Chpater 2, Using Modules: Don't Reinvent the Wheel

John M. Morrison

June 21, 2021

Contents

1 Introduction

Heretofore we have learned about the basics of a programming language. We now have the tools of conditional execution and functions to build programs. We are now going to proceed to learn how to use code created by others for our purposes, and how to distribute our code to others who might use it.‘

This is in keeping with the philosophy of “Use wheels, don’t reinvent them.” It is best to use an off the shelf solution to a problem if possible. However, we all know that we can sometimes learn by re-creating things for ourselves, which we will sometimes do so we can understand how they work.

Let us begin by turning Python into a scientific calculator.

2 OK, TI89, Time to Sweat!

A *module* in Python is a file containing code for functions. Python has a sizable number of built-in modules; to use them we must use the *import statement* in our programs.

2.1 How do I Import a Module?

To avail ourselves of scientific calculator functions, we just import the `math` module. One way to import the math module is to type

```
import math
```

at the beginning of our program. This is the *regular import*.

A second way is to use the *intimate import*

```
from math import *
```

Let us do a quick comparison. We can use `dir()` to see what symbols are visible at any time. Here we do the regular import; the only new symbol visible is that of `math`; we did a `dir()` before and after the import to see this. However, if we do `dir(math)`, we can see all of the mathematical goodness inside.

```
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__']
```

```
>>> import math
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'math']
>>> dir(math)
['__doc__', '__name__', '__package__', 'acos', 'acosh',
'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil',
'copysign', 'cos', 'cosh', 'degrees', 'e', 'exp',
'fabs', 'factorial', 'floor', 'fmod', 'frexp',
'fsum', 'hypot', 'isinf', 'isnan', 'ldexp', 'log',
'log10', 'log1p', 'modf', 'pi', 'pow', 'radians',
'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

To use a math function with the regular import, we must use the prefix `math.`; think of this as the genitive case, i.e., `math.` means `math`'s. Observe that `math.pi` is just a number; it is not a function. The name math provides a namespace; it is an umbrella under which a group of symbols live. When we write `math.sqrt(5)` we are using the full name of the function.

```
>>> math.sqrt(5)
2.2360679774997898
>>> math.log10(100)
2.0
>>> math.pi
3.1415926535897931
```

We can also by import all names in math directly into our session by using the intimate import as follows.

```
from math import *
```

Now let us view the visible symbols with `dir()`. The names of the math functions are directly visible in our session.

```
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__',
'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2',
'atanh', 'ceil'' 'copysign', 'cos', 'cosh', 'degrees',
'e', 'exp', 'fabs', 'factorial', 'floor', 'fmod',
'frexp', 'fsum', 'hypot', 'isinf', 'isnan', 'ldexp',
'log', 'log10', 'log1p', 'modf', 'pi', 'pow', 'radians',
'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

We can use them without `math.`; in fact, using them with `math.` causes an error.

```
>>> sqrt(5)
2.2360679774997898
>>> log10(100)
2.0
>>> pi
3.1415926535897931
>>> math.pi
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'math' is not defined
>>>
```

Ooh, goodie! We have to type less! Isn't that nice? On the other hand, there is a devil to pay. You have many more symbols visible; all of the symbols in `math` are unprotected in your program. Let's play Indiana Legislature. Continuing the last session, we can redefine π! Now our program believes π to be 3.

```
>>> pi = 3
>>> pi
3
>>> pi*10*10
300
```

You can repeat similar foolishness the regular import. However, you know you are in dangerous waters reassigning a value in an existing module. This is deliberate vandalistic stupidity which will result in nasty self-inflicted wounds.

The other gaffe, `pi = 3` could be unintentional if you did not know the math module contained a constant named pi. Such an unfortunate occurrence is called a *namespace collision*, and it produces terrible programming errors that are incredibly hard to ferret out.

Hence it is best to use the regular import when writing code into programs. The intimate import is fine for informal testing and exploration especially in Python's interactive mode, and it saves typing. In general, keeping the number of visible symbols small is like driving in light traffic: less congestion and confusion ensue.

If you only want a couple of `math` functions, you can use the one-at-a-time import like so.

```
from math import sin, cos
```

During your program, you may use `sin` or `cos` on a first-name basis and you do not clutter your namespace with lots of functions you are not going to use. In general, you should strive to keep your symbol table as small as possible, but the convenience generated in this situation is probably worth the small number of new symbols.

As a matter of style, it is a good idea to place your `import` statements at the top of your program. This is part of Google's style rules and we shall adhere to it here.

2.2 Exploring the `math` Module

Most objects output by functions in this module are `float` objects. Since we are just test-driving in an interactive session, the intimate import is fine.

```
>>> from math import *
>>> ceil(1.5)
2
>>> ceil(5)
5
>>> ceil(-7.5)     #negatives, but this is right!
-7                 #round up looks funny on
>>> type(ceil(5))
<class 'int'>
```

Now let's round down.

```
>>> floor(-7.5)
-8
>>> floor(4.6)
4
```

Python 2 vs. 3 Note In Python 2, `floor` and `ceil` return `floats`. In Python 3, they return an `int`.

Here is absolute value. Be aware that all math functions return floating-point numbers.

```
>>> fabs(-2)
2.0
```

Here is the natural exponential function.

```
>>> exp(1)
2.7182818284590451
>>> exp(2)
7.3890560989306504
>>> exp(.693)
1.9997056605411638
```

The log function is the natural log function. However, by using a second argument, you can compute logs of any base.

```
>>> log(1000)
6.9077552789821368
>>> log(1000,10)         #log base 10
2.9999999999999996
>>> log(1000)/log(10)    #also log base 10 by change-of-base
2.9999999999999996
```

Here we see powers and a separate square-root function.

```
>>> pow(5,6)
15625.0
>>> sqrt(4)
2.0
```

We saw this trick for computing square roots before.

```
>>> 3**.5
1.7320508075688772
>>> 3**(1/2)
1.7320508075688772
```

Beware, if you are using Python 2, that $1/2$ evaluates to zero! This is what would happen in a Python 2 session.

```
>>> 3**.5
1.7320508075688772
>>> 3**(1/2)
1
```

Here we show inverse sine and arcsine. All angles are in radians. In this example we see that $\arcsin(1) = \pi/2$.

```
>>> asin(1)
1.5707963267948966
```

And here is inverse cosine; we are seeing here that $\arccos(1) = 0$ and $\arccos(-1) = \pi$.

```
>>> acos(1)
0.0
>>> acos(-1)
3.1415926535897931
```

Finally we see that arctan(1) = $\pi/4$, arctan(4/3) = .92729521800161219, and that in degrees, arctan(1) = 45° .

Programming Exercises These exercises will acquaint you with the `math` module.

1. What do `degrees` and `radians` do?
2. What does `fmod` do?
3. How can you compute a log of any base using only the natural log function `log`?
4. Write a function `length(a,b,theta)` which, given side-angle-side data on a triangle, computes the length of the third side.
5. Does the math library have a cube root function? a square root function?
6. At the interactive prompt, type

   ```
   print(math.log__doc__)
   ```

   What happens? Try this on some other functions. Did you learn something useful? If you did, put it in yoour pocket and don't forget about it.

3 Reading Python Documentation

How do you find out what sort of modules Python has in its standard libraries? You can go to the Python site, `http://docs.python.org/3/`.

Under Indices and Tables, you will see the Global Module Index. Click on that link and be delivered here, `http://docs.python.org/3/py-modindex.html`. This has an alphabetical index of all Python modules. Navigate to `math` by using the alphabetical link area at the top of the page, and click on `m`, then you will see the `math` link.

You will see complete information about math functions Type some of them them into an interactive session and see them at work.

Python has hundreds of modules you can use to do a wide array of programming tasks. We will discuss the module `random` later in this chapter; it is handy for writing simple games. We encourage you to go for a little walk in the documentation and experiment. Just look for the `random` module under the letter r. You should do some exploring. The exercises below will have you poking through the `datetime` and `os` modules. Do not skip them; you will get to learn about some handy new toys.

Programming Exercise It's shiny objects time! These exercises all you to see some cool tools for doing an assortment of programming tasks. At this time, you can begin reading Chapter 4 to learn more. To start these exercises visit the documentation under `datetime`, and scroll down to Date Objects. Then use the directive `from datetime import date` in an interactive session.

1. Get Python to print today's date.
2. Make your birth date by typing something like

   ```
   d = date(1776, 7, 4)
   ```

   Note this is 4 July 1776. Subtract this from `today`. What do you see?
3. How many days old are you? Can you figure out when you will be 30000 days old?
4. What day of the week were you born on?
   For the next couple of exercises use `import os`
5. What happens when you enter `os.listdir()` into an interactive session. How about `os.listdir("..")` Try this on an absolute path in your file system. Windoze users should remember to use a raw string. Can you use this to count the number of files in a directory?
6. Pick a file in your cwd and call `os.stat()` on it. Then exit python and do an `ls -li` on it.. What do you see? Check the man page for `ls` and see what the `-i` option does.
7. At the interactive prompt, import `os` and then do a `dir(os)`. Check out some of the goodies inside. Make a junk directory and put some empty files in it. See if you can remove or rename files using the `os` module.
   Now use `import os.path`
8. How do you check to see if a file by a given name exists?
9. Choose a file that exists in your python processes's `cwd` and see if you can print out its absolute path.
10. For a file or directory on your machine, how do you tell if it is a file or if it is a directory?
11. Import the `statistics` module and see if you can use it to compute the mean, median and standard deviation of numbers in a list.
12. For the functions you have tried here, print out their docstrings and see what those say. Just use `.__doc__`. For example, try `print(os.listdir.__doc__)`.

4 Creating your own Modules

You can place functions in programs and use them repeatedly. You can also call one function from another function. Enter the following program in a file called `fun.py`.

```
#!/usr/bin/python
def square(x):
        return x*x
def hypot(x,y):
        return (square(x) + square(y))**(.5)
```

The file `fun.py` is a *module* of code containing two functions. You will often see that programmers will organize related groups of functions into modules in exactly this manner.

Run the program by entering

```
$ python fun.py
```

at the UNIX command line. If you entered the program correctly, you will see that *nothing appears to have happened.* What did happen is that your functions were stored in memory. Since you never called them, your program ends before they get used.

Now let's see how to use our module in another program. Create this file and name it `foo.py`; make sure it is in the same directory as your module `fun.py`.

```
#!/usr/bin/python
import fun
print (fun.square(5))
print (fun.hypot(5,12))
```

At the UNIX command line type

```
$python foo.py
```

You will see

```
25
13.0
```

appear on the screen. Let's now deconstruct. The directive `import fun` tells Python to load the code from the file `fun.py`; the functions you created in `fun.py` are now available. The command `print fun.square(5)` means, "pass the value 5 to `fun.py`'s `square` function". If you forget the `"fun."` before `square` you will get this error message.

```
>>> square(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'square' is not defined
>>>
```

Python is saying, "I have never heard of the square function!" It is expecting your program `foo.py` to have a square function. You must tell python *whose* function you are calling. Here we see Python's namespace mechanism at work.

This naming of function works much like the naming of people. Since you have a first and last name, if your full name is called in a crowded room, it is fairly unlikely that two people will respond, whereas if just a first name is called, several people are likely to respond and confusion is likely to ensue. Your last name behaves as a namespace. We now demonstrate this action.

Begin by modifying your `foo.py` to look like this.

```
#!/usr/bin/python
import fun
def square(x):
    return "This square function is not from fun.py!"
print (fun.square(5))
print (fun.hypot(5,12))
print (square(5))
```

Run this from the command line and you see the following

```
$ python foo.py
25
13.0
This square function is not from fun.py
```

These two lines

```
print (fun.square(5))
print (fun.hypot(5,12))
```

call `fun.py`'s `square` and `hypot` functions. However the last line

```
print (square(5))
```

moduleName.function(args)

If you list your files at the command line using `ls`, you will see that a side effect of importing `fun.py` into `foo.py` is that you will have the file `fun.pyc`. If you try to open this file with `vi`, you will see a bunch of gibberish. This file is called a *precompiled Python file*; it is an executable file that encodes the Python file `fun.py` in a format called *Python byte code*.

5 Preconditions, Postconditions, and Documentation

It is easy to enhance the usefulness of your Python functions by giving them a it docstring, or documentation string. This is done in the following manner.

```
def benjaminFranklin(key, string, kite):
    """This function performs scientific experiments and then during the
American Revolution, it woos ladies in Paris to get France in the war."""
    return "I am on the C-Note"
```

The docstring must go right after the function header in triple–quotes. Users can see it by printing `benjaminFranklin.__doc__` as shown in this interactive session. This should be familiar if you did the last two sets of exercises. Notice how using the `print` command causes the newline to be expanded.

```
>>> def benjaminFranklin(key, string, kite):
...     """This function performs scientific experiments and
... then during the American Revolution, it woos ladies
... in Paris to get France into the war."""
...     return "I am on the C-Note"
...
>>> benjaminFranklin.__doc__
'This function performs scientific experiments and\n then
during the American Revolution, it woos ladies\n in
Paris to get France into the war.'
>>> print (benjaminFranklin.__doc__)
This function performs scientific experiments and
then during the American Revolution, it woos ladies
in Paris to get France into the war.
>>>
```

In a docstring, you can state *preconditions* and *postconditions*.

Preconditions can be specifications on the arguments passed to the function; more generally, they are conditions that should be satisfied before the function is called. You may treat preconditions as being "God-given" when you are writing your function. They are the rules you lay down for the proper use of your function. Woe betide the abuser of your function, for he is likely to get exactly what he deserves! For him, it's *Caveat emptor!*

If a function is unfamiliar to you, read its docstring, and make sure you are playing by the rules. Fail to do so at your own risk. You should try printing out some docstrings from various functions in various libraries. Here are two quick examples.

```
>>> import math
>>> print(math.cos.__doc__)
cos(x)

Return the cosine of x (measured in radians).
>>> import os.path
>>> print(os.path.exists.__doc__)
Test whether a path exists.  Returns False for broken symbolic links
>>>
```

Postconditions describe the action of a function; they are things that are true after the function is done running. You should describe any return value and any side-effects of calling the function. We will specify all functions by their preconditions and postconditions. For a well-designed function, specifying the function in this way should be a simple matter. This description needs to be absolutely complete. This is the standard: *Another person should be able to use your function properly by reading your function's docstring.*

Place these conditions in your docstring. Remember that the docstring must occur at the first line of the function, inside of triple-quotes. Remember the purpose of creating modules is to create re-usable chunks of code. If you document your modules properly, other programmers can use your code base on their projects.

Here are two programs; enter these and save them in the same directory. Notice how you make the second and subsequent lines in your docstring flush-left. Also note the use of proper indentation.

docstring.py

```
#!/usr/bin/python
def square(x):
    """Precondition:  x is a number
Postcondition:  The square of x is returned to the caller.
This is a pure function. """
    return x*x
```

doo.py

```
#!/usr/bin/python
import docstring
print ("Here is the docstring:  ")
print (docstring.square.__doc__)
print (docstring.square(5))
```

At the UNIX command line type

```
$ python doo.py
Here is the docstring
Precondition:  x is a number
Postcondition:  x*x is returned to the caller.
This is a pure function.`
25
```

Exercises Place the functions listed here in a Python program so you can test them as you create them. Give appropriate pre and post conditions in a docstring for each function.

1. Write a Python function `lastChar` that takes a nonempty string as an argument and which returns its last character.
2. Write a Python function `sumOfThree` that takes three numbers are arguments and returns their sum
3. Write a Python function `repeat` that takes a string and a non-negative integer as an argument, and which returns the string repeated the integer number of times.

Here are some sample outputs you can test against.

```
lastChar("platypus") -> "s"
lastChar("a") -> "a"
sumOfThree(4,5,6.7) -> 15.7
sumOfThree(-1,0,1) -> 0
repeat("ab", 5) -> "abababababab"
repeat("moo ", 6) -> "moo moo moo moo moo moo "
```

6 Integrated Testing of Modules

The best way to write a function is to use the "Mean Teacher" procedure. In this procedure we administer the test, fail the student, then teach the lesson. We keep failing the "student" (function) until it passes with a score of 100%. In books on software development, you will hear this referred to as *test-driven development*. It is a very widespread practice in professional software development shops.

1. Give the function a name and an argument list.
2. Describe preconditions and postconditions for the function. These will completely specify the desired behavior of the function. If you are unable to do this easily and succinctly, you do not understand the task at hand well enough to carry it out.

3. Write test code for the function. Give it a *pro forma* return value called a *stub*. A good return value for a stub a nominal value of the correct return type for your function. On that line make a comment `#TODO`. You can use the search facility in `vi` to search for the string `#TODO` to find any incomplete functions in your module of code.
4. Run this code *before coding your function*. Make sure you extirpate any syntax errors and that your program compiles happily before continuing to the next step. Be merciless and seek out all potential weaknesses you can think of.
5. Code each function, and test it. Progress to another as each function passes the tests.
6. If all goes well, your test code performs as expected as each function is implemented.

6.1 An Example

Here is an example of good test code for the `lastChar` function residing in the last collection of exercises. Notice the return value for the function that makes it obvious we are not done yet. Place this in a program `lastCharShell.py`.

```
def lastChar(x):
    """prec:  x is a nonempty string
postc:  returns the last character of x"""
    return "TODO"
x = "moose"
print (f"lastChar({x}) = {lastChar(x)}, expected:  e")
x = "This is an sentence."
print (f"lastChar({x}) = {lastChar(x)}, expected:  .")
x = "DYSTOPIA"
print (f"lastChar({x}) = {lastChar(x)}, expected:  A")
```

The code fragment

```
x = "This is an sentence."
print (f"lastChar({x}) = {lastChar(x)}, expected:  .")
```

is called a *test case*. The collection of test code illustrated here is called a *test harness* or *test suite*. This test harness has three test cases in it. As your repertoire of programming skill grows, you will need to think carefully about the murphology of your functions. Try test cases that are likely to "break" your functions. For example, if a function contains conditional logic, have at least one test case for each possible path of execution. Test any border cases. Be merciless. However, play fairly and do not violate the preconditions of your function. Remember, good teachers are tough but fair.

If you implement the function correctly you should get this output.

```
$ python lastCharShell.py
lastChar(moose) =  e , expected:  e
lastChar(This is an sentence.) =  . , expected:  .
lastChar(DYSTOPIA) =  A , expected:  A
```

This procedure ensures the maximum integrity for your functions. In the professional world, you are often required to furnish test harnesses for any functions you write. By so doing you provide a starting point for others who may be using your functions to figure out what goes wrong if code depending on your functions should develop bugs. Your colleagues will be aware of the vulnerabilities you have tested, and they will not waste expensive and valuable time retracing old territory.

If you are creating modules of functions for a program, you will want to test them severely to ensure that they are reliable tools for your project. This is what is meant by the maxim "Creates strong black boxes and test them mercilessly. Then rely on them."

What if I want to import a module and not see the test code? You have been using functions from day one, but didn't know it. How can this be? Here is our first program `hello.py`. Yeah, `print` is a function, but that's not the point.

```
#!/usr/bin/python
print ("Hello, world!")
```

It appears that the `print` statement is outside of any function. However, it is not. There is a super-secret function with the ungainly name of `__main__` that actually contains the call to `print`. We will refer to `__main__` henceforth as the *main routine* of a program.

Say we want to test the square function. Here is a full-blown module.

```
def square(x):
    """Precondition:  x is a number
Postcondition:  returns the square of x.
This is a pure function"""
    return x*x

if __name__ == "__main__":
    t = 1
    print("PASS" if square(t) == 1 else "FAIL")
    t = 0
```

```
print("PASS" if square(t) == 0 else "FAIL")
t = -1
print("PASS" if square(t) == 1 else "FAIL")
```

If you run this directly, it will execute the test code. If you import it into another module, the test code will be suppressed. This is the best way to keep test code for modules.

Exercise Write test harnesses for each of the functions you developed in the last section. Give them appropriate docstrings. Import the functions into another module and use the trick we have just seen here so the test code does not run in the program into which it is imported.

7 A Chinese Wall

Functions store procedures we want to use to accomplish tasks. A large program may contain hundreds, or thousands of functions.

This brings us back to the symbol table. If a program contains many functions, how do we avoid confusion about variable names? How do we avoid conflicts where we use the same variable name in two places and cause untold vicious errors and unfindable bugs? We could spend vast volumes of time keeping track of variable names we have used! What if we are importing a module? Is there documentation telling you all of the local variables so you can avoid problems? Why haven't we done this yet?

If we do things correctly, this fear will prove to be entirely unfounded and the need for any such documentation will not exist. Phew! Functions provide a means of managing the visibility of symbols in a program.

For good style and sanity, all programs you write should have the following elements in this order.

1. Put all import statements first.
2. Define all of your functions.
3. Put the "main routine" of your code last. This begins with the first unintended statement outside of any function.

It is possible to violate this discipline, and that can indeed leave you sorry. Using this scheme will give you the best possible management of symbol visibility. If you hear anyone say "global variable," immediately place a paper bag over your head and run away as fast as you can.

Using this system, here is what happens. Python starts and all of is built-in methods are visible. The code for each of your imports and functions is

processed. The functions are then visible for the entire lifetime of the program. In particular, when you are inside of any function, all the other functions in the module as well as all of the imported modules, and Python's built-in methods are visible. This is really what we want to be visible. We shall term this collection of visible stuff the *global symbol table*.

If your program has a very large number of functions, you should think about organizing them into modules and importing the modules, to keep the size of the global symbol table manageable by giving functions surnames.

We have the protection of the *Chinese Wall*: every function, during its execution, has its own private symbol table. We shall see that when a function is called, control of the program passes into that function. During this time, the function can only see user–defined variables in its private symbol table, and the items present in the global symbol table. However, it cannot see any of the variables inside of other functions. This relieves us from worrying about conflicts between variable names. This is also the reason you can use functions from other modules and be completely oblivious to the variable names used inside of them.

Consider this example, `twoFunctions.py`.

```
#!/usr/bin/python
def square(x):
    y = x*x
        return y
def cube(x):
        return x*x*x
```

When you are coding inside of a function, you can treat anything found in a function's argument list as a variable; these variables, which are properly called *parameters*, are only visible inside of the function. You can also create variables inside of functions; such variables are called *local variables*. Notice that the name `x` is used for both of these functions' arguments. Yet, because of the Chinese Wall, these `x` variables are separate. One is `cube`'s `x` and the other is `square`'s `x`. We will say that variables created inside of a function are *local* to that function. You can see that the function `square` has a local variable named `y`. Local variables inside of any function are invisible outside of that function. You can only interact with the function by passing data to its arguments. *There are no user-serviceable parts inside!*

The *scope* of a variable is the region of code in which the variable is visible. Python is said to have *function scope*, since the lifetime of a variable is from the time of its creation until the time at which the function containing it returns. When a function returns, its local variables are said to *go out of scope*.

Since the main routine contains all your code that is outside of any function and it constitutes a real function, variables created inside of main routine are

separate from any variable created inside of any function in your program. This, again, is the Chinese Wall at work.

The Chinese Wall is helpful, because it precludes namespace collisions in which two objects of the same name could cause confusion. Since any given function will only have a few variables and arguments inside of it, this relieves you of the massive headache of remembering all the variable names you ever used and keeping them from fighting with each other. It makes writing programs simpler, more fun, and basically ... possible.

The Chinese Wall is the reason that we can concern ourselves with the behavior of a function and we can forget about the specifics of its implementation.

In practice, functions should seldom be more than a screenful of code. Write functions that perform one specific task. This is the *Principle of Atomicity of Purpose*. Make functions as general and extensible as possible without rendering them confusing. There is a delicate art that manages the compromise between completeness of a function and efficiency and manageability.

Next, we discuss the call stack, we shall look at the life cycles of symbol tables in detail. There you will see very specifically how functions keep variable names separate.

8 Stack Frames and the Function Call Stack

The mechanism of the call stack will help you to understand how a chain of function calls builds and resolves. Understanding this process is absolutely key to understanding much of what happens later.

Although this section contains a lot of minute detail, it is important to read through it carefully to grasp the key ideas. Your patience will be rewarded when you program, and with insight into how things work in Python.

A *stack* of objects is a data structure containing a list of objects; think of the list as building from the bottom upwards. Visualize it just as you would a stack of books. Putting an item on the top of the stack called *pushing* the item onto the stack. Removing an item on the stack is called *popping* the item from the stack. A stack is either empty (no items on it) or nonempty. Simply looking at and reporting the item on top is called *peeking* at the stack. Trying to pop an item from an empty stack or trying to peek at it is an error.

A list in Python can act as a stack. Observe what we do here. A list has a `pop` method; this method removes the last item of the list as a side-effect and returns that list item as a return value.

```
>>> x = [1,2,3]
>>> x
[1, 2, 3]
```

```
>>> y = x.pop()
>>> y
3
```

Here is how to peek at the top of the stack; simply view the last item. This is called "peeking."

```
>>> x[-1]
2
```

Finally, here is how we can push, or add an item to the top of the stack. We just press the existing `append` method into service.

```
>>> x.append("seahorse")
>>> x
[1, 2, 'seahorse']
```

Function calls in Python are controlled by a stack called the *call stack*. Recall we spoke of the heap and the stack before; the call stack is the stack that stores function calls, and its stores the memory addresses of objects being pointed at by variables.

Visible Names At any time in a program's execution there is one source of names that is visible for the entire lifetime of the program; this is the global symbol table.

After your import statements, come your functions. These all are placed in the global symbol table as Python sees them; this is how functions can see each other from within their bodies. Variables local to the main routine go in the global symbol table too, but you should only read their values when outside of the main routine. Attempting to change the values attached to these variables within a function other than the main routine is a dangerous and foolish thing to do. Do not do it.

Can I see this? Yes, use `print(dir())` and you can see all visible names. If you import a module, seeing inside of it is simple. For example, if you have imported the module `os`, just enter `dir(os)` and you will see the goodness inside.

Remember, all Python programs begin executing in the main routine; prior to that, they are just reading things into memory. When a function is called, its *stack frame* or *activation record* is pushed onto the function call stack.

The main routine lives at the bottom of the call stack. It occupies a special stack frame called the *global frame*. Items in this frame are visible throughout a program's execution.

You should think of a stack frame as a box that contains information about the function, including function's symbol table and a pointer that bookmarks where the the caller was in its execution. This pointer allows the caller to resume execution exactly where it left off, and it is known as the *return address*.

When a function returns, its stack frame is popped from the function call stack, its return value, if any, is returned to its caller, and its stack frame is orphaned. All variables local to a function die when the function returns. This may orphan objects on the heap, which will make them susceptible to garbage collection. The function may have side effects, which can live beyond the function's (or program's) lifetime.

When the main routine returns, the global frame is popped from the call stack and program's execution terminates. Then operating system then discards the process generated by the program and deallocates its virtual memory.

The Chinese wall is enforced, because only the global symbol table and the variables in the currently working function are the only ones visible. This currently working function is the one whose frame is on the top of the call stack. Each function can only see its own private symbol table and the global symbol table.

We shall take a detailed look at this phenomenon with this sample program, `fAndG.py`.

```
def f(x):
    y = x*x
    return y*y
def g(x):
        z = f(x + 1)
    return z*4
#Here is the main routine
x = 2
print ("Whoopie!")
```

This program begins to run. The code for the functions `f`and `g`is recognized and then `f`and `g`, along with Python's built-in modules comprise the global symbol table. Then the main routine begins. The first line causes it its frame to look like this.

```
global frame:
    x -> 2
```

The second line causes the function `print` to be called, so a frame is created for it and it is pushed onto the stack. It will have an argument, whose name we don't know but will assume is `s`.

```
print:
    s -> "Whoopie!"
```

```
global frame:
    x -> 2
```

Then `print` causes `"Whoopie!"` to be put to stdout, and it returns. Its frame is then popped from the stack.

```
global frame:
    x -> 2
```

What's different now is that we are just beyond the `print` function call in the main routine. Since the main routine has run out of code, it has a tacit return and the program ends. This feature is common to all functions; they return automatically if they run out of code. If this occurs, they return the Python graveyard object `None` to the caller.

Since neither `f` nor `g` was called, they played no role in the execution of the program. They were visible and memory was allocated for their procedure but they remained unused. Now we will add a little code to make things more interesting.

```
def f(x):
    y = x*x
    return y*y
def g(x):
z = f(x + 1)
    return z*4
#Here is the main routine
x = 2
print ("Whoopie!")
print (f(x))
```

We start off just like last time after the first line.

```
global frame:
    x -> 2
```

Next, we call `print`. Our stack now looks like this.

```
print:
    s -> "Whoopie!"
```

```
global frame:
    x -> 2
```

Next, `print` puts its string to the screen and it runs out of code. It has a tacit return and control passes back to the global frame right where it left off.

```
global frame:
    x -> 2
```

The next line of code is `print(f(x))`. We push `print` onto the stack.

```
print:
    s -> f(2)
```

```
global frame:
    x -> 2
```

The `print` frame has no clue what to do; it sees a call to `f` while it is building its arguement list. What to do? Push `f`'s frame onto the stack with argument `x = 2`

```
f:
    x -> 2
```

```
print:
    s -> f(2)
```

```
global frame:
    x -> 2
```

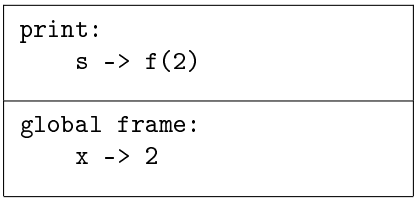
The function `f` is now controlling execution. The line `y = x*x` is now executed. We see that since `x` is pointing at 2, `x*x` evaluates to 4. That gets put in the frame's local symbol table.

```
f:
    x -> 2
    y -> 4
```

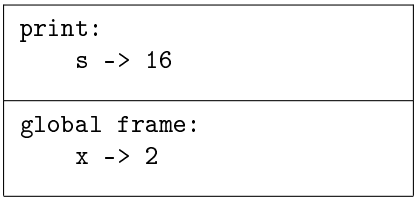
```
print:
    s -> f(2)
```

```
global frame:
    x -> 2
```

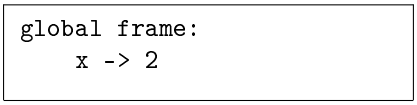
The next line is `f` is `return y*y`, so the value 4 is fetched from the local symbol table, `y*y` evaluates to 16, it is returned to the caller, and the frame for `f` is popped from the stack.



Where'd the 16 go? The return address in the stack frame for `f` tells exactly where to resume execution in `print`'s frame. That frame was just building it argument list, so that's what happens. Our diagram looks like this.



Now `print` knows exactly what to do with 16; it puts it to the screen, then its frame is popped. We are now here.



The return address puts us just beyond the last line of code in the main routine. The program terminates. What got put to the screen?

```
Whoopie!
16
```

Next we will call the function `g` and see the enchilada in its peppery entirety.

```
def f(x):
    y = x*x
    return y*y
def g(x):
        z = f(x + 1)
    return z*4
#Here is the main routine
x = 2
print "Whoopie!"
print (g(x))
```

In the first two lines of the main routine, the local variable `x` is set to `2`, `print` gets pushed onto the stack, then `Whoopie!` is put to stdout, and then `print` gets popped from the stack, just as we saw in the previous example. This leaves our stack looking like this.

```
global frame:
    x -> 2
```

Next we come to this line `print(g(x))`. We push `print` onto the stack.

```
print
    s -> g(2)
```

```
global frame:
    x -> 2
```

Next, the function g is called for the value `x = 2`. The call stack now looks like this

```
g:
    x -> 2
```

```
print
    s -> g(2)
```

```
main routine:
    x -> 2
```

The next line of `g` has the instruction `z = f(x + 1)`; the expression `x + 1` evaluates to 3 and the call `f(3)` is made.

```
f:
    x -> 3
```

```
g:
    x -> 2
```

```
print
    s -> g(2)
```

```
main routine:
    x -> 2
```

Now `f`'s frame is in control since it's on top. The line `z = f` The call `f(3)` initializes `f`'s `y` to `x*x`, which evaluates to 9. We therefore augment its symbol table int he next stack diagram.

```
f:
    x -> 3
    y -> 9
```

```
g:
    x -> 2
```

```
print
    s -> g(2)
```

```
main routine:
    x -> 2
```

The next line to execute is in `f`, which is `return y*y`. This pops `f` from the stack and returns 81 to the caller. We now have this.

```
g:
    x -> 2
    z -> 81
```

```
print
    s -> g(2)
```

```
main routine:
    x -> 2
```

The caller, `g`, resumes where it left off; it must return `4*y`, which evaluates to 324. That value goes back to `print` which was just building its argument list. We now have this.

```
print
    s -> 324
```

```
main routine:
    x -> 2
```

Now `print` puts 324 to the screen and it runs out of code. This triggers a tacit return and its framea is popped.

```
print
    s -> 324
```

```
main routine:
    x -> 2
```

The main routine now ends, as does program execution. Here is what got put to the screen.

```
Whoopie!
324
```

8.1 A Summary

Python reads all imported modules into program memory. It then places all functions you have defined in the global symbol table. Execution of a program begins in the main routine. If there is no main routine, the program will not do anything. When a function is called, its stack frame is pushed onto the call stack. The only data visible are the data internal to that function, and global symbol table.

The function can see its private symbol table and its instruction pointer that knows what instruction it is currently executing. When a function returns, its return value (if any) is returned to the caller and its frame is popped off the top of the stack.

The next lower function on the stack gets control of the program and resumes right where it left off; a record of this is embodied in the frame's instruction pointer.

The call stack is a LIFO (last in first out) system. When a function call is made, we say the stack is *building*; when one returns the stack is *unwinding*. If a function has no `return` statement, it automatically returns when its code reaches an end; otherwise, *A function's control terminates as soon as it encounters a* `return` *statement*. Throughout the lifetime of a program, the call stack may build and partially unwind many times. It is only when the main routine returns that the program's life ends.

8.2 A Terminology Roundup

There are some terms we use for functions; we will list them here for your convenience.

pass When you call a function as we did with `hypot(5,12)`, you are said to be *passing* the values `5` and `12` to the function. You can pass literals, as we did here, variables, or more generally, you can pass expressions to functions. For instance `hypot(2 + 2, 4 - 1)` will return 5.0. These expressions can contain variables. Everything is evaluated before it is passed to the function; that is the value ultimately passed to a function.

This includes variables; as a result, a function gets a copy of the value a variable holds passed it as an argument, not a copy the object the variable points at. What is passed, is in fact, the memory address of the object. In programming language terminology, Python is a purely *pass–by–value* language.

Do not confuse this with the Python keyword `pass`, which is the do-nothing statement.

argument list The argument list for a function is just the comma-separated list of stuff that appears in the parentheses after the `def` declaration. Each item in the list, needless to say, is called an "argument." The names given arguments are called *parameters* or `formal parameters`. The arguments supplied by the caller are passed to the parameters in the callee.

call You are *calling* a function when you use it. If you do something with `hypot(5,12)` you are said to be *calling* the `hypot` function.

8.3 The Python Ternary Operator

This is a convenience that can make the writing of very simple if-else progressions very succinct. You will see it in OPC (other peoples' code) so you need to be aware of it. You may even acquire a taste for it yourself.

We show the absolute value function written in this style

```
def absoluteValue(x):
    return x if x >= 0 else -x
```

what you see here is the Python ternary operator at work. Its general form is

```
exprTrue if predicate  else exprFalse
```

The predicate must be... a predicate, i.e. a boolean valued expressoin. If the predicate evaluates to true, the expression evaluates to `exprTrue`. Otherwise, it evaluates to `exprFalse`.

Programming Exercises Try wrting these with if-else and with the ternary operator. Use the Mean Teacher technique to create these functions. Make sure you test each branch of execution and any border cases.

1. Write this function.

   ```
   def parroty(n):
       """Precondition: n is a nonnegative integer
   Postcondition:  returns the string "EVEN" if n is even and "ODD"
   otherwise.  """
       return TODO
   ```

2. Consider the function

$$f(x) = \begin{cases} -(x+4) & \text{if } x < -4 \\ 0 & \text{if } -4 \leq x < 4 \\ x - 4 & \text{if } x \geq 4. \end{cases}$$

Code it with simple ifs and possibly an else. Can you implement it using the ternary operator?

9 Some Types are Smarter than Others

For objects of all Python types, a variable stores its object's location in memory. It does not store the object itself. Being aware of this will make some seemingly confusing issues very clear later on.

All variables in Python point at *objects* in memory. An object has three attributes:

- **identity** An object is a well–defined region of memory. It is. Variables can point at objects; if an object is not pointed at by any variable, it is said to be *orphaned*. Memory occupied by orphaned objects is reclaimed by Python's garbage collector.
- **state** An object has a current *state*; for a string, the state is just the sequence of characters stored by the string. Objects often hold data; these data reflect the state of the object. The state of an object is what the object "knows."
- **behavior** We saw in the exercises that a string can return a copy of itself with all alphabetical characters capitalized. This is achieved by a *string method*. Methods are functions that are attached to objects that can accomplish tasks. An object of a given type can have zero or more methods. The minimal behavior of any object is that it knows its type. The methods that an object can perform constitute is behaviors.

Let's show a concrete example with a brief interactive Python session. We shall discuss some of the methods which strings have. Suppose you have a string named `stringName`. The proper usage for invoking any string method on `stringName` is

```
stringName.methodName(arguments....)
```

In an earlier exercise, we saw that strings have two methods named `upper` and `lower`; these methods respectively, return a copy of the string with all alphabetical characters to upper or lower case. Non-alphabetical characters are ignored. These methods do not change the string they are called upon.

```
>>> x = "Zolyvars"
>>> x.upper()
'ZOLYVARS'
>>> x.lower()
>>> 'zolyvars'
'zolyvars'
>>> "EwrEW####@##".upper()
'EWREW####@##'
>>> x = "Zolyvars"
Zolyvars
```

Grammatically the `x.` is `x` in the genitive case: you can parse it as "`x`'s". Hence, when you say `x.lower()`, you are saying, "Call `x`'s method lower." Since `x` is a string, its `lower` method returns a copy of the string in lower–case letters.

It is important to remember that methods are *functions* that are attached to objects; this explains the empty parentheses after `lower` and `upper`. Forgetting the parentheses leads to ugly-looking output; try it for yourself and see! Since no further information is required for these functions to do their jobs, they have empty argument lists. Since they are invoked by typing `x.`, you know that these methods have complete access to the state of `x`. Methods provide a means by which you can send a message to an object. You call methods on an object via the object's name; that is, you use the variable name currently pointing at the object.

Let us now look at another string method, `find` which allows you to search for one string inside of another. The `find` method returns an integer that is the index of the location of the sought substring. If it does not find the substring, it returns a `-1`. We begin with a little interactive session.

```
>>> x = "abcdefghijklmnopqrstuvwxyz"
>>> x.find("p")  #call x's find method for "p"
15
```

Observe that `find` takes an argument that is a string. Since it found the string `"p"` starting at index `15` in `x`, it returned a `15`. Now we will send `find` snark-hunting.

```
>>> x.find("1")
-1
```

We got back a `-1` since we did not find the numerical character `1` in the string `x`. Next we find the string "ijk" starting at index 8 of x.

```
>>> x.find("ijk")
8
```

A substring must be contiguous. It is not enough just for its characters to appear in order. Here we are told that `"aeg"` is not found.

```
>>> x.find("aeg")
-1
```

The `find` method is quite a bit richer than its sibling methods `upper` and `lower`. The `find` method requires a string be passed it as an argument. The call `x.find("p")` says, "Hey `x`, find the index in yourself at which the string 'p' starts." Since the string "p" is present in `x` at index 15, that index is returned to the caller (your interactive session). On the next line we attempt to find the numerical character "1" in the string `x`; since it is not present, `find` returns the *sentinel value* -1 to tell the caller it did not find the requested string in `x`. The call `x.find("ijk")` found the string "ijk" inside of `x` at location 8. When you ask `find` to find, it looks for a *contiguous* substring. The call `x.find("aeg")` returns the sentinel value -1, because `"aeg"` is not a contiguous substring of `x`.

There are numerous methods for strings. To see them all, visit the Python site find §3.6.1, String Methods. You will need to to this to work the programming exercises given below.

Programming Exercises Open the String Methods documentation and an interactive Python session. Write the functions described below by using the String Methods class. Develop test cases *before* placing any code in the function. Give each function a method stub. Remember to compile as you go make it easy to find and squash errors. Some test cases are provided for you.

```
def countCharsIn(c, someString):
    """precondtion: c is a one-character string, someString is
a string.
postcondition:  returns the number of times c appears in
someString"""
def hasExtension(fileName, ext):
   """precondition:  filename is a string that is a file name
and ext is a file extension.
postcondition:  return True if fileName ends with a period (.),
then the extension.  Returns false if fileName contains any
spaces in it regardless of its suffix."""
def chompEnds(c, someString:
   """precondition:  c is a one-character string that appears
at least twice in someString.
postcondition:  returns the substring of someString beginning
after the first instance of c and ending before the last
instance of c in someString.  """
```

```
print ("countChars(\"a\", \"alabaster\") = ",
countChars("a", "alabaster"), "expected:  3")
print ("hasExtension(\"cat.cpp\", \"cpp\") = ",
countChars("cat.cpp", "cpp"), "expected:  True")
print ("hasExtension(\"siamese cat.cpp\", \"cpp\") = ",
countChars("siamese cat.cpp", "cpp"), "expected:  False")
```

10 Format Strings

Format strings can give your test code a simpler and cleaner appearance. We show an example then deconstruct.

```
>>> x = "f(%d) = %d " %(10, 10*10)
>>> print(x)
f(10) = 100
>>> import math
>>> y = "sin(%5.4f) = %5.4f" %(math.pi, math.sin(math.pi))
>>> print(y)
sin(3.1416) = 0.0000
```

The variable x winds up pointing at a string. The construct `%d` is called a *format specifier*; in this case we use `%d` for an integer (or long). The format specifier is a a placeholder that says, "expect a decimal (base 10) integer here." After the close quote on the string, you put a `%` sign then a tuple. The length of the tuple must match the number of format specifiers. The items in the tuple are dropped into place in order. The tuple may contain objects of various types. The whole thing is formatted as a string in the end. If you are using only one format specifier, you may use a single variable after the outer `%`. How do you make a `%`? Use `%%`.

In the string `y`, we used format specifier `%5.4f`. The `f` means, "Expect a floating point number." the 5.4 means give the number a minimum width of 5 and show 4 figures after the decimal. Format specifiers begin with a `%`. Then you can place a flag which can be any of the following.

#	Use alternate form.
0	Pad with zeroes for numerical values
+	Put a + or a - in front of numerical values
-	Left justify within width.

You can then specify a minimum width (an integer) and then for numbers with decimals, use a .(integer) to specify how many places to show beyond the

decimal point. Finally you specify the type of datum to expect. This table shows the most common ones.

`d`	base 10 (decimal) integer
`o`	base 8 (octal) integer
`x`	base 16 (hex) integer, 0x prefix, lower case
`X`	base 16 (hex) integer, 0x prefix, upper case
`e, f`	Floating point decimal number, lower case e
`E`	Floating point decimal number, upper case E
`g`	Floating point decimal number, lower case e, if scientific notation is used
`G`	Floating point decimal number, upper case E, if scientific notation is used
`c`	a single character
`s`	a string