

Lecture 2

Functions & Modules

Want to be a part of the largest network
of campus women in STEM?



THE SCIENTISTA FOUNDATION

WOMEN IN STEM

Accepting applications until
11:59PM on Feb 12 2017
Apply here: bit.ly/2ksNk3k

Information Session
4:45-5:45PM on Feb 1
Goldwin Smith G76



ScientistaCornellUniversity@gmail.com



[/ScientistaCornell](https://www.facebook.com/ScientistaCornell)



[@Scientista_Cornell](https://www.instagram.com/Scientista_Cornell)

Labs Start this Week

- Labs are Wednesday here Thurston 205
 - Can do them at home (instructions on web)
 - Bring a laptop if you come to the lab
- Lab is due **before beginning** of next lab
 - Can check off *on the day it is handed out*
 - Can check off during *consultant hours*
- **Cannot spend lab time on previous lab**

Announcements

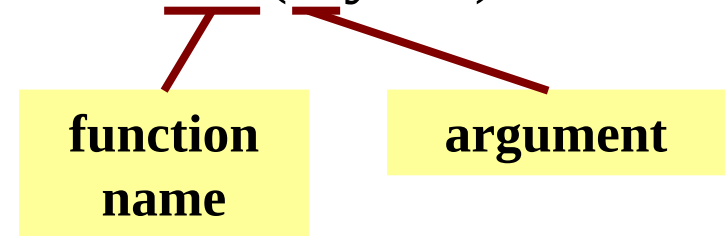
If Not Done Already

Reading (Optional)

- Enroll in Piazza
- Sign into CMS
 - To make sure you've been added
- Website
 - Text
 - Video!
- Readings this week
 - Chapter 1 (browse)
 - Chapter 2 (in detail)
 - Chapter 3 (in detail)
- Next week
 - Sections 8.1, 8.2, 8.4, 8.5

Function Calls

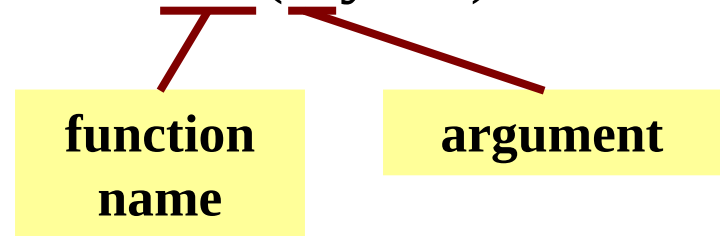
- Python supports expressions with math-like functions
 - A function in an expression is a **function call**
 - Will explain the meaning of this later
- Function expressions have the form **fun**(x,y,...)



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

Function Calls

- Python supports expressions with math-like functions
 - A function in an expression is a **function call**
 - Will explain the meaning of this later
- Function expressions have the form **fun**(x,y,...)



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

Arguments can be any **expression**

Built-In Functions

- You have seen many functions already
 - Type casting functions: `int()`, `float()`, `bool()`
 - Dynamically type an expression: `type()`
 - Help function: `help()`
 - Quit function: `quit()`
- `print <string>` is **not** a function call
 - It is simply a statement (like assignment)
 - But it is in Python 3.x: `print(<string>)`

Arguments go in (),
but `name()` refers to
function in general

Built-in Functions vs Modules

- The number of built-in functions is small
 - <http://docs.python.org/2/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
```

```
8/29/16  
>>> math.cos(math.pi)
```

Example: Module math

```
>>> import math
```

To access math functions

```
>>> 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
```

```
8/29/16  
>>> math.cos(math.pi)
```

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
```

```
3.141592653589793
```

```
8/29/16  
>>> math.cos(math.pi)
```

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
```

Module has variables too!

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

```
3.141592653589793
```

```
8/29/16 >>> math.cos(math.pi)
```

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
```

Module has variables too!

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

```
3.141592653589793
```

```
8/29/16  
>>> math.cos(math.pi)
```

Other Modules

- **io**
 - Read/write from files
- **random**
 - Generate random numbers
 - Can pick any distribution
- **string**
 - Useful string functions
- **sys**
 - Information about your OS

Reading the Python Documentation

The screenshot shows a web browser displaying the Python 2.7.3 documentation for the `math` module. The browser's address bar shows `http://docs.python.org/library/math.html`. The page has a dark blue sidebar on the left with a 'Table Of Contents' for the `math` module, listing sub-sections like 'Number-theoretic and representation functions', 'Power and logarithmic functions', 'Trigonometric functions', 'Angular conversion', 'Hyperbolic functions', 'Special functions', and 'Constants'. Below the sidebar, the main content area is titled '9.2. math — Mathematical functions'. It contains introductory text about the module's availability and its relationship to the C standard, followed by a section '9.2.1. Number-theoretic and representation functions'. This section lists several functions: `math.ceil(x)`, `math.copysign(x, y)`, `math.fabs(x)`, `math.factorial(x)`, and `math.floor(x)`, each with a brief description of its behavior. A search bar is located at the bottom left of the sidebar. A red box highlights the URL `http://docs.python.org/library` in the bottom right corner of the page.

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.

9.2.1. Number-theoretic and representation functions

`math.ceil(x)`
Return the ceiling of `x` as a float, the smallest integer value greater than or equal to `x`.

`math.copysign(x, y)`
Return `x` with the sign of `y`. On a platform that supports signed zeros, `copysign(1.0, -0.0)` returns `-1.0`.

New in version 2.6.

`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.

New in version 2.6.

`math.floor(x)`
Return the floor of `x` as a float, the largest integer value less than or equal to `x`.

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

Reading the Python Documentation

9.2. math — Mathematical functions — Python v2.7.3 documentation

python library

Python v2.7.3 documentation » The Python Standard Library » 9. Numeric and Mathematical Modules » previous | next | modules | index

Table Of Contents

9.2. math — Mathematical functions

- 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
- 9.2.5. Hyperbolic functions
- 9.2.6. Special 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.

This Page

Report a Bug

Show Source

Quick search

Go

Enter search terms or a module, class or function name.

Return `x` with the sign of `y`. On a platform that supports signed zeros, `copysign(1.0, -0.0)` returns `-1.0`.

New in version 2.6.

`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.

New in version 2.6.

`math.floor(x)`

Return the floor of `x` as a float, the largest integer value less than or equal to `x`.

Reading the Python Documentation

The image shows a screenshot of the Python 2.7.3 documentation for the `math` module. The browser address bar shows `http://docs.python.org/library/math.html`. The page title is "9.2. math — Mathematical functions". The left sidebar contains a "Table Of Contents" for the `math` module, listing 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", "9.2.5. Hyperbolic functions", and "9.2.6. Special functions". The main content area starts with "9.2. math — Mathematical functions" and describes the module's purpose. It then lists functions provided by the module, including `ceil`, `copysign`, `factorial`, and `floor`. Annotations are present: a green speech bubble labeled "Function name" points to `math.ceil(x)`; a green speech bubble labeled "Possible arguments" points to the parameter `x`; a green speech bubble labeled "Module" points to `math`; a green speech bubble labeled "What the function evaluates to" points to the description of the `factorial` function; and a green speech bubble labeled "Function name" points to the `factorial` function name. A green box at the bottom right contains the URL `http://docs.python.org/library`.

Function name

Possible arguments

Module

What the function evaluates to

`math.ceil(x)`

Return the ceiling of `x` as a float, the smallest integer value greater than or equal to `x`.

`math.factorial(x)`

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

New in version 2.6.

`math.floor(x)`

Return the floor of `x` as a float, the largest integer value less than or equal to `x`.

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

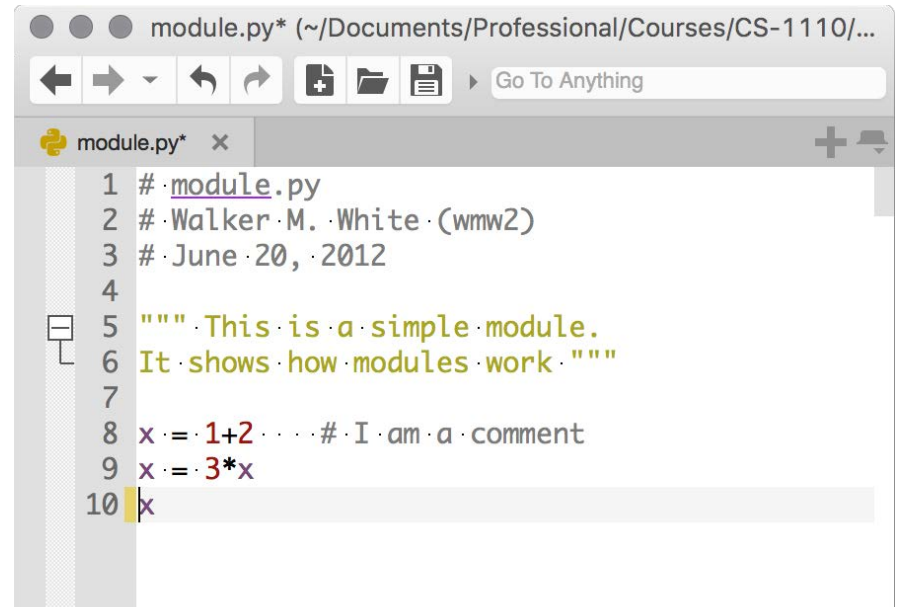
Interactive Shell vs. Modules



A screenshot of a terminal window titled 'wmwhite — python — 52x25'. The window shows the output of a 'python' command, including version information (Python 2.7.12, Anaconda 4.1.1) and a prompt. The user enters several lines of code: 'x = 1+2', 'x = 3*x', and 'x'. The output shows the variable 'x' being updated and then printed as '9'.

```
Last login: Fri Jul 29 21:42:45 on ttys002
[wmwhite@Ryleh]:~ > python
Python 2.7.12 |Anaconda 4.1.1 (x86_64)| (default, Jul 2 2016, 17:43:17)
[GCC 4.2.1 (Based on Apple Inc. build 5658) (LLVM build 2336.11.00)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
Anaconda is brought to you by Continuum Analytics.
Please check out: http://continuum.io/thanks and https://anaconda.org
>>> 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 text editor window titled 'module.py* (~/.Documents/Professional/Courses/CS-1110/...)'. The editor shows a Python file with comments and a simple module structure. The code includes a docstring and two lines of code that calculate 'x'.

```
1 # module.py
2 # Walker M. White (wmw2)
3 # June 20, 2012
4
5 """This is a simple module.
6 It shows how modules work."""
7
8 x = 1+2 # I am a comment
9 x = 3*x
10 x
```

- **Write in a text editor**
 - We use Komodo Edit
 - But anything will work
- Load module with import

Using a Module

Module Contents

```
# module.py
```

```
""" This is a simple module.  
It shows how modules  
work"""
```

```
x = 1+2
```

```
x = 3*x
```

```
x
```

Using a Module

Module Contents

module.py

Single line comment
(not executed)

""" This is a simple module.

It shows how modules
work"""

x = 1+2

x = 3*x

x

Using a Module

Module Contents

```
# module.py
```

Single line comment
(not executed)

```
""" This is a simple module.  
It shows how modules  
work"""
```

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

```
x = 1+2
```

```
x = 3*x
```

```
x
```

Using a Module

Module Contents

```
# module.py
```

Single line comment
(not executed)

```
""" This is a simple module.  
It shows how modules  
work"""
```

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

```
x = 1+2
```

```
x = 3*x
```

```
x
```

Commands
Executed on import

Using a Module

Module Contents

```
# module.py
```

Single line comment
(not executed)

```
""" This is a simple module.  
It shows how modules  
work"""
```

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

```
x = 1+2
```

Commands
Executed on import

```
x = 3*x
```

```
x
```

Not a command.
import **ignores this**

Using a Module

Module Contents

```
# module.py
```

```
""" This is a simple module.  
It shows how modules  
work"""
```

```
x = 1+2
```

```
x = 3*x
```

```
x
```

Python Shell

```
>>> import module
```

```
>>>x
```

Using a Module

Module Contents

```
# module.py
```

```
""" This is a simple module.  
It shows how modules  
work"""
```

```
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

```
# module.py
```

```
""" This is a simple module.  
It shows how modules  
work"""
```

```
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>
```

```
module.x
```

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

```
>>>
```

```
9
```

Using a Module

Module Contents

```
# module.py
```

```
""" This is a simple module.  
It shows how modules  
work"""
```

```
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>
```

```
module.x
```

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

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

```
9
```

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 *
```

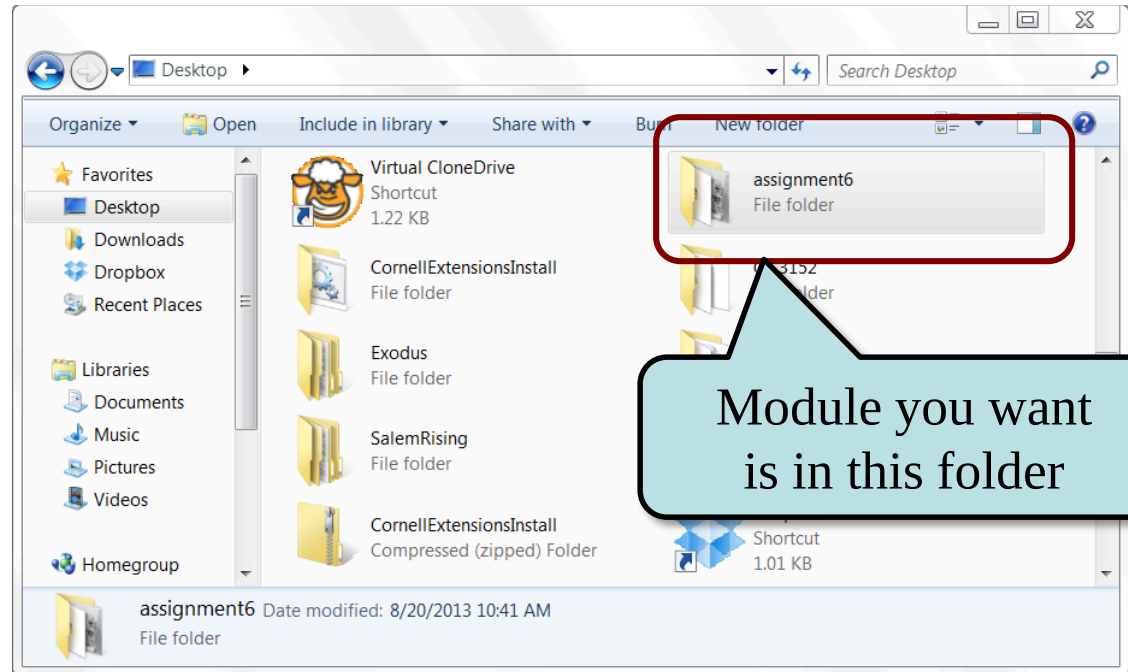
```
>>> cos(pi)
```

```
-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, tkturtle
 - Both have func. Turtle()

Modules Must be in Working Directory!



Modules Must be in Working Directory!

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

Module you want is in this folder

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.y



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.y



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

Scripts and Print Statements

module.py

```
# module.py
```

```
""" This is a simple module.  
It shows how modules work"""
```

```
x = 1+2
```

```
x = 3*x
```

```
x
```

script.py

```
# script.py
```

```
""" This is a simple script.  
It shows why we use print"""
```

```
x = 1+2
```

```
x = 3*x
```

```
print x
```


Scripts and Print Statements

module.py

```
# module.py
```

```
""" This is a simple module.  
It shows how modules work"""
```

```
x = 1+2
```

```
x = 3*x
```

```
x
```

script.py

```
# script.py
```

```
""" This is a simple script.  
It shows why we use print"""
```

```
x = 1+2
```

```
x = 3*x
```

```
print x
```



Only difference

Scripts and Print Statements

module.py

```
modules — -bash — 62x24
[wmwhite@Ryleh]:modules > python module.py
[wmwhite@Ryleh]:modules > █
```

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

script.py

```
modules — -bash — 62x24
[wmwhite@Ryleh]:modules > python script.py
9
[wmwhite@Ryleh]: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 — -bash — 62x24
[wmwhite@Ryleh]:modules > python module.py
[wmwhite@Ryleh]:modules > █
```

```
modules — -bash — 62x24
[wmwhite@Ryleh]:modules > python script.py
9
[wmwhite@Ryleh]:modules > █
```

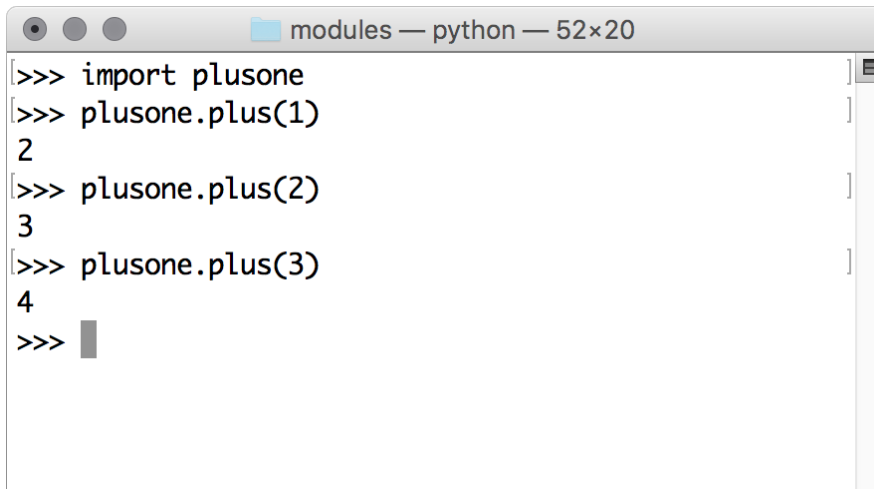
When you run a script,
only statements are executed

- 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)

Next Time: Defining Functions

Function Call

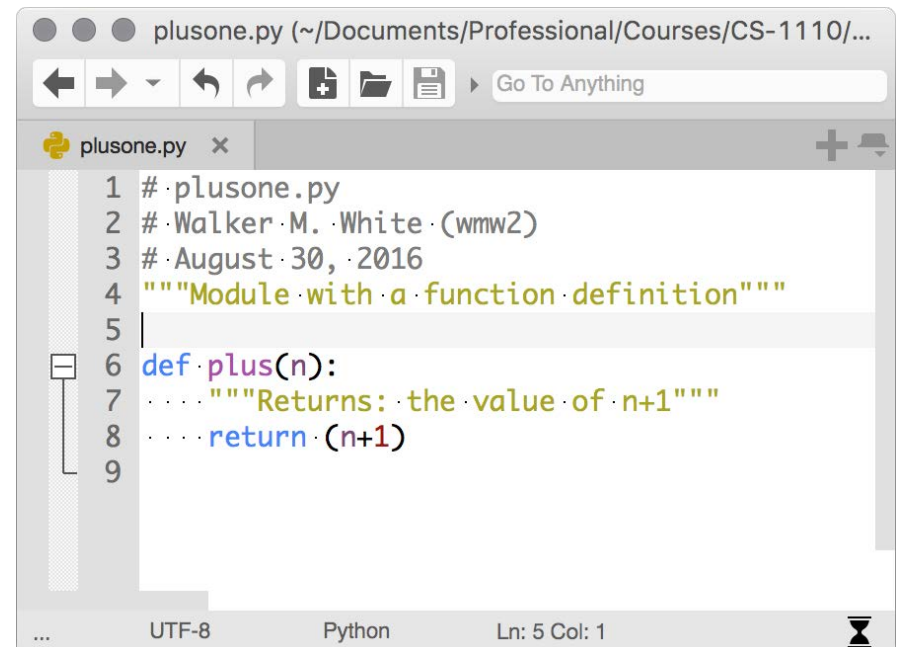
- Command to **do** the function
- Can put it anywhere
 - In the Python shell
 - Inside another module



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

Function Definition

- **define** the function
- Belongs inside a module

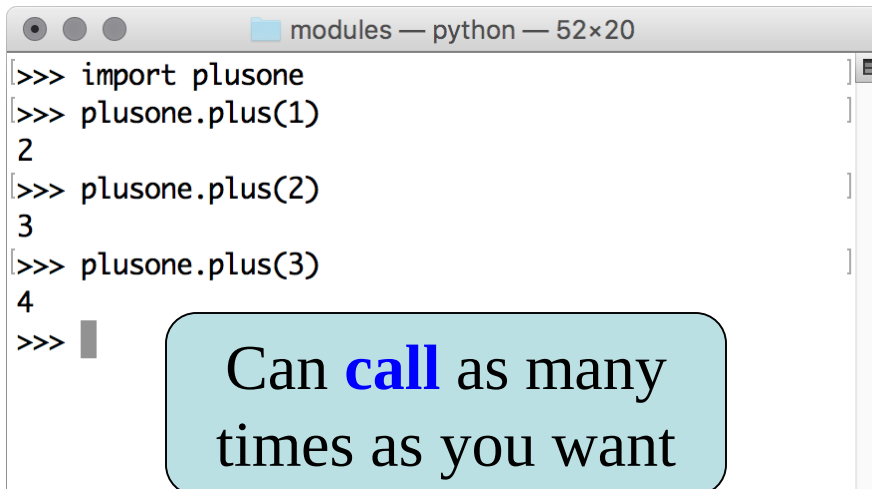


```
1 # plusone.py
2 # Walker M. White (wmw2)
3 # August 30, 2016
4 """Module with a function definition"""
5
6 def plus(n):
7     """Returns the value of n+1"""
8     return (n+1)
9
```

Next Time: Defining Functions

Function Call

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

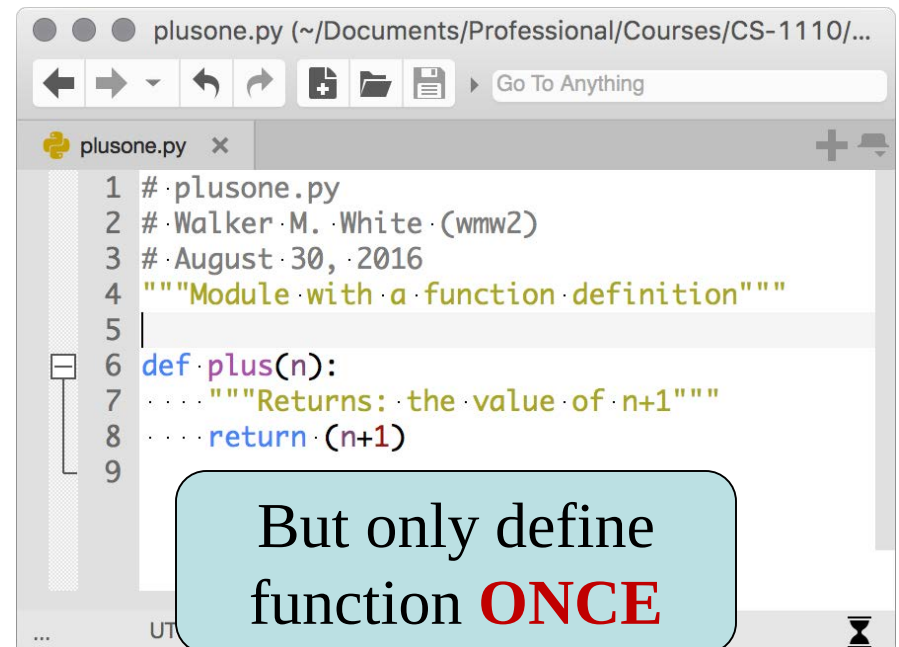


```
>>> import plusone
>>> plusone.plus(1)
2
>>> plusone.plus(2)
3
>>> plusone.plus(3)
4
>>> 
```

Can **call** as many times as you want

Function Definition

- Define the function
- Belongs inside a module



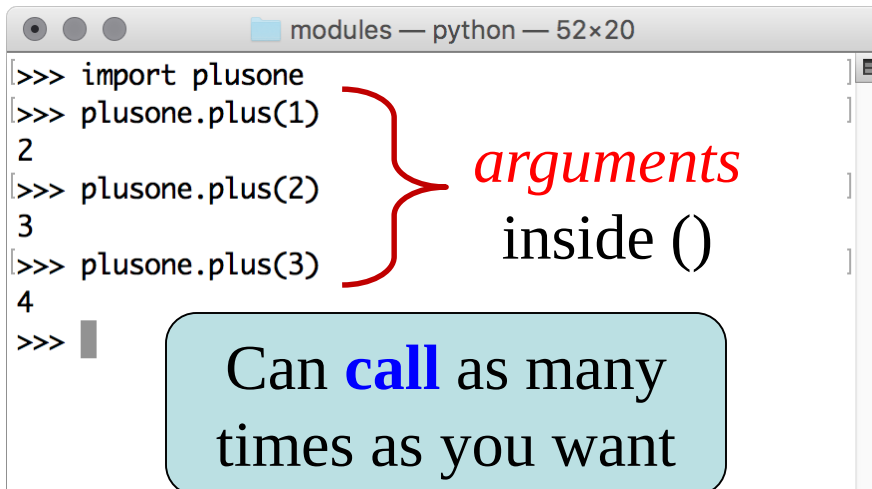
```
1 # plusone.py
2 # Walker M. White (wmw2)
3 # August 30, 2016
4 """Module with a function definition"""
5
6 def plus(n):
7     """Returns the value of n+1"""
8     return (n+1)
9 
```

But only define function **ONCE**

Next Time: Defining Functions

Function Call

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



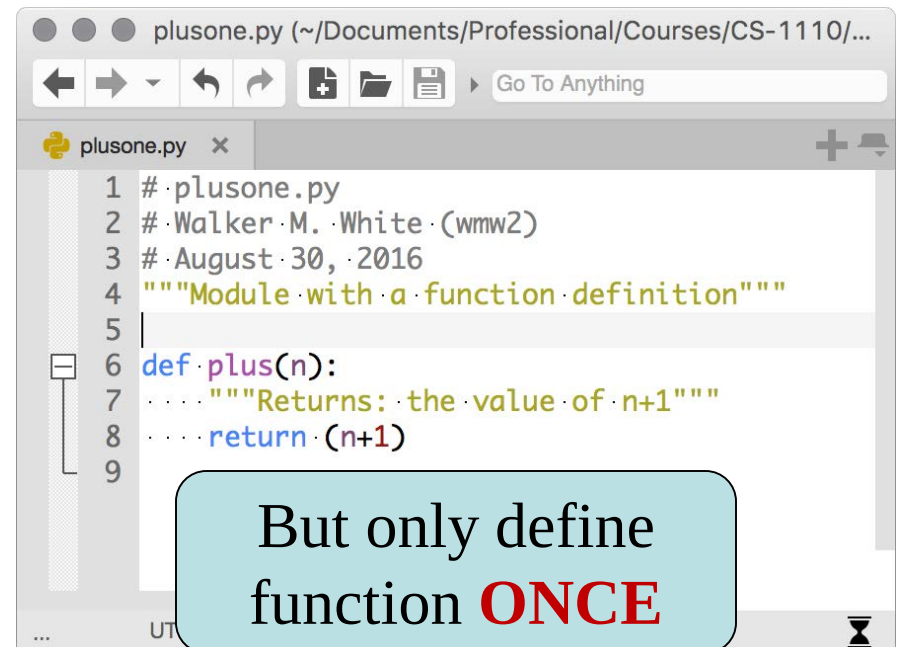
```
>>> import plusone
>>> plusone.plus(1)
2
>>> plusone.plus(2)
3
>>> plusone.plus(3)
4
>>>
```

arguments
inside ()

Can **call** as many times as you want

Function Definition

- **define** the function
- Belongs inside a module



```
1 # plusone.py
2 # Walker M. White (wmw2)
3 # August 30, 2016
4 """Module with a function definition"""
5
6 def plus(n):
7     """Returns the value of n+1"""
8     return (n+1)
9
```

But only define function **ONCE**

Functions and Modules

- Purpose of modules is **function definitions**
 - Function definitions are written in module file
 - Import the module to call the functions
- Your Python workflow (right now) is
 1. Write a function in a module (a .py file)
 2. Open up the Terminal/Command Prompt
 3. Move to the directory with this file
 4. Start Python (type python)
 5. Import the module
 6. Try out the function