



<http://www.cs.cornell.edu/courses/cs1110/2022sp>

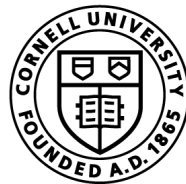
Lecture 3:

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]



<http://www.cs.cornell.edu/courses/cs1110/2022sp>

Lecture 3:

Functions & Modules

(Sections 3.1-3.3)

Have these ready for today's lecture:

- Materials for note taking (e.g., lecture slides on paper or pdf)*
- Terminal or Powershell, ready to start Python*
- Atom Editor*

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



Lecture Afterthoughts

There were many questions about the math module function **ceil()**. Since it's not a commonly known function, we've replaced it with **sqrt()**, which takes the square root of a number (Recall: $\sqrt{9} = 3$ because 3^2 (or 3×3) = 9).

We hope this is a better example function!

This is not a math class! We just needed an example to work with. 😊

Announcements

Can't see the zoom polls?

- Using zoom in web browser? It doesn't show polls
- It's okay! Please still work out the answer.
- We aren't counting participation, so no credit is lost!

Lab Deadlines

- Lab 1: Feb 2
 - Lab 2: Feb 3
 - Lab 3: Feb 4
 - Lab 4: Feb 7
- extensions*
- New labs this week!*

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  
>>>
```

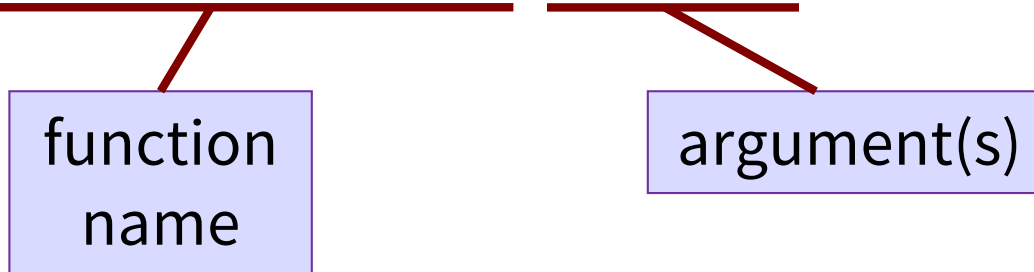
Ahh, much better.

All is right in the world.

Function Calls

- Function expressions have the form:

best_function_ever(x, y, ...)



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

Visualize the execution!

A diagram illustrating the state of variables after execution. It consists of four rows, each with a variable name and a value inside a blue-outlined box. The first row shows 'x' followed by a box containing '5'. The second row shows 'y' followed by a box containing '4'. The third row shows 'bigger' followed by a box containing '5'. The fourth row shows 'a' followed by a box containing '3'.

Play-by-Play of Python interactive mode

```
>>> x = 5
>>> y = 4
>>> bigger = max(x, y)
>>> bigger
5
>>> a = round(3.14159265)
>>> a
3
```

Executes 3 assignment statements

Evaluates the expression and reports

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 space-saving 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

What Python can access directly

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

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

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
>>>
```

What Python can access directly

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

```
x 7
```

```
math
```

```
sqrt()
log()
```

```
e 2.718281
```

```
pi 3.14159
```

```
...
```

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
```

What Python can access directly

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

```
x 7 3.14159
```

```
math
```

```
sqrt()
log()
```

```
e 2.718281
```

```
pi 3.14159
```

```
...
```



module help

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

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

```
Terminal — less · python — 80x24
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>

The screenshot shows a web browser displaying the Python documentation for the `math` module. The browser's address bar shows the URL `docs.python.org/3.7/library/math.html`. The page header includes navigation links for Python version (3.7.6), language (English), and documentation structure (Documentation » The Python Standard Library » Numeric and Mathematical Modules »). A search box is also present.

The main content area is titled `math` — Mathematical functions. It begins with a paragraph stating that this module provides access to mathematical functions defined by the C standard. It then explains that these functions cannot be used with complex numbers and that the `cmath` module should be used instead for complex numbers. A paragraph follows, stating that the following functions are provided by this module, except when explicitly noted otherwise, all return values are floats.

The page is divided into sections, with the first being "Number-theoretic and representation functions". Under this section, three functions are listed:

- `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`.

The left sidebar contains a "Table of Contents" for the `math` module, listing categories such as "Number-theoretic and representation functions", "Power and logarithmic functions", "Trigonometric functions", "Angular conversion", "Hyperbolic functions", "Special functions", and "Constants". It also includes links for "Previous topic" (`numbers` — Numeric abstract base classes) and "Next topic" (`cmath` — Mathematical functions for complex numbers). At the bottom of the sidebar, there are links to "Report a Bug" and "Show Source".



A Closer Reading of the Documentation

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

The image shows a screenshot of the Python documentation for the `math` module. The page title is "math — Mathematical functions". The main content area shows the function `math.sqrt(x)` with the description "Return the square root of x." Four callout boxes are overlaid on the page:

- Function name**: Points to `math.sqrt(x)`.
- Possible arguments**: Points to `x` in `math.sqrt(x)`.
- Module**: Points to `math` in `math.sqrt(x)`.
- What the function evaluates to**: Points to the description "Return the square root of x."

Other visible text on the page includes "Table of Contents", "math — Mathematical functions", "Number-theoretic and representation functions", "This module provides access to", "the `cmath` module which support complex numbers as much mathematics as possible. The complex result allows earlier programmers can determine how to use the module. Otherwise, all return values are floats.", "Previous", "numbers", "base class", "Next", "cmath", "function", "This Page", "Report a Bug", "Show Source", "math.ceil(x)", "Return the ceiling of x.", "x.__ceil__()", "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

We'll use these many of these this semester.

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 x
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
```

We recommend Atom...
...but any editor will work

Typing in Interactive Mode vs. Writing a Module

Python Interactive Mode

```
wmwhite — python — 52x25
[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 x
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
 - 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

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



What does the docstring do?

Module Text File

```
# my_module.py

"""This is a simple module
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
```

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

built-in fns...

my_module

x ~~3~~ 9



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
```

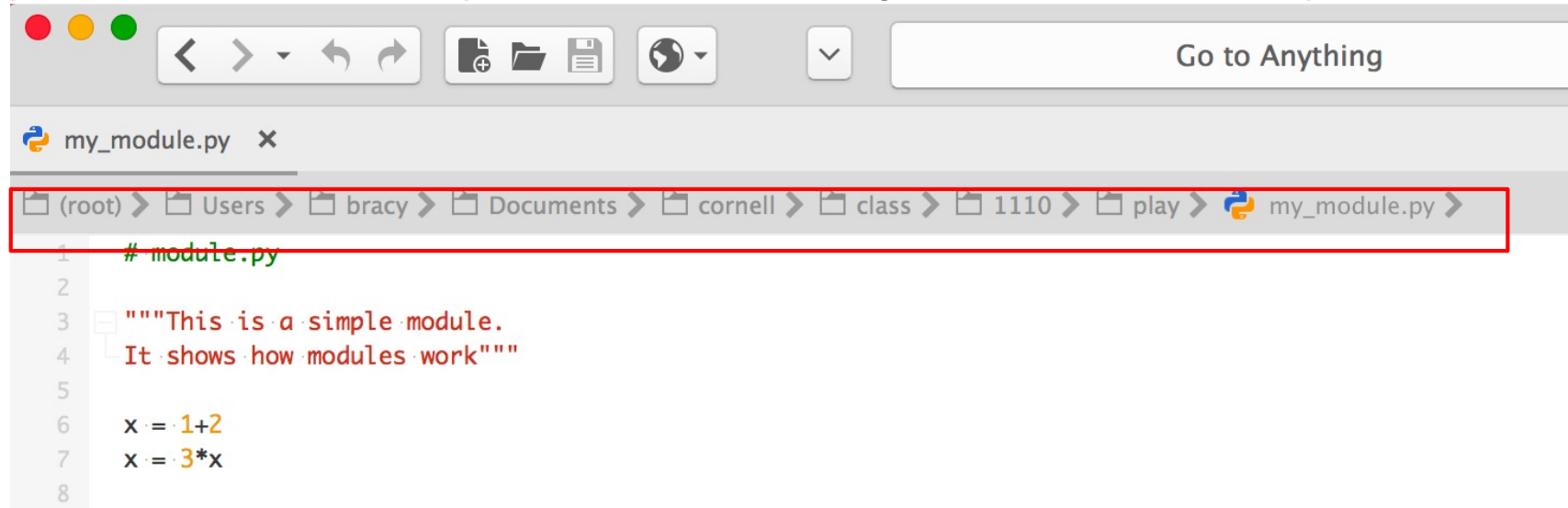
After you hit "Return" here
what will python print next?

- (A) >>>
- (B) 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"

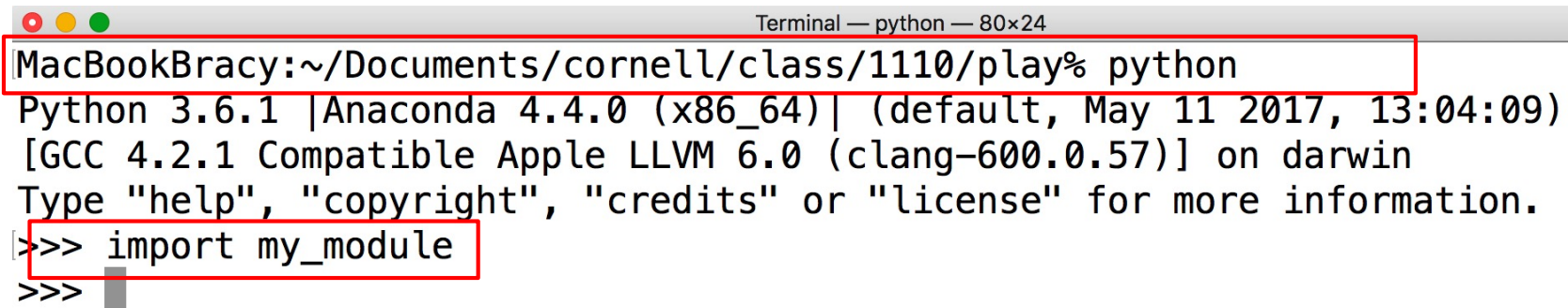
If `my_module.py` in directory/folder **play**:



The screenshot shows a code editor window with the file `my_module.py` open. The breadcrumb navigation path is highlighted with a red box: `(root) > Users > bracy > Documents > cornell > class > 1110 > play > my_module.py`. The code content is as follows:

```
1 # 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
```

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



The screenshot shows a terminal window with the following output:

```
Terminal — python — 80x24
MacBookBracy:~/Documents/cornell/class/1110/play% 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.
>>> import my_module
>>>
```

Rule #2: You must **import**

Windows command line
(Mac looks different)

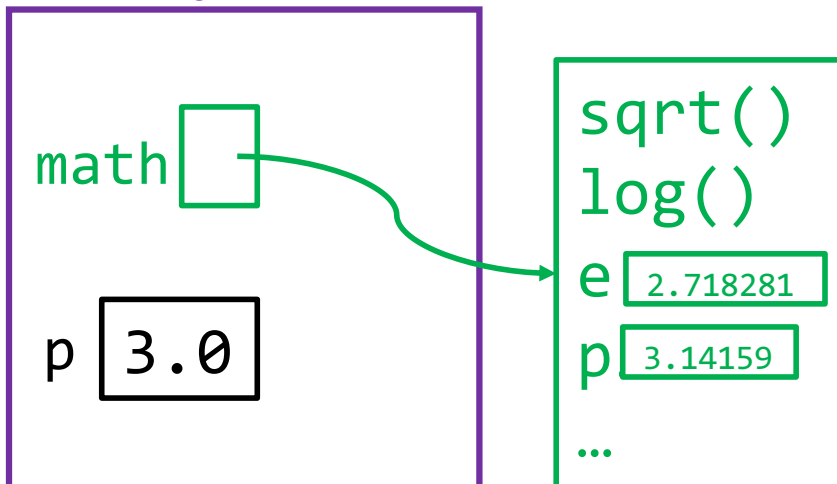
With import

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

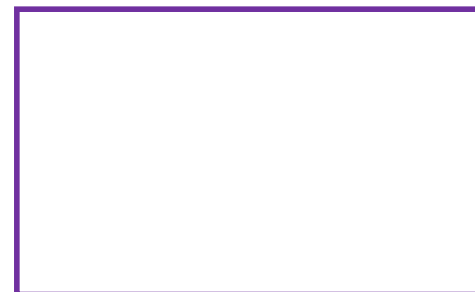
Without import

```
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



What Python can access directly

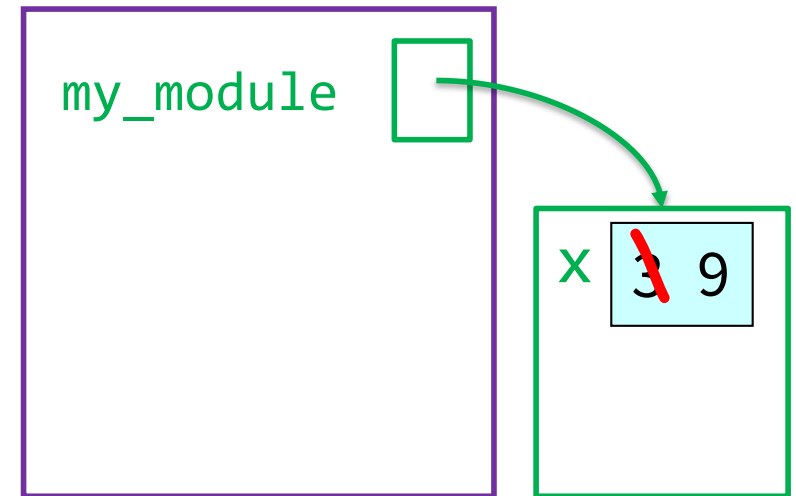


*Python
unaware of
what "math" is*

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)*

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
```

*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
```

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
```

e was overwritten!

What Python can access directly

```
e 12345 2.71828182845
sqrt()
log()
pi 3.141592653589793
...
```

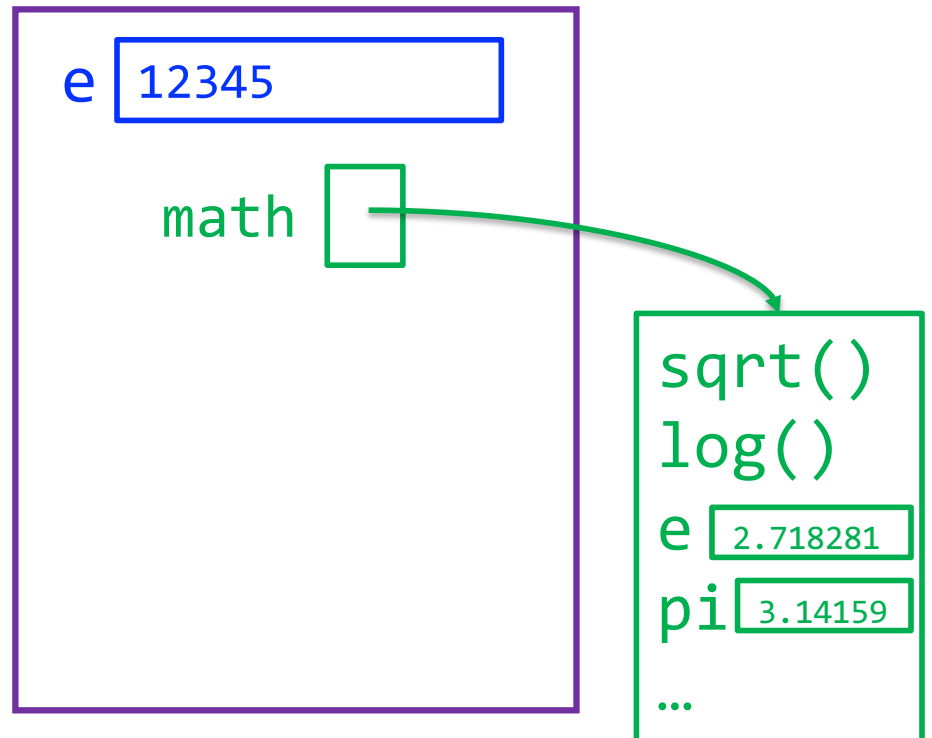
*Do you know every mathematical constant?
Might not want to import them all.*

Avoiding `from` keeps variables separate

Example:

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

What Python can access 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)

This is as far as we got in the Lecture on 2/1/22.

The following slides will be useful for and covered in this week's lab.

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"

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

x ~~3~~ 9

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**

Python Interactive Mode

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

Executes 2 assignment statements

Evaluates the expression and reports

No output!

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:\>
```

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

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