Lecture 3:
Functions & Modules
(Sections 3.1-3.3)

CS 1110
Introduction to Computing Using Python

Cornell Bowers CIS
Computer Science

[E. Andersen, A. Bracy, D. Gries, L. Lee, S. Marschner, C. Van Loan, W. White]

iClicker Question #1:

Have you successfully installed Python?

A. Yes. I did it all on my own.
B. Yes. I had some trouble at first, but I figured it out with some help.
C. No. I'm still struggling with my PC.
D. No. I'm still struggling with my Mac.
E. I'll be using the lab computers, not my own.

Common Python Gotcha

```
> 1+1
> Command not found: 1+1
>
```

What in the world?!?

Take a step back.

Where am I?

If you don't see **>>>**

you are not in python interactive mode!

```
> python
>>> 1 + 1
2
>>>
```

Ahh, much better.

All is right in the world.

Function Calls

- Function calls have the form:

```
best_function_ever(x,y,…)
```

function name

argument(s)

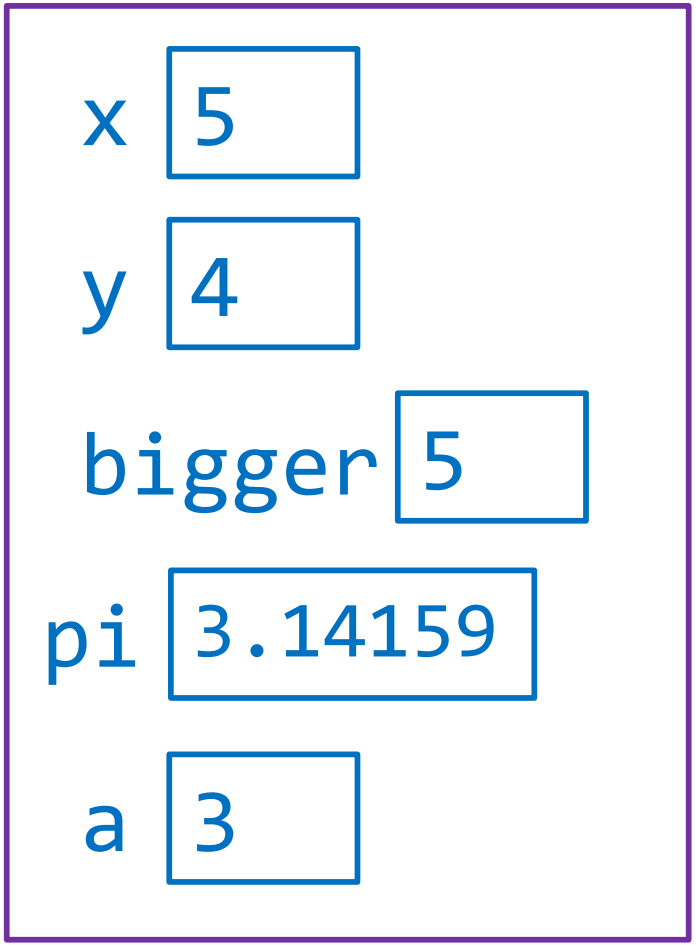
- Arguments
  - Separated by commas
  - Can be any expression

A function might have 0, 1, … or many arguments

Two math functions built into Python

```
>>> x = 5
>>> y = 4
>>> bigger = max(x, y)
>>> bigger
5
>>> pi = 3.14159
>>> a = round(pi)
>>> a
3
```

Visualize the execution!



Play-by-Play of Python interactive mode

```
>>> x = 5
>>> y = 4
>>> bigger = max(x, y)
>>> bigger
5
>>> pi = 3.14159
>>> a = round(pi)
>>> a
3
```

Executes 3 assignment statements

Evaluates the expression and reports

Executes 1 assignment statement

Executes 1 assignment statement

Evaluates the expression and reports

Python interactive mode reports the value to be helpful

Always-available Built-in Functions

- You have seen many functions already
  - Type casting functions: `int()`, `float()`, `bool()`
  - Get type of a value: `type()`
  - Exit function: `exit()`

Empty parens are a human convention to indicate something is a function.

- Longer list:

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

Visualizing functions & variables **(1)**

Running Example:

1. Built-in functions

- Available as soon as you start python
- We don't usually draw them, but they are technically there

```
C:\> python
>>>
```

What Python can access directly

```
int()
float()
str()
type()
print()
…
```

Visualizing functions & variables (2)

Running Example:

1. Built-in functions
2. Define a new variable

```
C:\> python
>>> x = 7
>>>
```

What Python can access directly

```
int()
float()
str()
type()
print()
…
x  7
```

Modules: libraries and scripts

- Many more functions available via built-in ***modules***
  - "Libraries" of functions and variables
- To access a module in Python, use **`import`** command:

  `import` *<module name>*

  Can then access functions like this:

  <module name>.*<function name>*(*<arguments>*)

  Example:

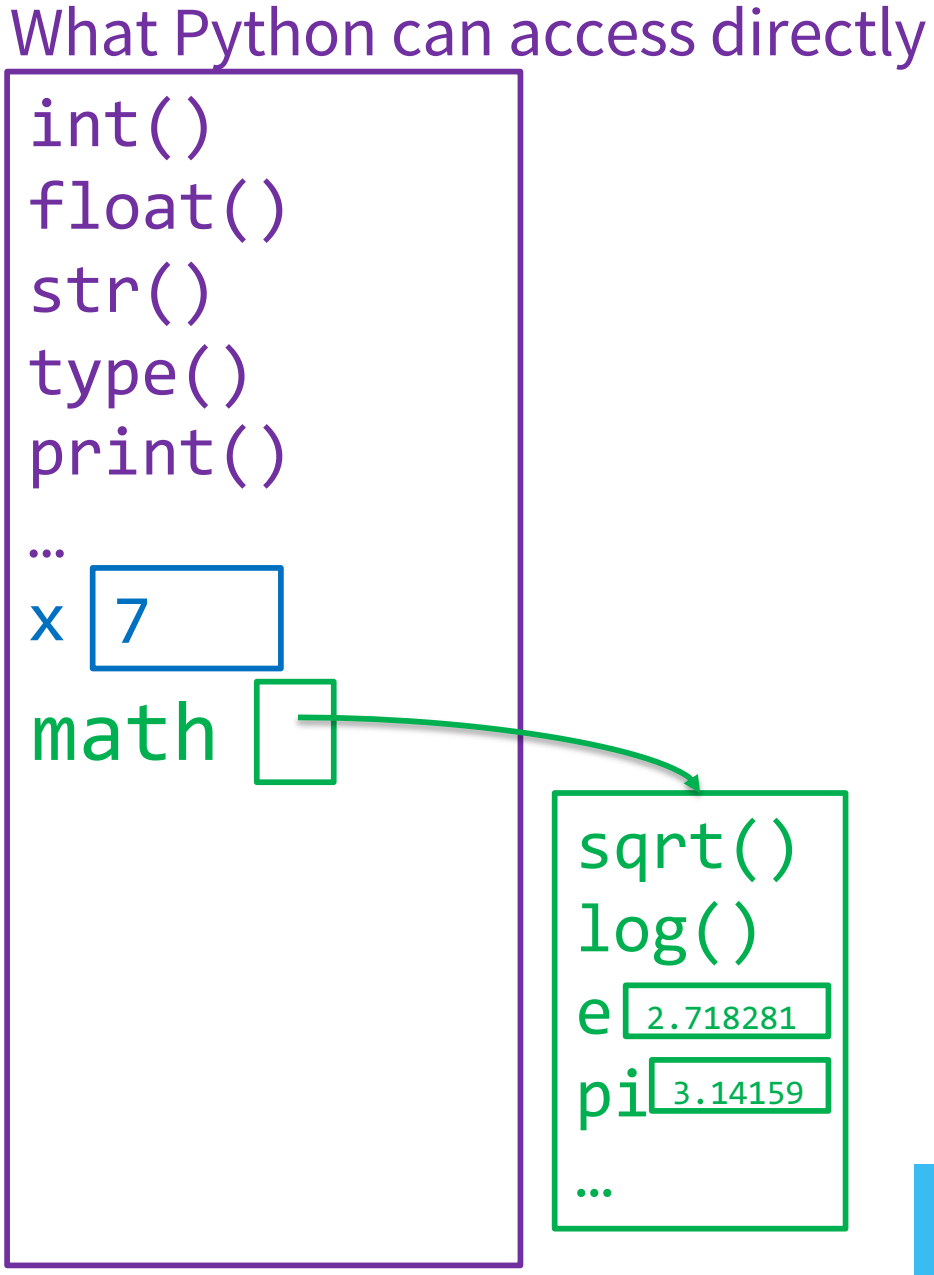
```
>>> import math
>>> p = math.sqrt(9.0)
>>> p
3.0
```

Visualizing functions & variables (3)

Running Example:

1. Built-in functions
2. Define a new variable
3. Import a module

```
C:\> python
>>> x = 7
>>> import math
>>>
```



Module Variables

- Modules can have variables, too
- Can access them like this:

  <module name>.*<variable name>*

- **Example**:

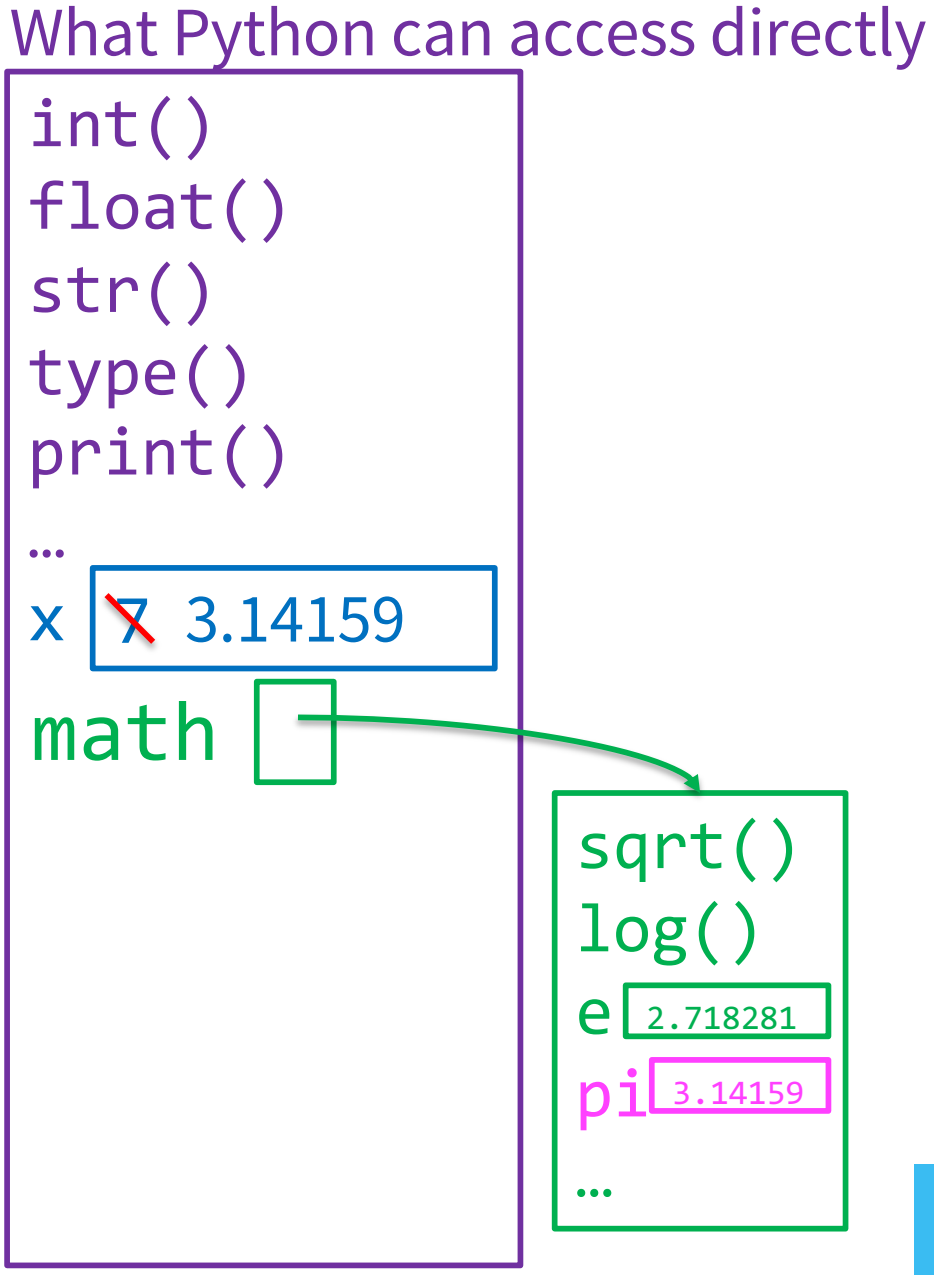
```
>>> import math
>>> math.pi
3.141592653589793
```

Visualizing functions & variables (4)

Running Example:

1. Built-in functions
2. Define a new variable
3. Import a module
4. Use a module variable

```
C:\> python
>>> x = 7
>>> import math
>>> x = math.pi
```



Make your Own Module!

Why?

Python Interactive Mode:

- Good for scratch work!
  - quickly testing something
- **Not** typically how we'll write programs.

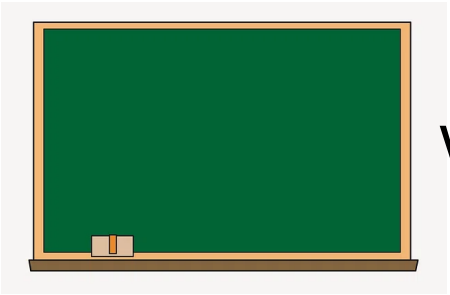
We'll want to write our code in a text file using a text editor.

my_module.py

```
# my_module.py

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

x = 1+2
x = 3*x
```

We recommend Pulsar…

…but any editor will work



VS

Typing in Interactive Mode vs. Writing a Module

Python Interactive Mode

```
wmwhite — python — 52×25
[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
>>>
```

- Type python at command line
- Type commands after **>>>**
  - type line-by-line, again and again
- Python executes as you type

Module

my_module.py

```
# my_module.py

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

x = 1+2
x = 3*x
```

- Written in text editor
  - write once, go back and edit
  - run repeatedly
- Can load with **`import`**
- Python executes statements when **`import`** is called

Section 2.4 in your textbook discusses a few differences

iClicker Question #2:

Have you successfully installed Pulsar?

A. Yes. I did it all on my own.
B. Yes. I had some trouble at first, but I figured it out with some help.
C. No. I'm still struggling.
D. I'll be using a different text editor, not Pulsar.
E. I'll be using the lab computers, not my own.

my_module.py

Module Text File

```
# my_module.py

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

x = 1+2
x = 3*x
```

Single line comment
starts with **#**
(not executed)

Docstring
(note the Triple Quotes)
A multi-line comment.
Useful for *code documentation.*

Commands
Executed on import

Ways of Executing Python Code

1. running the Python Interactive Shell
2. NEW: importing a module

Importing a module from inside Python (1)

Module Text File

my_module.py

```
# my_module.py

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

x = 1+2
x = 3*x
```

Python Interactive Mode

```
C:\> python
>>> import my_module
```

Needs to be the same name as the file ***without the ".py"***

Importing a module from inside Python (2)

Module Text File

my_module.py

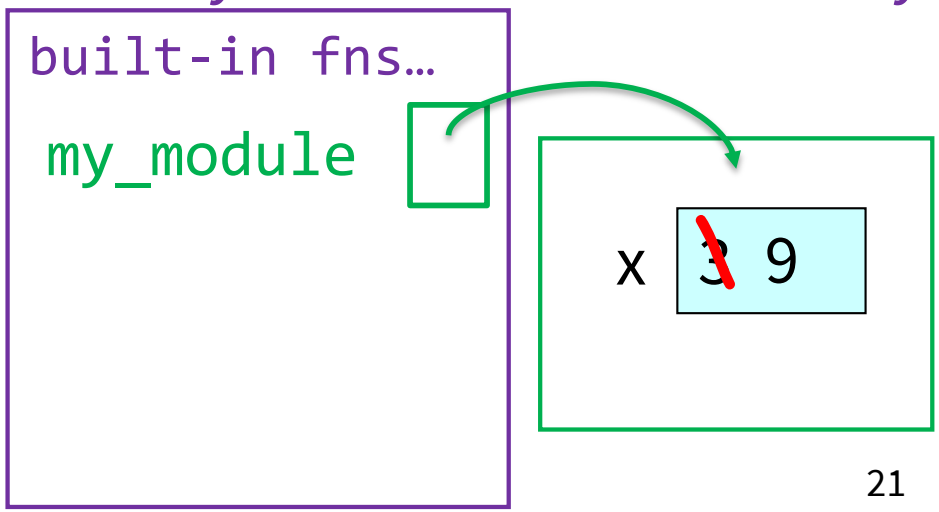
```
# my_module.py

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

x = 1+2
x = 3*x
```

variable x stays "within" the module

Python Interactive Mode

```
C:\> python
>>> import my_module
>>> my_module.x
9
```

What Python can access directly



Clicker Question!

Module Text File

```
# fah2cel.py


"""Convert 32 degrees
Fahrenheit to degrees
Celsius"""


f= 32.0
c= (f-32)*5/9
```

Python Interactive Mode

```
C:\> python
>>> import fah2cel
```

After you hit "Return" here what will python print next?

(A) >>>

(B) 32.0
0.0
>>>

(C) an error message

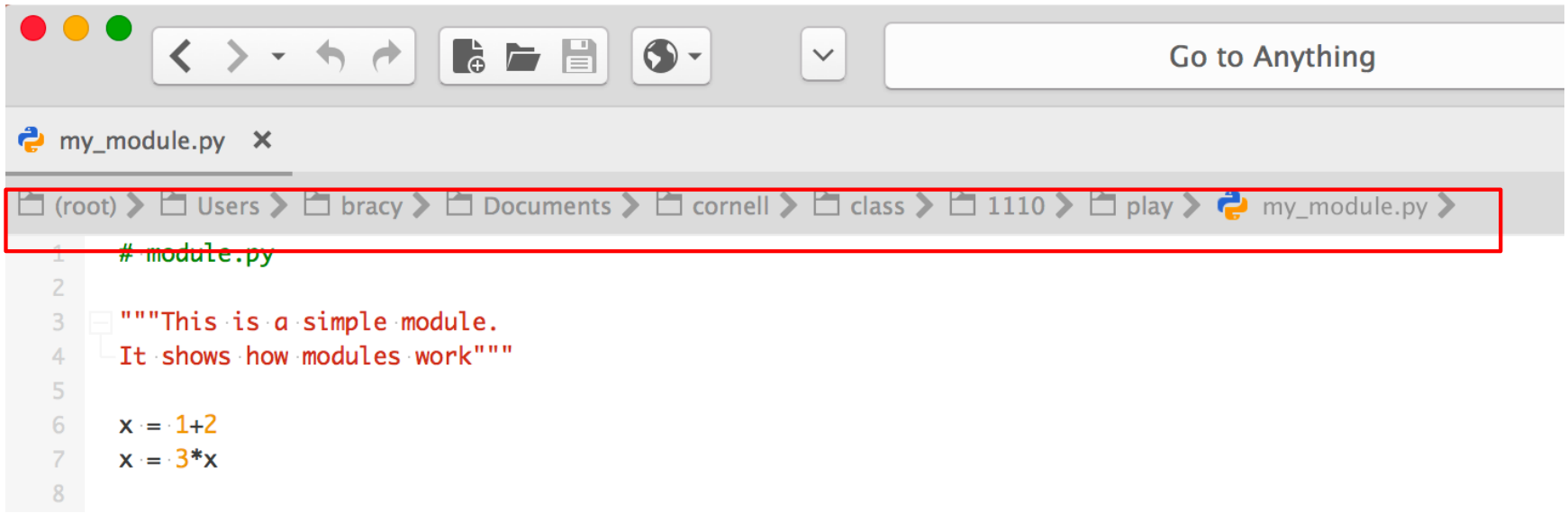
(D) The text of fah2cel.py

(E) Sorry, no clue.

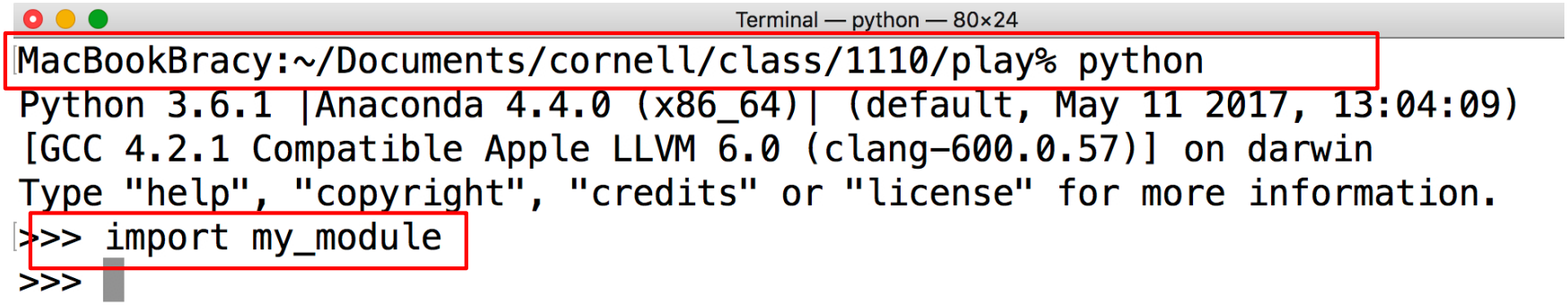
Rule #1: Modules must be in Working Directory*

*the directory where you typed "python"

If **my_module.py** in directory/folder **play**:



Then you must run **python** from the folder **play**:



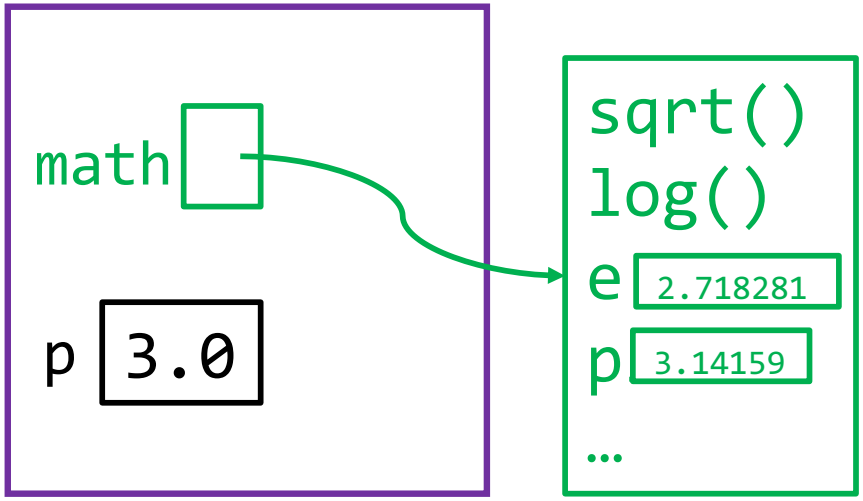
Rule #2: You must **import**

Windows command line
(Mac looks different)

With import

```
C:\> python
>>> import math
>>> p = math.sqrt(9.0)
>>> p
3.0
```

What Python can access directly



Without import

```
C:\> python
>>> math.sqrt(9.0)
Traceback (most recent call
last):
  File "<stdin>", line 1,
in <module>
NameError: name 'math' is
not defined
```

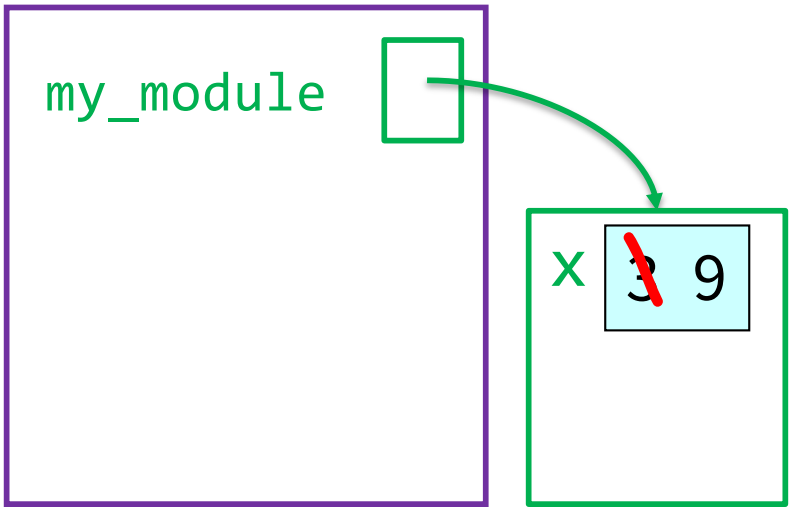
What Python can access directly

Python unaware of what "math" is

Rule #3: You Must Use the Module Name

```
C:\> python
>>> import my_module
>>> my_module.x
9
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

What Python can access directly



Python unaware of what "x" is (it cannot access it directly)

Ways of Executing Python Code

1. running the Python Interactive Shell
2. importing a module
3. NEW: running a script (a different kind of module)

Running a Script

- From the command line, type:

  `python <script filename>`

- Example:

  C:\> python my_module.py

From the command line, use **full** filename, *with* ".py"

Modules: Libraries vs. Scripts

Library

- Provides functions, variables
- **`import`** it into Python shell, don't include ".py"
- Within Python shell you have access to the functions and variables of the imported module

Script

- Behaves like an application
- At command line prompt, Tell python to run the file (use full filename, including ".py")
- After running the app you're back at the command line (not in Python shell)

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

Common Command shell Gotcha

```
>>> python last_task.py
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'python' is not defined
>>>
```

Rule #1 of running a script from the command line is making sure you are in the command line!

If you see **>>>** *you are in* ***python interactive mode****, But you wanted to be* ***outside*** *of Python!*

```
>>> exit()
C:\> python last_task.py
[..some output…]
```

Running a Script

- From the command line, type:

  `python <script filename>`

From the command line, use **full** filename, *with* ".py"

- Example:

  C:\> python my_module.py

  C:\>

  looks like nothing happened!

- Actually, something did happen
  - Python executed all of my_module.py

Clicker Question

Module Text File

```
# my_module.py

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

x = 1+2
x = 3*x
```

Command Line

```
C:\> python my_module.py
C:\> my_module.x
```

After you hit "Return" here what will be printed next?

(A) >>>

(B) 9
>>>

(C) an error message

(D) The text of my_module.py

(E) Sorry, no clue.

Running my_module.py as a script

Module Text File

```
# my_module.py

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

x = 1+2
x = 3*x
```

Command Line

```
C:\> python my_module.py
C:\>
```

What Python can access directly

x ~~3~~ 9

When the script ends:

- All memory used by `my_module.py` is deleted
  - Includes all variables
- There is no evidence that the script ran!

Creating Evidence that the Script Ran

- New (very useful!) command: `print`

```
print (<expression>)
```

- **`print`** evaluates the `<expression>` and writes the value to the console

my_module.py vs. script.py

my_module.py

```
# my_module.py

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

x = 1+2
x = 3*x
```

script.py

```
# script.py

"""A simple script.
Shows why we use print"""

x = 1+2
x = 3*x
print(x)
```

Only difference!

Running script.py as a script

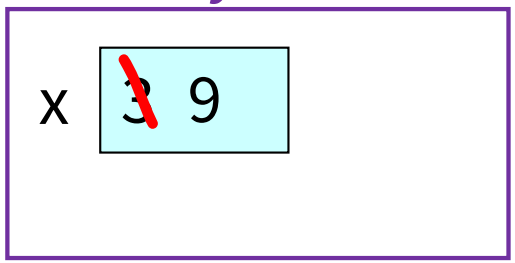
script.py

```
# script.py
"""A simple script.
Shows why we use print"""

x = 1+2
x = 3*x
print(x)
```

Command Line

```
C:\> python script.py
9
C:\>
```

What Python can access directly



When the script ends:

- All memory used by `script.py` is deleted
  - Includes all variables
- But the print statement leaves evidence that it ran

Interactive mode **evaluates & reports**
Script mode only **evaluates**

(both execute assignment statements)

Python Interactive Mode

```
C:\> python
>>> x = 1+2
>>> x = 3*x
>>> x
9
>>>
```

Executes 2 assignment statements

Evaluates the expression and reports

script2.py

```
# script2.py


"""A simple script.
Shows why we use print"""


x = 1+2
x = 3*x
x
```

Executes 2 assignment statements

Evaluates the expression

Command Line

```
C:\> python script2.py
C:\>
```

No output!

Finding your way around a Python module

module help

After importing a module, see what functions and variables are available:

```
>>>  help(<module name>)
```

Terminal — less ◂ python — 80×24

```
Help on module math:

NAME
    math

MODULE REFERENCE
    https://docs.python.org/3.6/library/math

    The following documentation is automatically generated from the Python
    source files.  It may be incomplete, incorrect or include features that
    are considered implementation detail and may vary between Python
    implementations.  When in doubt, consult the module reference at the
    location listed above.

DESCRIPTION
    This module is always available.  It provides access to the
    mathematical functions defined by the C standard.

FUNCTIONS
    acos(...)
        acos(x)

        Return the arc cosine (measured in radians) of x.
```

Reading the Python Documentation

https://docs.python.org/3/library/math.html

docs.python.org/3.7/library/math.html

Python » English 3.7.6 Documentation » The Python Standard Library » Numeric and Mathematical Modules » Quick search

Table of Contents

math — Mathematical functions
- Number-theoretic and representation functions
- Power and logarithmic functions
- Trigonometric functions
- Angular conversion
- Hyperbolic functions
- Special functions
- Constants

Previous topic

numbers — Numeric abstract base classes

Next topic

cmath — Mathematical functions for complex numbers

This Page

Report a Bug
Show Source

math — Mathematical functions

This module 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.

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*)

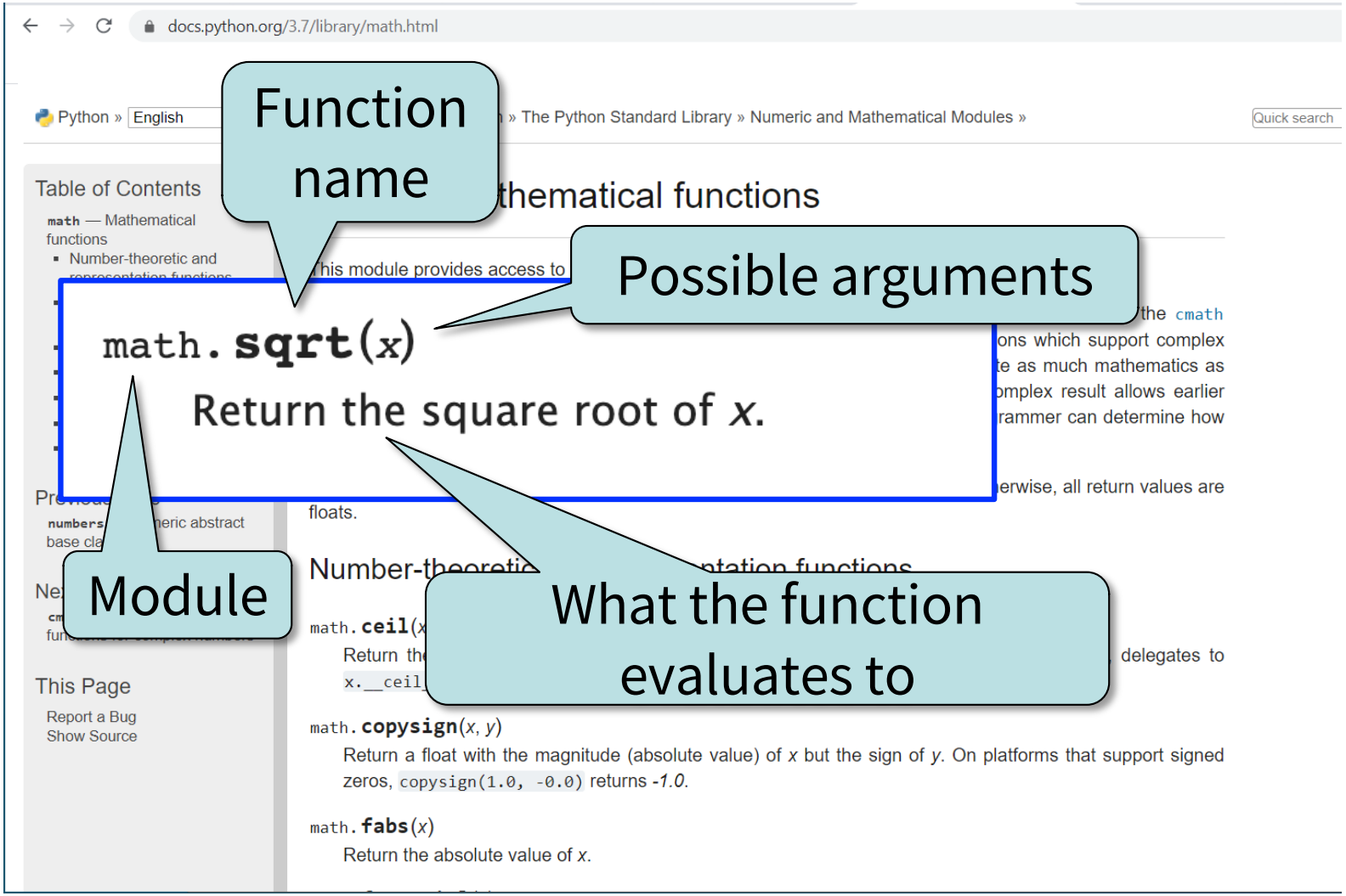
Return a float with the magnitude (absolute value) of *x* but the sign of *y*. On platforms that support signed zeros, `copysign(1.0, -0.0)` returns *-1.0*.

`math.`**`fabs`**(*x*)

Return the absolute value of *x*.

A Closer Reading of the Documentation

https://docs.python.org/3.7/library/math.html



What does the docstring do?

Module Text File

```
# my_module.py

"""This is a simple modul
It shows how modules work

x = 1+2
x = 3*x
```

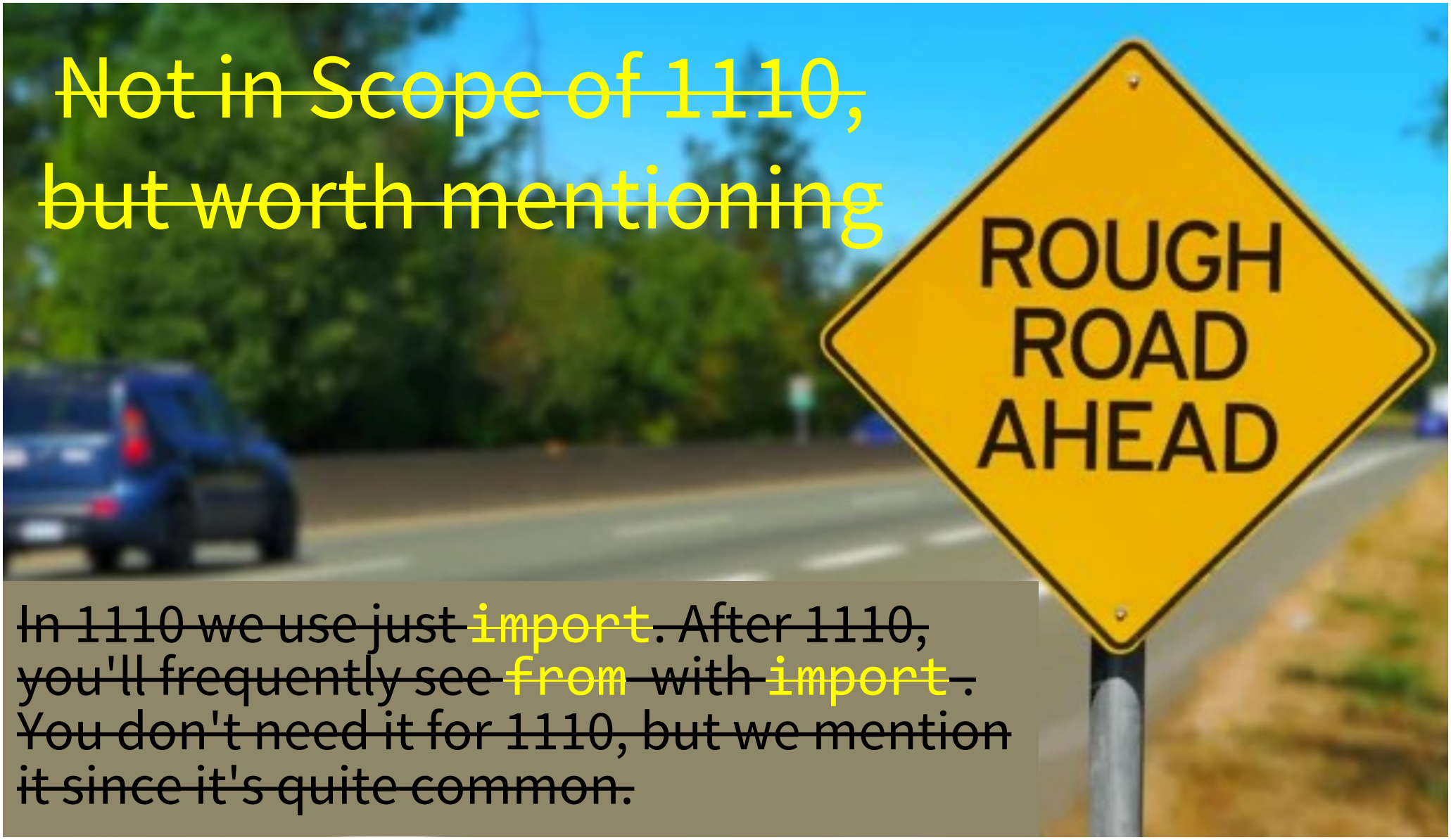
```
>>> import my_module
>>> help(my_module)
```

```
Help on module my_module:

NAME
    my_module

DESCRIPTION
    This is a simple module.
    It shows how modules work

DATA
    x = 9
```

Other Useful Modules

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

We'll use these many of these this semester.



~~Not in Scope of 1110, but worth mentioning~~

~~In 1110 we use just `import`. After 1110, you'll frequently see `from` with `import`. You don't need it for 1110, but we mention it since it's quite common.~~

Sorry folks, I was trying to simplify this lecture and forgot that we actually *do* use `from X import Y` in Lab 3 this year. Sorry for the confusion. I still maintain you won't see this construct *much*, but I definitely cannot claim we won't use it in 1110. We will this week.

`from` command (1)

You can also import like this:

`from` <module> `import` <function name>

Example:

```
C:\> python
>>> from math import pi
>>> pi
3.141592653589793
```

What Python can access directly

```
pi  3.141592653589793
```

`pi` *gets copied from the math module to the "purple box"*

No longer need the module name!

from command (2)

You can also import *everything* from a module:

```
from <module> import *
```

Example:

```
C:\> python
>>> from math import *
>>> pi
3.141592653589793
>>> sqrt(pi)
1.7724538509055159
```

everything *gets copied from the math module to the "purple box"*

What Python can access directly

```
sqrt()
log()
e   2.718281828459045
pi  3.141592653589793
…
```

Module functions now behave like built-in functions

Dangers of Importing Everything

Example:

```
C:\> python
>>> e = 12345
>>> from math import *
>>> e
2.71828182845
```

the variable e *is overwritten by this import!*

What Python can access directly

e ~~12345~~ 2.71828182845

sqrt()
log()
pi 3.141592653589793
…

Do you know the name of every mathematical constant? Might not want to import them all in case they overwrite one of your variables.

Avoiding **from** keeps variables separate

Example:

```
C:\> python
>>> e = 12345
>>> import math
>>> math.e
2.718281828459045
>>> e
12345
```