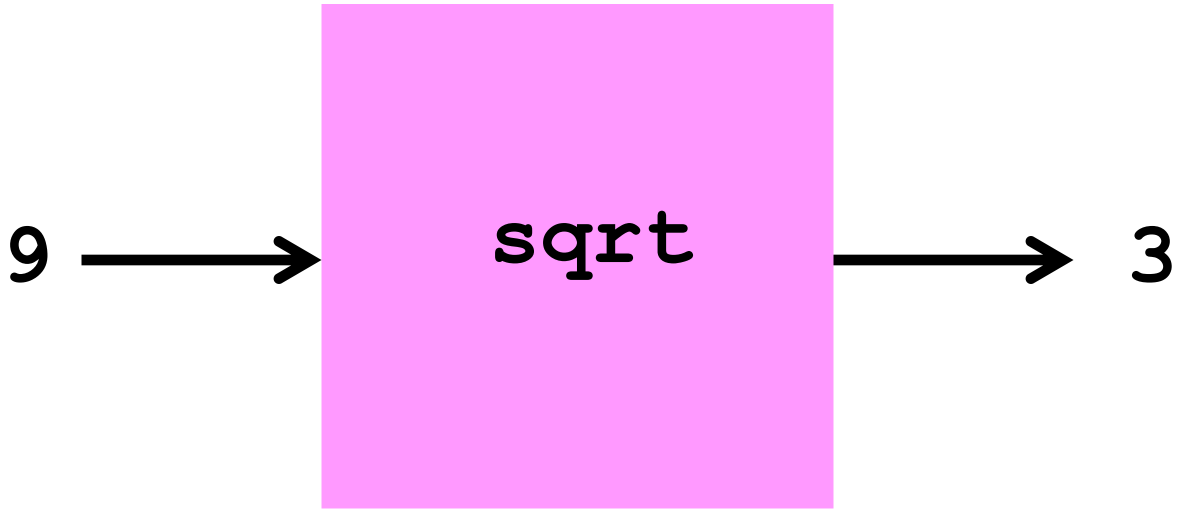
# 4. Using Modules and Functions

Topics:

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

## The Usual Idea of a Function



A factory that has inputs and builds outputs.

# Why are Functions So Important?

One reason is that they hide detail and enable us to think at a higher level.

Who wants to think about how to compute square roots in a calculation that involves other more challenging things.

```
r = (sqrt(250+110*sqrt(5))/20)*E
```

## A Point of View

To write a function is to package a computational idea in a way that others can use it.

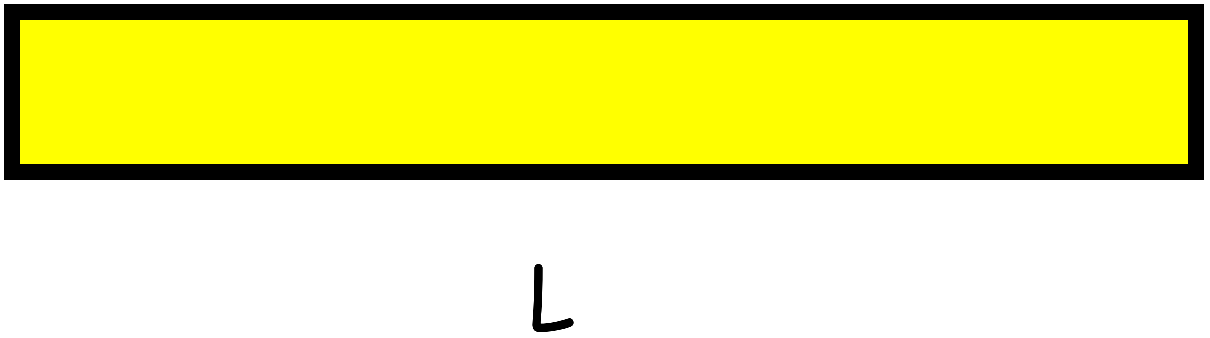
Take sqrt…

## Insight

The act of computing the square root of a number x is equivalent to building a square whose area is x.

If you can build that square and measure its side, then you have sqrt(x).

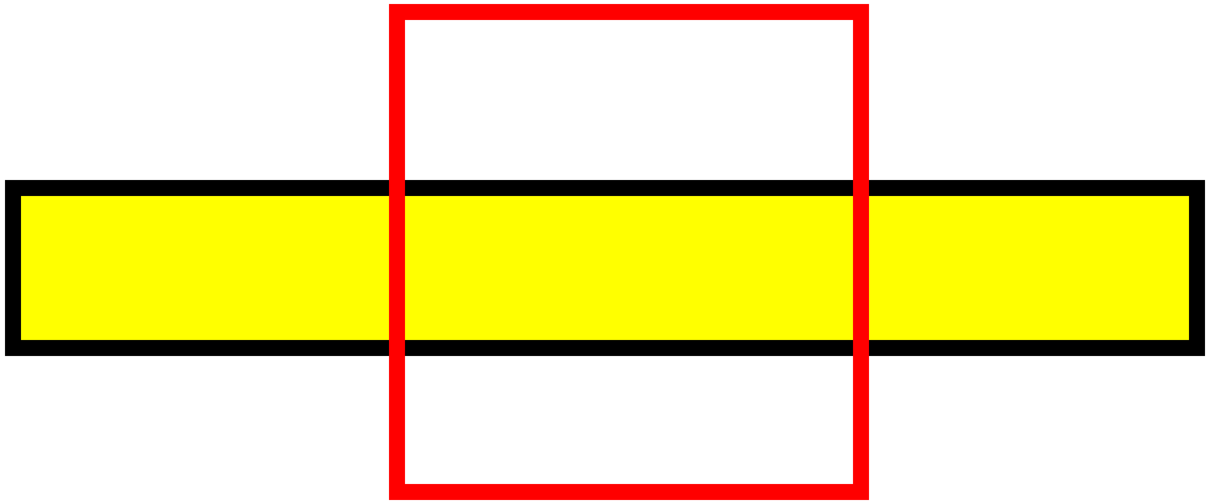
## Making a Given Rectangle "More Square"



$x/L = W$

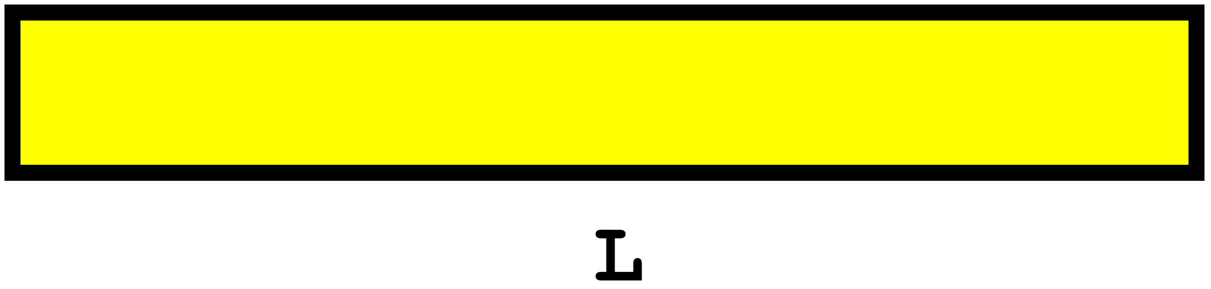
How can we make this rectangle
" more square" while preserving its area?

## Observation



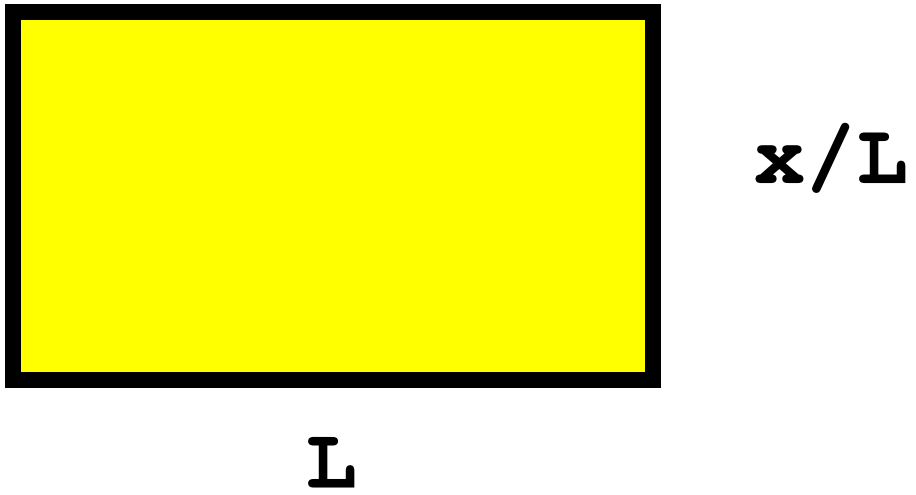
If the square and rectangle have area x, then we see that the sqrt(x) is in between L and W.

# Recipe for an Improved L

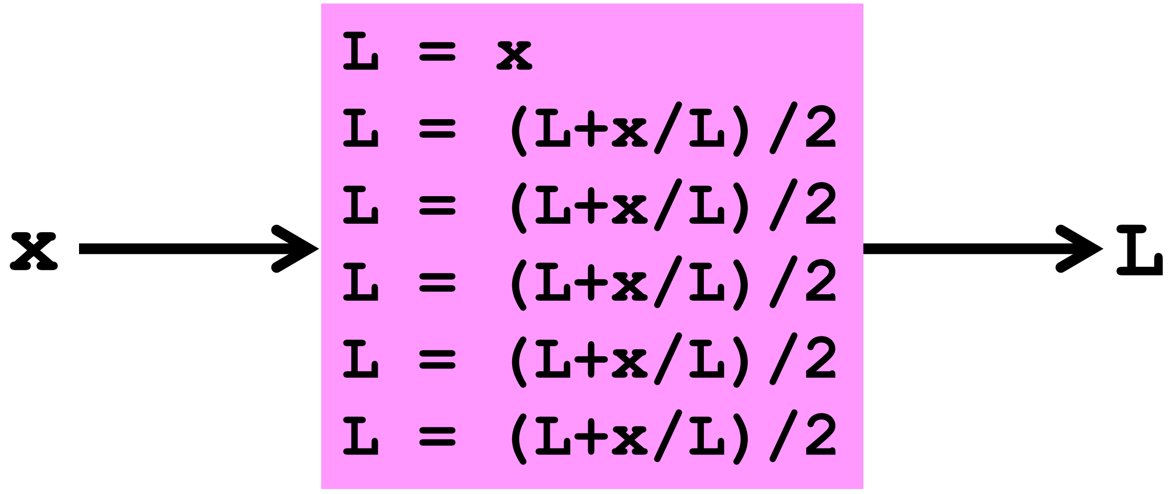


```
L = (L+x/L)/2
```

Take the average of the length and width



# The Usual Idea of a Function



```
L = x
L = (L+x/L)/2
L = (L+x/L)/2
L = (L+x/L)/2
L = (L+x/L)/2
L = (L+x/L)/2
```

A factory that has inputs and builds outputs.

How do we make something like 🟪 in Python?

# Talking About Functions

A function has a name and arguments.

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

name arguments

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
>>> b = 7
>>> m = max(a**b,b**a)
>>> diff = abs(a**b-b**a)
```

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

# Functions in Mathematics vs Functions in Python

So far our examples look like the kind of functions that we learn about in math.

> "In comes one or more numbers and out comes a number."

However, the concept is more general in computing as we will see throughout the course.

## The Number of Arguments is Sometimes Allowed to Vary

```
>>> a = 5
>>> b = 6
>>> c = 7
>>> d = 8
>>> m = max(a**d,d**a,b**c,c**b)
>>> n = max(a*b*c*d,500)
```

The max function can have an arbitrary number of arguments

## 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

A function may only accept 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)
```

## Type-Conversion Functions

Keep in mind that **`int`**, **`float`**, and **`str`** are also "built-in" functions.

```
a = float(22)/float(7)
b = int(100*a)
s = 'pi = ' + str(b)
```

They convert representations. "In comes an **`int`** and out comes a **`float`** that represents the same value"

## 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 in CS 1110:

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

# The CS1110 Plan for Learning about Functions in Python

1. Practice using the `math` module. Get solid with the import mechanism.

2. Practice using the `simpleMath` module. Get solid with how functions are defined.

3. Practice designing and using your own “math-like” functions.

# The Plan Cont'd

4. Practice using the `simpleGraphics` module. Get solid using procedures that produce graphical output.

5. Practice using methods from the string class.

6. Practice using the `simpleDate` module. Get solid with how methods and objects are defined.

Procedures and Methods are special types of functions.

## The Plan Cont'd

Over the entire semester we keep revisiting the key ideas to see how they play out in more complicated situations.

All along the way we develop skills for

1. Designing Functions
2. Testing Functions

# By Analogy

Tricycle in the Driveway. And then…

Tricycle on the sidewalk. And then…

2-wheeler w/ trainers. And then…

2-wheeler no turning. And then…

2-wheeler and turning in street. And then…

2-wheeler w/ derailleur. And eventually…

Tour de France*

*But only if you "test positive" for Python!

## Let's Start by Revisiting `import`

We have already used `import`:

```
kepler.py
```

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

## 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) |
| `tan(x)` | the tangent of x (radians) |
| `atan2(x,y)` | the angle whose tangent is y/x |

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

## Finding Out What's in a Module?

If a module is part of your Python installation, then you can find out what it contains like this:

```
>>> help('random')
```

But if the module is "famous" (like all the ones we will be using), then just Google it.

## What's in a Module?

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

```
>>> help('math.sqrt')
```

What's With the "dot" Notation: **`math.sqrt?`**

# Need a Lot of Stuff from a Module? A Tempting Shortcut…

CarelessKepler.py

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

You now have permission to use everything in the math module by its name. However, this can open the door to name conflict. More later

# Need a Lot of Stuff from a Module? A Safer Way…

CarefulKepler.py

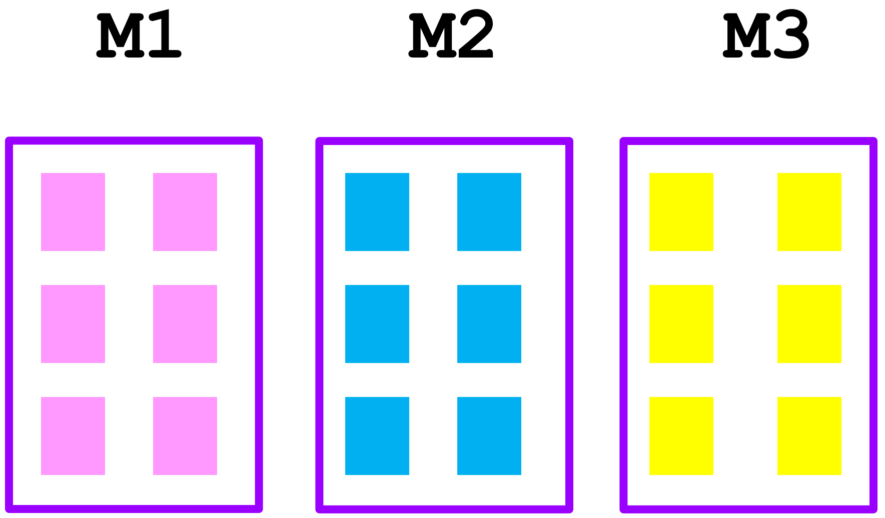
```
import math
        :
r = (math.sqrt(250+110*math.sqrt(5))/20)*E
x = math.cos(math.pi*math.log(r))
        :
```

You now have permission to use everything in the math module.
But you must use its "full name." The "dot notation" does this.

# Appeciating "Full Names"

Your code

```
import M1
import M2
import M3
      :
```

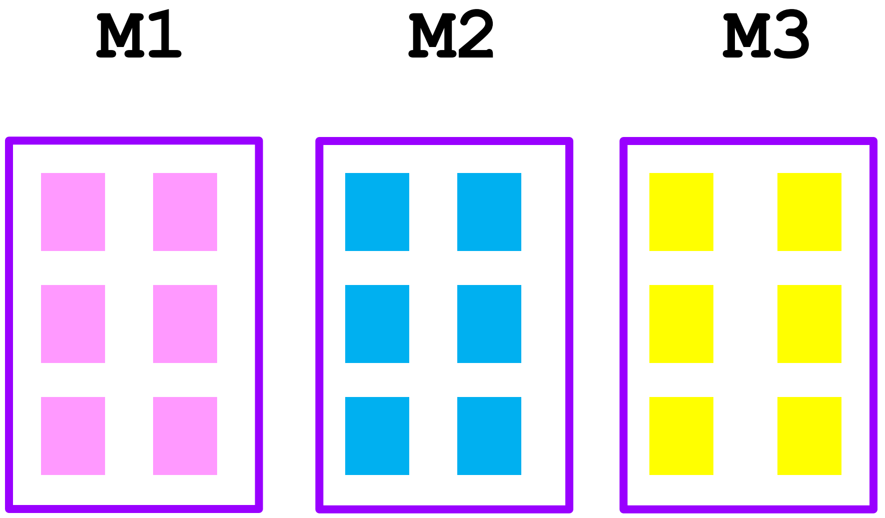


Unambiguous names in your code even if some of the module functions have the same name.

# Appeciating “Full Names“

Your code

```
from M1 import *
from M2 import *
from M3 import *
      :
```

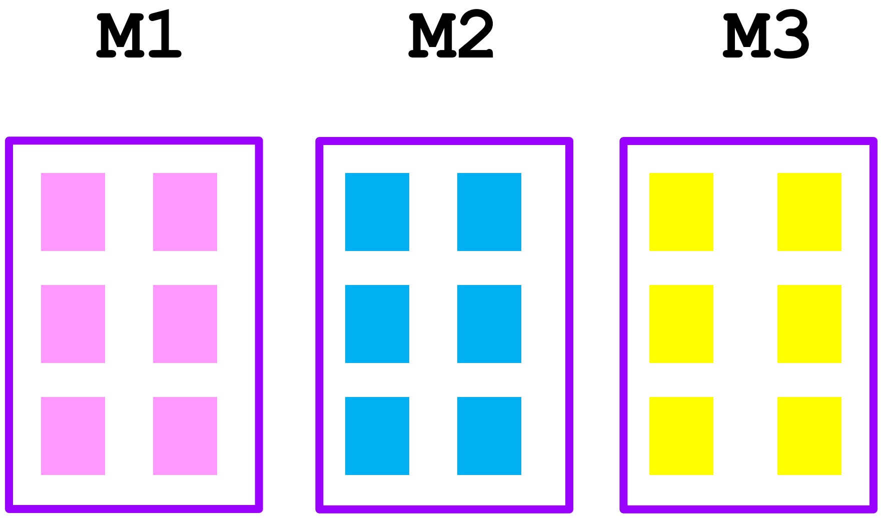


Now function calls in your code can be ambiguous.
Easy to lose track of things if **M1**, **M2**, and **M3**
include tons of functions.

## Appeciating “Full Names“

Your code

```
from M1 import f1
from M2 import f2
from M3 import f2
      :
```

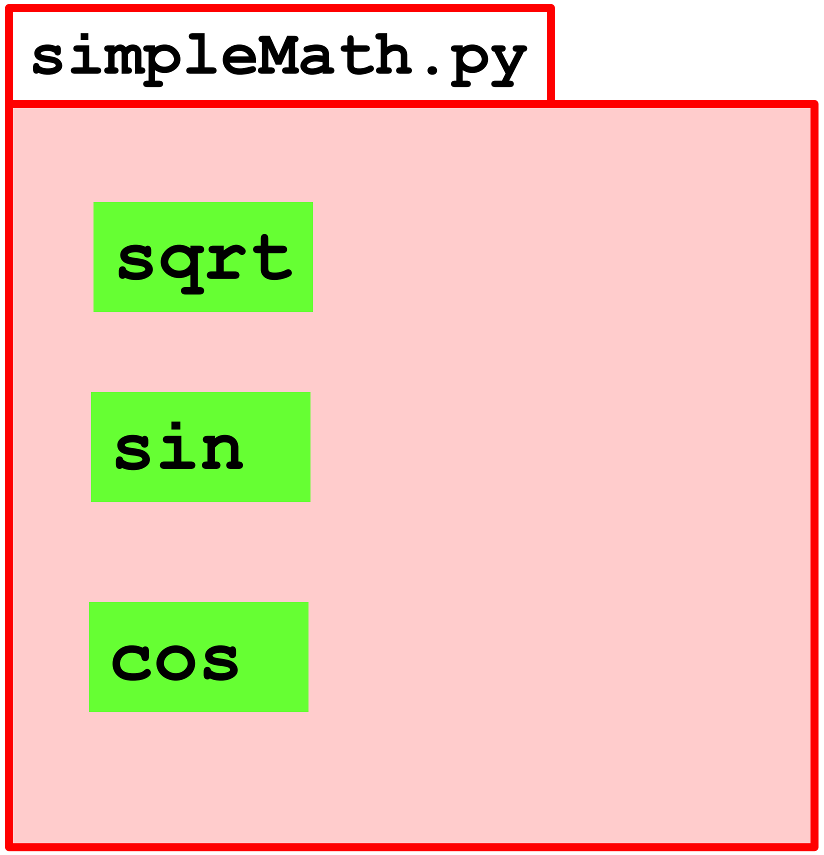


Selective importing is ok since you are “on top of” exactly what is being imported. And you can use the short name, e.g., `f1` instead of `M1.f1`

The time has come to see how functions are actually defined.

To do this we introduce a small "classroom" module that we call **`simpleMath`**.

# Visualizing `simpleMath.py`



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

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

## How are Functions Defined?

Let's look at the three function definitions in **`simpleMath`** not worrying (for now) about their inner workings.

This plays nicely with the following fact: you can use a function without understanding how it works.

I can drive a car without knowing what is under the hood.

# A Square Root Function

```
def sqrt(x):
    x = float(x)
    L = x
    L = (L + x/L)/2
    L = (L + x/L)/2
    L = (L + x/L)/2
    L = (L + x/L)/2
    L = (L + x/L)/2
    return L
```

The function header begins with **`def`**.

It indicates the name of the function and its arguments.

Note the colon and indentation.

# A Square Root Function

```
def sqrt(x):
    x = float(x)
    L = x
    L = (L + x/L)/2
    L = (L + x/L)/2
    L = (L + x/L)/2
    L = (L + x/L)/2
    L = (L + x/L)/2
    return L
```

This is the body of the function.

It computes a value L (hopefully a good square root.)

The calling program will be informed of this value because of the **`return`** statement.

## 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

Again, do not worry about the math behind the implementations.

# 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

# Fruitful Functions

All three of these functions are fruitful functions.

Fruitful functions return a value.

Not all functions are like that.

We will discuss the mechanics of how fruitful functions return values later.

## Making Functions Usable

Again, the great thing about functions in programming is that you can use a function without understanding how it works.

However, for this to be true the author(s) of the function must communicate how-to-use information through docstrings and comments. There are set ways (rules) for doing this.

## Rule 1. The Module Starts With Authorship Comments

```
# simpleMath.py
# Charles Van Loan (cfv3)
# January 2, 2015
""" Module to illustrate three simple
math-type functions.

Very crude implementations for the
square root, cosine, and sine
functions."""
```

Module Name, author(s), last-modified date.
And we follow that format in CS 1110.

## Rule 2. The Module Specification

```
# simpleMath.py
# Charles Van Loan (cfv3)
# January 2, 2015
""" Module to illustrate three simple
math-type functions.

Very crude implementations for the
square root, cosine, and sine
functions."""
```

Short line, blank line, longer comments. This is displayed when you type this: **`help('simpleMath')`**

## Rule 3. Each Function Starts with a Docstring "Specification"

```
def sqrt(x):
	"""Returns an approximate square
	root of x.

	Performs five steps of rectangle
	averaging.

	Precondition: The value of x is a
	positive number."""
```

Short summary that states what the function returns. Also called the post condition.

# Rule 3. Each Function Starts with a Docstring "Specification"

```
def sqrt(x):
    """Returns an approximate square
    root of x.

    Performs five steps of rectangle
    averaging.

    Precondition: The value of x is a
    positive number."""
```

Longer prose giving further useful information to the person using the function.

## Rule 3. Each Function Starts with a Docstring "Specification"

```
def sqrt(x):
      """Returns an approximate square
      root of x.

      Performs five steps of rectangle
      averaging.

      Precondition: The value of x is a
      positive number."""
```

Conditions that the arguments must satisfy if the function is to work. Otherwise, no guarantees.

## The Specification for cos(x)

```
def cos(x):
      """Returns an approximation to the
      cosine of x.

      Uses a degree-6 polynomial.

      Precondition: x is a number that
      represents a radian value."""
```

## The Specification for sin(x)

```
def sin(x):
    """Returns an approximation to the
    sine of x.

    Uses a degree-7 polynomial.

    Precondition: x is a number that
    represents a radian value."""
```

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

## Check out Square Root

Show_simpleMath.py

```
import math
import simpleMath
                :
x = input('x = ')
MySqrt = simpleMath.sqrt(x)
TrueSqrt = math.sqrt(x)
                :
```

# Check out Square Root

Sample Output

```
x = 25
simpleMath.sqrt(x) =   5.00002318
      math.sqrt(x) =   5.00000000
```

## Check out Cosine and Sine

Show_simpleMath.py

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

## 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
```

## Summary

1. How to gain access to functions in other modules using `import`.

2. How to define a function using `def`.

3. How to document modules and functions through structured doc strings.

## Terminology

**argument**

An expression that occurs within the parentheses of a method call. The following call has two arguments: x+y and w+z: min(x+y,w+z)

Reference: http://www.cs.cornell.edu/Courses/cs1110/2015sp/materials/definitions.php

## Terminology

**docstring**

A string literal that begins and ends with three quotation marks. Document strings are used to write function specs and are displayed by the help() command.

Reference: http://www.cs.cornell.edu/Courses/cs1110/2015sp/materials/definitions.php

## Terminology

**fruitful function**

A function that terminates by executing a return statement, giving an expression whose value is to be returned. Fruitful functions (possibly) return a value other than None.

Reference: http://www.cs.cornell.edu/Courses/cs1110/2015sp/materials/definitions.php

## Terminology

**function**

A set of instructions to be carried out. A function is analogous to a recipe in a cookbook. We often separate functions into fruitful functions and procedures.

Reference: http://www.cs.cornell.edu/Courses/cs1110/2015sp/materials/definitions.php

## Terminology

**function body**

A function consists of a function header followed by the function body, which is indented by four spaces under the header. When the function is called, its body is executed.

Reference: http://www.cs.cornell.edu/Courses/cs1110/2015sp/materials/definitions.php

## Terminology

**function name**

The name of the method, defined in the function header.

Reference: http://www.cs.cornell.edu/Courses/cs1110/2015sp/materials/definitions.php

## Terminology

**function specification**

The specification of a function defines what the function does. It is used to understand how to write calls on the function. It must be clear, precise, and thorough, and it should mention all parameters, saying what they are used for. It may be given in terms of a precondition and postcondition. Function specifications are typically written as a docstring.

Reference: http://www.cs.cornell.edu/Courses/cs1110/2015sp/materials/definitions.php

## Terminology

**import**

The import statement has one of the following forms:

**import** <*module*> # encapsulate contents in module folder

**from** <*module*> **import** * # pull everything into global space

Reference: http://www.cs.cornell.edu/Courses/cs1110/2015sp/materials/definitions.php

## Terminology

**module**

A file containing global variables, functions, classes and other Python code. The file containing the module must be the same name as the module and must end in ".py" A module is used by either importing it or running it as a script.

Reference: http://www.cs.cornell.edu/Courses/cs1110/2015sp/materials/definitions.php

## Terminology

**parameter**

A variable that is declared within the parentheses of the header of a function or method.

Reference: http://www.cs.cornell.edu/Courses/cs1110/2015sp/materials/definitions.php

## Terminology

**post-condition**

An assertion that indicates what is to be true at the end of execution of a function body or, more generally, of any sequence of statements.

Reference: http://www.cs.cornell.edu/Courses/cs1110/2015sp/materials/definitions.php

## Terminology

**precondition**

An assertion that indicates what is to be true at the beginning of execution of a function body.

Reference: http://www.cs.cornell.edu/Courses/cs1110/2015sp/materials/definitions.php

## Terminology

**return statement**

A statement which terminates the execution of the function body in which it occurs. If it is followed by an expression, then the function call returns that value. Otherwise, the function call returns None.

Reference: http://www.cs.cornell.edu/Courses/cs1110/2015sp/materials/definitions.php