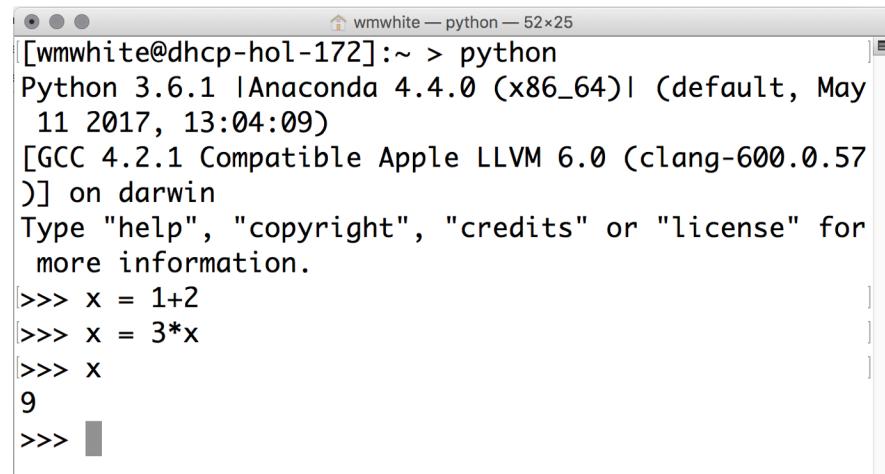


# Not-So-Mini-Lecture 6

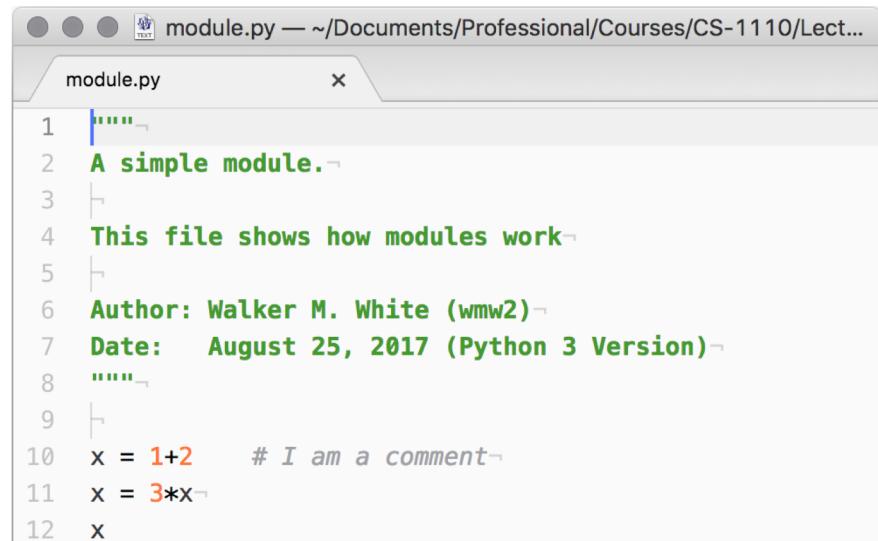
## Modules & Scripts

# Interactive Shell vs. Modules



A screenshot of a terminal window titled "wmwhite — python — 52x25". The window shows the following text:

```
[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
>>>
```



A screenshot of a code editor window titled "module.py". The file contains the following Python code:

```
"""
A simple module.

This file shows how modules work

Author: Walker M. White (wmw2)
Date: August 25, 2017 (Python 3 Version)

"""
x = 1+2      # I am a comment
x = 3*x
x
```

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

- **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
```

```
"""
```

```
# 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>
```

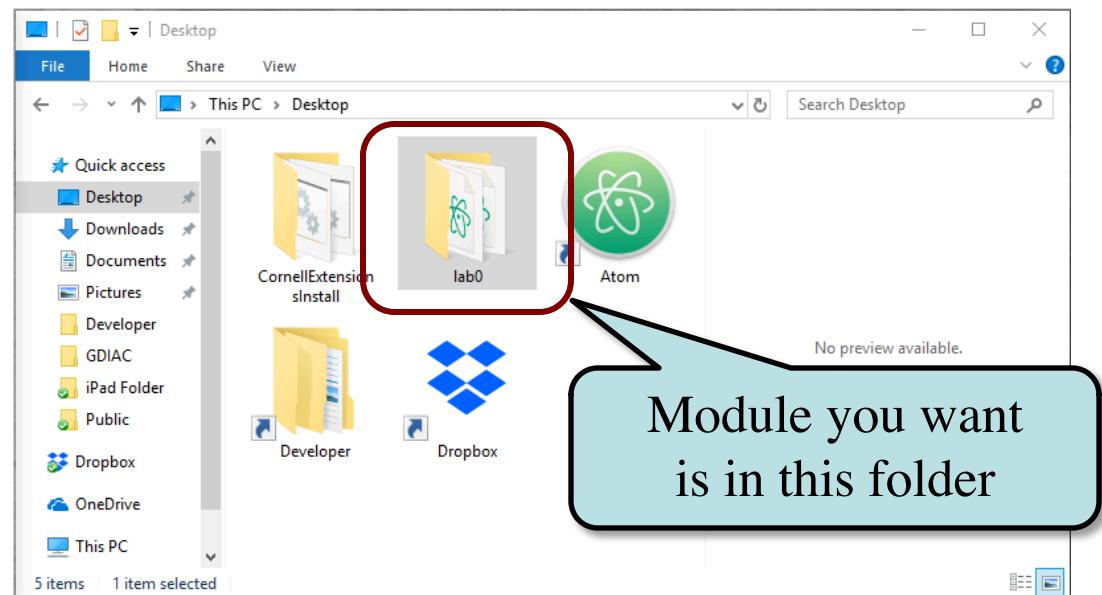
```
NameError: name 'x' is not defined
```

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

```
9
```

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

# Modules Must be in Active Directory!



# Modules Must be in Active Directory!

The screenshot shows a Windows desktop environment. In the center is a File Explorer window titled 'Desktop' with the path 'This PC > Desktop'. It displays several icons: 'CornellExtension sinstall' (yellow folder), 'lab0' (grey folder, circled in red), 'Atom' (green icon), 'Developer' (yellow folder), 'Dropbox' (blue icon), and 'Public' (yellow folder). A callout bubble points to the 'lab0' folder with the text 'Module you want is in this folder'. Below the File Explorer is a Windows PowerShell window with the following text:

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

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

A callout bubble at the bottom left of the slide contains the text '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
```

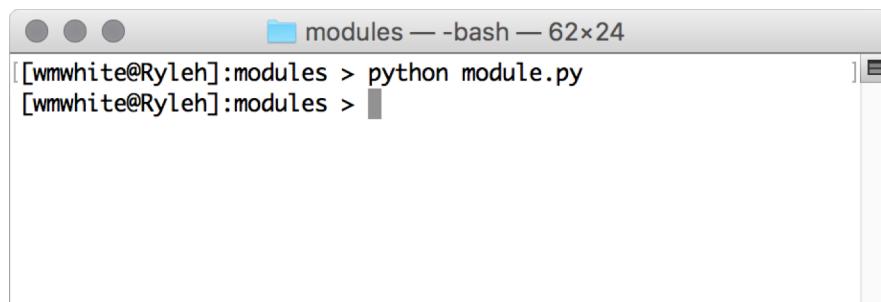
```
x = 3*x
```

print(x)



# Scripts and Print Statements

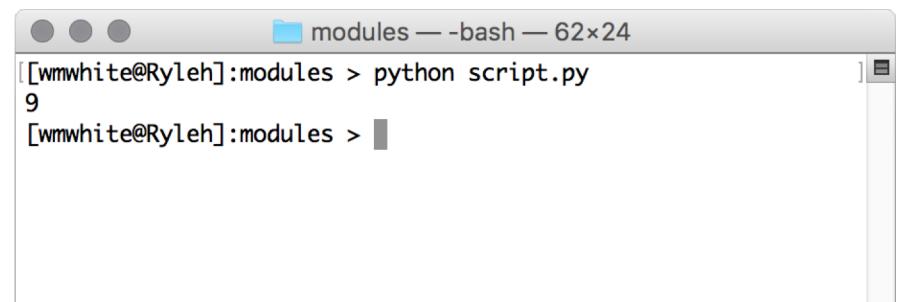
module.py



A screenshot of a terminal window titled "modules — bash — 62x24". The window shows the command "[wmwhite@Ryleh]:modules > python module.py" followed by a blank line, indicating no output.

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

script.py

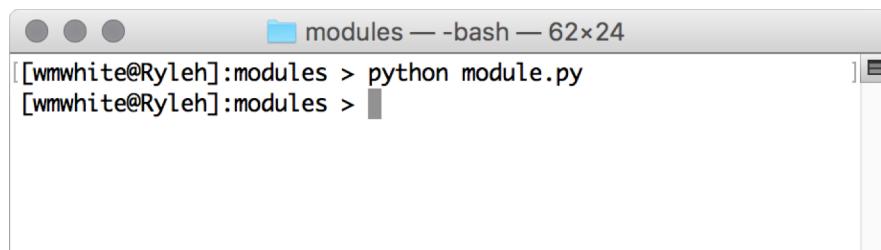


A screenshot of a terminal window titled "modules — bash — 62x24". The window shows the command "[wmwhite@Ryleh]:modules > python script.py" followed by the number "9", which is the value assigned to the variable x in the script.

- We see something this time!
- Python did the following:
  - Executed the **assignments**
  - Executed the last line  
(Prints the contents of x)

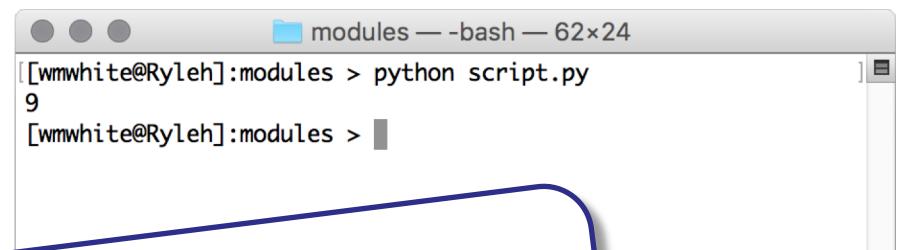
# Scripts and Print Statements

module.py



A screenshot of a terminal window titled "modules — bash — 62x24". The command entered is "python module.py". The output shows two lines of text: "[wmwhite@Ryleh]:modules > python module.py" and "[wmwhite@Ryleh]:modules >".

script.py



A screenshot of a terminal window titled "modules — bash — 62x24". The command entered is "python script.py". The output shows three lines of text: "[wmwhite@Ryleh]:modules > python script.py", "9", and "[wmwhite@Ryleh]:modules >".

- Looks like a module
- Python ignores the last line ('x' is not a statement)
- Executed the assignments
- Skipped the last line

When you run a script,  
only statements are executed

- this time!
- 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'
```

9/7/18

# Making a Script Interactive

---

====

A script showing off input.

This file shows how to make a script interactive.

====

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

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

```
Give me something: Hello
```

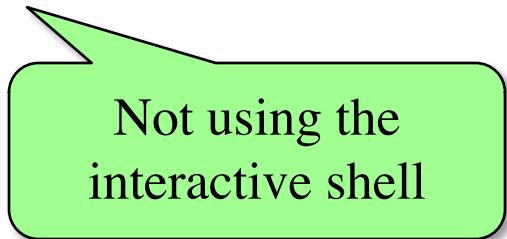
```
You said: Hello
```

```
[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.