http://www.cs.cornell.edu/courses/cs1110/2021sp

Lecture 3: Functions & Modules

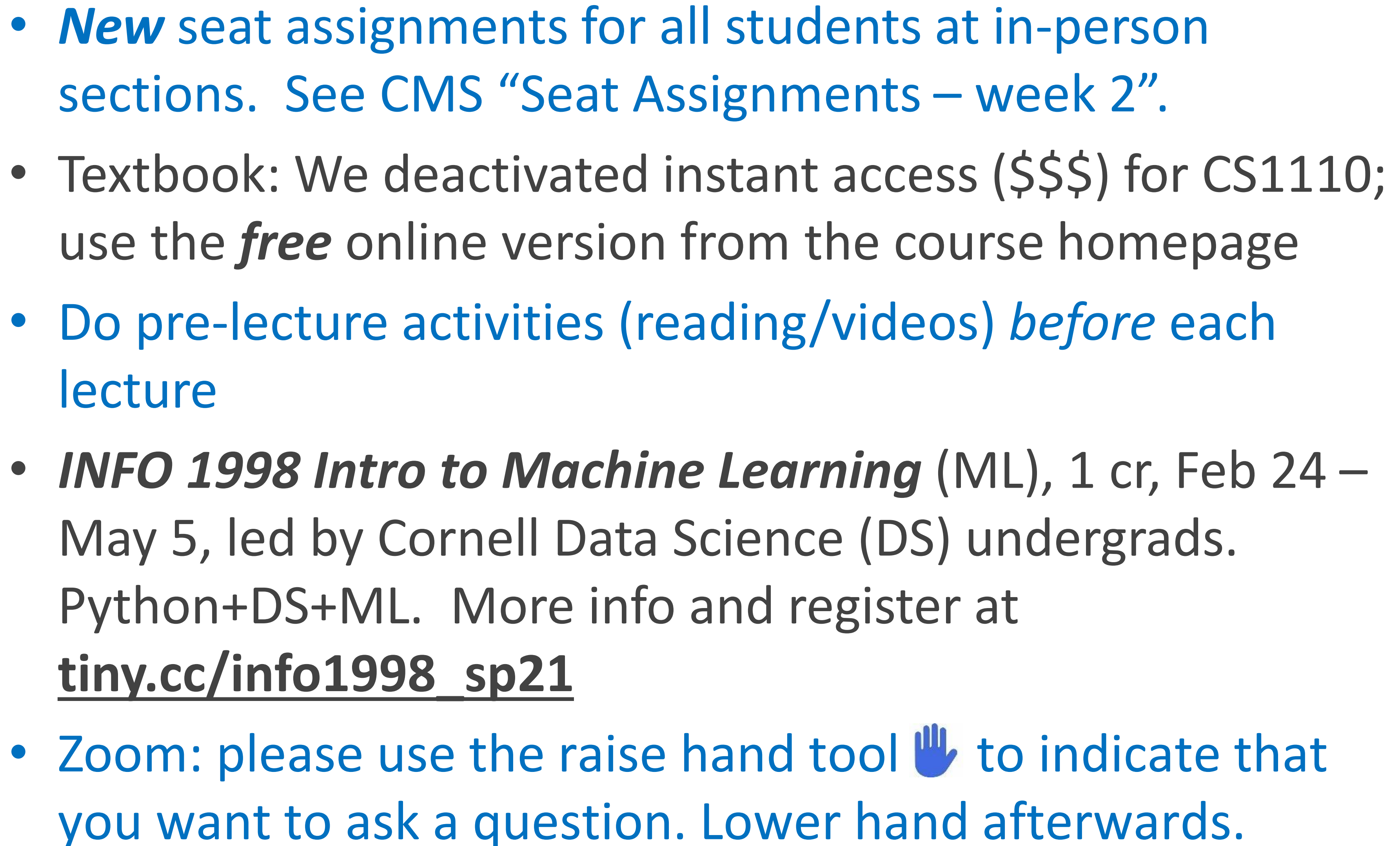
(Sections 3.1-3.3, 2.4)

CS 1110

Introduction to Computing Using Python

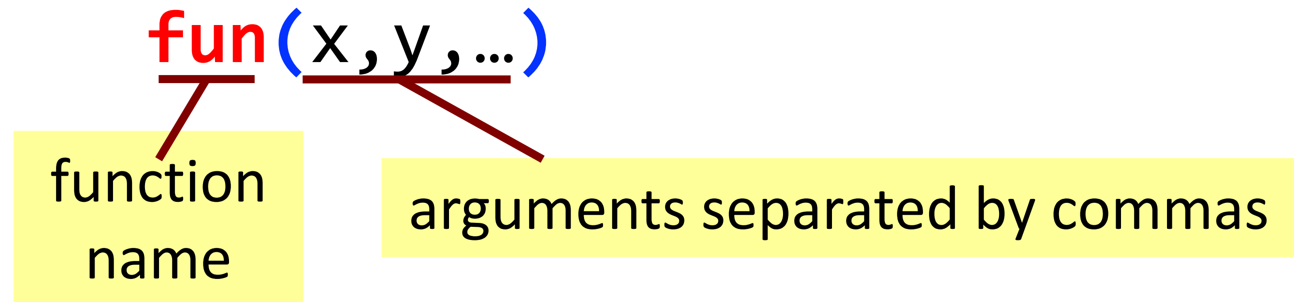
[E. Andersen, A. Bracy, D. Fan, D. Gries, L. Lee,
S. Marschner, C. Van Loan, W. White]

Announcements/Reminders



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 available

- So far just built-ins

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

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

Visualizing functions & variables available

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

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

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

Visualizing functions & variables available

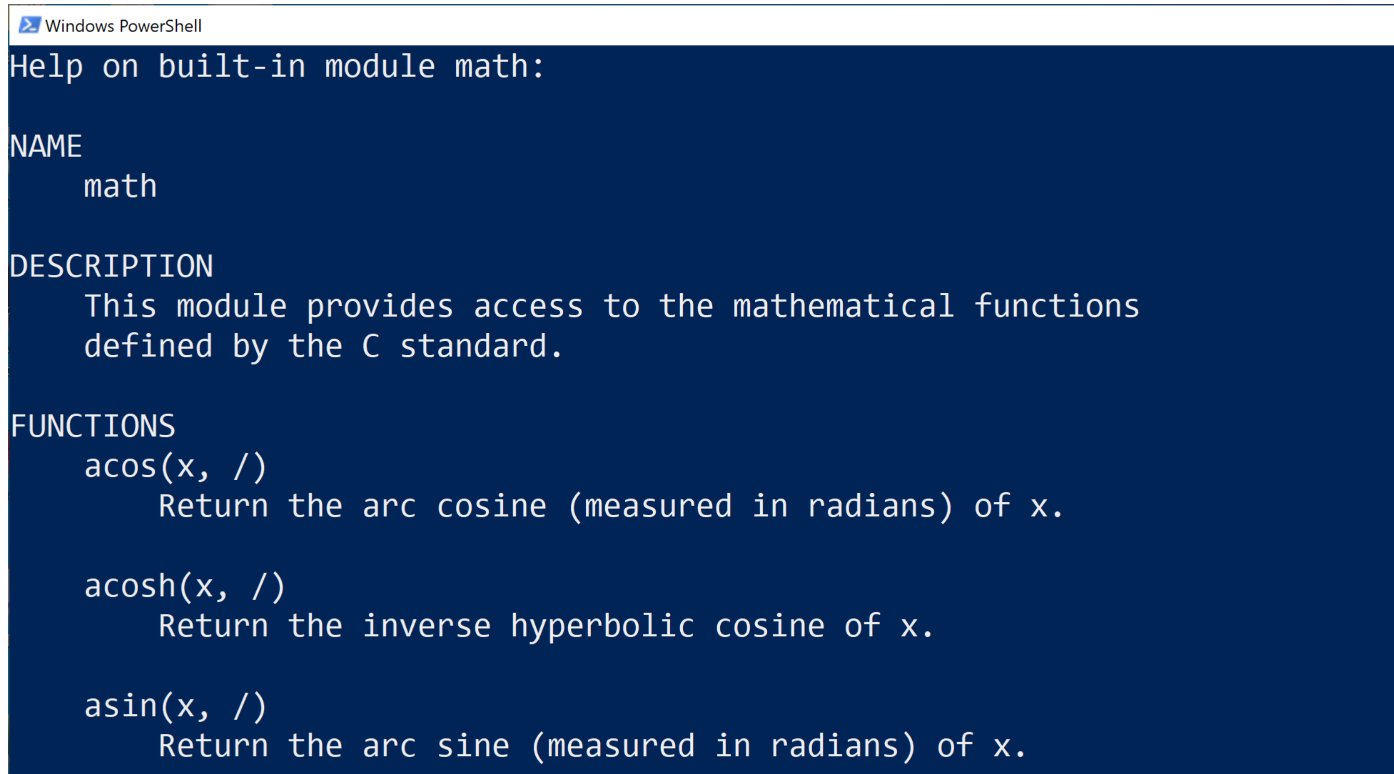
- So far just built-ins
- Now we've defined a new variable
- Now we've imported a module

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

module help

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

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



Reading the Python Documentation

https://docs.python.org/3.7/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*.

Reading the Python Documentation

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

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

3.7.6 Documentation » The Python Standard Library » Numeric and Mathematical Modules »

Quick search

Function name

Possible arguments

`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.

Module

What the function evaluates to

math — Mathematical functions

Previous topic

numbers — Numeric abstract base classes

Next topic

cmath — Mathematical functions for complex numbers

This Page

Report a Bug

Show Source

The following functions are provided by this module ... 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*.

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

```
# my_module.py

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

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

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

```
>>> terminal time >>>
```

my_module.py

What's in the module

```
# 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

my_module.py — C:\Users\Daisy\OneDrive\cs1110sp20\lectures\03-fns_modules — Atom

File Edit View Selection Find Packages Help

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
```

Windows PowerShell

```
PS C:\Users\Daisy\OneDrive\cs1110sp20\lectures\03-fns_modules> python
Python 3.7.4 (default, Aug  9 2019, 18:34:13) [MSC v.1915 64 bit (AMD64)] :: An
Type "help", "copyright", "credits" or "license" for more information.
>>> import my_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 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 Shell

```
>>> import my_module
```

my_module

x ~~3~~ 9

variable x stays "within" the module

Clicker Question!

Module Text

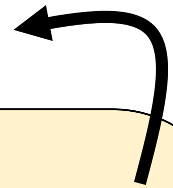
```
# fah2cel.py

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

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

Python Shell

```
>>> import fah2cel
```



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

(A) >>>

(B) 0.0
>>>

(C) an error message

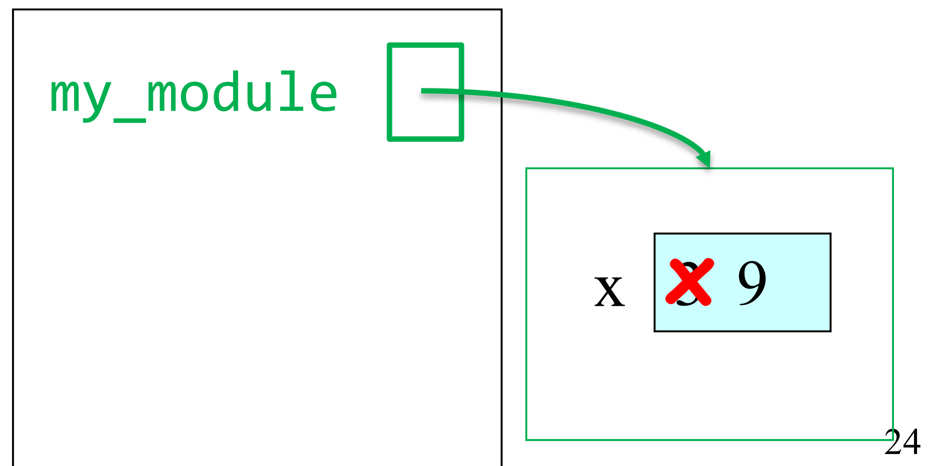
(D) The text of fah2cel.py

(E) Sorry, no clue.

Using a Module (my_module.py)

Module Text	Python Shell
# my_module.py	>>> import my_module
	>>> my_module.x
"""This is a simple module.	9
It shows how modules work"""	

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



You must import

Windows command line
(Mac looks different)

With import

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


Without import

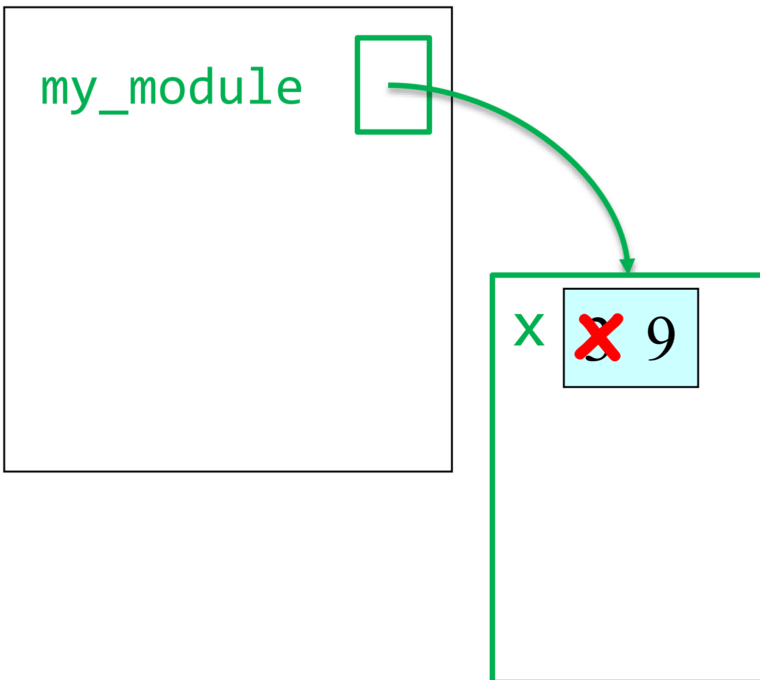
```
C:\> python
>>> math.ceil(3.14159)
Traceback (most recent call last):
  File "<stdin>", line 1, in
<module>
NameError: name 'math' is not
defined
```

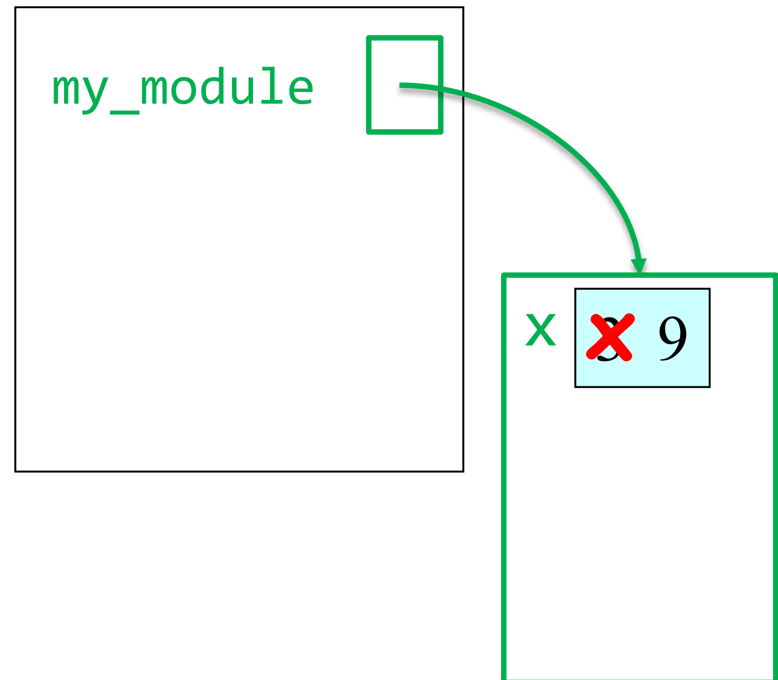
Python unaware of what “math” is

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 Shell

Windows PowerShell

```
>>> 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

FILE
```

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

pi 3.141592653589793

from command

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

  from <module> import *

- **Example**:

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

ceil()
sqrt()
e 2.718281828459045
pi 3.141592653589793
…

Module functions now behave like built-in functions

Dangers of Importing Everything

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

e was overwritten!

e 2.718281828459045

ceil()

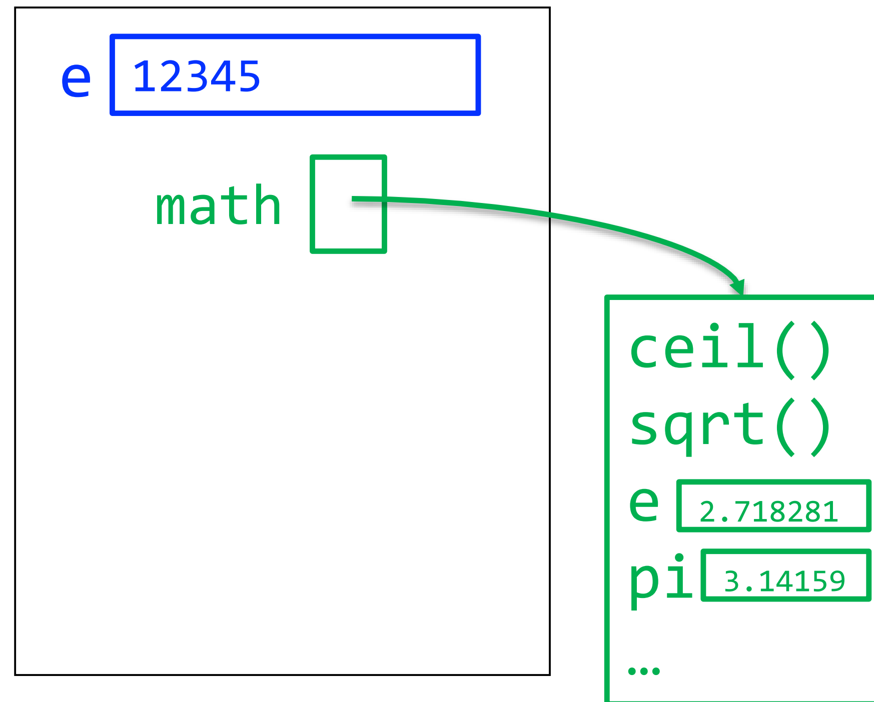
sqrt()

pi 3.141592653589793

…

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:\>

- 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	C:\> python module.py

```
# my_module.py

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

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

C:\> python module.py

Python does not execute (because of #)

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

Python executes this.

Python executes this.

x ~~3~~ 9

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
```

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

fah2cel.py

```
# fah2cel.py

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

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

Command Line

```
C:\> python fah2cel.py
C:\> fah2cel.c
```

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

(A) >>>
(B) 0.0
>>>
(C) an error message
(D) The text of fah2cel.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)
```

Syntax:
print (<expression>)

Running script.py as a script

Command Line	script.py

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


C:\>
```

```python
# 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	script.py
C:\> python	# script.py
>>> x = 1+2	
>>> x = 3*x	""" This is a simple script.
>>> x	It shows why we use print"""
9	
>>> print(x)	x = 1+2
9	x = 3*x
>>>	print(x)
	# note: in script mode, you will
	# not get output if you just type x

Modules vs. Scripts

Module	Script
• Provides functions, variables	• Behaves like an application
• import it into Python shell	• Run it from command line
⇨ Within Python shell you have access to the functions and variables of the imported module	⇨ After running the app you're back at the command line (not in Python shell)

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