http://www.cs.cornell.edu/courses/cs1110/2019sp

Lecture 3: Functions & Modules

(Sections 3.1-3.3)

CS 1110

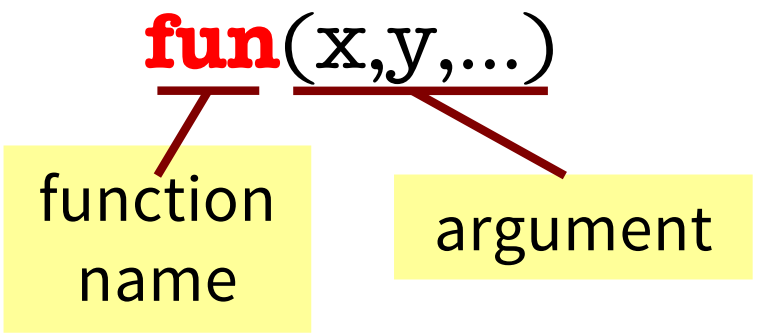
Introduction to Computing Using Python

Cornell CIS
COMPUTING AND INFORMATION SCIENCE

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

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/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

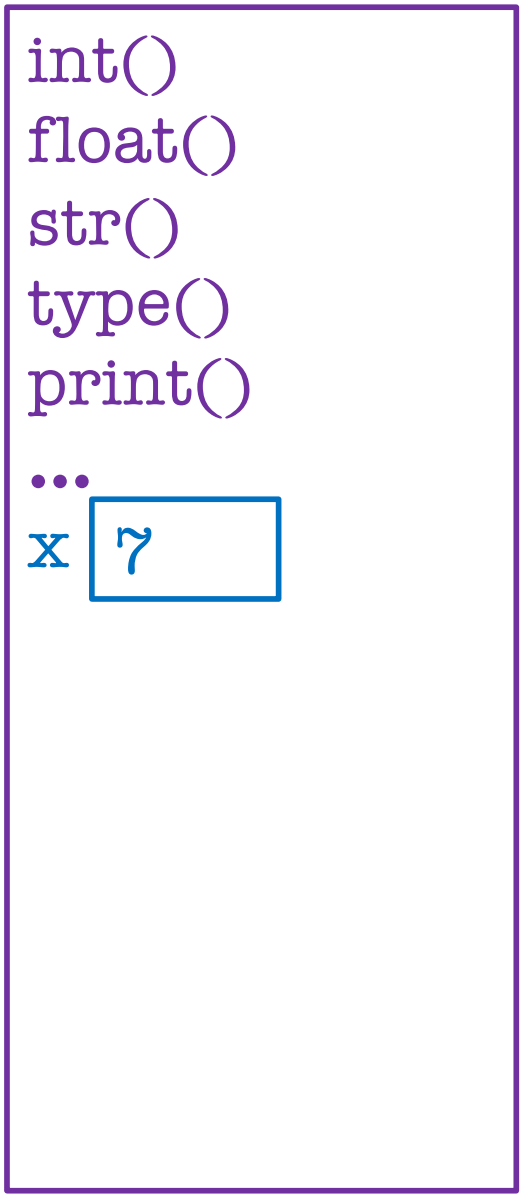
- So far just built-ins

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

Visualizing functions & variables

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

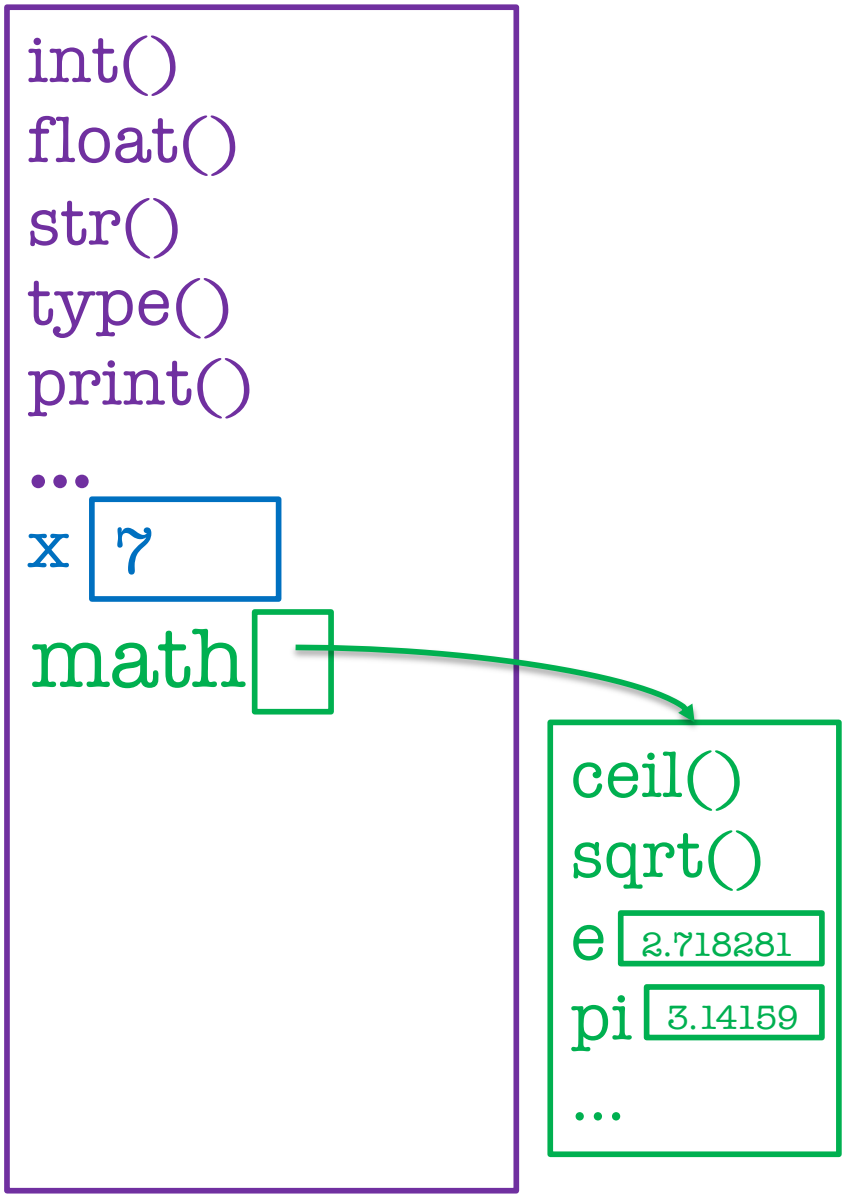
```
C:\> python
>>> x = 7
>>>
```



Visualizing functions & variables

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

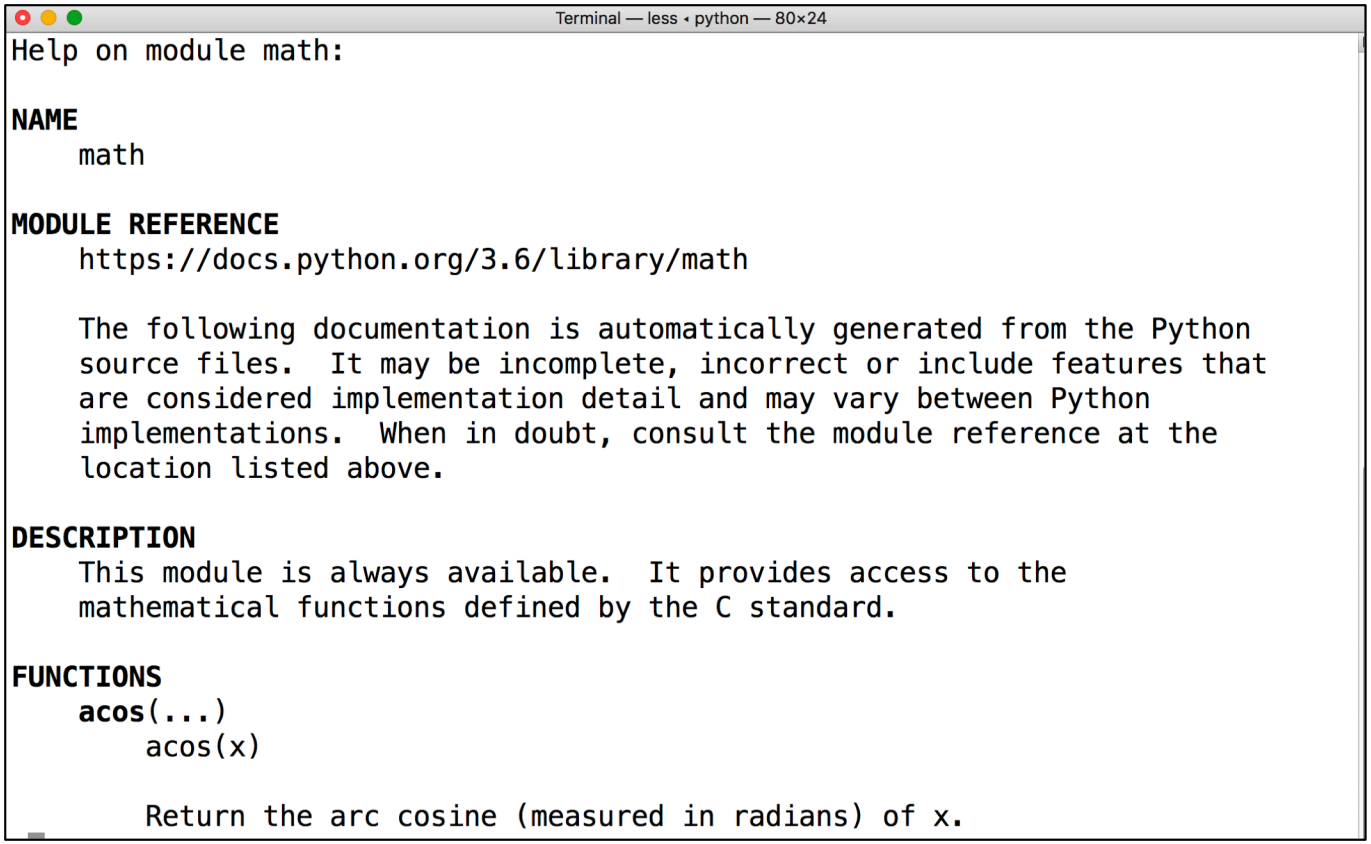
```
C:\> python
>>> x = 7
>>> import math
>>>
```



module help

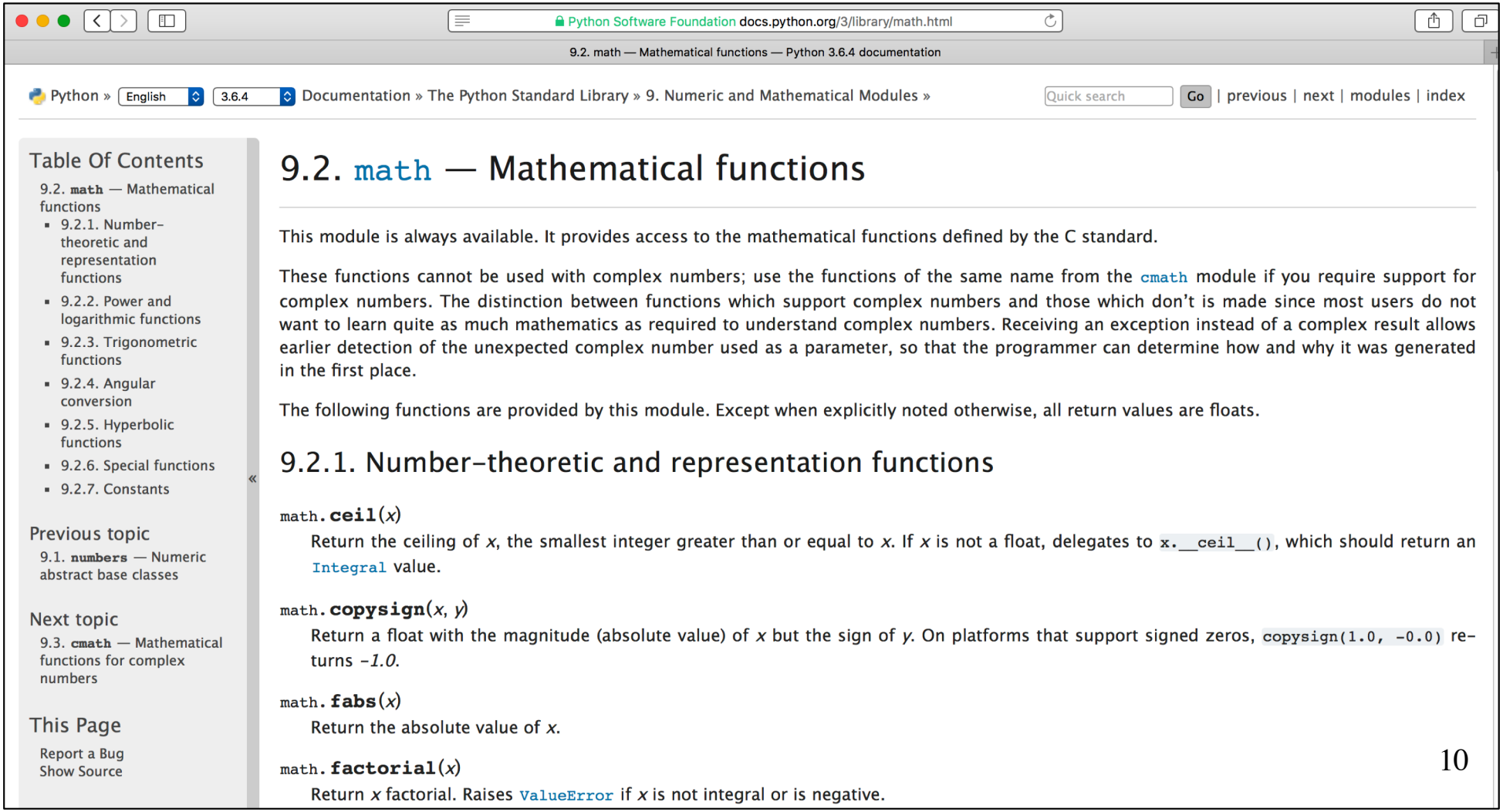
After importing a module, see what functions and variables are available:

>>> help(*<module name>*)



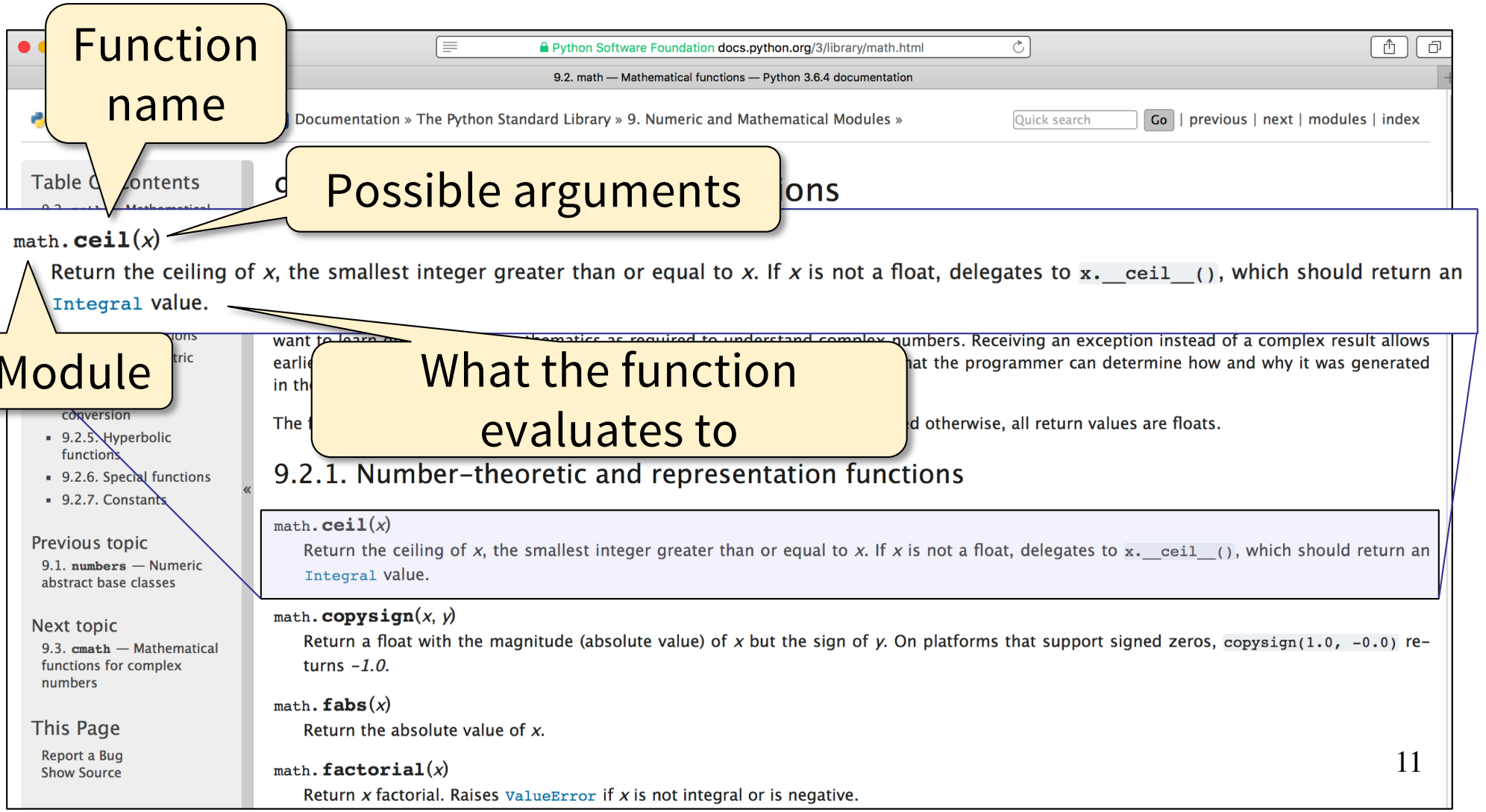
Reading the Python Documentation

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



A Closer Reading of the Python Documentation

https://docs.python.org/3/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

```
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 >>>
- 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
```

Single line comment
(not executed)

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

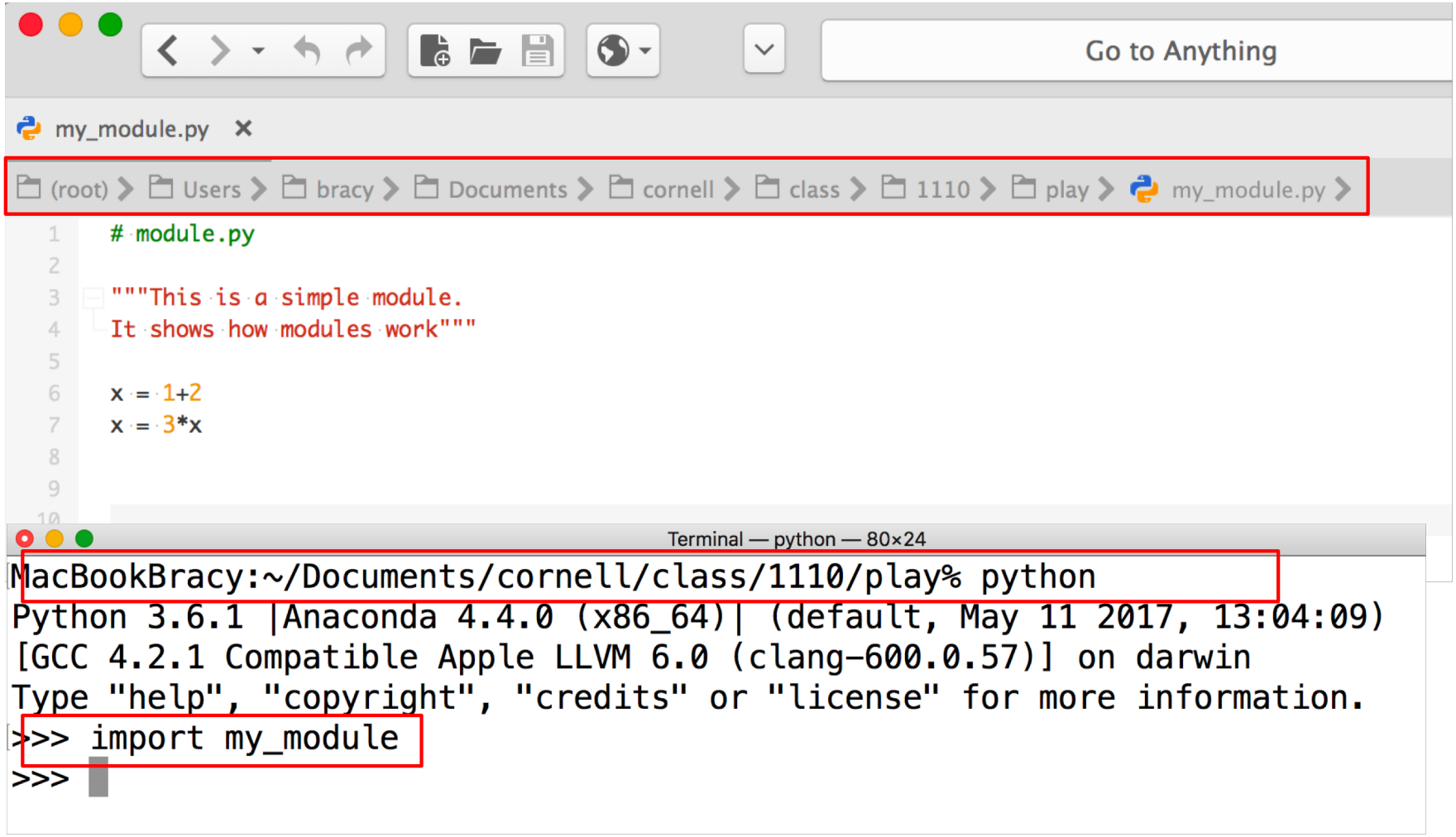
Docstring
(note the Triple Quotes)
Acts as a multi-line comment
Useful for *code documentation*

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

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

```
# 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 #)

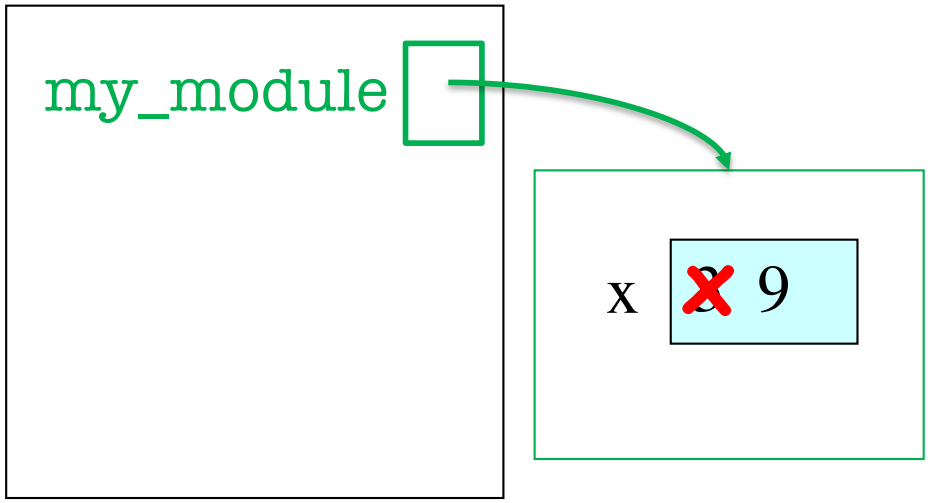
Python does not execute (because of """ and """)

Python executes this.

Python executes this.

Python Command Shell

```
>>> 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

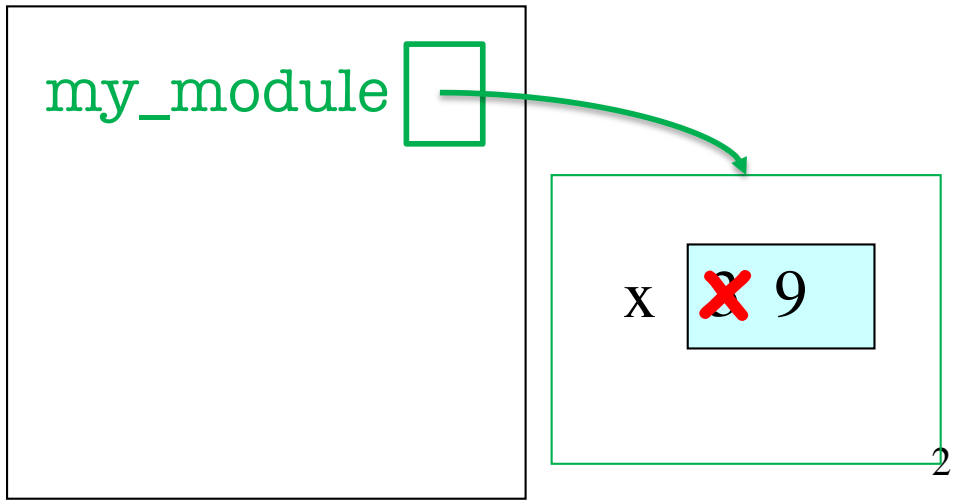
```
# my_module.py


"""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

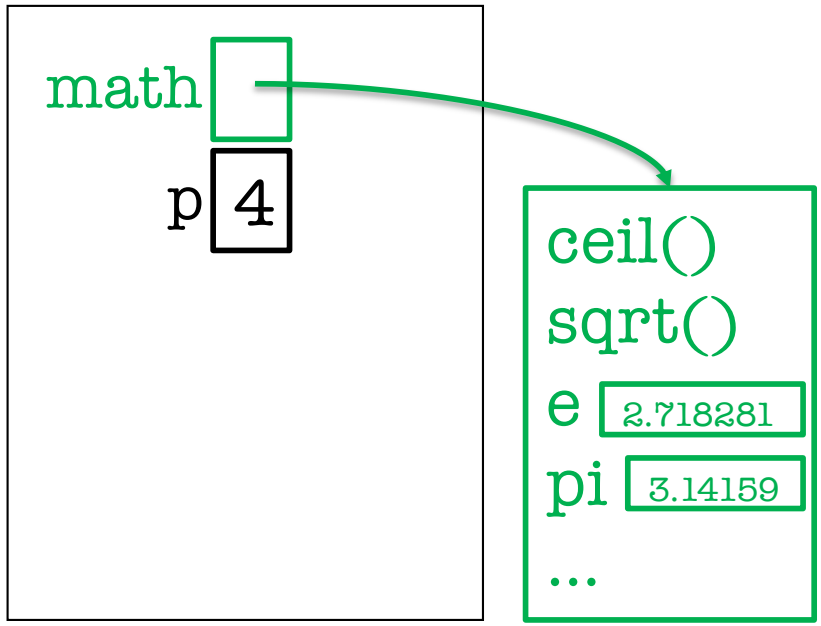


You must import

Windows command line
(Mac looks different)

With import

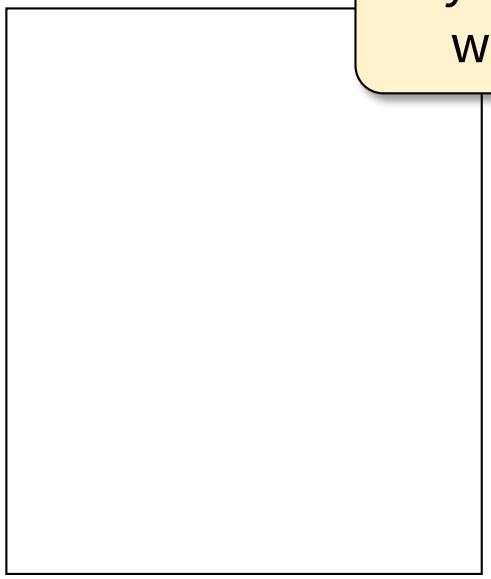
```
C:\> python
>>> import math
>>> p = math.ceil(3.14159)
>>> p
4
```



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
```

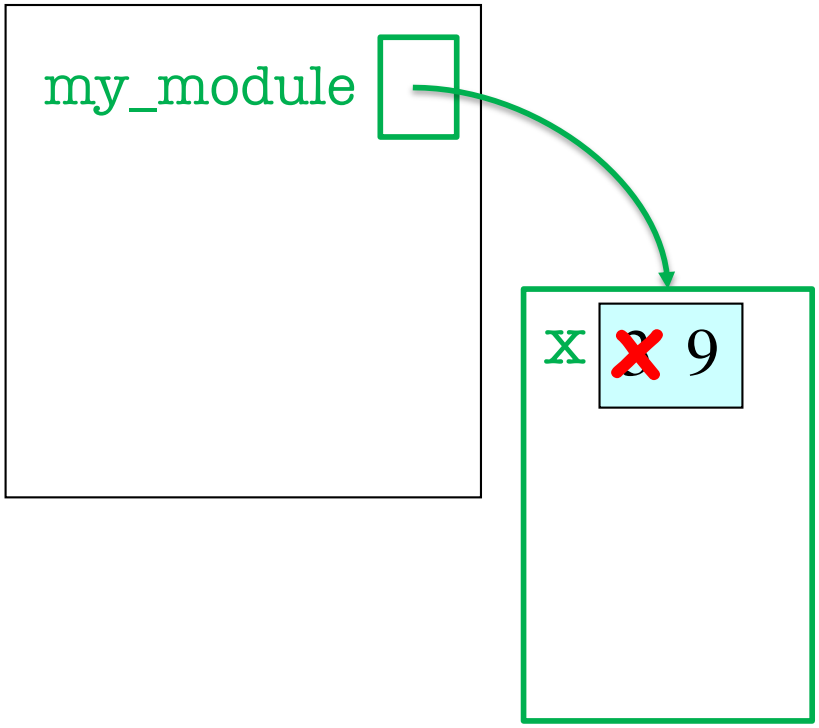
Python unaware of what “math” is

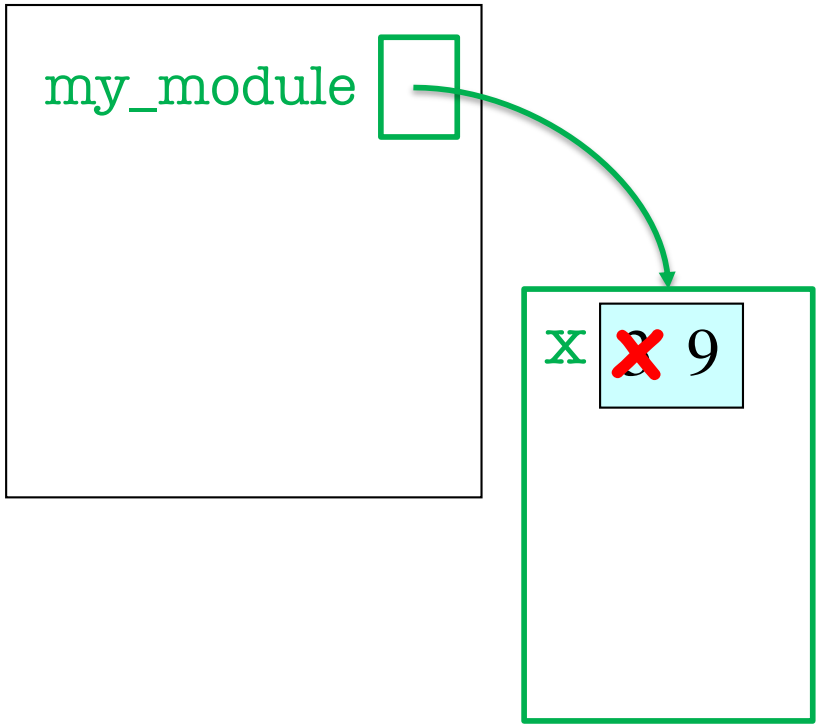


You Must Use the Module Name

```
>>> 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?

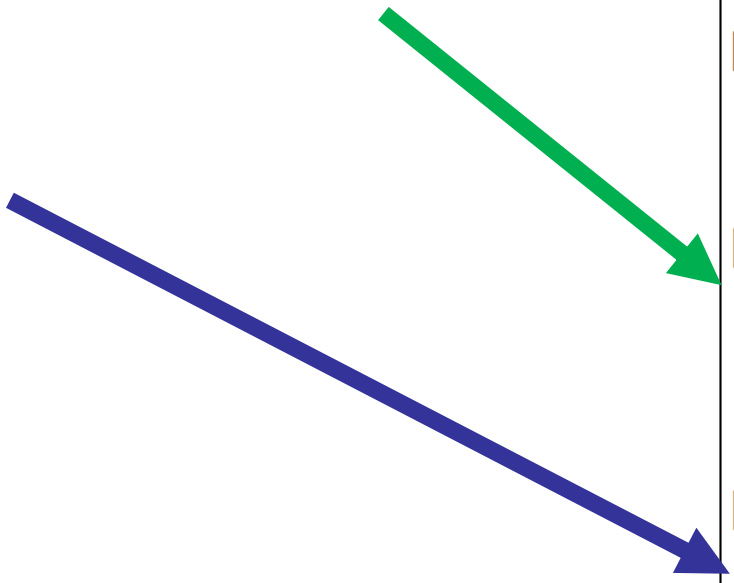
Module Text

```
# my_module.py


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


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

Python Command Shell

```
>>> 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
```



from command

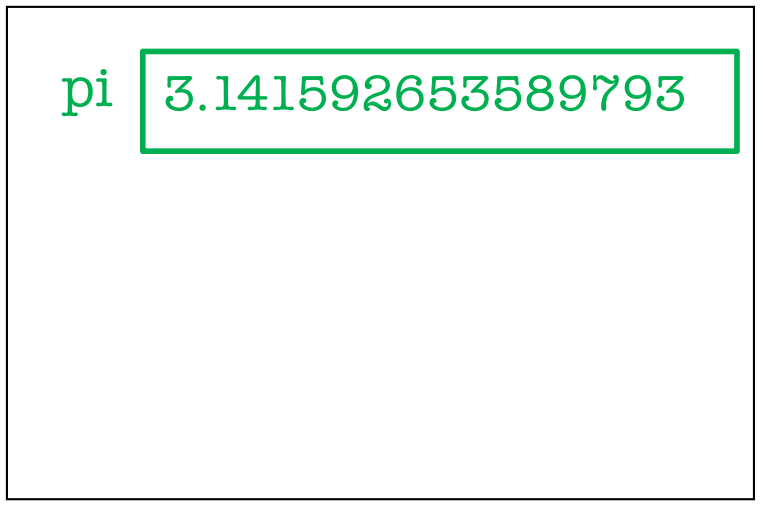
- You can also import like this:

  from <module> import <function name>

- **Example**:

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

no longer need the module name



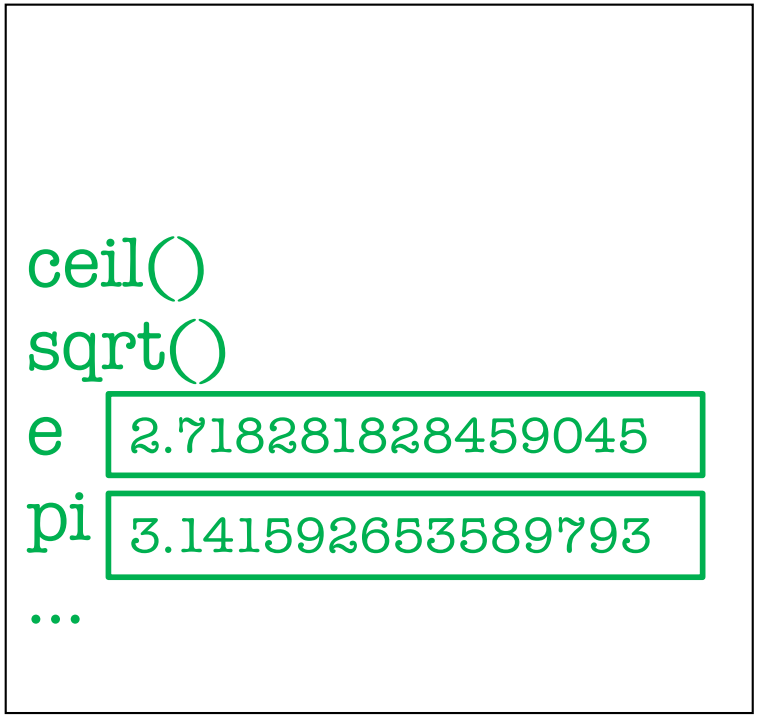
from command

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

  from <module> import *

- **Example**:

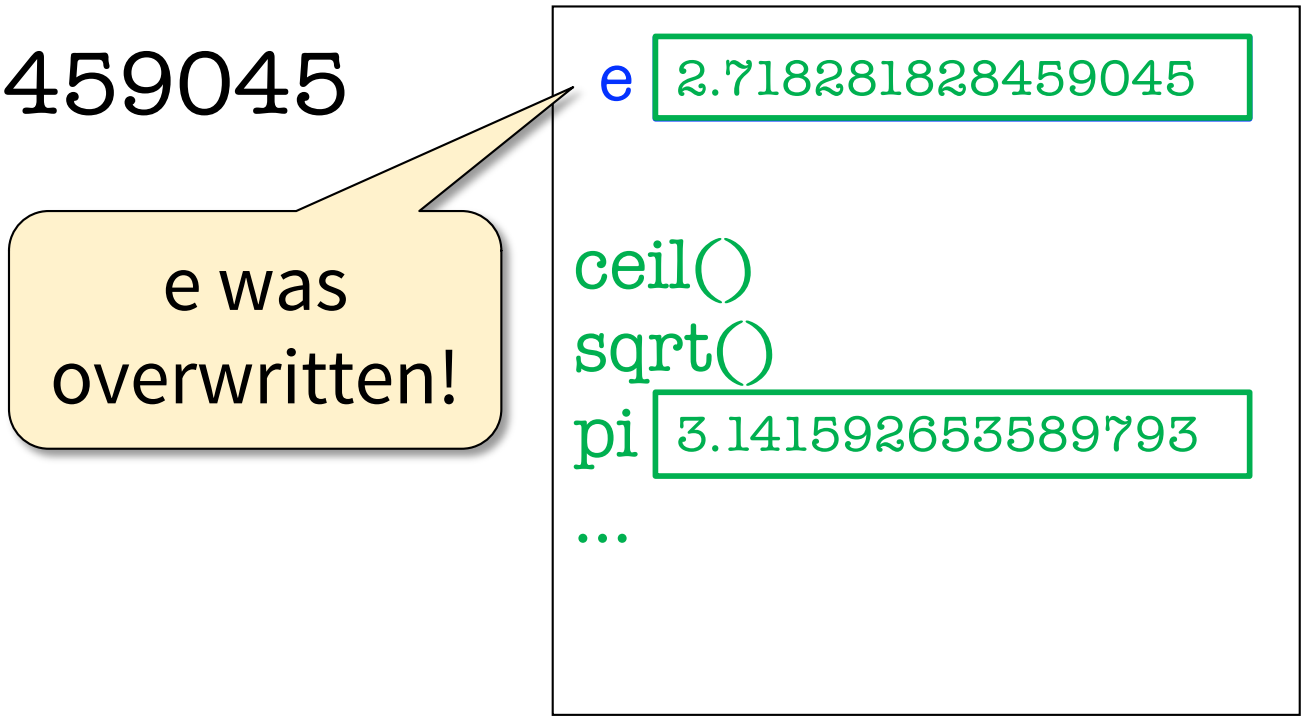
```
>>> from math import *
>>> pi
3.141592653589793
>>> ceil(pi)
4
```



Module functions now behave like built-in functions

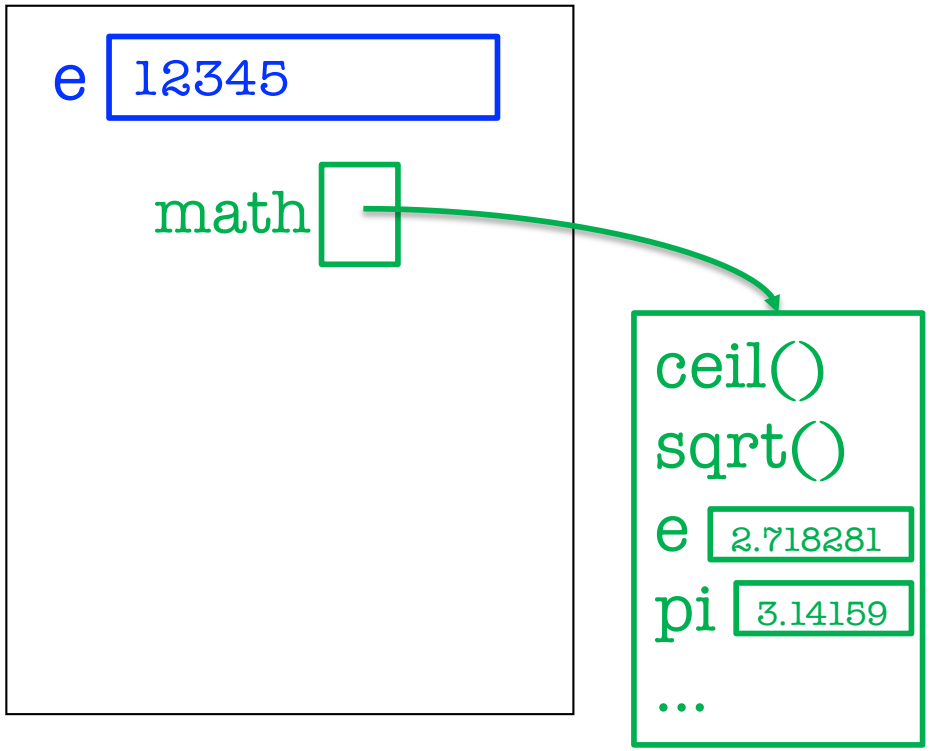
Dangers of Importing Everything

```
>>> 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

```
# 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.

Command Line

x ~~3~~ 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.

Clicker Answer

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.