

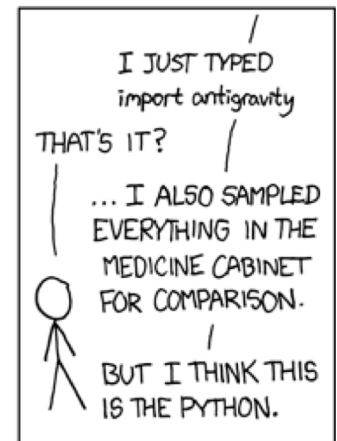
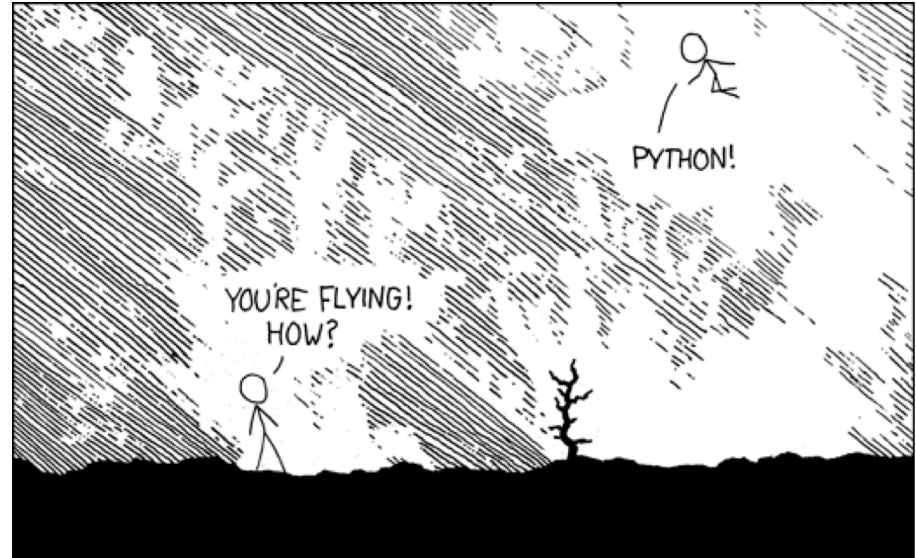
Lecture 3

Functions & Modules

(Optional) Readings

Reading for Next Week

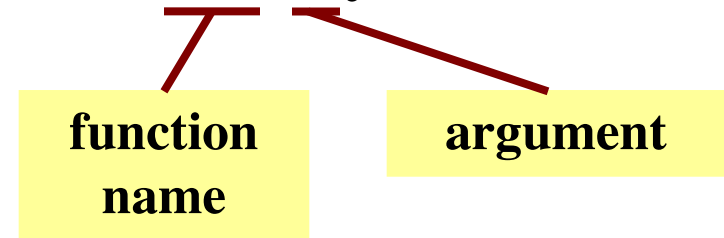
- Chapter 3 in the text
 - But can skip section 3.9
- Browse the Python API
 - Will learn what that is today
 - Do not need to read all of it
- Sections 8.1, 8.2, 8.5, 8.8
 - Strings are needed for A1
 - But Chap 8 mixes easy stuff with advanced stuff



[xkcd.com]

Function Calls

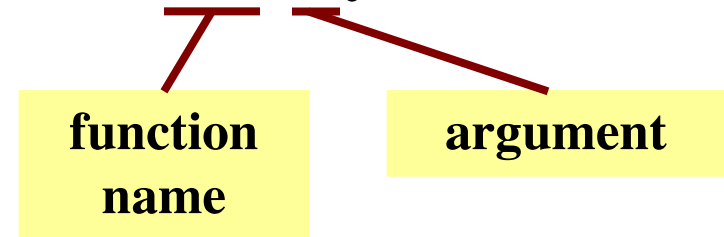
- Python supports expressions with math-like functions
 - A function in an expression is a **function call**
 - Will explain the meaning of this later
- Function expressions have the form **fun(x,y,...)**



- **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()`
- One of the most important functions is `print()`
 - `print(exp)` displays value of `exp` on screen
 - Will see later why this is important

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/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)
```

```
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
```

```
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

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

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()

Reading the Python Documentation

The screenshot shows a web browser displaying the Python documentation page for the `math` module. The browser's address bar shows the URL `docs.python.org/3/library/math.html`. The page title is "9.2. math — Mathematical functions — Python 3.6.2 documentation". The left sidebar contains a "Table Of Contents" for the `math` module, listing sections from 9.2.1 to 9.2.7. The main content area is titled "9.2. math — Mathematical functions" and contains the following text:

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`, the smallest integer greater than or equal to `x`. If `x` is not a float, delegates to `x._ceil_()`, which should return an `Integral` value.

`math.copysign(x, y)`
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`.

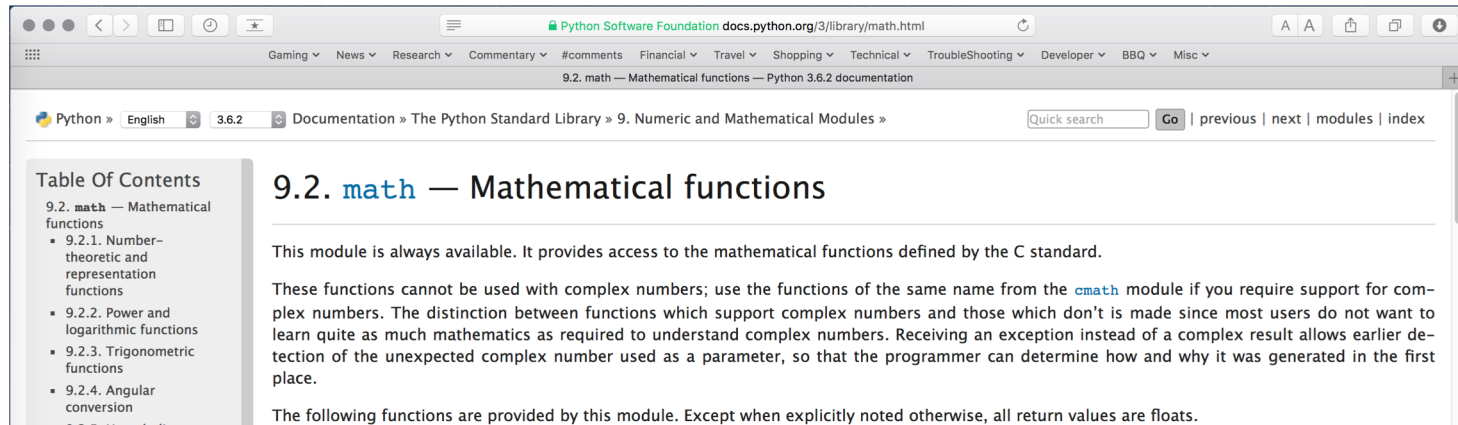
`math.factorial(x)`
Return `x` factorial. Raises `ValueError` if `x` is not integral or is negative.

`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

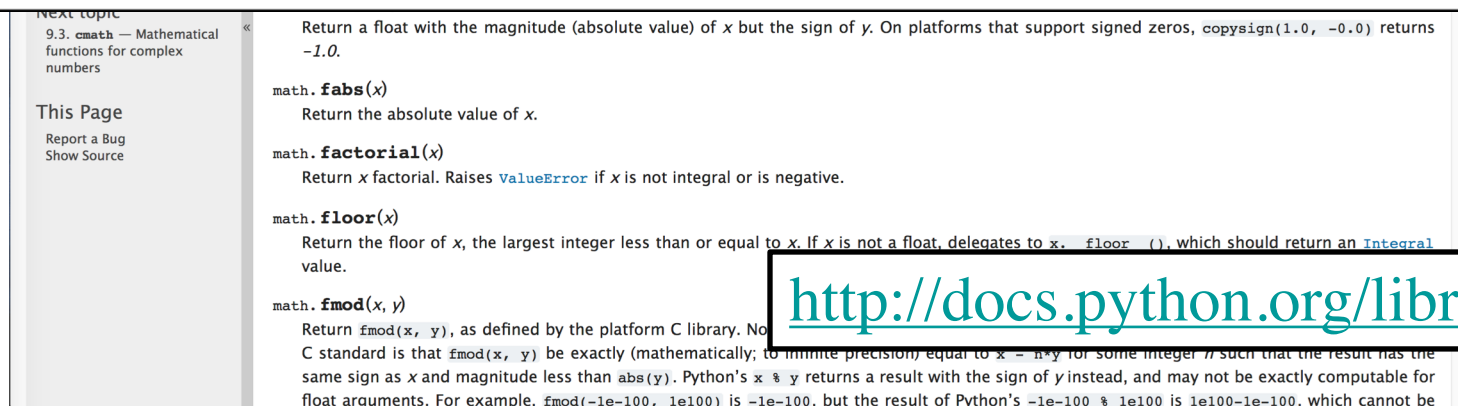
A red box highlights the URL `http://docs.python.org/library` in the bottom right corner of the page.

Reading the Python Documentation



`math.ceil(x)`

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



Reading the Python Documentation

The screenshot shows the Python documentation page for `math.ceil(x)`. The page title is "9.2. math — Mathematical functions". The main heading is `math.ceil(x)`. The description says: "Return the ceiling of `x`, the smallest integer greater than or equal to `x`." The callouts point to various parts of the page:

- Function name**: Points to `math.ceil(x)`.
- Possible arguments**: Points to the description "Return the ceiling of `x`, the smallest integer greater than or equal to `x`."
- Module**: Points to the `math` module name in the function signature.
- What the function evaluates to**: Points to the return value description "Return `x` factorial. Raises `ValueError` if `x` is not integral or is negative."

The URL <http://docs.python.org/library> is highlighted in a box at the bottom right of the screenshot.

Interactive Shell vs. Modules

```
wmwhite — python — 52x25
[wmwhite@dhcp-hol-172]:~ > python
Python 3.6.1 |Anaconda 4.4.0 (x86_64)| (default, May
 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57
)] 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

```
module.py — ~/Documents/Professional/Courses/CS-1110/Lect...
module.py 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 Atom Editor
 - 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.
```

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

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

```
"""
```

Single line comment
(not executed)

```
# This is a comment
```

```
x = 1+2
```

Commands
Executed on import

```
x = 3*x
```

```
x
```


Using a Module

Module Contents

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

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

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

```
"""
```

Single line comment
(not executed)

```
# This is a comment
```

```
x = 1+2
```

Commands
Executed on import

```
x = 3*x
```

```
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>
```

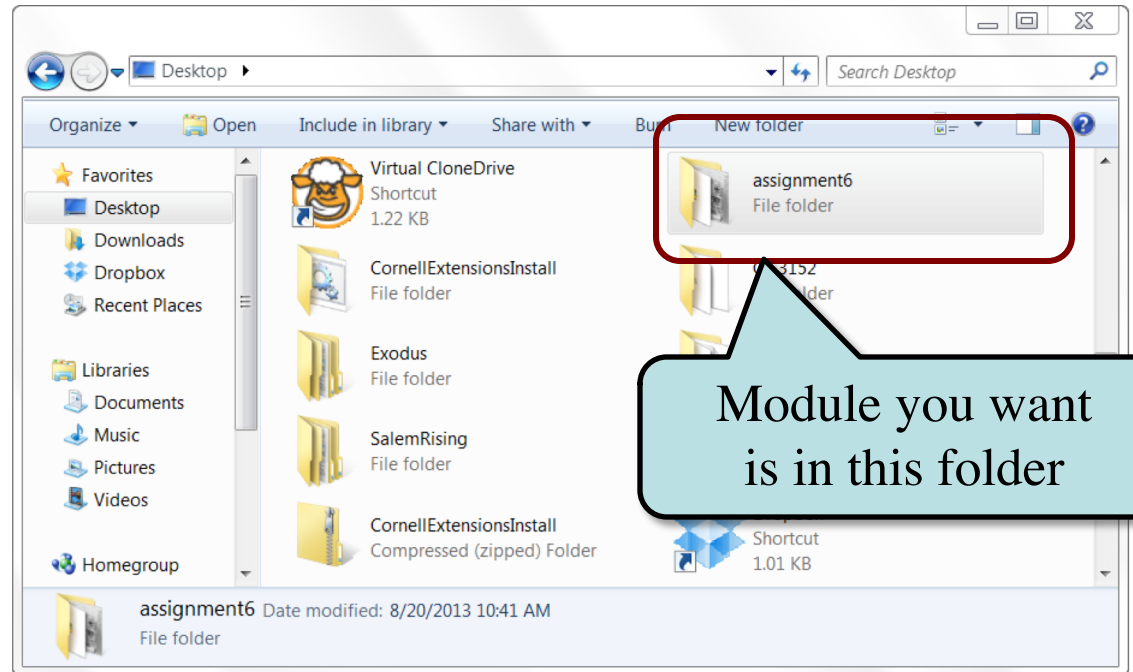
```
NameError: name 'x' is not defined
```

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

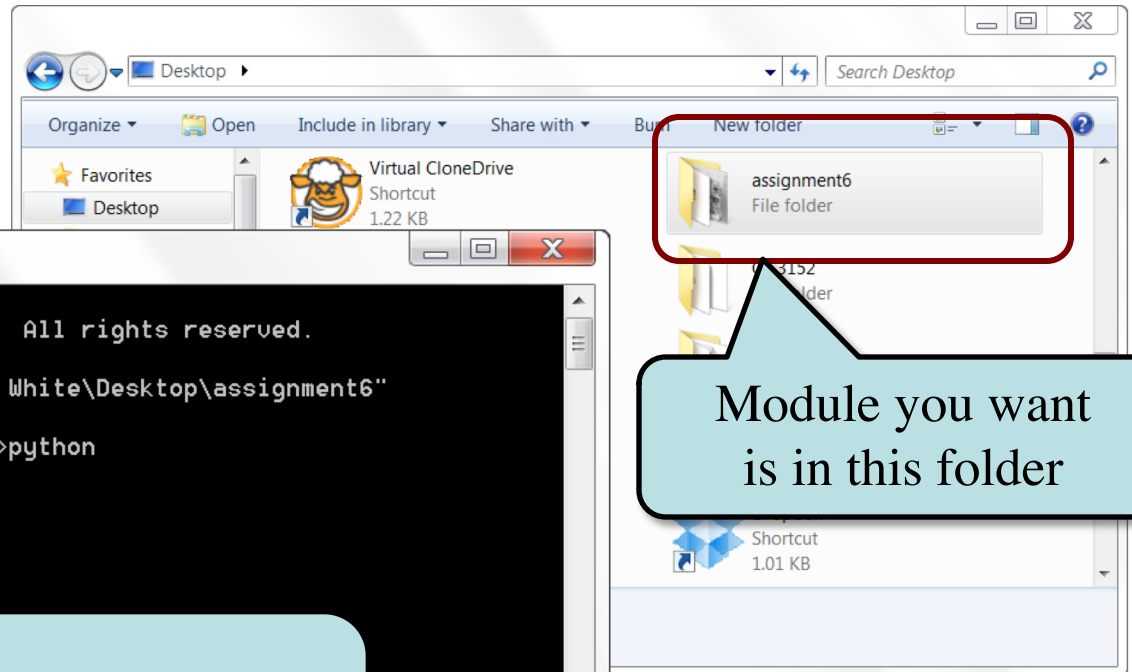
```
9
```

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

Modules Must be in Working Directory!



Modules Must be in Working Directory!



Command Prompt

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Walker White>cd "C:\Users\Walker White\Desktop\assignment6"
C:\Users\Walker White\Desktop\assignment6>python
```

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 — -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 a script
- Python executed the following:
 - Executed the assignments
 - Skipped the last line ('x' is not a statement)

- Python executed the following:
 - 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'
```

```
8/30/18
```

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
```

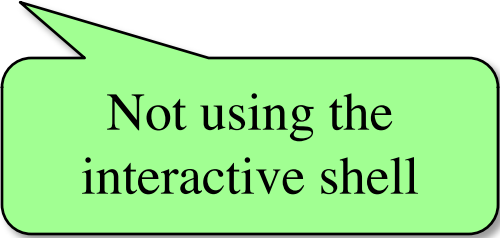
```
You said: Hello
```

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

Function Definition

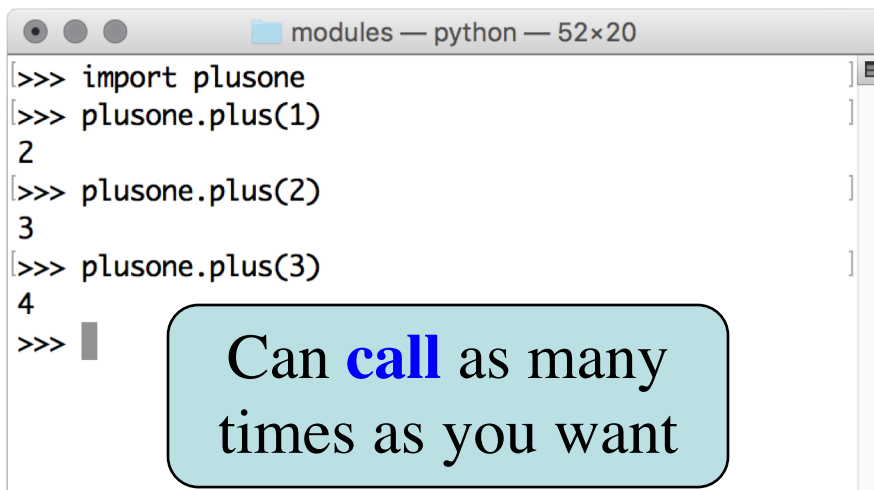
- Command to **do** the function
- Belongs inside a module

```
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

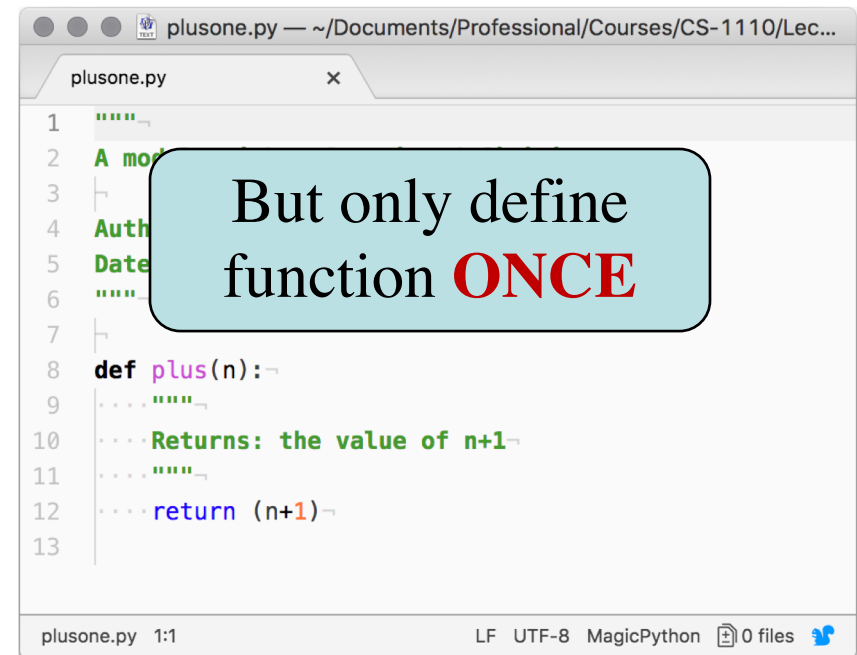


```
>>> import plusone
>>> plusone.plus(1)
2
>>> plusone.plus(2)
3
>>> plusone.plus(3)
4
>>> █
```

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

Function Definition

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



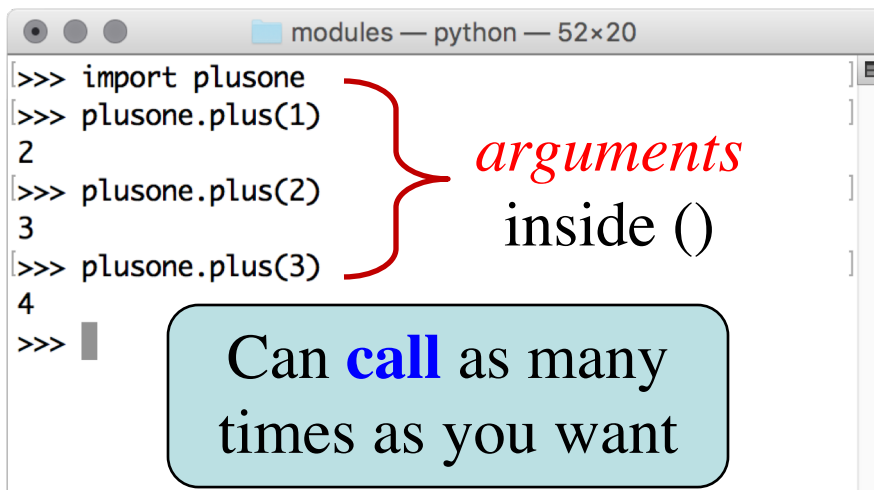
```
1 """
2 A mod
3
4 Auth
5 Date
6 """
7
8 def plus(n):
9     """
10    Returns: the value of n+1
11    """
12    return (n+1)
13
```

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



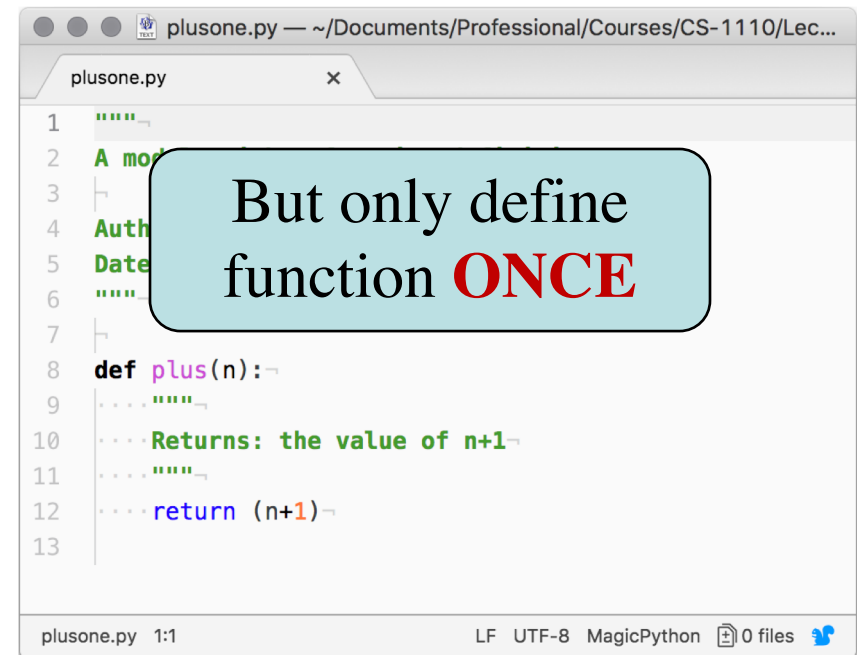
```
>>> import plusone
>>> plusone.plus(1)
2
>>> plusone.plus(2)
3
>>> plusone.plus(3)
4
>>>
```

arguments
inside ()

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

Function Definition

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



```
1 """
2 A mod
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
```

But only define function **ONCE**