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

- ***New*** seat assignments for all students at in-person sections. See CMS "Seat Assignments – week 2".
- Textbook: We deactivated instant access ($$$) for CS1110; use the ***free*** online version from the course homepage
- Do pre-lecture activities (reading/videos) *before* each lecture
- ***INFO 1998 Intro to Machine Learning*** (ML), 1 cr, Feb 24 – May 5, led by Cornell Data Science (DS) undergrads. Python+DS+ML. More info and register at **tiny.cc/info1998_sp21**
- Zoom: please use the raise hand tool to indicate that you want to ask a question. Lower hand afterwards.

Function Calls

- Function expressions have the form:

`fun(x,y,…)`

function name

arguments separated by commas

- 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

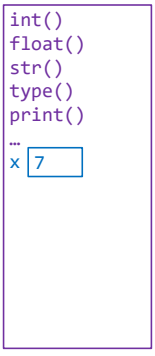
- So far just built-ins

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

Visualizing functions & variables available

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

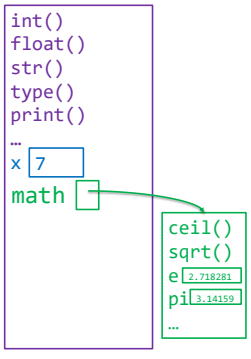
```
C:\> python
>>> x = 7
>>>
```



Visualizing functions & variables available

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

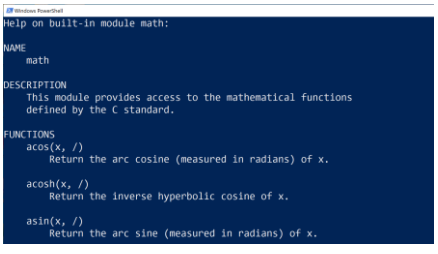
```
C:\> python
>>> x = 7
>>> import math
>>>
```



module help

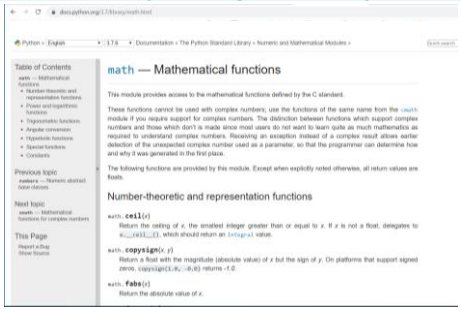
After importing a module, can see what functions and variables are available:

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



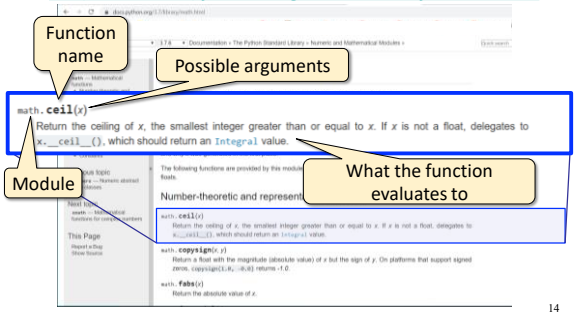
Reading the Python Documentation

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



Reading the Python Documentation

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



Other Useful Modules

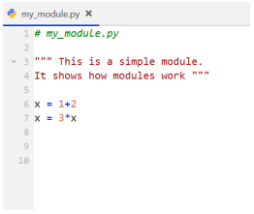
- io
  - Read/write from files
- random
  - Generate random numbers
  - Can pick any distribution
- string
  - Useful string functions
- sys
  - Information about your OS

Making your Own Module

Write in a text editor

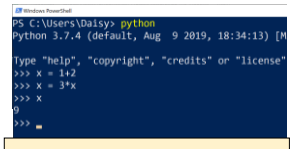
We recommend Atom…

…but any editor will work



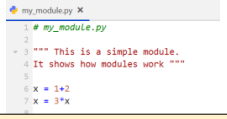
Interactive Shell vs. Modules

Python Interactive Shell



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

Module



- 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
```

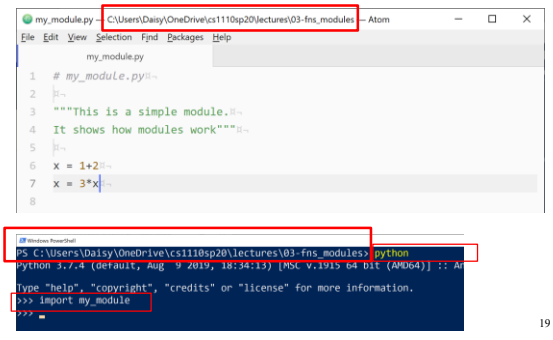
Single line comment (not executed)

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



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 #)

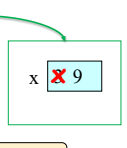
Python does not execute (because of """ and """)

Python executes this.

Python executes this.

Python Shell

```
>>> import my_module
```



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

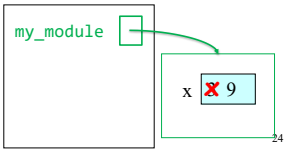
```
# my_module.py

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

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

Python Shell

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

variable we want to access

module name

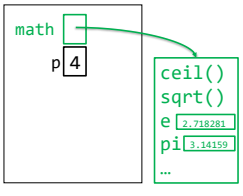


You must import

Windows command line (Mac looks different)

With import

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



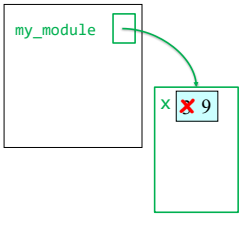
Without import

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

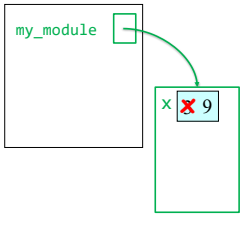
Python unaware of what "math" is

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 Shell

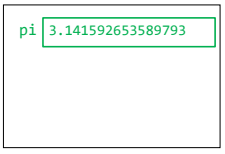
```
>>> import my_module
>>> help(my_module)
Help on module my_module:

NAME
    my_module

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

DATA
    x = 9

FILE
```

from command

- You can also import like this:
  from <module> import <function name>
- **Example**:

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

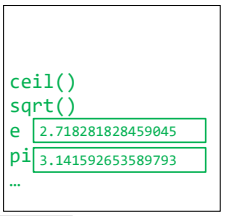
no longer need the module name



from command

- You can also import *everything* from a module:
  from <module> import *
- **Example**:

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

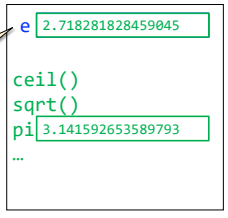


Module functions now behave like built-in functions

Dangers of Importing Everything

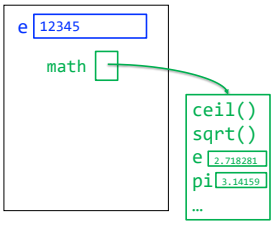
```
>>> e = 12345
>>> from math import *
>>> e
2.718281828459045
```

e was overwritten!



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

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

x ~~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
```

Only difference

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

```
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	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.