http://www.cs.cornell.edu/courses/cs1110/2020sp

Lecture 3: Functions & Modules

(Sections 3.1-3.3)

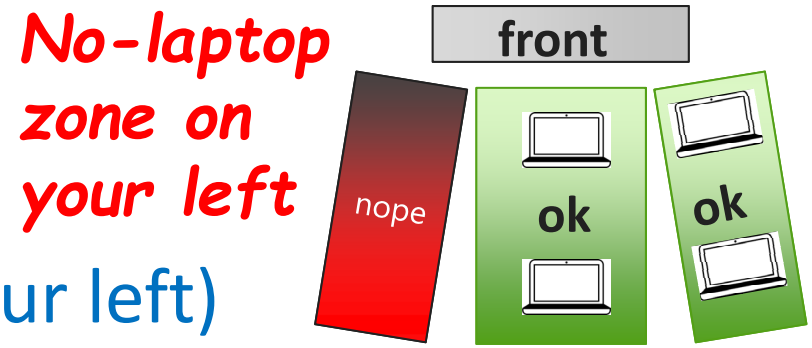
CS 1110

Introduction to Computing Using Python

Revisions after lecture are shown in **orange**.
We will say more about running a script (last few slides) next lecture.

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

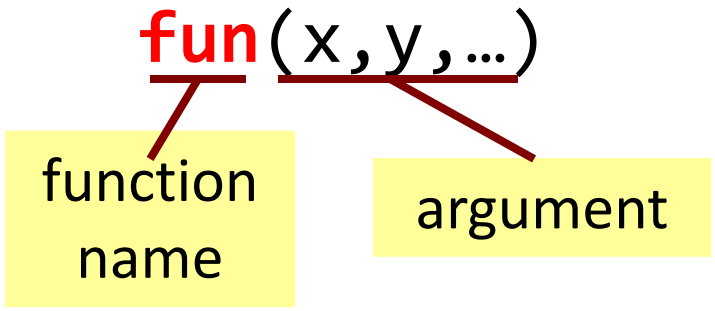
Announcements



- No laptop use stage right (your left)
- We will use clickers, but not for credit. Therefore no need to register your clicker.
- Textbook is terse, so don't worry if you don't understand everything that you've read
  - We teach the material in lecture
  - But good to have read it beforehand so that you have a 2nd look at the material in class
  - We'll also have exercises/questions in class to help you learn the material.
- Before next lecture, read Sections 3.4-3.11

Function Calls

- Function expressions have the form:



- Some math functions built into Python:

```
>>> x = 5
>>> y = 4
>>> bigger = max(x, y)
>>> bigger
5
```

```
>>> a =
round(3.14159265)
>>> a
3
```

Arguments can be any expression

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()

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

- Longer list:

http://docs.python.org/3.7/library/functions.html

Modules

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

  `import` *<module name>*

  Can then access functions like this:

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

  Example:

```
>>> import math
>>> p = math.ceil(3.14159265)
>>> p
4
```

Module Variables

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

  <module name>.*<variable name>*

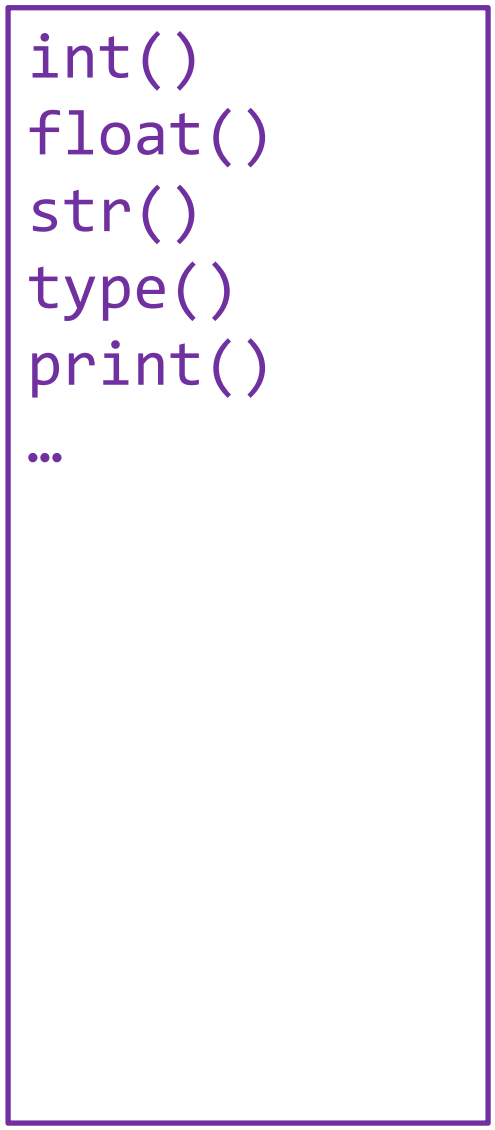
- **Example**:

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

Visualizing functions & variables

- So far just built-ins

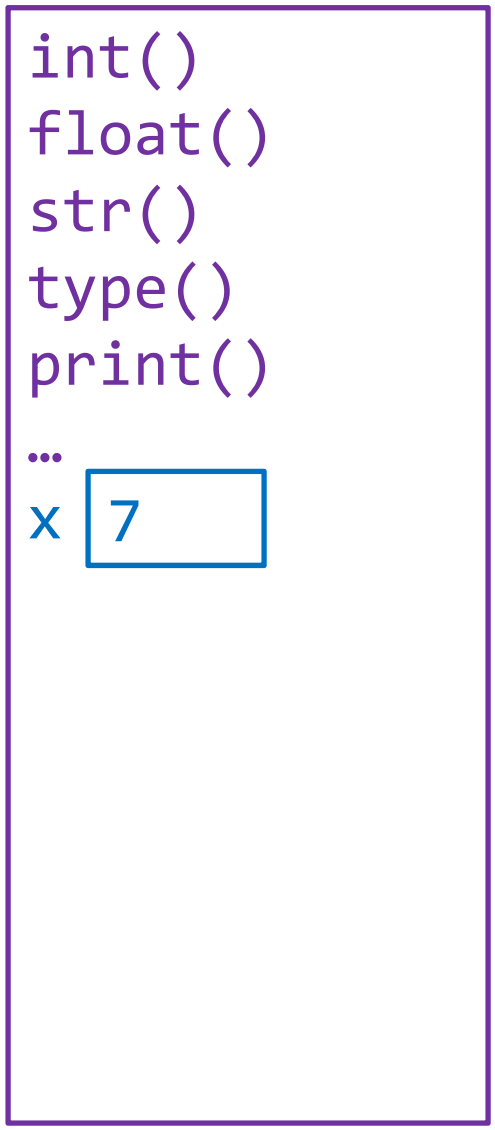
```
C:\> python
>>>
```



Visualizing functions & variables

- So far just built-ins
- Now we've defined a new variable

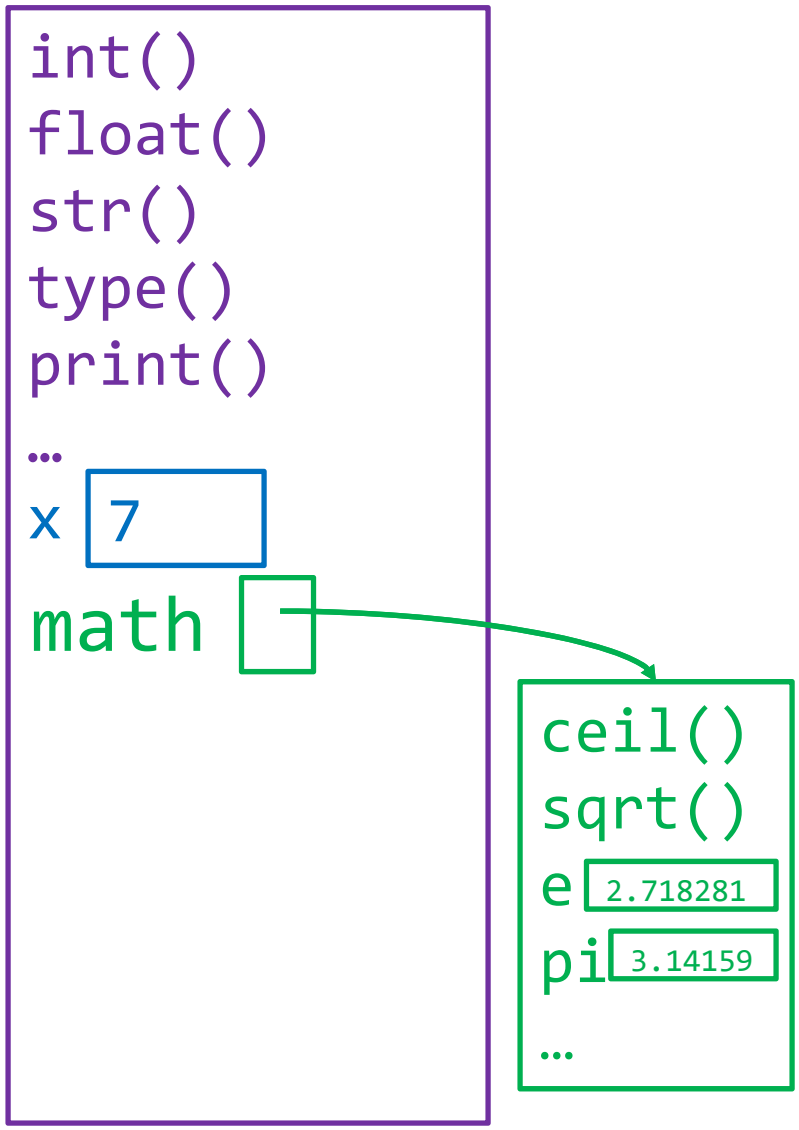
```
C:\> python
>>> x = 7
>>>
```



Visualizing functions & variables

- So far just built-ins
- Now we've defined a new variable
- Now we've imported a module

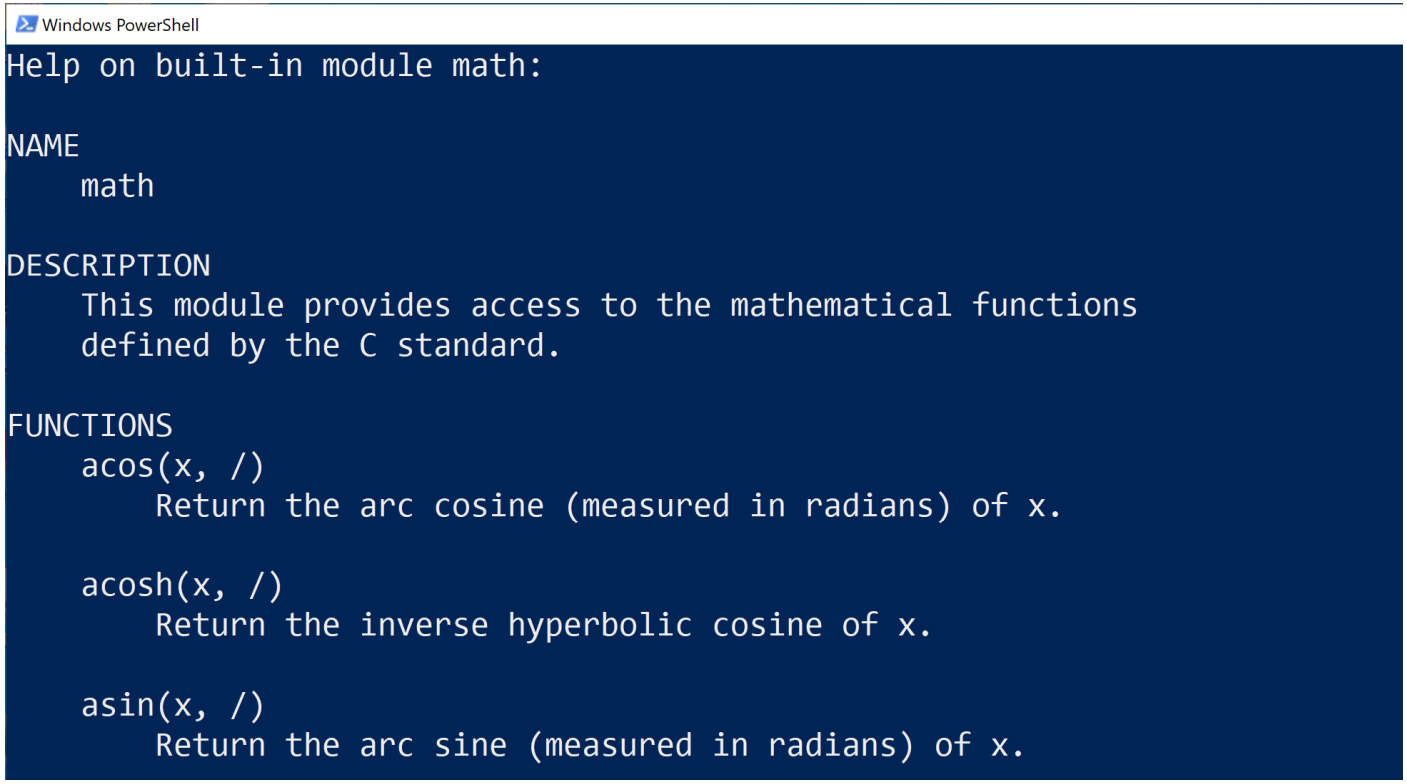
```
C:\> python
>>> x = 7
>>> import math
>>>
```



module help

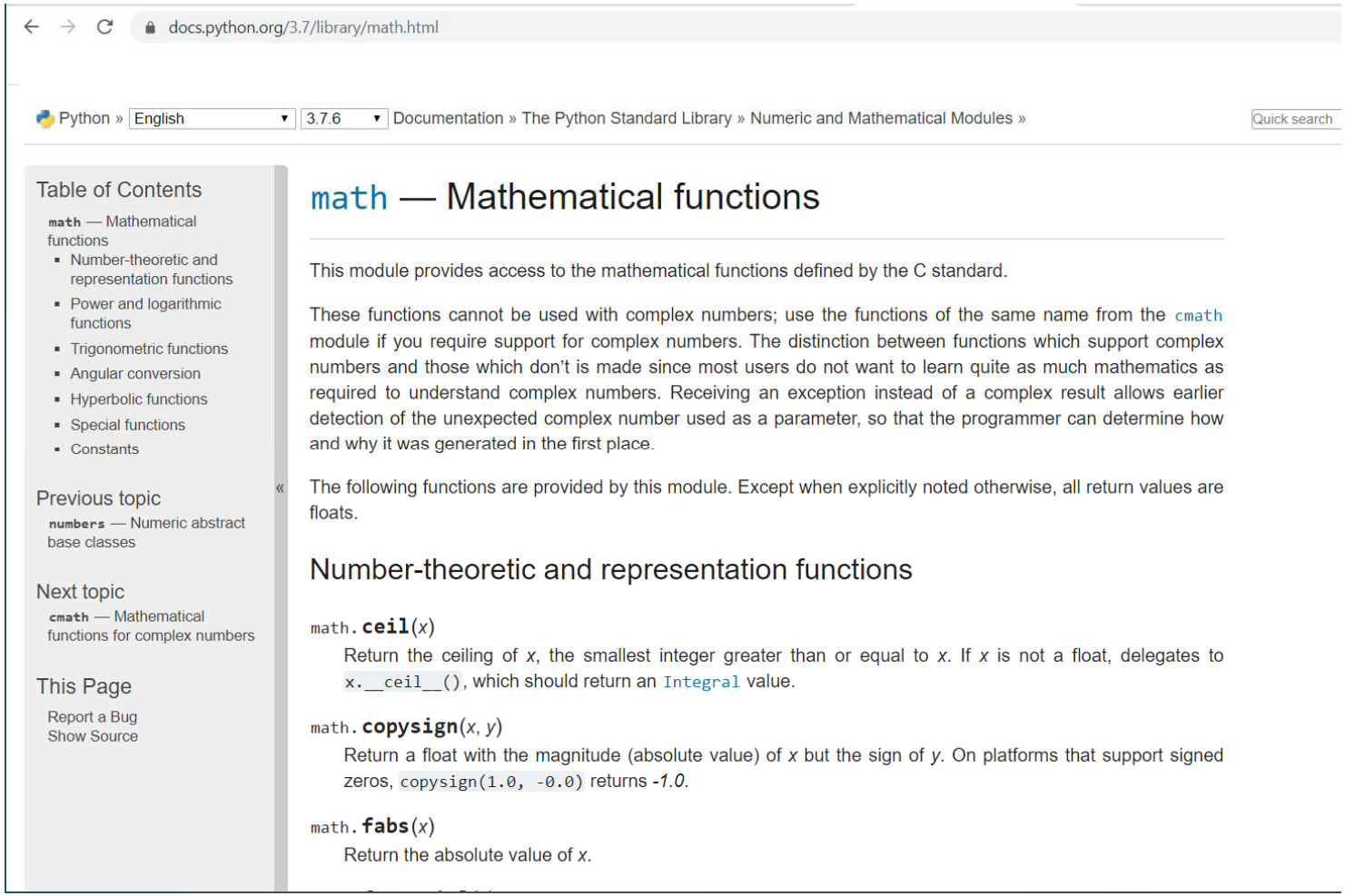
After importing a module, see what functions and variables are available:

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



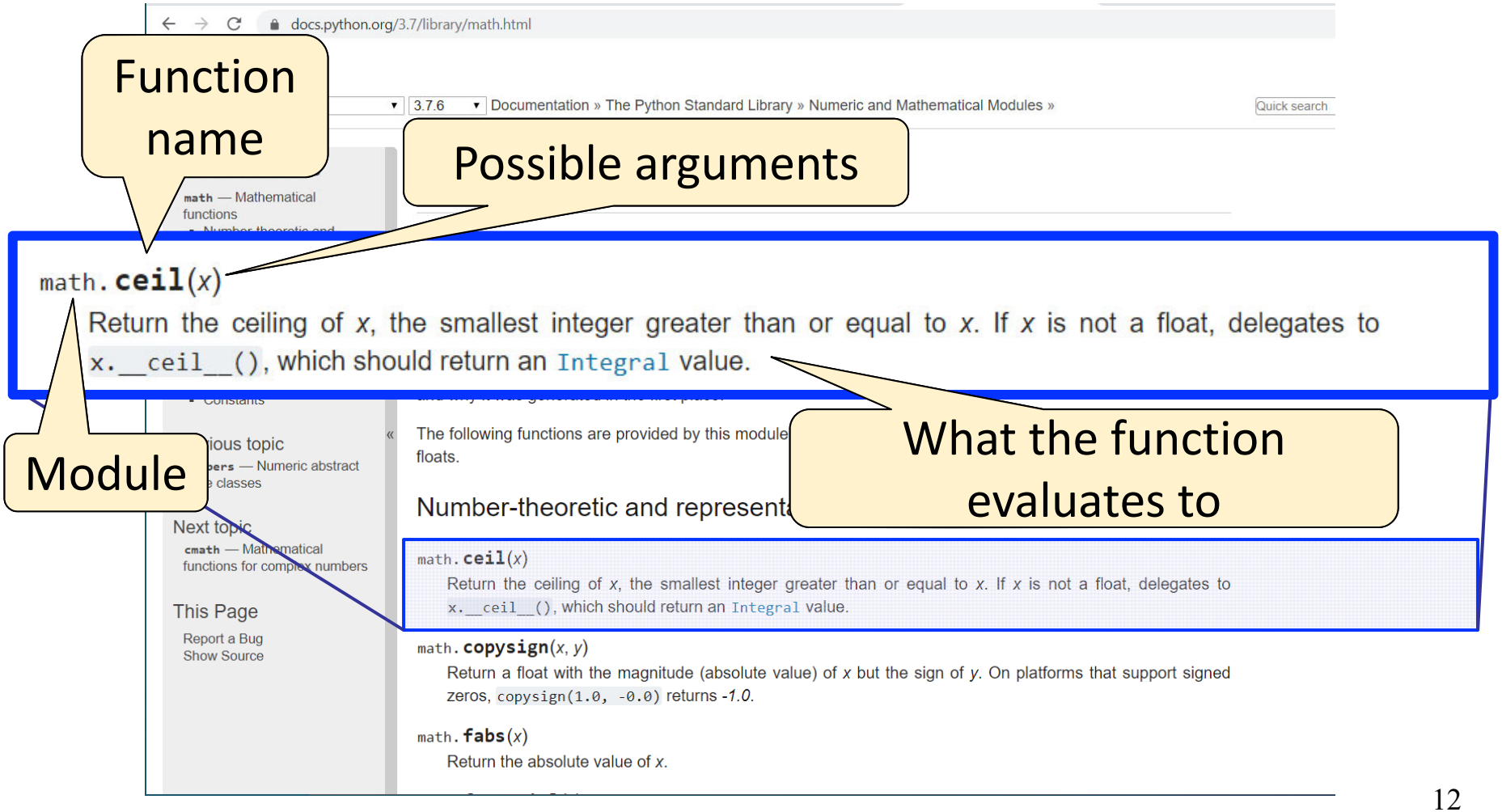
Reading the Python Documentation

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



Reading the Python Documentation

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



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

Making your Own Module

Write in a text editor

We recommend Atom…

…but any editor will work

my_module.py

```
1  # my_module.py
2
3  """ This is a simple module.
4  It shows how modules work """
5
6  x = 1+2
7  x = 3*x
8
9
10
```

Interactive Shell vs. Modules

Python Interactive Shell

```
Windows PowerShell
PS C:\Users\Daisy> python
Python 3.7.4 (default, Aug  9 2019, 18:34:13) [M

Type "help", "copyright", "credits" or "license"
>>> x = 1+2
>>> x = 3*x
>>> x
9
>>> _
```

- Type `python` at command line
- Type commands after >>>
- Python executes as you type

Module

my_module.py

```
1  # my_module.py
2
3  """ This is a simple module.
4  It shows how modules work """
5
6  x = 1+2
7  x = 3*x
```

- Written in text editor
- Loaded through `import`
- Python executes statements when `import` is called

Section 2.4 in your textbook discusses a few differences

my_module.py

Module Text

```
# my_module.py

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

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

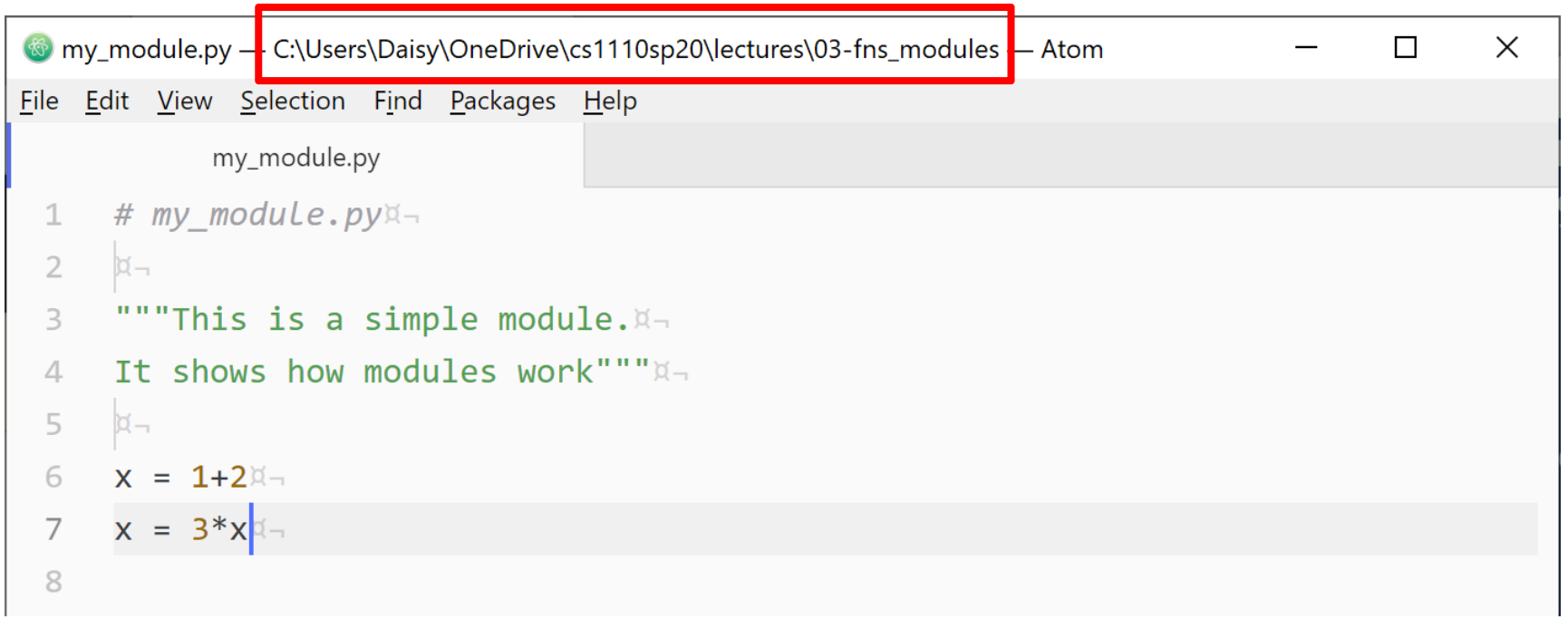
Single line comment
(not executed)

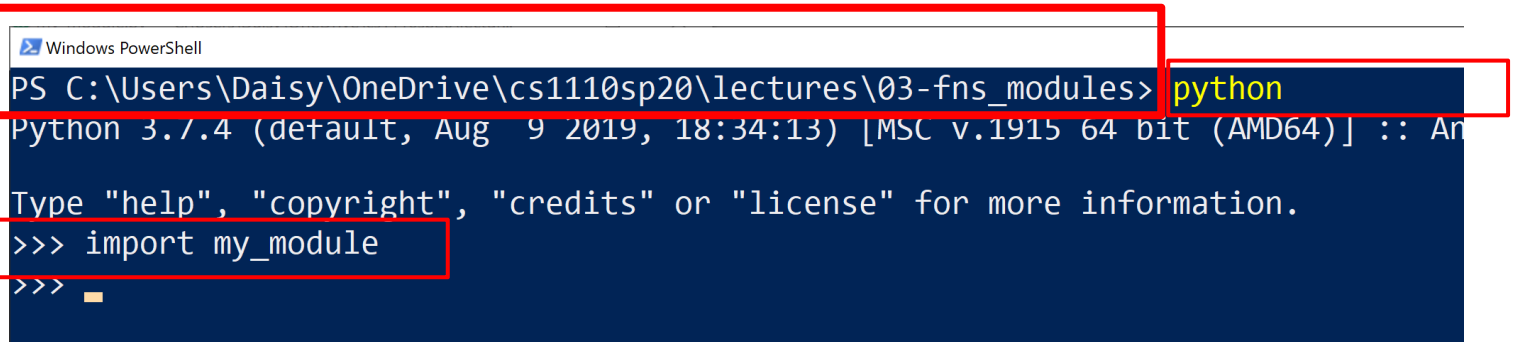
Docstring
(note the Triple Quotes)
Acts as a multi-line comment
Useful for *code documentation*

Commands
Executed on `import`

Modules Must be in Working Directory!

Must run python from same folder as the module





Using a Module (my_module.py)

Module Text

```
# my_module.py

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

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

Python Command Shell

```
>>> import my_module
```

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

On import….

Module Text

Python Command Shell

```
# my_module.py

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

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

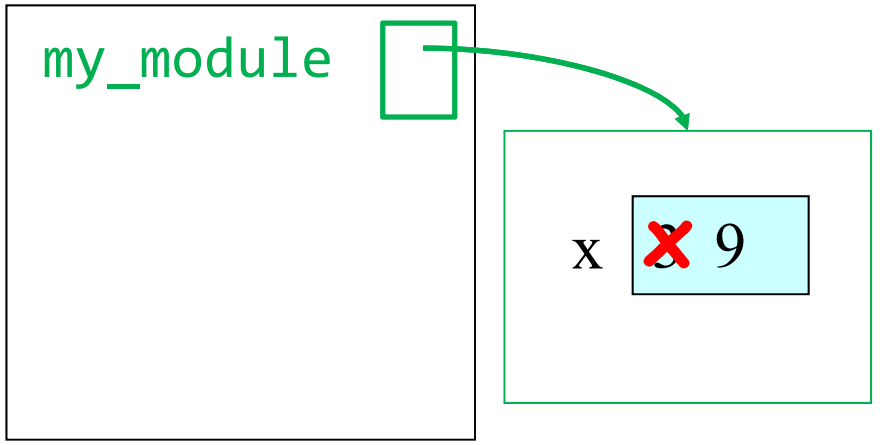
Python does not execute (because of #)

Python does not execute (because of """ and """)

Python executes this.

Python executes this.

```
>>> import my_module
```



variable x stays “within” the module

Clicker Question!

Module Text

```
# my_module.py

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

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

Python Command Shell

```
>>> import my_module
```

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

(A) >>>

(B) 9
>>>

(C) an error message

(D) The text of my_module.py

(E) Sorry, no clue.

Clicker Answer

Module Text

```
# my_module.py

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

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

Python Command Shell

```
>>> import my_module
```

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

(A) >>>

(B) 9
>>>

(C) an error message

(D) The text of my_module.py

(E) Sorry, no clue.

Using a Module (my_module.py)

Module Text

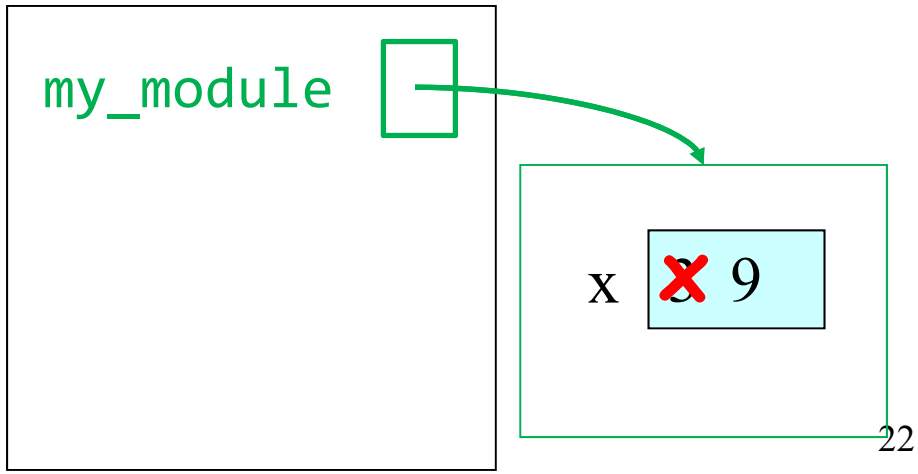
```
# my_module.py

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

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

Python Command Shell

```
>>> import my_module
>>> my_module.x
9
```

variable we want to access

module name

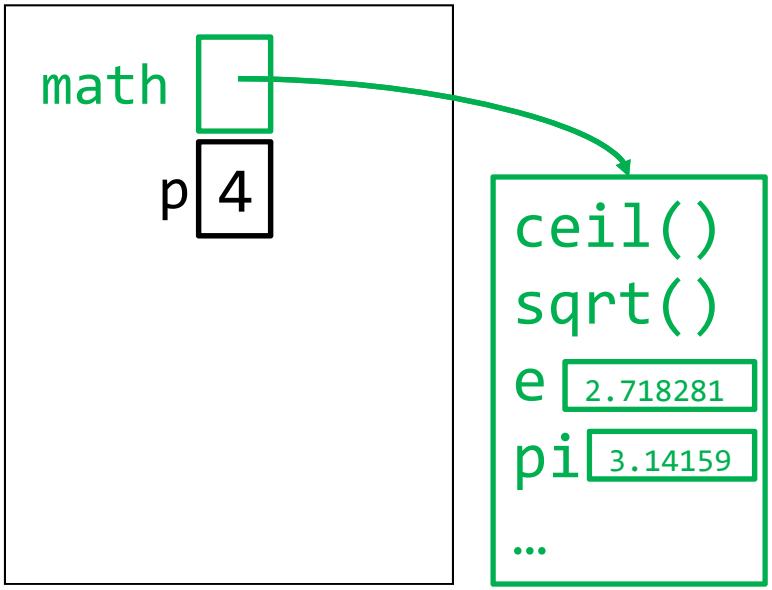


You must import

Windows command line
(Mac looks different)

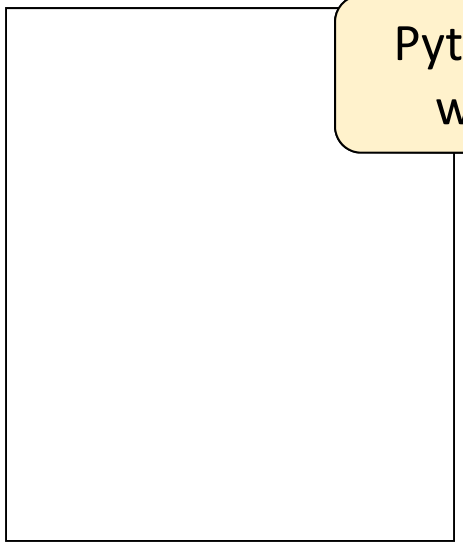
With import

```
C:\> python
>>> import math
>>> p = math.ceil(3.14159)
>>> p
4
```



Without import

```
C:\> python
>>> math.ceil(3.14159)
Traceback (most recent call last):
  File "<stdin>", line 1, in
<module>
NameError: name 'math' is not
defined
```

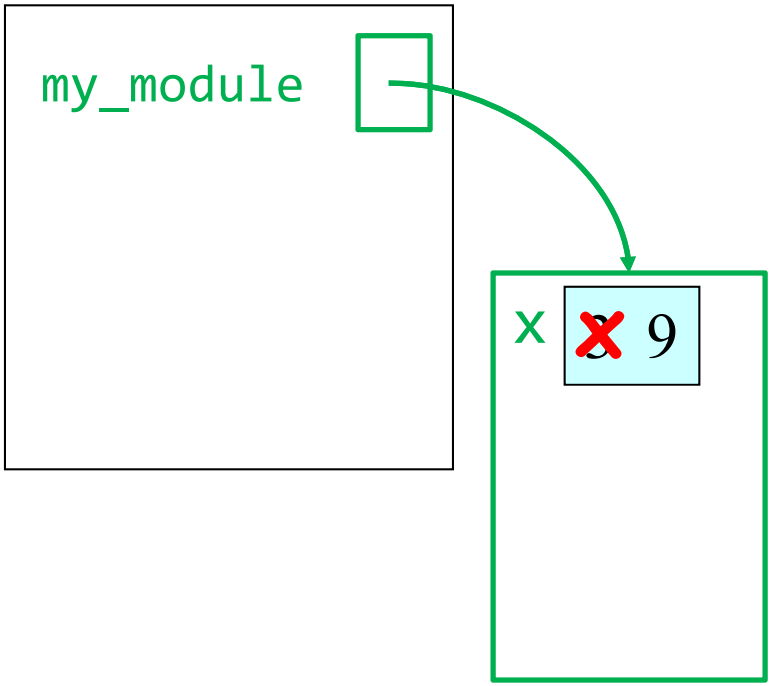


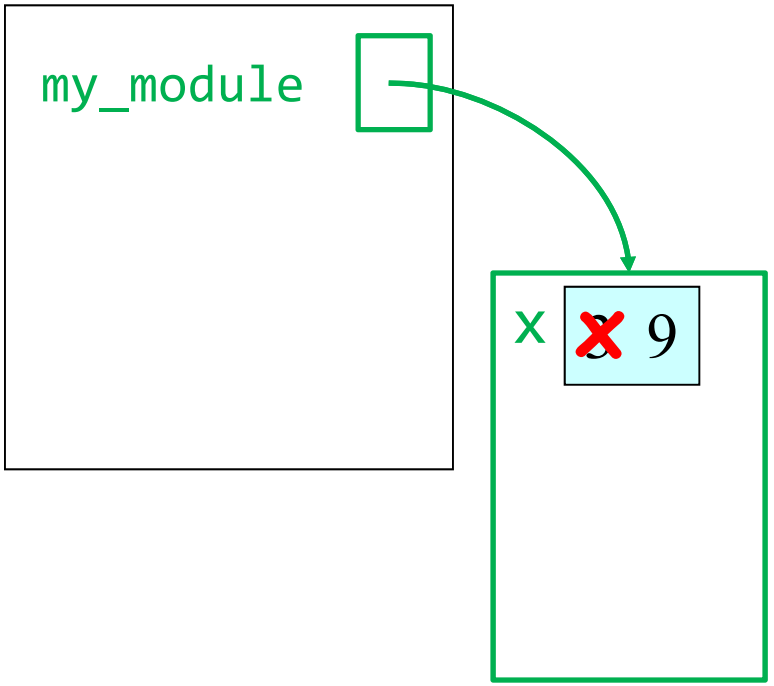
Python unaware of what "math" is

You Must Use the Module Name

```
>>> import my_module
>>> my_module.x
9
```

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



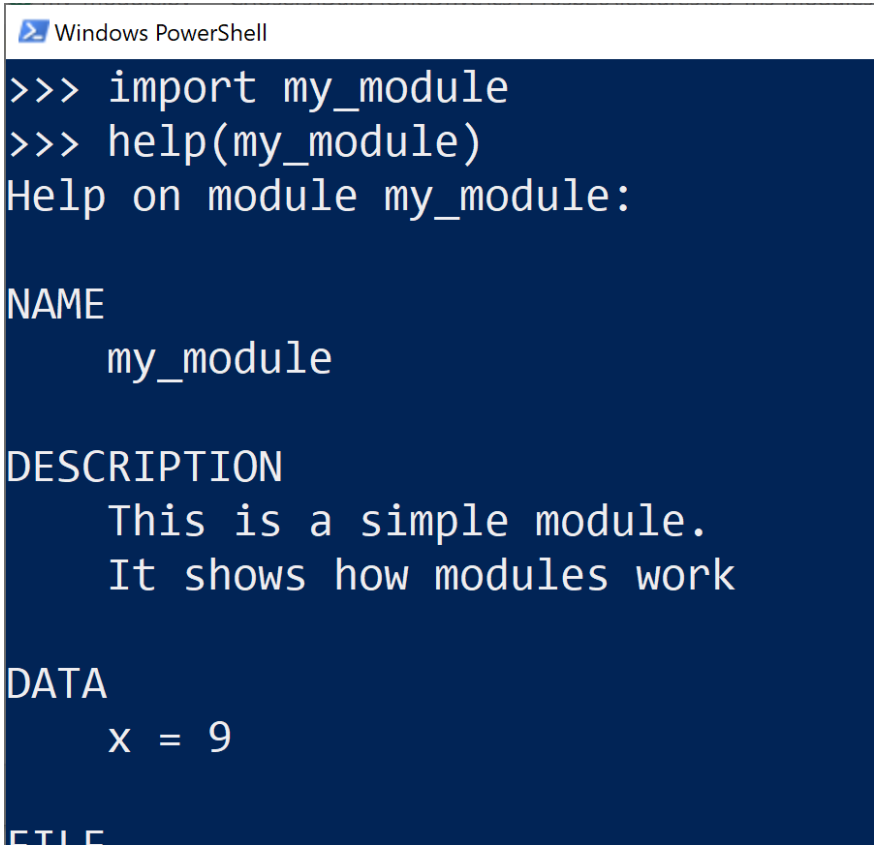


What does the docstring do?

Module Text

```
# my_module.py

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

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

Python Command Shell



from command

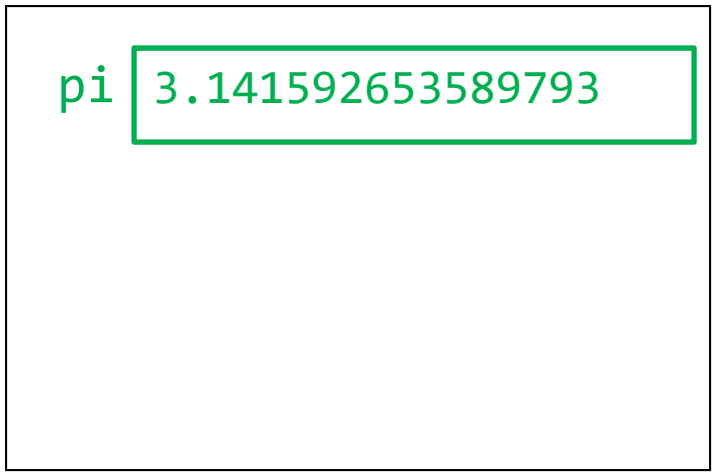
- You can also import like this:

  from <module> import <function name>

- **Example**:

```
>>> from math import pi
>>> pi
3.141592653589793
```

no longer need the module name



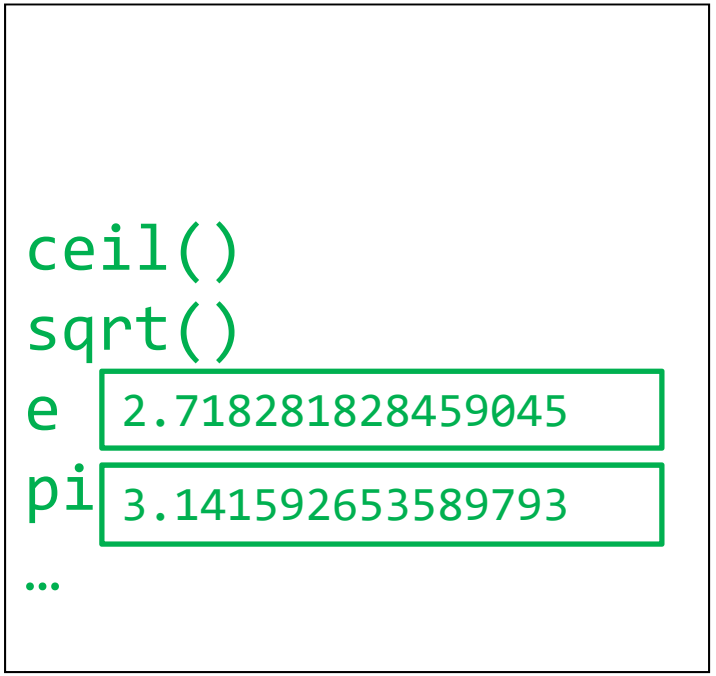
from command

- You can also import *everything* from a module:

  from <module> import *

- **Example**:

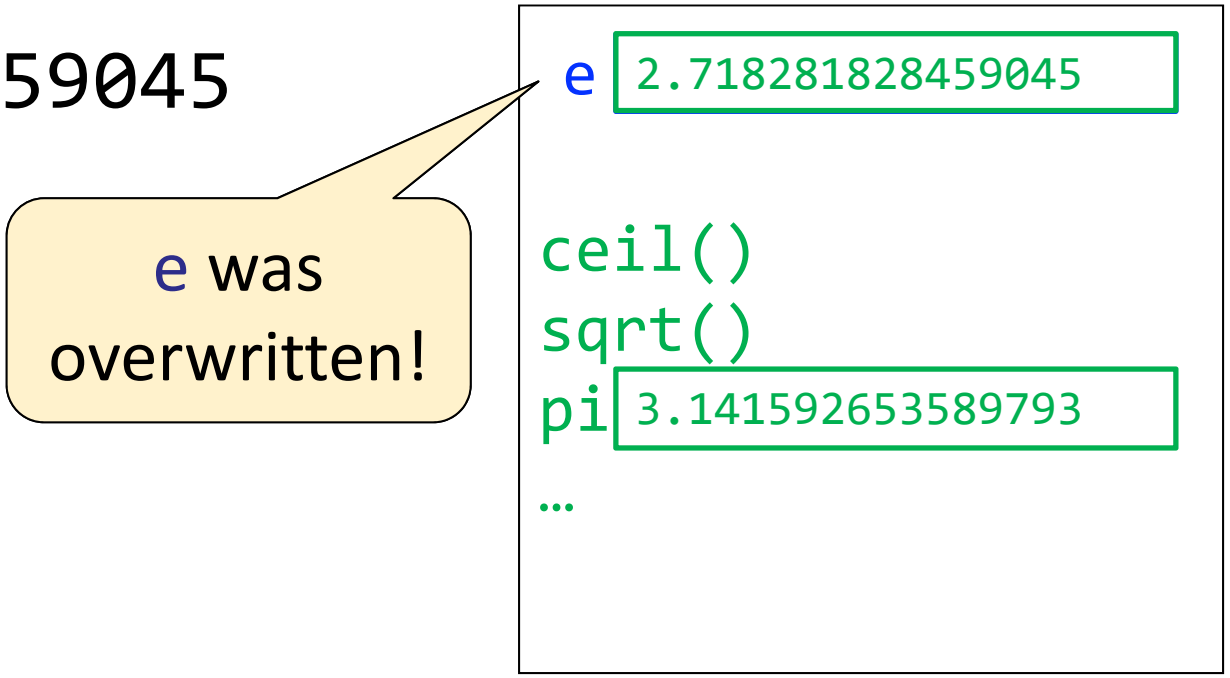
```
>>> from math import *
>>> pi
3.141592653589793
>>> ceil(pi)
4
```



Module functions now behave like built-in functions

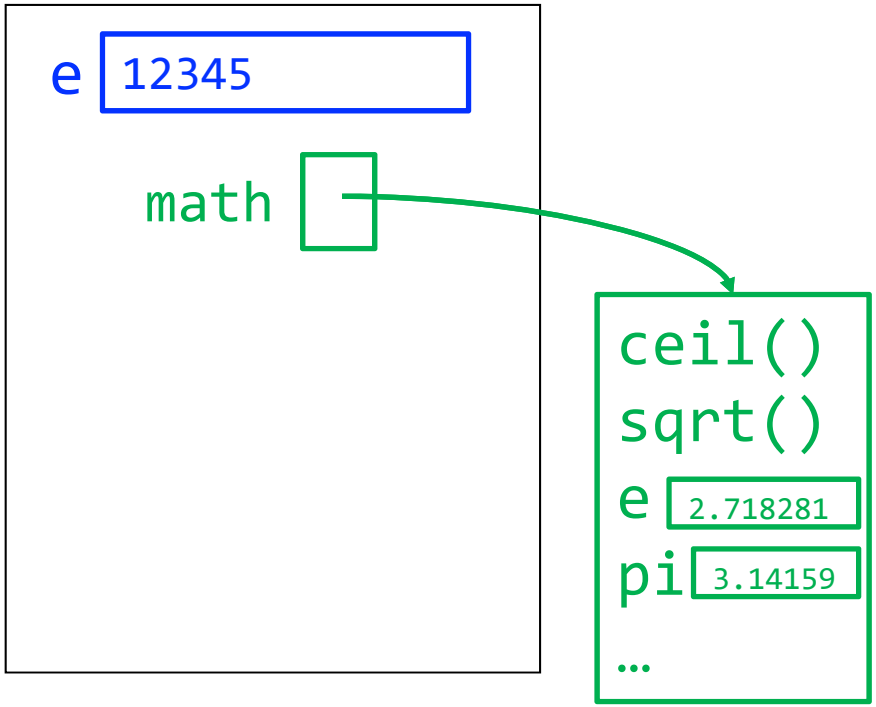
Dangers of Importing Everything

```
>>> e = 12345
>>> from math import *
>>> e
2.718281828459045
```



Avoiding from Keeps Variables Separate

```
>>> e = 12345
>>> import math
>>> math.e
2.718281828459045
>>> e
12345
```



Ways of Executing Python Code

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

Running a Script

- From the command line, type:

  python <script filename>

- Example:

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

looks like nothing happened

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

Running my_module.py as a script

my_module.py

Command Line

```
# my_module.py
```

Python does not execute (because of #)

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

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

Python does not execute (because of """ and """)

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

Python executes this.

Python executes this.

x ~~2~~ 9

Running my_module.py as a script

my_module.py

```
# my_module.py

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

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

Command Line

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

when the script ends, all memory used by my_module.py is deleted

thus, all variables get deleted (including x)

so there is no evidence that the script ran

Clicker Question

my_module.py

```
# my_module.py

"""This is a simple
my_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.

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

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

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

Running script.py as a script

Command Line

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


C:\>
```

script.py

```
# script.py

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

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

Subtle difference about script mode

Interactive mode

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

script.py

```
# script.py

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

x = 1+2
x = 3*x
print(x)
# note: in script mode, you will
# not get output if you just type x
```

Modules vs. Scripts

Module

- Provides functions, variables
- import it into Python shell

Script

- Behaves like an application
- Run it from command line

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