

CS 110:

Introduction to Computing Using Python

Lecture 3

Functions & Modules

[Andersen, Gries, Lee, Marschner, Van Loan, White]

Announcements

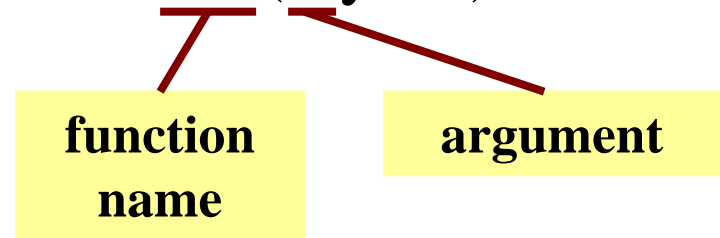
- Be checking course page for announcements
 - <http://www.cs.cornell.edu/courses/cs1110/2017sp/>
- AEW workshops still have space
 - ENGRG 1010
 - can enroll through Student Center
 - 1-credit S/U course
 - 2-hour weekly workshop
 - work on problem sets related to the week's content

Things to Do Before Next Class

- Read the rest of Chapter 3
 - Can skip 3.10
- You should be using the *First Edition* of the textbook
 - Second edition is for Python 3

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,...)



- **Examples** (math functions that work in 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/2/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)

(startup text omitted)

```
>>> import math
```

```
>>> math.cos(2.0)
```

```
-0.4161468365471424
```

Without import

C:\> python

Python is unaware of
what "math" is

(startup text omitted)

```
>>> 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
```

module help

- After importing a module, can see what functions and variables are available with:
`help(<module name>)`

```
>>> import math
>>> help(math)
Help on built-in module math:

NAME
    math

FILE
    (built-in)

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.

    acosh(...)
        acosh(x)

        Return the inverse hyperbolic cosine of x.

    asin(...)
        asin(x)

        Return the arc sine (measured in radians) of x.
```

Reading the Python Documentation

<https://docs.python.org/2/library/math.html>

The screenshot shows the Python documentation page for the `math` module. The page is titled "9.2. math — Mathematical functions" and is part of the "9. Numeric and Mathematical Modules" section. The left sidebar contains a "Table of Contents" with links to various sub-sections, a "Previous topic" link to "9.1. numbers — Numeric abstract base classes", and a "Next topic" link to "9.3. cmath — Mathematical functions for complex numbers". The main content area includes a "This Page" section with links to "Report a Bug" and "Show Source", and a "Quick search" section with a search box and a "Go" button. The main text describes the `math` module, stating it is always available and provides access to mathematical functions defined by the C standard. It notes that these functions cannot be used with complex numbers and that 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. The page lists several functions: `ceil(x)`, `copysign(x, y)`, `fabs(x)`, `factorial(x)`, `floor(x)`, `fmod(x, y)`, and `frexp(x)`. Each function is described with its purpose and any relevant details, such as raising `ValueError` or returning a float.

Python » 2.7.13 » Documentation » The Python Standard Library » 9. Numeric and Mathematical Modules » [previous](#) | [next](#) | [modules](#) | [index](#)

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` as a float, the smallest integer value greater than or equal to `x`.

math.copysign(x, y)
Return `x` with the sign of `y`. On a platform that supports signed zeros, `copysign(1.0, -0.0)` returns `-1.0`.

New in version 2.6.

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.

New in version 2.6.

math.floor(x)
Return the floor of `x` as a float, the largest integer value less than or equal to `x`.

math.fmod(x, y)
Return `fmod(x, y)`, as defined by the platform C library. Note that the Python expression `x % y` may not return the same result. The intent of the C standard is that `fmod(x, y)` be exactly (mathematically; to infinite precision) equal to `x - n*y` for some integer `n` such that the result has the same sign as `x` and magnitude less than `abs(y)`. Python's `x % y` returns a result with the sign of `y` instead, and may not be exactly computable for float arguments. For example, `fmod(-1e-100, 1e100)` is `-1e-100`, but the result of Python's `-1e-100 % 1e100` is `1e100-1e-100`, which cannot be represented exactly as a float, and rounds to the surprising `1e100`. For this reason, function `fmod()` is generally preferred when working with floats, while Python's `x % y` is preferred when working with integers.

math.frexp(x)
Return the mantissa and exponent of `x` as the pair `(m, e)`. `m` is a float and `e` is an integer such that `x == m * 2**e` exactly. If `x` is zero, returns `(0.0, 0)`, otherwise `0.5 <= abs(m) < 1`. This is used to "pick apart" the internal representation of a float in a portable way.

Reading the Python Documentation

<https://docs.python.org/2/library/math.html>

The screenshot shows the Python documentation for the `math` module. A callout labeled "Function name" points to `math.ceil(x)`. A callout labeled "Possible arguments" points to the parameter `x` in the function signature. A callout labeled "Module" points to the `math` module name in the page header. A callout labeled "What the function evaluates to" points to the description of the `math.fmod(x, y)` function.

Python » 2.7.13 » Documentation » The Python Standard Library » 9. Numeric and Mathematical Modules » previous | next | modules | index

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` as a float, the smallest integer value greater than or equal to `x`.

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

math.fmod(x, y)
Return `fmod(x, y)`, as defined by the platform C library. Note that the Python expression `x % y` may not return the same result. The intent of the C standard is that `fmod(x, y)` be exactly (mathematically; to infinite precision) equal to `x - n*y` for some integer `n` such that the result has the same sign as `x` and magnitude less than `abs(y)`. Python's `x % y` returns a result with the sign of `y` instead, and may not be exactly computable for float arguments. For example, `fmod(-1e-100, 1e100)` is `-1e-100`, but the result of Python's `-1e-100 % 1e100` is `1e100-1e-100`, which cannot be represented exactly as a float, and rounds to the surprising `1e100`. For this reason, function `fmod()` is generally preferred when working with floats, while Python's `x % y` is preferred when working with integers.

math.frexp(x)
Return the mantissa and exponent of `x` as the pair `(m, e)`. `m` is a float and `e` is an integer such that `x == m * 2**e` exactly. If `x` is zero, returns `(0.0, 0)`, otherwise `0.5 <= abs(m) < 1`. This is used to "pick apart" the internal representation of a float in a portable way.

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 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
```

Interactive Shell vs. Modules

Python Interactive Shell

```
C:\>python
Python 2.7.12 |Anaconda 4.1.1 (64-bit)| (default, Jun 29 2016, 11:07:13) [MSC v.1500 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
Anaconda is brought to you by Continuum Analytics.
Please check out: http://continuum.io/thanks and https://anaconda.org
>>> x = 1+2
>>> x = 3*x
```

- Type python at command line
- Type commands after >>>
- 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
8
```

- Written in text editor
- Loaded through import
- Python executes statements when import is called

module.py

Module Text

```
# module.py
```

Single line comment
(not executed)

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

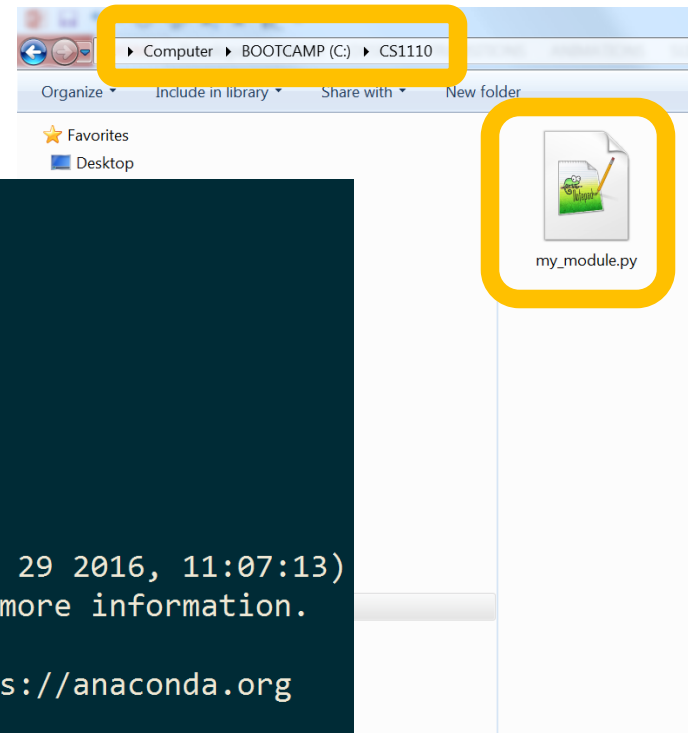
Docstring (note the Triple Quotes)
Acts as a multiple-line comment
Useful for *code documentation*

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

Commands
Executed on import

Modules Must be in Working Directory!

- Must run python from the same folder as the module



```
Directory of c:\CS1110
02/01/2017 10:14 PM <DIR> .
02/01/2017 10:14 PM <DIR> ..
02/01/2017 09:17 PM my_module.py
                   1 File(s)          130 bytes
                   2 Dir(s)  74,767,933,440 bytes free

c:\CS1110>python
Python 2.7.12 |Anaconda 4.1.1 (64-bit)| (default, Jun 29 2016, 11:07:13)
Type "help", "copyright", "credits" or "license" for more information.
Anaconda is brought to you by Continuum Analytics.
Please check out: http://continuum.io/thanks and https://anaconda.org
>>>
```

Using a Module (module.py)

Module Text

```
# module.py
```

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

```
x = 1+2
```

```
x = 3*x
```

Python Command Shell

```
>>>
```

Using a Module (module.py)

Module Text

```
# module.py
```

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

```
x = 1+2
```

```
x = 3*x
```

Python Command Shell

```
>>> import module
```

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

On import....

Module Text

Python Command Shell

```
# module.py
```

Python does not execute (because of #)

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

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

```
x = 1+2
```

Python executes this.

x 3

```
x = 3*x
```

Python executes this.

x ~~3~~ 9

This variable x stays “within” the module

Using a Module (module.py)

Module Text

```
# module.py
```

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

```
x = 1+2
```

```
x = 3*x
```

Python Command Shell

```
>>> import module
```

```
>>> module.x
```

module name

The variable we want to access

Using a Module (module.py)

Module Text

```
# module.py
```

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

```
x = 1+2
```

```
x = 3*x
```

Python Command Shell

```
>>> import module
```

```
>>> module.x
```

```
9
```

You Must import



```
C:\> python
```

```
>>> import module
```

```
>>> module.x
```

```
9
```

```
C:\> python
```

```
>>> module.x
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

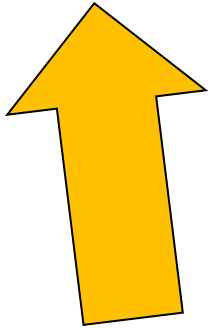
```
NameError: name 'module' is not  
defined
```

You Must Use the Module Name

```
>>> import module
```

```
>>> module.x
```

```
9
```



```
>>> import 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

```
# module.py
```

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

```
x = 1+2
```

```
x = 3*x
```

Python Command Shell

```
>>> import module  
>>> help(module)  
Help on module module:  
  
NAME  
    module  
  
FILE  
    c:\cs1110\module.py  
  
DESCRIPTION  
    This is a simple module.  
    It shows how modules work  
  
DATA  
    x = 9
```

from command

- You can also import like this:

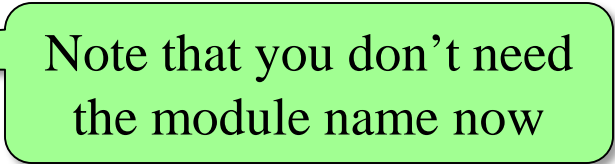
```
from <module> import <function name>
```

- **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:**

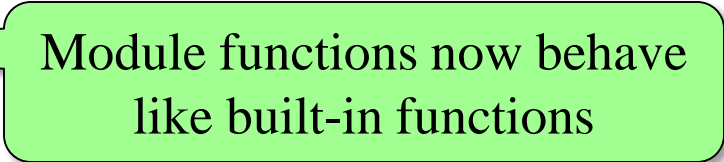
```
>>> from math import *
```

```
>>> pi
```

```
3.141592653589793
```

```
>>> cos(pi)
```

```
-1.0
```



Module functions now behave like built-in functions

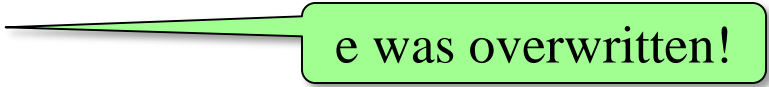
Dangers of Importing Everything

```
>>> e = 12345
```

```
>>> from math import *
```

```
>>> e
```

```
2.718281828459045
```



e was overwritten!

Avoiding from Keeps Variables Separate

```
>>> e = 12345
```

```
>>> import math
```

```
>>> math.e
```

```
2.718281828459045
```

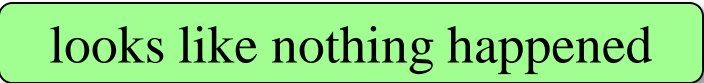
```
>>> e
```

```
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 module.py
C:\> 
- Actually, something did happen
 - Python executed all of module.py

Running module.py as a script

module.py

Command Line

```
# module.py
```

Python does not execute (because of #)

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

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

```
x = 1+2
```

Python executes this.

x 3

```
x = 3*x
```

Python executes this.

x ~~3~~ 9

Running module.py as a script

module.py

```
# module.py
```

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

```
x = 1+2
```

```
x = 3*x
```

Command Line

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

```
C:\>
```

when the script ends, all memory
used by module.py is deleted

thus, all variables get deleted
(including x)

so 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

module.py vs. script.py

module.py

```
# 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
```

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
```

Modules vs. Scripts

Module

- Provides functions, variables
- import it into Python shell

Script

- Behaves like an application
- Run it from command line

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