

Lecture 3

# **Functions & Modules**

# Labs this Week

---

- Lab 1 is due at the **beginning** of your lab
  - If it is not yet by then, you cannot get credit
  - Only exception is for students who added late (Those students should talk to me)
- Should spend time *entirely* on Lab 2
  - Getting behind this early is bad
  - We are getting you ready for Assignment 1

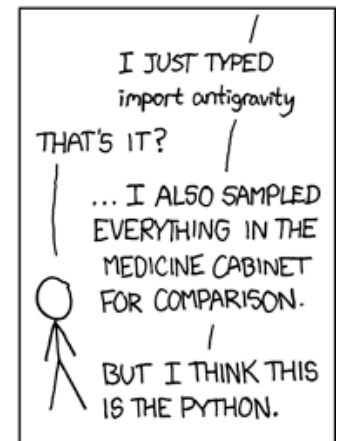
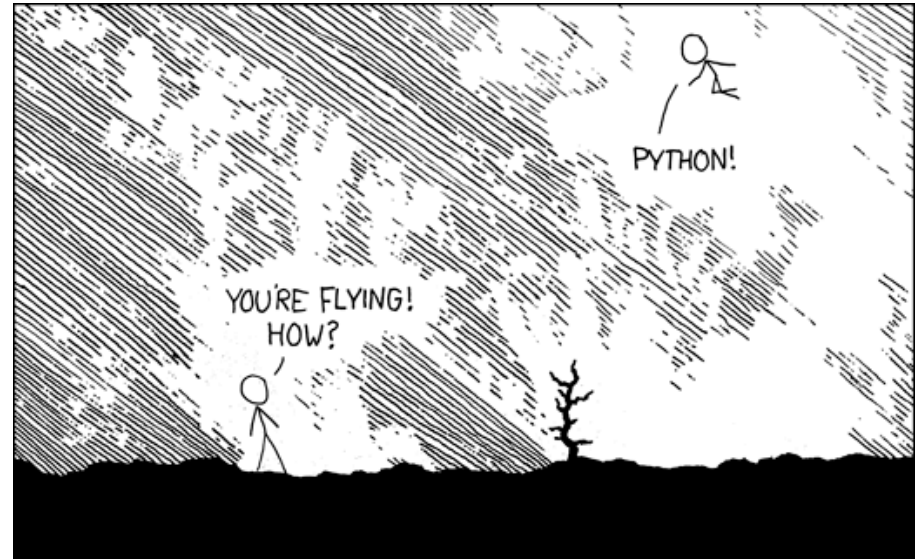
# Readings for Next Few Lectures

## Thursday Reading

- Chapter 3
  - But can skip 3.10
- Browse the Python API
  - Will learn what that is today
  - Do not need to read all of it

## Next Week

- Sections 8.1, 8.2, 8.4, 8.5

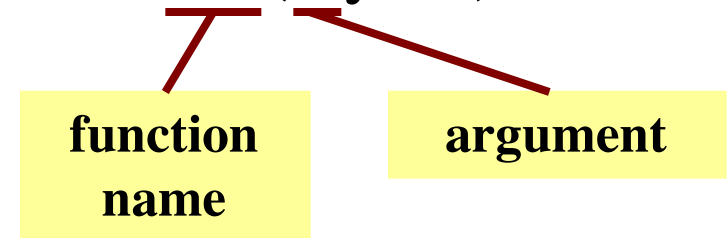


[xkcd.com]

# Function Calls

---

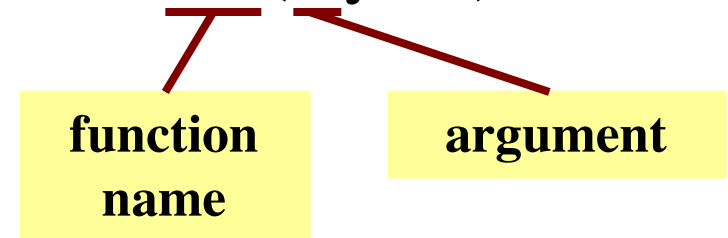
- Python supports expressions with math-like functions
  - A function in an expression is a **function call**
  - Will explain the meaning of this later
- Function expressions have the form **fun(x,y,...)**



- **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
```

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

```
-1.0
```



# 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
```

```
>>> 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)
```

```
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

- `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 window displaying the Python 2.7.3 documentation for the `math` module. The page title is "9.2. math — Mathematical functions — Python v2.7.3 documentation". The URL in the address bar is <http://docs.python.org/library/math.html>. The page content includes a "Table Of Contents" on the left, a main heading "9.2. math — Mathematical functions", and a sub-heading "9.2.1. Number-theoretic and representation functions". The main text describes the `math` module and lists several functions: `math.ceil(x)`, `math.copysign(x, y)`, `math.fabs(x)`, `math.factorial(x)`, and `math.floor(x)`. A search box is visible at the bottom left, and a URL box is at the bottom right.

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

The screenshot shows a web browser window displaying the Python documentation for the `math` module. The page title is "9.2. math — Mathematical functions — Python v2.7.3 documentation". The URL in the address bar is `http://docs.python.org/library/math.html`. The page content includes a "Table Of Contents" on the left, a main heading "9.2. math — Mathematical functions", and a description of the module. A callout box highlights the `math.ceil(x)` function, and another callout box highlights the URL `http://docs.python.org/library`.

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

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

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

<http://docs.python.org/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 "Table Of Contents" on the left, a "9.2. math — Mathematical functions" section header, and a description of the module. The page also lists several functions, including `math.ceil(x)`, `math.factorial(x)`, and `math.floor(x)`. The page is annotated with several callouts:

- A green callout bubble labeled "Function name" points to the `math.ceil(x)` function name.
- A green callout bubble labeled "Possible arguments" points to the `x` argument in `math.ceil(x)`.
- A white callout box labeled "Module" points to the `math` module name in the function signature.
- A green callout bubble labeled "What the function evaluates to" points to the description of the `math.ceil(x)` function: "Return the ceiling of `x` as a float, the smallest integer value greater than or equal to `x`."
- A white callout box labeled "http://docs.python.org/library" points to the URL of the Python documentation page.



# Interactive Shell vs. Modules

```
wmwhite — python — 52x25
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

```
module.py* (~/Documents/Professional/Courses/CS-1110/...)
module.py* 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>
```

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

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

```
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>
```

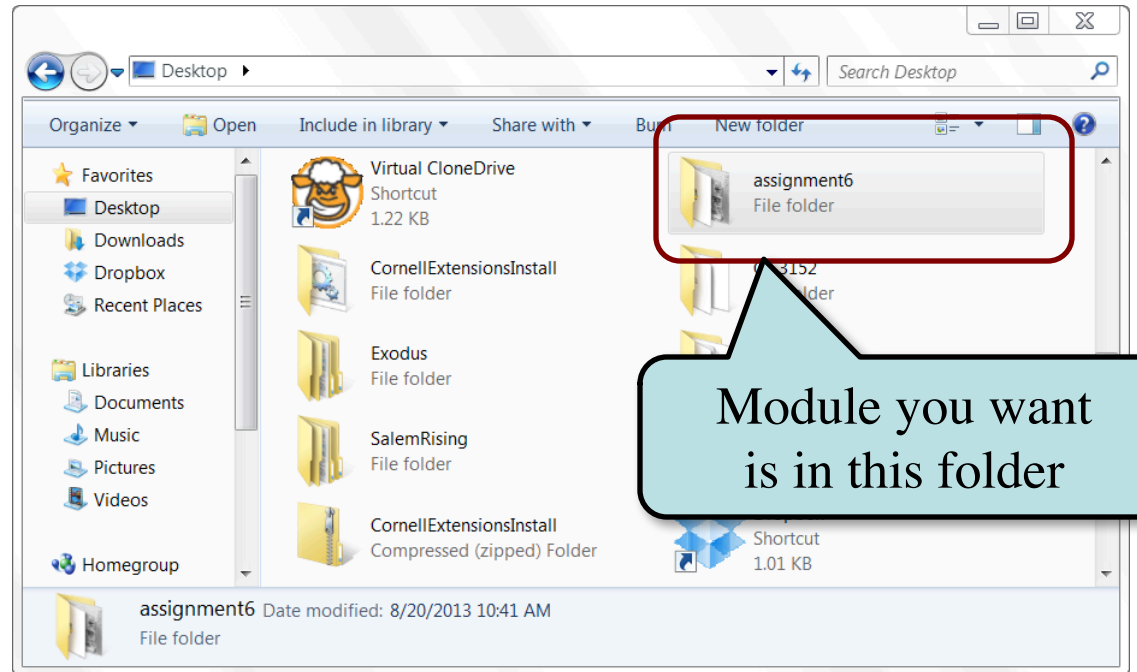
```
NameError: name 'x' is not defined
```

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

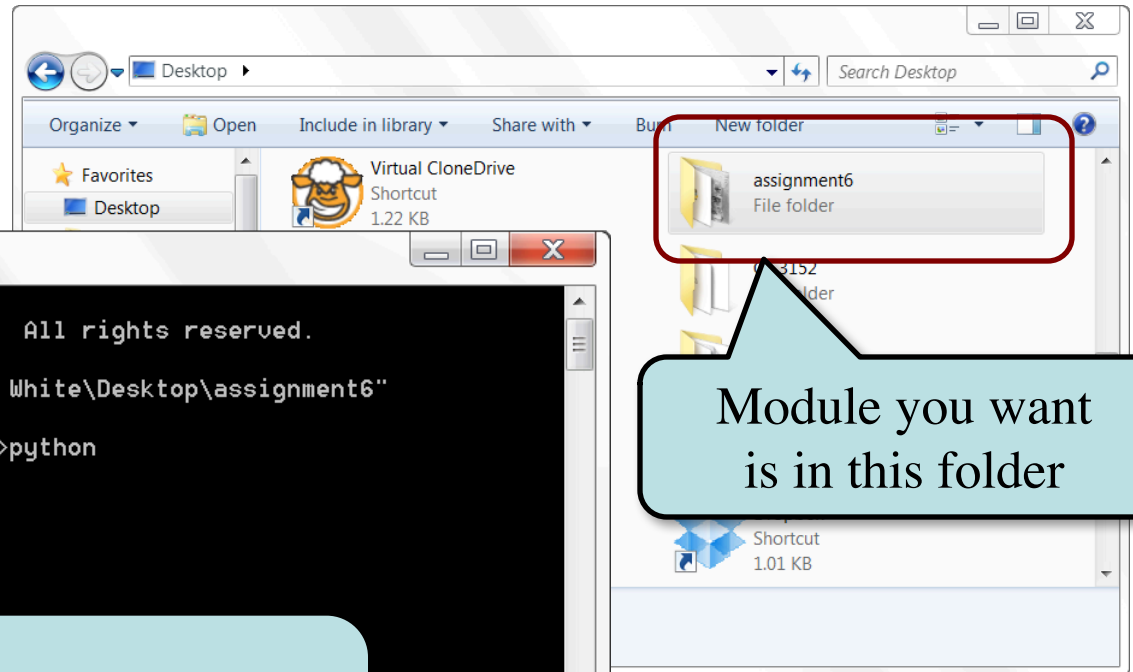
```
9
```

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

# Modules Must be in Working Directory!



# Modules Must be in Working Directory!



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

# 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 vs. Scripts

---

## Module

---

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

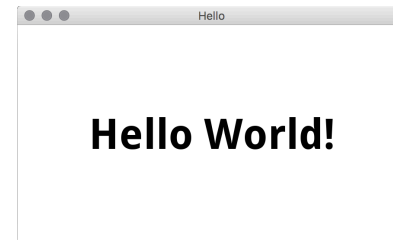
```
>>> import temp
>>> temp.to_fahrenheit(100)
212.0
>>>
```

## Script

---

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

```
python helloApp.py
```



# Modules vs. Scripts

## Module

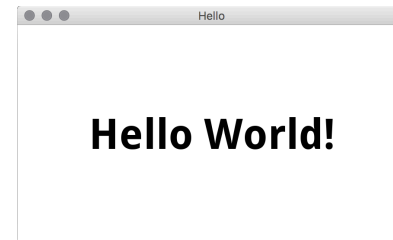
- Provides functions, variables
  - **Example:** temp.py
- import it into Python shell

```
>>> 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

---

```
# 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

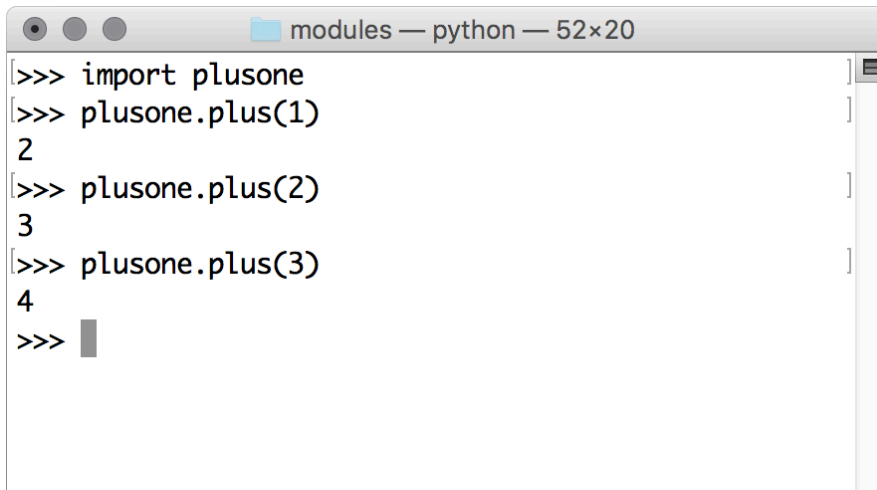
- Looks like it executed the last line
- Python skipped the last line ('x' is not a statement)

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

# Next Time: Defining Functions

## Function Call

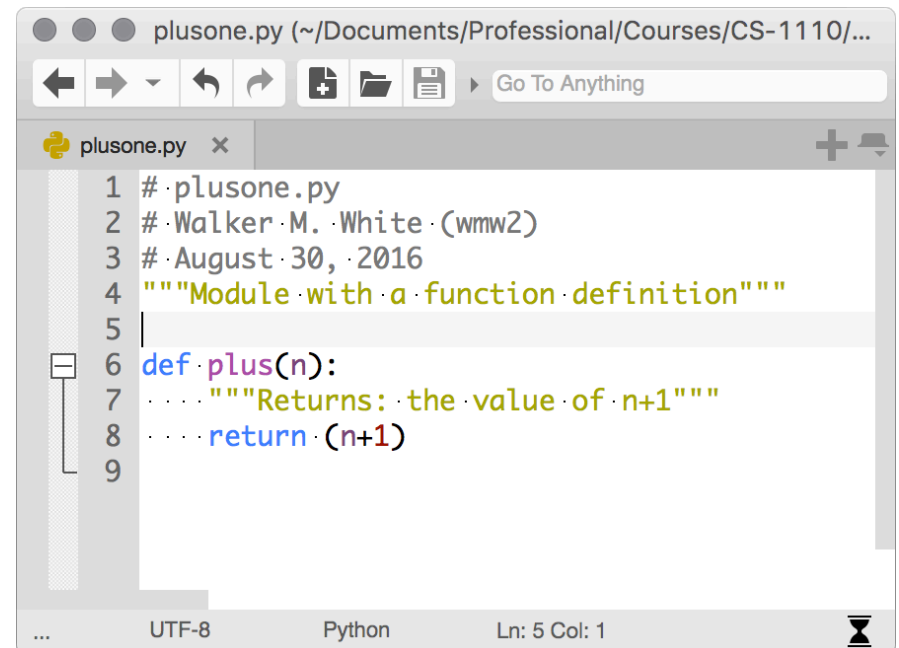
- Command to **do** the function
- Can put it anywhere
  - In the Python shell
  - Inside another module



```
modules — 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

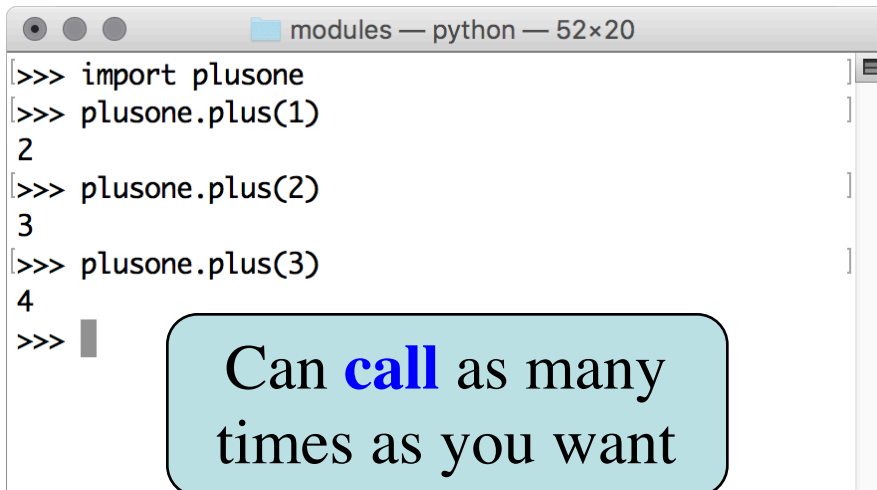


```
plusone.py (~/Documents/Professional/Courses/CS-1110/...
plusone.py x
1 # plusone.py
2 # Walker M. White (wmw2)
3 # August 30, 2016
4 """Module with a function definition"""
5
6 def plus(n):
7     """Returns the value of n+1"""
8     return (n+1)
9
... UTF-8 Python Ln: 5 Col: 1
```

# Next Time: Defining Functions

## Function Call

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

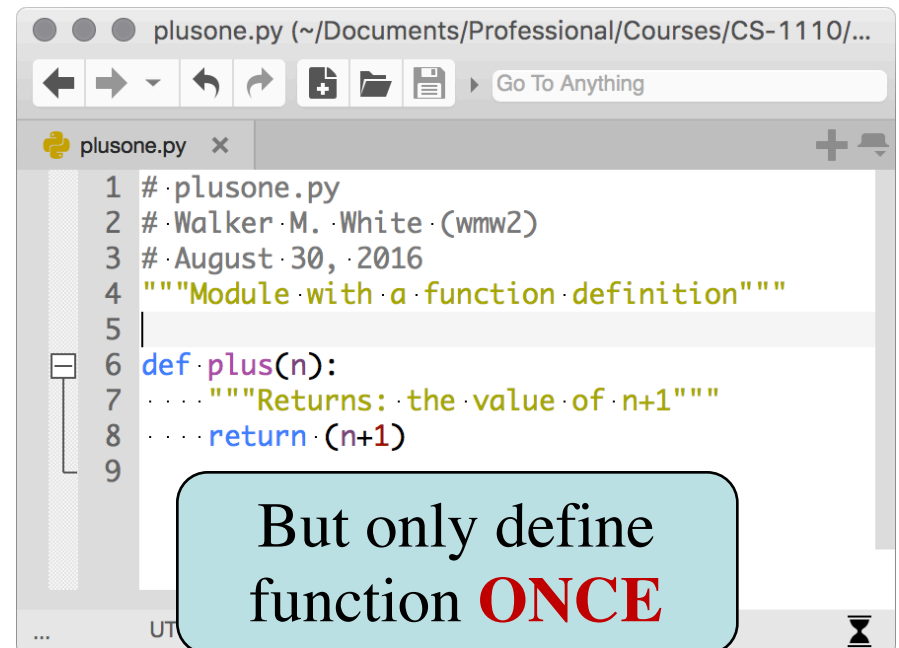


```
>>> import plusone
>>> plusone.plus(1)
2
>>> plusone.plus(2)
3
>>> plusone.plus(3)
4
>>> 
```

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

## Function Definition

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



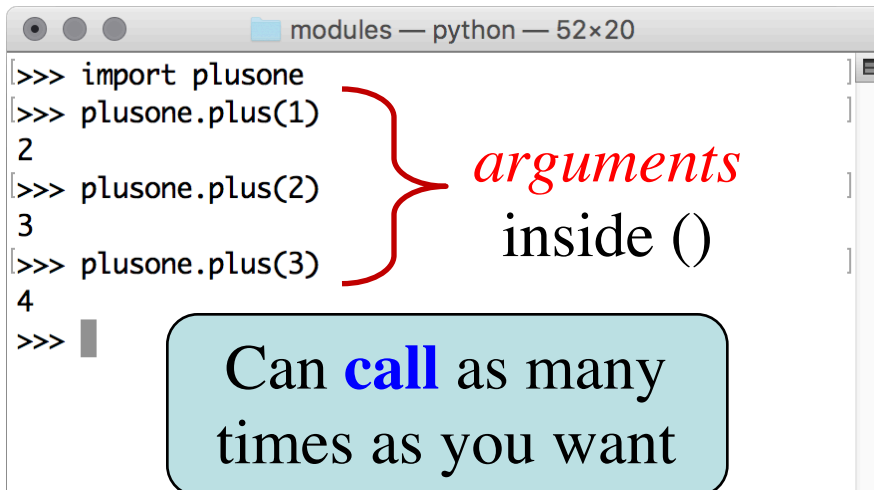
```
1 # plusone.py
2 # Walker M. White (wmw2)
3 # August 30, 2016
4 """Module with a function definition"""
5
6 def plus(n):
7     """Returns the value of n+1"""
8     return (n+1)
9
```

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



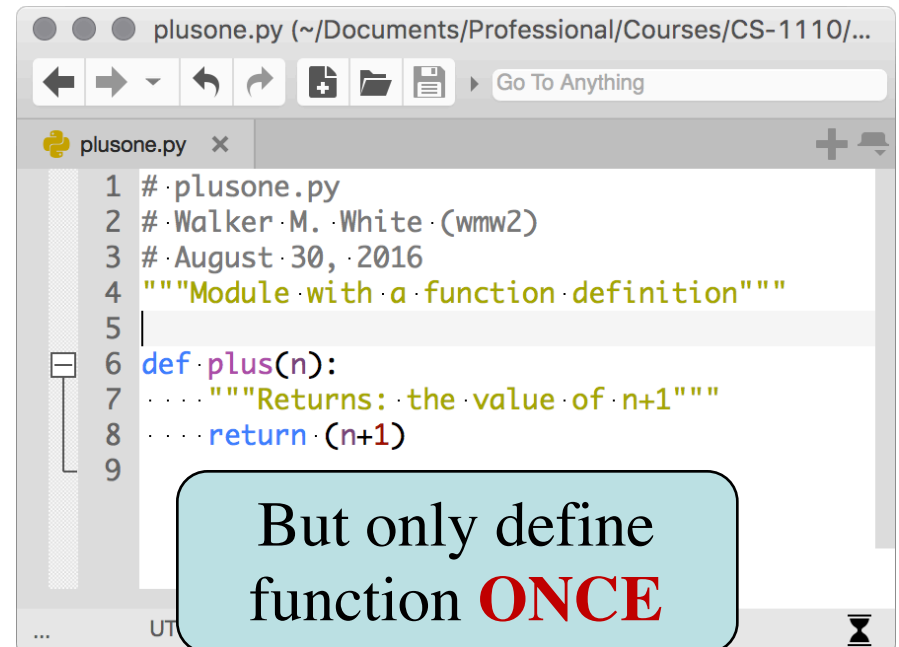
```
>>> import plusone
>>> plusone.plus(1)
2
>>> plusone.plus(2)
3
>>> plusone.plus(3)
4
>>>
```

*arguments*  
inside ()

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

## Function Definition

- Command to **do** 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