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

Lecture 3: Functions & Modules

(Sections 3.1-3.3)

CS 1110
Introduction to Computing Using Python

[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:

fun(x,y,...)

function name

argument

- Some math functions built into Python:
  - `round(2.34)`
  - `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()`

Arguments go in (), but **name()** refers to function in general

- Longer list:

http://docs.python.org/3/library/functions.html

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:

  `import` *<module name>*

- Can then access functions like this:

  <module name>.*<function name>*(*<arguments>*)

- **Example**:

```
>>> import math
>>> math.cos(2.0)
-0.4161468365471424
```

Necessity of import

With import

C:\> python

This is the Windows command line (Mac looks different)

```
>>> import math
>>> math.cos(2.0)
-0.4161468365471424
```

Without import

C:\> python

Python is unaware of what “math” is

```
>>> math.cos(2.0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'math' is not defined
```

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

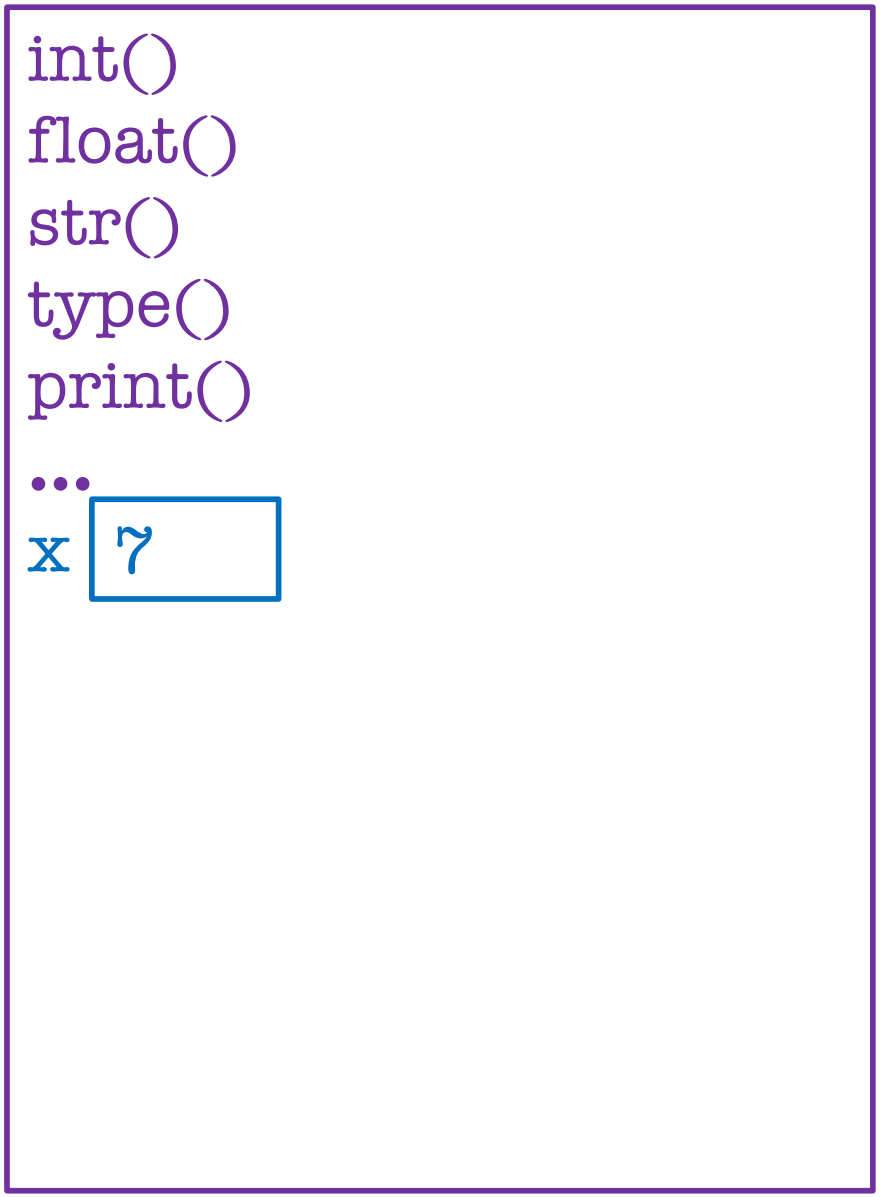
```
> python
>>>
```

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

Visualizing functions & variables

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

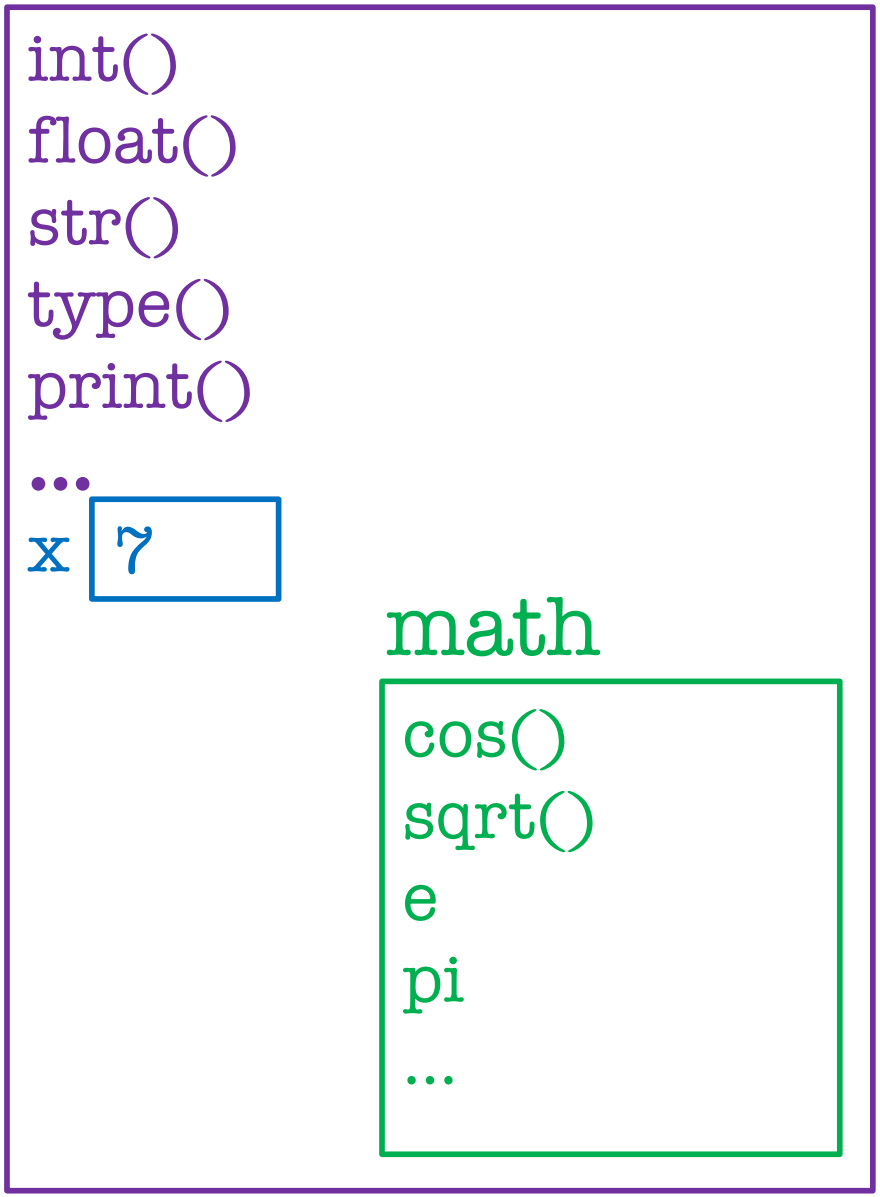
```
> python
>>> x = 7
>>>
```



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
>>>
```



module help

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

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

```
Terminal — less • python — 80×24
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

Python Software Foundation docs.python.org/3/library/math.html

9.2. math — Mathematical functions — Python 3.6.4 documentation

Python » English 3.6.4 Documentation » The Python Standard Library » 9. Numeric and Mathematical Modules » Quick search Go | previous | next | modules | index

Table Of Contents

Previous topic

9.1. numbers — Numeric abstract base classes

Next topic

9.3. cmath — Mathematical functions for complex numbers

This Page

Report a Bug
Show Source

9.2. math — Mathematical functions

This module is always available. It 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.

9.2.1. 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*.

math.**factorial**(*x*)
Return *x* factorial. Raises `ValueError` if *x* is not integral or is negative.

A Closer Reading of the Python Documentation

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

Function name

Possible arguments

Module

What the function evaluates to

Python Software Foundation docs.python.org/3/library/math.html

9.2. math — Mathematical functions — Python 3.6.4 documentation

Documentation » The Python Standard Library » 9. Numeric and Mathematical Modules » Quick search Go | previous | next | modules | index

Table Contents

- 9.2.4. Conversion
- 9.2.5. Hyperbolic functions
- 9.2.6. Special functions
- 9.2.7. Constants

Previous topic

9.1. numbers — Numeric abstract base classes

Next topic

9.3. cmath — Mathematical functions for complex numbers

This Page

Report a Bug
Show Source

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.

9.2.1. 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*.

math.**factorial**(*x*)

Return *x* factorial. Raises `ValueError` if *x* is not integral or is negative.

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

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

```
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

```
# my_module.py

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

x = 1+2
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

"""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

Go to Anything

my_module.py

(root) > Users > bracy > Documents > cornell > class > 1110 > play > my_module.py

```
# module.py

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

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

Terminal — python — 80×24

```
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
>>>
```

Using a Module (my_module.py)

Module Text	Python Command Shell
# my_module.py	>>>
"""This is a simple module. It shows how modules work""" x = 1+2 x = 3*x	

Using a Module (my_module.py)

Module Text	Python Command Shell
# my_module.py	>>> import my_module

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

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

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

On import….

Module Text	Python Command Shell

```
# 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.

x ~~3~~ 9

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
```

module name

The variable we want to access

Using a Module (my_module.py)

Module Text	Python Command 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	

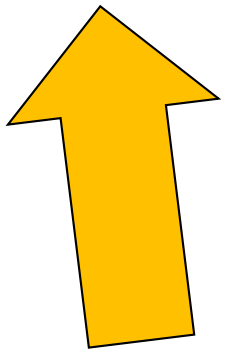
You Must import

```
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

```
# 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:

  `from <module> import *`

- **Example**:

```
>>> 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

```
>>> 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	Command Line

```
# 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.

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)
```

Only difference

Running script.py as a script

Command Line	script.py
C:\> python script.py 9 C:\>	# 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	Script
• Provides functions, variables • `import` it into Python shell	• Behaves like an application • Run it from command line

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

Next Time: Defining Functions

- Today we created a module with a *variable*
- Have not discussed how to make a *function*
- **Example**:

```
>>> import math
>>> math.cos(2.0)
-0.4161468365471424
```

we want to make functions like this