4. Modules and Functions

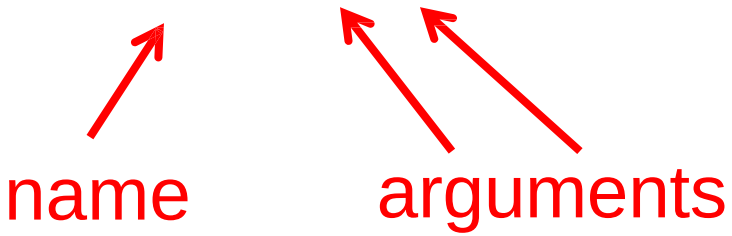
Topics:

- Modules
- Using `import`
- Using functions from `math`
- A first look at defining functions

Talking About Functions

A function has a name and arguments.

```
m = max(x,y)
```



We say that `max(x,y)` is a function call.

Built-in Functions

The list of "built-in" Python functions is quite short.

Here are some of the ones that require numerical arguments:

`max, min, abs, round`

```
abs(-6) is 6          max(-3,2) is 2          min(9,-7) is -7
round(6.3) is 6.0     round(3.5) is 4.0       round(-6.3) is -6.0
```

Calling Functions

```
>>> a = 5


>>> diff = abs(a)
```

In a function call, arguments can be expressions. Thus, the value of **a** is passed as an argument to **abs**.

The Built-In Function `len`

```
>>> s = 'abcde'
>>> n = len(s)
>>> print n
5
```

A function can have a string argument.

"In comes a string and out comes its length (as an `int`)"

Functions and Type

Sometimes a function only accepts arguments of a certain type. E.g., you cannot pass an `int` value to the function `len`:

```
>>> x = 10
>>> n = len(x)
TypeError: Object of the type int
                    has no len()
```

Functions and Type

On the other hand, sometimes a function is designed to be flexible regarding the type of values it accepts:

```
>>> x = 10
>>> y = 7.0
>>> z = max(x,y)
```

Here, max is returning the larger of two values and it does not care if one has type int and the other has type float.

Type-Conversion Functions

Three important built-in functions convert types: **`int`**, **`float`**, and **`str`**.

```
>>> a = float(22)/float(7)
>>> a
3.142857142857143
>>> b = int(100*a)
>>> b
314
>>> c = '100*pi = ' + str(b)
>>> c
'100*pi = 314'
```

Some Obvious Functions are not in the "Core" Python Library!

```
>>> x = 9
>>> y = sqrt(x)
NameError: name 'sqrt' not defined
```

How can we address this issue?

Modules

A way around this is to **`import`** functions (and other things you may need) from "modules" that have been written by experts.

Recall that a **`module`** is a file that contains Python code.

That file can include functions that can be imported for your use.

Widely-Used Modules

A given Python installation typically comes equipped with a collection of standard modules that can be routinely accessed.

Here are some that we will use:

`math`	`numpy`	`urllib2`
`string`	`scipy`	`PIL`
`random`	`timeit`	`datetime`

Import instruction

If you want to use the square root function from the math module, then it must be imported:

MyModule.py

```
from math import sqrt
          :
r = sqrt(250+110*sqrt(5))/20
          :
```

Useful functions in `math`

`ceil(x)`	the smallest integer >= x
`floor(x)`	the largest integer <= x
`sqrt(x)`	the square root of x
`exp(x)`	e**x where e = 2.7182818284…
`log(x)`	the natural logarithm of x
`log10(x)`	the base-10 logarithm of x
`sin(x)`	the sine of x (radians)
`cos(x)`	the cosine of x (radians)

Legal: `from math import sin,cos,exp,log`

floor, ceil, int, round

x	floor(x)	ceil(x)	round(x)	int(x)
2.9	2.0	3.0	3.0	2
2.2	2.0	3.0	2.0	2
2	2.0	2.0	2.0	2
2.5	2.0	3.0	3.0	2
-3.9	-4.0	-3.0	-4.0	-3
-3.2	-4.0	-3.0	-3.0	-3

What's in a Module?

If you know the name of a particular function and want more information:

```
>>> help('math')
```

Calling a function from a Module: Method 1

MyModule.py

```
from math import *
          :
r = sqrt(250+110*sqrt(5))/20
x = cos(pi*log(r))
          :
```

This is handy. You now have permission to use everything in the math module by its name. However, this can open the door to name conflict if the imported module is big like math.

Calling a function from a Module: Method 2

MyModule.py

```
import math
          :
r = math.sqrt(250+110*math.sqrt(5))/20
x = math.cos(math.pi*math.log(r))
          :
```

You again have permission to use everything in the math module by its name. But you must use its "full name" and that involves using the "dot notation."

Calling a function from a Module: Method 3

MyModule.py

```
from math import sqrt, pi, cos, log
          :
r = sqrt(250+110*sqrt(5))/20

 x = cos(pi*log(r))
          :
```

Here you take only what you need from the source module. You get to use "nice" names without using the dot notation. The danger of name conflicts minimized because you are explicitly aware of what is imported.

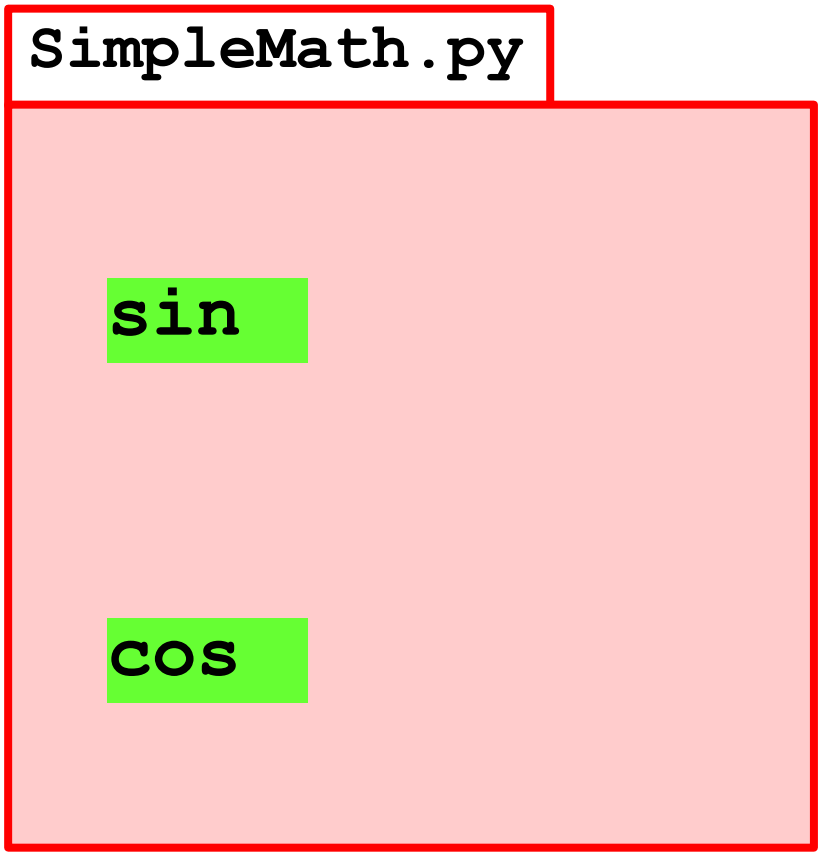
Building Your Own Functions

We will build functions that already exist in Python for the purpose of comparison. Among these functions are the **Sine** and **Cosine** mathematical functions.

We will write it into a script named:

SimpleMath

Visualizing `SimpleMath.py`



Recall that a module is simply a .py file that contains Python code.

This particular module houses three functions: `sin` and `cos`

The Cosine and Sine Functions

```
def cos(x):
    x = float(x)
    y = 1.0-(x**2/2)+(x**4/24)-(x**6/720)
    return y
```

```
def sin(x):
    x = float(x)
    y = x-(x**3/6)+(x**5/120)-(x**7/5040)
    return y
```

They too have headers

DO NOT WORRY ABOUT THE MATH. THIS IS ABOUT THE STRUCTURE OF PYTHONFUNCTIONS

The Cosine and Sine Functions

```
def cos(x):
    x = float(x)
    y = 1.0-(x**2/2)+(x**4/24)-(x**6/720)
    return y
```

```
def sin(x):
    x = float(x)
    y = x-(x**3/6)+(x**5/120)-(x**7/5040)
    return y
```

They too have bodies

Now let's compare these functions in the **`SimpleMath`** module with their counterparts in the **`math`** module.

Check out Cosine and Sine

SimpleMath.py

```
import math
import SimpleMath
x = float(input('theta (degrees)='))
x = (math.pi*x)/180
MyCos = SimpleMath.cos(x)
TrueCos = math.cos(x)
MySin = SimpleMath.sin(x)
TrueSin = math.sin(x)
            :
```

Check out Cosine and Sine

Sample Output…

```
theta (degrees) = 60
SimpleMath.cos(theta) =    0.49996457
      math.cos(theta) =    0.50000000
SimpleMath.sin(theta) =    0.86602127
      math.sin(theta) =    0.86602540
```