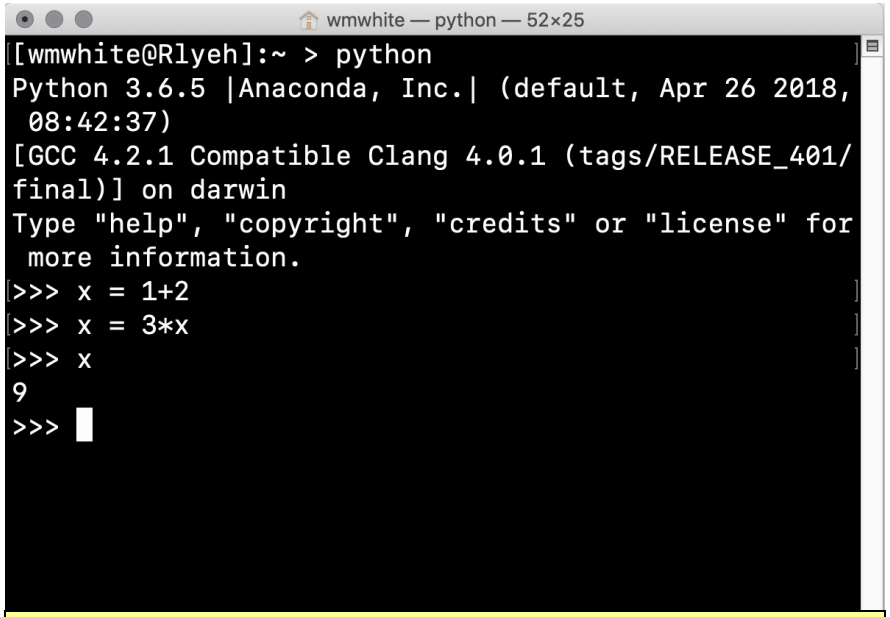


# Reading the Python Documentation

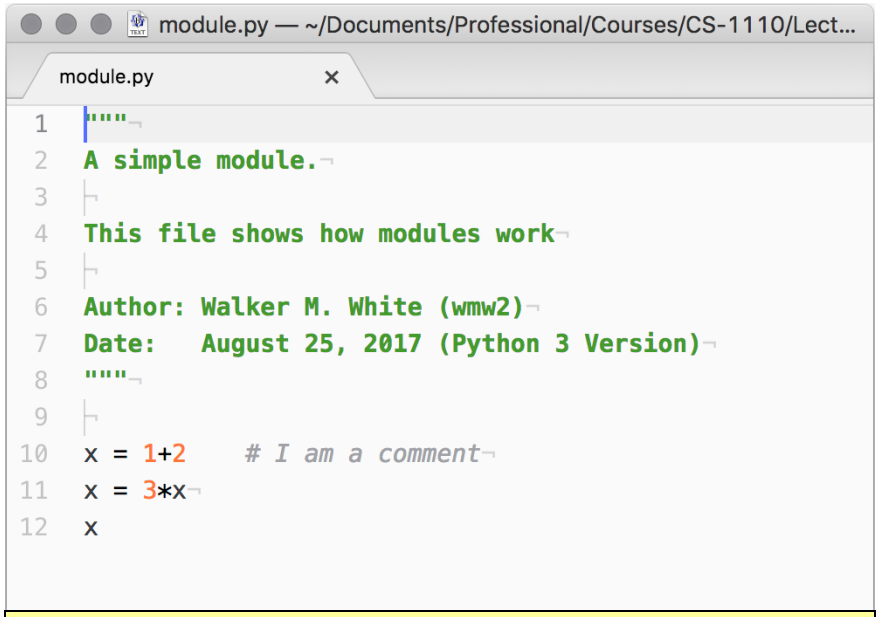



http://docs.python.org/3/library

# Interactive Shell vs. Modules




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




- **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
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
```

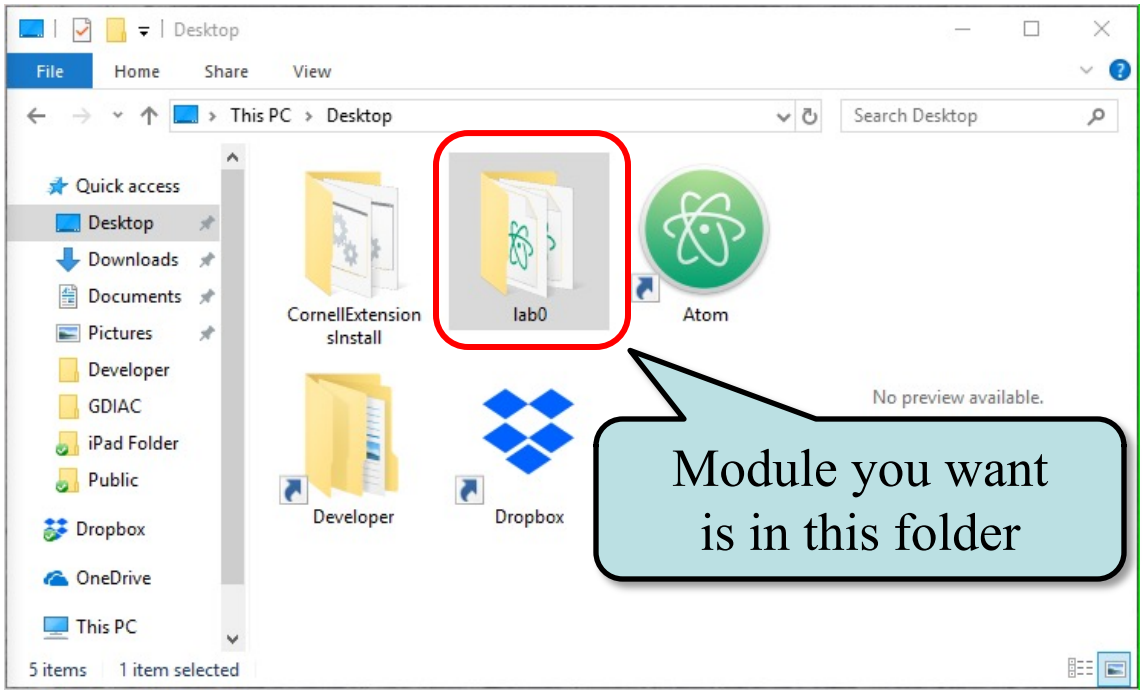
"**Module data**" must be prefixed by module name

Prints **docstring** and module contents

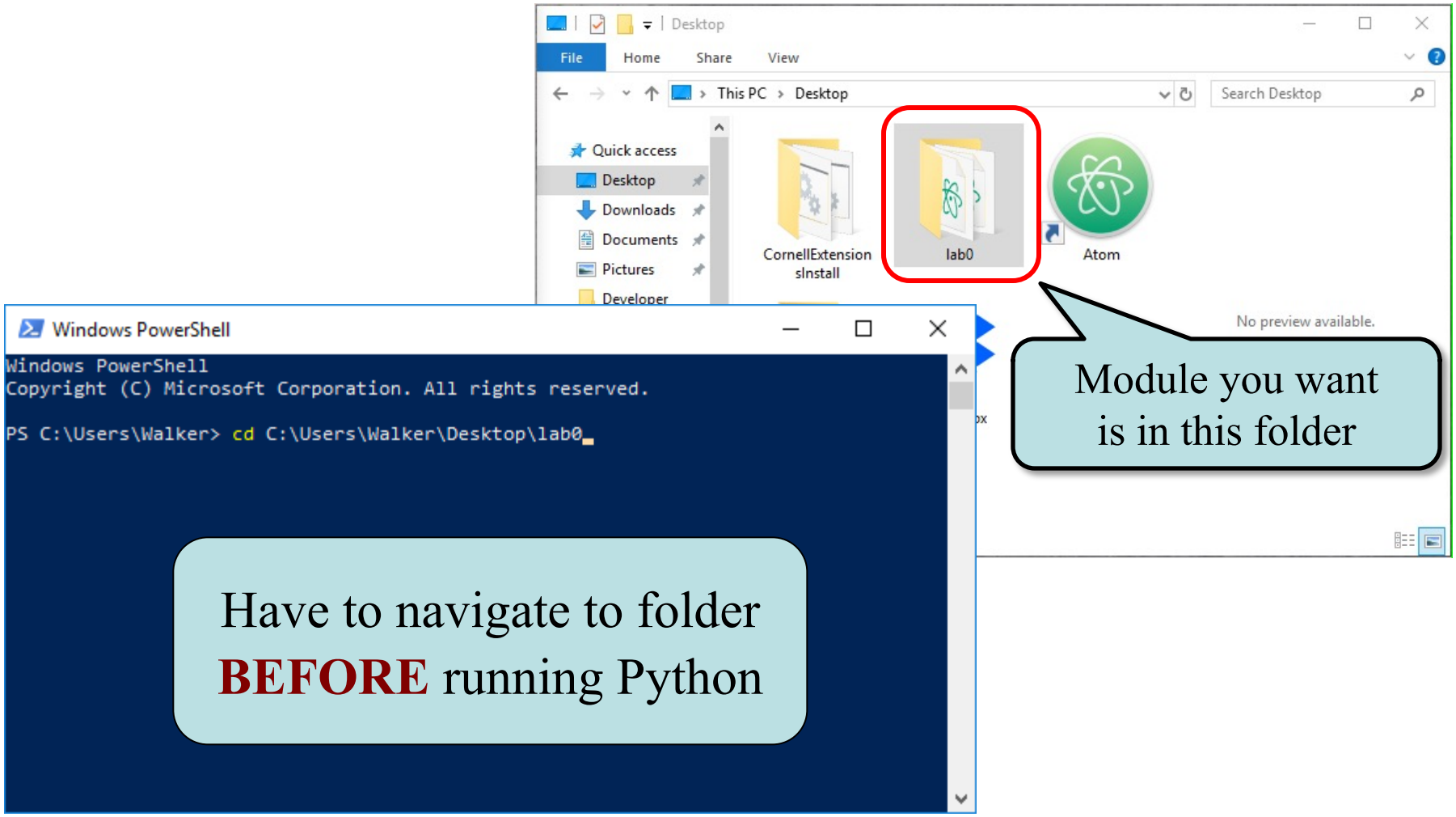
## Python Shell

```
>>> 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

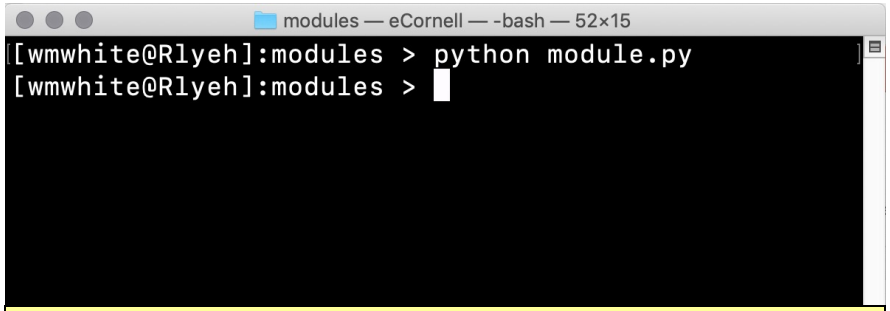
```
""" A simple script.

This file shows why we use print
"""

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

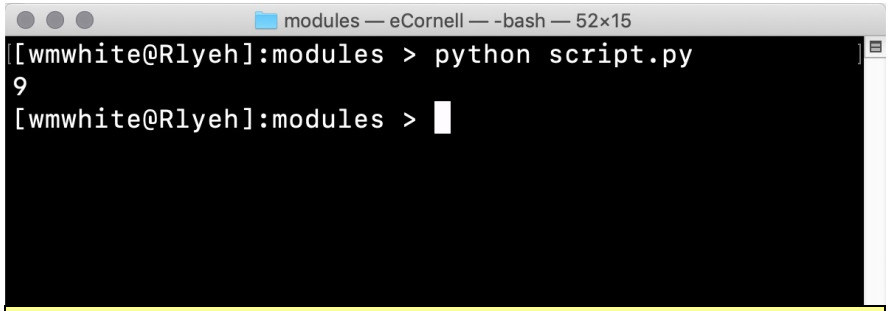
# Scripts and Print Statements

## module.py




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

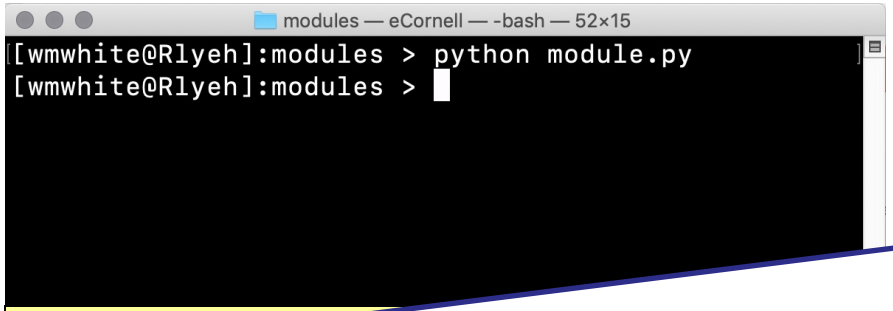
## script.py




- 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




- Looks l
- Python
  - Execu… assignments
  - Skipped the last line
    (‘x’ is not a statement)

## script.py


…his time!

…llowing:

- Executed the assignments
- Executed the last line
  (Prints the contents of x)

When you run a script,
only statements are executed

# User Input

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

No space after the prompt.

Proper space after prompt.

Assign result to variable.

# 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'
>>> x + 1
TypeError: must be str, not int
>>> x = int(x)
>>> x+1
4
```

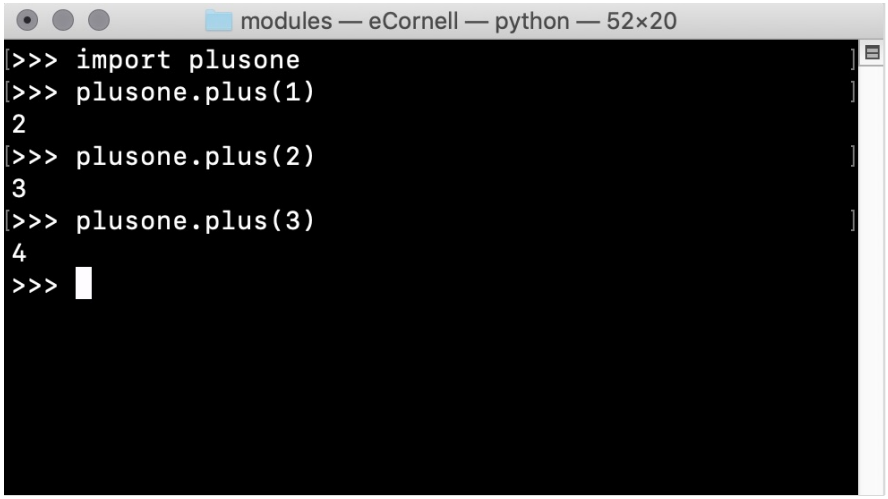
Value is a string.

Must convert to int.

# Next Time: Defining Functions

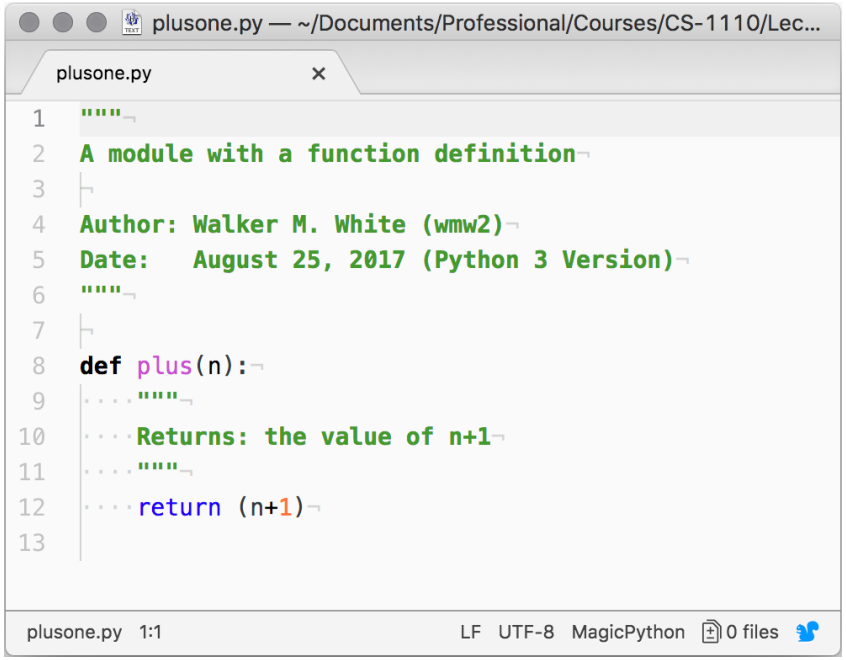
## Function Call

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




## Function Definition

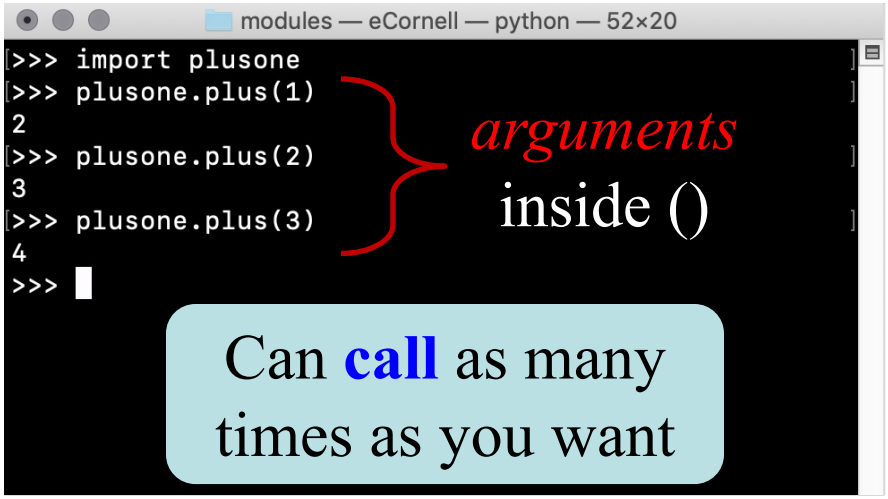
- Command to **do** the function
- Belongs inside a module



# Next Time: Defining Functions

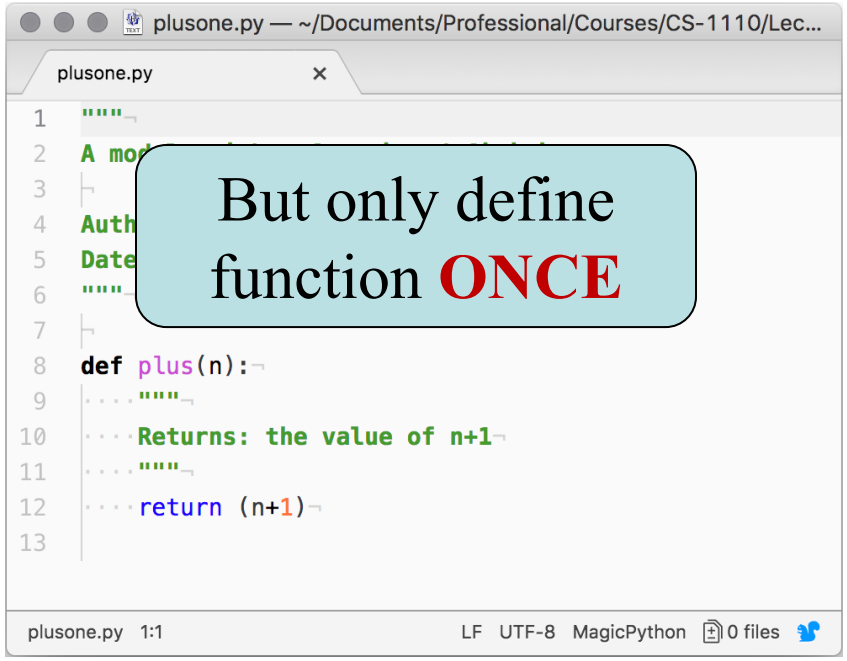
## Function Call

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




## Function Definition

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



# Polling (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 functions 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 functions 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 **CORRECT**
B: False
C: An error!
D: Nothing!
E: I do not know