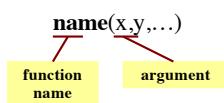


## Function Calls

- Python supports expressions with math-like functions
  - A function in an expression is a *function call*
- Function calls** have the form



- Arguments** are
  - Expressions**, not values
  - Separated by commas
- Examples:**
  - round(2.34)
  - max(a+3,24)

## Built-in Functions vs Modules

- The number of built-in functions is small
  - <http://docs.python.org/3/library/functions.html>
- Missing a lot of functions you would expect
  - Example:** cos(), sqrt()
- Module:** file that contains Python code
  - A way for Python to provide optional functions
  - To access a module, the import command
  - Access the functions using module as a *prefix*

## Example: Module math

```

>>> import math
To access math
functions
>>> math.cos(0)
Functions
require math
prefix!
1.0
>>> cos(0)
Module has
variables too!
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'cos' is not defined
>>> math.pi
3.141592653589793
>>> math.cos(math.pi)
-1.0
  
```

### Other Modules

- os**
  - Information about your OS
  - Cross-platform features
- random**
  - Generate random numbers
  - Can pick any distribution
- intros**
  - Custom module for the course
  - Will be used a lot at start

## Reading the Python Documentation

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, as the functions of the same name from the cmath module, if you require support for complex numbers, use the functions of the cmath module. Note that the module doesn't import these names even though they exist quite as much mathematics as required to understand complex numbers. Revising the execution instead of a complex result allows further development of the associated complex number used as a parameter, so that the programme can determine how and only it was generated in the first place.

**math.ceil(x)**

Return the ceiling of  $x$ , the smallest integer greater than or equal to  $x$ .

For floating point numbers, this is the same as math.floor(x). For integers and other objects which have the \_\_ceil\_\_ method, this is the result of applying the \_\_ceil\_\_ method. On platforms that support signed zeros, math.ceil(-0.0) returns -0.0.

**math.fabs(x)**

This function returns the absolute value of  $x$ . The result is a float, even if  $x$  is an integer or long integer.

**math.fmod(x, y)**

This function is equivalent to the C standard fmod(x, y), as defined by the platform C library. Note that the C standard is that fmod(0.0, 0.0) is undefined. Python's implementation follows this standard, returning a result with the sign of  $x$ , and may not be exactly comparable to other programs. For example, fmod(-1.0, 0.7) returns -0.10000000000000002, but the result of Python's -1.0%0.7 is -0.7, which cannot be represented as a float.

<http://docs.python.org/3/library/math.html>

## Interactive Shell vs. Modules

- Launch in command line
- Type each line separately
- Python executes as you type

- Write in a code editor**
  - We use Atom Editor
  - But anything will work
  - Load module with import

## Using a Module

### Module Contents

''' A simple module.

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

This file shows how modules work

'''

# This is a comment  
# This is a comment (not executed)

x = 1+2  
Commands  
Executed on import

x = 3\*x  
Not a command.  
import ignores this

## Using a Module

### Module Contents

""" A simple module.

This file shows how modules work  
"""

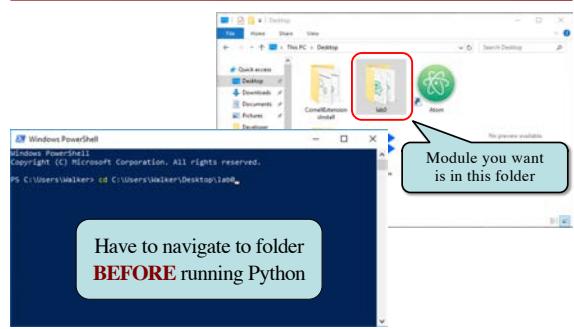
# This is  
x = 1+2  
x = 3\*x  
x

**Module data** must be  
prefixed by module name  
Prints **docstring** and  
module contents

### Python Shell

```
>>> import module
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
>>> >>> module.x
9
>>> help(module)
```

## Modules Must be in Working Directory!



## Modules vs. Scripts

### Module

- Provides functions, variables
  - Example:** temp.py
- import it into Python shell
 

```
>>> import temp
>>> temp.to_fahrenheit(100)
212.0
>>>
```

### Script

- Behaves like an application
  - Example:** helloApp.py
- Run it from command line:
 

```
python helloApp.py
```

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

## Scripts and Print Statements

### module.py

""" A simple module.

This file shows how modules work  
"""

```
# This is a comment
x = 1+2
x = 3*x
x
```

### script.py

""" A simple script.

This file shows why we use print  
"""

```
# This is a comment
x = 1+2
x = 3*x
print(x)
```

Only difference

## User Input

```
>>> input("Type something")
```

Type somethingabc

No space after the prompt.

```
>>> input("Type something: ")
```

Type something: abc

Proper space after prompt.

```
>>> x = input("Type something: ")
```

Type something: abc

Assign result to variable.

## Numeric Input

- input returns a string
  - Even if looks like int
  - It cannot know better

- You must convert values
  - int(), float(), bool(), etc.
  - Error if cannot convert

- One way to program
  - But it is a *bad* way
  - Cannot be automated

```
>>> x = input("Number: ")
```

Number: 3

```
>>> x
```

'3'

Value is a string.

```
>>> x + 1
```

TypeError: must be str, not int

```
>>> x = int(x)
```

```
>>> x+1
```

4

Must convert to int.