Lecture 3:
Functions & Modules
(Sections 3.1-3.3)

CS 1110
Introduction to Computing Using Python

http://www.cs.cornell.edu/courses/cs1110/2018sp

[E. Andersen, A. Bracy, D. Gries, L. Lee, S. Marschner, C. Van Loan, W. White]
CS1110 Spring 2018 Announcements

Check course page for announcements!

http://www.cs.cornell.edu/courses/cs1110/2018sp

ENGRG 1010. AEW workshops still space

- can enroll through Student Center
- 1-credit S/U course
- 2-hour weekly workshop
- work on related problem sets

Full? Or need a different time?

https://tinyurl.com/aew-request
Things to Do Before Next Class

- Read Sections 3.4-3.11

- If you haven’t already:
  - install Anaconda Python & Komodo on your machine
  - **play** around with python a bit!
Function Calls

• Python supports expressions with math-like functions
  ▪ A function in an expression is a function call
  ▪ Will explain the meaning of this later
• Function expressions have the form:
  \[ \text{fun}(x,y,...) \]

  - function name
  - argument

• Some math functions built into Python:
  ▪ \text{round}(2.34)
  ▪ \text{max}(a+3,24)

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()

• Longer list:
  http://docs.python.org/3/library/functions.html

Arguments go in (), but name() refers to function in general
The Lack of Built-in Functions

- Python contains few built-in functions
- Missing many functions you would expect
  - *Example: cos(), sqrt()*
- Many more functions are available through built-in *modules*
Modules

- “Libraries” of functions and variables
- To access a module, use the import command:
  ```python
  import <module name>
  ```
- Can then access functions like this:
  ```python
  <module name>.<function name>(<arguments>)
  ```
- **Example:**
  ```python
  >>> import math
  >>> math.cos(2.0)
  -0.4161468365471424
  ```
# Necessity of `import`

<table>
<thead>
<tr>
<th>With import</th>
<th>Without import</th>
</tr>
</thead>
<tbody>
<tr>
<td>C:&gt; python</td>
<td>C:&gt; python</td>
</tr>
<tr>
<td>This is the Windows command line (Mac looks different)</td>
<td>Python is unaware of what “math” is</td>
</tr>
<tr>
<td>&gt;&gt;&gt; import math</td>
<td>&gt;&gt;&gt; math.cos(2.0)</td>
</tr>
</tbody>
</table>
| >>> math.cos(2.0) | Traceback (most recent call last):
| -0.4161468365471424 | File "<stdin>", line 1, in <module>
| | NameError: name 'math' is not defined |
Module Variables

• Modules can have variables, too
• Can access them like this:
  \(<\text{module name}>.\text{<variable name>}>\)

• Example:
  
  >>> import math
  >>> math.pi

  3.141592653589793
Visualizing functions & variables

• So far just built-ins

> python
>>>
Visualizing functions & variables

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

> python
>>> x = 7

int()
float()
str()
type()
print()
...
x 7
Visualizing functions & variables

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

> python
>>> x = 7
>>> import math
>>>
After importing a module, see what functions and variables are available:

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

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
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 use Komodo Edit…

…but any editor will work

```python
# 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

- 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
my_module.py

Module Text

```python
# 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
```

19
Modules Must be in Working Directory!

Must run `python` from same folder as the module

```python
# module.py

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

x = 1+2
x = 3*x

>>> import my_module
>>>
Using a Module (my_module.py)

<table>
<thead>
<tr>
<th>Module Text</th>
<th>Python Command Shell</th>
</tr>
</thead>
<tbody>
<tr>
<td># my_module.py</td>
<td>&gt;&gt;&gt;</td>
</tr>
</tbody>
</table>
| """This is a simple module.
It shows how modules work"""" | |
| x = 1+2 | |
| x = 3*x | |

...
Using a Module (my_module.py)

<table>
<thead>
<tr>
<th>Module Text</th>
<th>Python Command Shell</th>
</tr>
</thead>
<tbody>
<tr>
<td># my_module.py</td>
<td>&gt;&gt;&gt; import my_module</td>
</tr>
<tr>
<td>&quot;&quot;&quot;This is a simple module.</td>
<td></td>
</tr>
<tr>
<td>It shows how modules work&quot;&quot;&quot;&quot;</td>
<td></td>
</tr>
<tr>
<td>x = 1+2</td>
<td></td>
</tr>
<tr>
<td>x = 3*x</td>
<td></td>
</tr>
</tbody>
</table>

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

Module Text

```python
# my_module.py

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

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

Python Command Shell

```python
>>> import module
Python does not execute (because of #)

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

Python executes this.

Python executes this.

variable x stays “within” the module
```
Clicker Question!

Module Text

```python
# my_module.py

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

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

Python Command Shell

```python
>>> 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.
```
# my_module.py

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

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

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)

<table>
<thead>
<tr>
<th>Module Text</th>
<th>Python Command Shell</th>
</tr>
</thead>
<tbody>
<tr>
<td># my_module.py</td>
<td></td>
</tr>
<tr>
<td>&quot;&quot;&quot;This is a simple module. It shows how modules work&quot;&quot;&quot;</td>
<td></td>
</tr>
<tr>
<td>x = 1+2</td>
<td></td>
</tr>
<tr>
<td>x = 3*x</td>
<td></td>
</tr>
</tbody>
</table>

>>> import my_module
>>> my_module.x

The variable we want to access

module name
Using a Module (my_module.py)

<table>
<thead>
<tr>
<th>Module Text</th>
<th>Python Command Shell</th>
</tr>
</thead>
<tbody>
<tr>
<td># my_module.py</td>
<td>&gt;&gt;&gt; import my_module</td>
</tr>
<tr>
<td>&quot;&quot;&quot;This is a simple module.</td>
<td>&gt;&gt;&gt; my_module.x</td>
</tr>
<tr>
<td>It shows how modules work&quot;&quot;&quot;</td>
<td>9</td>
</tr>
<tr>
<td>x = 1+2</td>
<td></td>
</tr>
<tr>
<td>x = 3*x</td>
<td></td>
</tr>
</tbody>
</table>
You Must import

C:\> python
>>> import my_module

>>> my_module.x
9

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

```python
# 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

Note that you don’t need the module name now
**from command**

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

- **Example:**
  ```python
  >>> from math import *
  >>> pi
  3.141592653589793
  >>> cos(pi)
  -1.0
  ```

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

```python
>>> 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

Command Line

C:\> python module.py

Python does not execute (because of #)

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

Python executes this.

x = 1 + 2

Python executes this.

x = 3 * x

x = 9
# my_module.py

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

x = 1+2
x = 3*x

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
<table>
<thead>
<tr>
<th>my_module.py</th>
<th>Command Line</th>
</tr>
</thead>
<tbody>
<tr>
<td># my_module.py</td>
<td>C:&gt; python my_module.py</td>
</tr>
<tr>
<td>&quot;&quot;&quot;This is a simple my_module.</td>
<td>C:&gt; my_module.x</td>
</tr>
<tr>
<td>It shows how modules work&quot;&quot;&quot;&quot;</td>
<td></td>
</tr>
<tr>
<td>x = 1+2</td>
<td></td>
</tr>
<tr>
<td>x = $3^x$</td>
<td></td>
</tr>
</tbody>
</table>

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.
my_module.py

```python
# my_module.py

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

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

Command Line

```bash
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
<table>
<thead>
<tr>
<th>my_module.py</th>
<th>script.py</th>
</tr>
</thead>
<tbody>
<tr>
<td># my_module.py</td>
<td># script.py</td>
</tr>
<tr>
<td>&quot;&quot;&quot; This is a simple module.</td>
<td>&quot;&quot;&quot; This is a simple script.</td>
</tr>
<tr>
<td>It shows how modules work&quot;&quot;&quot;&quot;</td>
<td>It shows why we use print&quot;&quot;&quot;&quot;</td>
</tr>
<tr>
<td>x = 1+2</td>
<td>x = 1+2</td>
</tr>
<tr>
<td>x = 3*x</td>
<td>x = 3*x</td>
</tr>
<tr>
<td>Only difference</td>
<td><strong>print</strong>(x)</td>
</tr>
</tbody>
</table>
# Running script.py as a script

<table>
<thead>
<tr>
<th>Command Line</th>
<th>script.py</th>
</tr>
</thead>
<tbody>
<tr>
<td>C:&gt; python script.py</td>
<td># script.py</td>
</tr>
<tr>
<td>9</td>
<td>&quot;&quot;&quot; This is a simple script.</td>
</tr>
<tr>
<td></td>
<td>It shows why we use print&quot;&quot;&quot;</td>
</tr>
<tr>
<td>C:&gt;</td>
<td>x = 1+2</td>
</tr>
<tr>
<td></td>
<td>x = 3*x</td>
</tr>
<tr>
<td></td>
<td>print(x)</td>
</tr>
</tbody>
</table>
## Subtle difference about script mode

<table>
<thead>
<tr>
<th>Interactive mode</th>
<th>script.py</th>
</tr>
</thead>
<tbody>
<tr>
<td>C:&gt; python</td>
<td># script.py</td>
</tr>
<tr>
<td>&gt;&gt;&gt;&gt; x = 1+2</td>
<td>&quot;&quot;&quot; This is a simple script.</td>
</tr>
<tr>
<td>&gt;&gt;&gt;&gt; x = 3*x</td>
<td>It shows why we use print&quot;&quot;&quot;</td>
</tr>
<tr>
<td>&gt;&gt;&gt;&gt; x</td>
<td>x = 1+2</td>
</tr>
<tr>
<td>9</td>
<td>x = 3*x</td>
</tr>
<tr>
<td>&gt;&gt;&gt;&gt; print(x)</td>
<td>print(x)</td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>&gt;&gt;&gt;&gt;</td>
<td># note: in script mode, you will</td>
</tr>
<tr>
<td></td>
<td># not get output if you just type x</td>
</tr>
</tbody>
</table>
# Modules vs. Scripts

<table>
<thead>
<tr>
<th>Module</th>
<th>Script</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Provides functions, variables</td>
<td>• Behaves like an application</td>
</tr>
<tr>
<td>• import it into Python shell</td>
<td>• Run it from command line</td>
</tr>
</tbody>
</table>

Files look the same. Difference is how you use them.
• Today we created a module with a \textit{variable}
• Have not discussed how to make a \textit{function}
• \textbf{Example}:
  
  >>> import math
  >>> math.cos(2.0)
  -0.4161468365471424
  
  we want to make functions like this