Python 101 (part 6): Hedgehogs, Pythons And Funky Chameleons

By Vikram Vaswani

Table of Contents

A Little Knowledge

If you've been paying attention these last few weeks, you should now know enough about Python to begin writing your own programs in the language. As a matter of fact, you might even be entertaining thoughts of cutting down your weekly visits to this Web site and doing away with this tutorial altogether.

Well, a very wise man once said that a little knowledge was a dangerous thing...and so, as your Python scripts become more and more complex, you're going to bump your head against the principles of software design, and begin looking for a more efficient way of structuring your Python programs.

Over the course of the next few articles, I'm going to take a look at the Python framework for basic software abstractions like functions and classes. Specifically, this tutorial will show you how to build your own functions, thereby adding an element of reusability to your code; you'll learn the difference between arguments and return values, find out how to handle function objects, and even visit some of the more important built−in methods.

Let's get cracking!

Cheating The Taxman

Ask a geek to define the term "function", and he'll probably tell you that a function is "a block of statements that can be grouped together as a named entity." Since this definition raises more questions than answers (the primary one being, what on earth are you doing hanging around with geeks in the first place), I'll simplify that definition a little: a function is simply a set of program statements which perform a specific task, and which can be "called", or executed, from anywhere in your program.

Every programming language comes with its own functions − you'll remember my frequent use of the print(), input() and range() functions from previous articles in this series − and typically also allows developers to define their own functions. For example, if I had a series of numbers, and I wanted to reduce each of them by 20%, I could pull out my calculator and do it manually....or I could write a simple Python function called cheatTheTaxman() and have it do the heavy lifting for me.

There are two important reasons why functions are a Good Thing. First, a user−defined function allows you to separate your code into easily identifiable subsections, thereby making it easier to understand and debug. And second, a function makes your program modular, allowing you to write a piece of code once and then re−use it multiple times within the same program.

Talking Movies

Let's take a simple example, which demonstrates how to define a function and call it from different places within your Python program:

```
#!/usr/bin/python

# define a function
def greatMovie():
print "Star Wars"

# main program begins here
print "Question: which is the greatest movie of all time?"

# call the function
greatMovie()

# ask another question
print "Question: which movie introduced the world to Luke
Skywalker, Yoda
and Darth Vader?"

# call the function
greatMovie()
```

Now run it − you should see something like this:

```
Question: which is the greatest movie of all time?
Star Wars
Question: which movie introduced the world to Luke Skywalker,
Yoda and
Darth Vader?
Star Wars
```

Let's take this line by line. The first thing I've done is define a new function with the "def" keyword; this keyword is followed by the name of the function (and optionally, one or more arguments). All the program code attached to the function is then indented within this block − this program code could contain loops, conditional statements, or calls to other functions. In the example above, my function has been named "greatMovie", and only contains a call to Python's print() function.

Here's the typical format for a function:

```
def function_name(optional function arguments):
'comment'
statement 1...
statement 2...
.
.
.
statement n...
```

The comment line above allows the developer to optionally include a description of the function; if I wanted to be really verbose, I could modify the function definition in the example above to read

```
# define a function
def greatMovie():
'this prints the name of my favourite movie'
print "Star Wars"
```

Call Me Sometime

Of course, defining a function is only half of the puzzle − for it to be of any use at all, you need to invoke it. In Python, as in Perl, PHP and a million other languages, this is accomplished by "calling" the function by its name, as I've done in the last line of the example above. Calling a user−defined function is identical to calling a built−in Python function like print() or type().

It should be noted that Python will barf if you attempt to call a function before it has been defined − look what happens when I try the following example:

```
#!/usr/bin/python

# main program begins here
print "Question: which is the greatest movie of all time?"

# call the function
greatMovie()

# ask another question
print "Question: which movie introduced the world to Luke
Skywalker, Yoda
and Darth Vader?"

# call the function
greatMovie()

# define a function
def greatMovie():
'this prints the name of my favourite movie'
print "Star Wars"
```

Here's the output:

```
Question: which is the greatest movie of all time?
Traceback (innermost last):
File "./test.py", line 8, in ?
greatMovie()
NameError: greatMovie
```

Another important thing to remember about functions is that they're, at heart, objects...just like everything else in Python. Consequently, they can be assigned to other objects, passed as values or data structures, and otherwise treated as though they were any other Python object.

The following example demonstrates this object−oriented behaviour, by assigning one function to another, and then using the new assignment to invoke the function.

```
#!/usr/bin/python

# define a function
def greatMovie():
'this prints the name of my favourite movie'
print "Star Wars"

# assign function as an object
myFavouriteMovie = greatMovie

print "Question: which is the greatest movie of all time?"

# call function by its new name
myFavouriteMovie()
```

And the output is:

```
Question: which is the greatest movie of all time?
Star Wars
```

Return To Me

Usually, when a function is invoked, it generates a "return value". This return value is either a built−in Python object called "None", or a value explicitly returned via the "return" statement. I'll examine both these a little further down − but first, here's a quick example of how a return value works.

```
#!/usr/bin/python

# define a function
def tempConv():
celsius = 35
fahrenheit = (celsius * 1.8) + 32
return fahrenheit

# assign return value to variable
result = tempConv()

print "35 Celsius is", result, "Fahrenheit"
```

The output is:

```
35 Celsius is 95.0 Fahrenheit
```

In this case, the value of the last expression evaluated within the function is assigned to the variable "result" when the function is invoked from within the program. Note that the return value must be explicitly specified within the function definition via the "return" statement; failure to do so will cause the function to return a null object named "None".

To illustrate this, consider the output if the "return" statement is eliminated from the function definition above.

```
35 Celsius is None Fahrenheit
```

Tall, Dark And Handsome

Return values from a function can even be substituted for variables anywhere in a program.

```
#!/usr/bin/python

# define a function
def marryMe():
if tall == 1 and dark == 1 and handsome == 1:
return "Yes!"
else:
return "I'm sorry, I just don't feel the same way about
you."

# set some variables
tall = 1
dark = 1
handsome = 1

# pop the question
print "Will you marry me?"

# use return value in a function call
print(marryMe())
```

Notice how the function is able to read the values of variables defined outside the function; this is related to variable scope in Python, and I plan to discuss it in detail a little further down.

Return values need not be numbers or strings alone − a function can just as easily return a list, dictionary or tuple, as demonstrated in the following example:

```
Python 1.5.2 (#1, Aug 25 2000, 09:33:37) [GCC 2.96 20000731
(experimental)] on linux-i386
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>> def returnList():
... return ["Huey", "Dewey", "Louie"]
...
>>> def returnTuple():
... return "macaroni", "spaghetti", "lasagne", "fettucine"
...
>>> def returnDictionary():
```

```
... return {"new hope":"Luke", "teacher":"Yoda", "bad
guy":"Darth"}
...
>>> print type(returnList())

>>> print type(returnTuple())

>>> print type(returnDictionary())

>>>
```

Arguing Your Case

Now, if you've been paying attention, you've seen how functions can help you segregate blocks of code, and use the same piece of code over and over again, thereby eliminating unnecessary duplication. But this is just the tip of the iceberg...

The functions you've seen thus far are largely static, in that the variables they use are already defined. But it's also possible to pass variables to a function from the main program − these variables are called "arguments", and they add a whole new level of power and flexibility to your code.

To illustrate this, let's go back a couple of pages and revisit the tempConv() function.

```
def tempConv():
celsius = 35
fahrenheit = (celsius * 1.8) + 32
return fahrenheit
```

As is, this function will always calculate the Fahrenheit equivalent of 35 degrees Celsius, no matter how many times you run it or how many roses you buy it. However, it would be so much more useful if it could be modified to accept *any* value, and return the Fahrenheit equivalent of that value to the calling program.

This is where arguments come in − they allow you to define placeholders, if you will, for certain variables; these variables are provided at run−time by the main program.

Let's now modify the tempConv() example to accept an argument.

```
#!/usr/bin/python

# set a variable − this could be obtained via input()
alpha = 40

# define a function
def tempConv(temperature):
fahrenheit = (temperature * 1.8) + 32
return fahrenheit

# call function with argument
result = tempConv(alpha)

# print
print alpha, "Celsius is", result, "Fahrenheit"
```

And now, when the tempConv() function is called with an argument, the argument is assigned to the placeholder variable "temperature" within the function, and then acted upon by the code within the function definition.

It's also possible to pass more than one argument to a function − as the following example demonstrates.

```
>>> def addIt(item1, item2, item3):
... result = item1 + item2 + item3
... return result
...
>>> addIt("hello", "−", "john")
'hello−john'
>>>
```

Note that there is no requirement to explicitly specify the type of argument(s) being passed to a function − since Python is a dynamically−typed language, it can automatically identify the variable type and act on it appropriately.

Enter The Funky Chameleon

The order in which arguments are passed to a function is important − the following example assumes that the name is passed as the first argument, and the age as the second.

```
>>> def User(name, age):
... print "Hello, " + name + ". How strange that we are both "
+ age +
" years old."
...
>>> User("Funky Chameleon", "23")
Hello, Funky Chameleon. How strange that we are both 23 years
old.
>>>
```

If you get the order wrong, you may get unexpected results.

```
>>> User("23", "Funky Chameleon")
Hello, 23. How strange that we are both Funky Chameleon years
old.
>>>
```

However, it's possible to override this behaviour in the function call itself, by specifying the arguments as name−value pairs.

```
>>> User(age="23", name="Funky Chameleon")
Hello, Funky Chameleon. How strange that we are both 23 years
old.
>>>
```

Additionally, you can specify certain arguments to be optional, by setting default values for them in the function definition. Consider the following example, which requires two arguments, a URL and a language to display it in. If the language argument is omitted, a default value is used.

```
>>> def displayURL(url, lang="English"):
... print "Displaying URL in " + lang
...
>>> displayURL("http://www.melonfire.com", "Spanish")
Displaying URL in Spanish
>>> displayURL("http://www.melonfire.com")
Displaying URL in English
>>>
```

Note, however, that the list of optional arguments must follow, not precede, the list of required arguments in the function definition.

All these examples have one thing in common − the list of arguments is fixed. Look what happens if you pass an extra argument to the User() function above,

```
>>> User("Harry the Hedgehog", "2", "male")
Traceback (innermost last):
File "", line 1, in ?
TypeError: too many arguments; expected 2, got 3
>>>
```

or miss out on a parameter when calling the function.

```
>>> User("Harry the Hedgehog")
Traceback (innermost last):
File "", line 1, in ?
TypeError: not enough arguments; expected 2, got 1
>>>
```

To handle situations such as there, Python functions come with a built−in secret weapon − the *variable argument. Consider the following function definition:

```
>>> def User(name, age, *more):
... print "Name:", name
... print "Age:", age
... for x in more:
```

```
... print "Additional argument:", x
...
```

In this case, extra arguments passed to the function will be stored in the variable "more" as a tuple; these arguments can then be extracted and processed using standard tuple operators or methods.

```
>>> User("Harry the Hedgehog", "2", "male")
Name: Harry the Hedgehog
Age: 2
Additional argument: male
>>> User("Harry the Hedgehog", "2", "male", "angry", "lives
next door")
Name: Harry the Hedgehog
Age: 2
Additional argument: male
Additional argument: angry
Additional argument: lives next door
>>>
```

Here's an example which demonstrates how this works.

```
>>> def Product(*numlist):
... product = 1;
... for x in numlist:
... product = product * x
... return product
...
>>>
```

In this case, you can call the Product() function with as many arguments as you like; the "for" loop will iterate through the "numlist" tuple and multiply each one by the previous result.

```
>>> Product(2, 3)
6
>>> Product(2)
2
>>> Product(2, 10, 10, 1)
```

```
200
>>> Product(2, 10, 10, 6326, 236237238923490234)
OverflowError: integer literal too large
>>>
```

Oh, well...

Flavour Of The Month

Let's now talk a little bit about the variables used within a function, and their relationship with variables in the main program. Unless you specify otherwise, the variables used within a function are local − that is, the values assigned to them, and the changes made to them, are restricted to the function space alone.

For a clearer example of what this means, consider this simple example:

```
#!/usr/bin/python

# set a variable outside the function
flavour = "tangerine"

# alter the variable within the function
def changeFlavour():
flavour = "raspberry"

# before invoking function
print "(before) Today's flavour is", flavour

# invoke function
changeFlavour()

# after invoking function
print "(after) Today's flavour is", flavour
```

And here's what you'll see:

```
(before) Today's flavour is tangerine
(after) Today's flavour is tangerine
```

As you can see, assignment to a variable within a function does not alter the same variable outside the function...unless you declare it with the "global" keyword.

```
#!/usr/bin/python

# set a variable outside the function
```

```
flavour = "tangerine"

# alter the variable within the function
def changeFlavour():
global flavour
flavour = "raspberry"

# before invoking function
print "(before) Today's flavour is", flavour

# invoke function
changeFlavour()

# after invoking function
print "(after) Today's flavour is", flavour
```

And now, when you run it,

```
(before) Today's flavour is tangerine
(after) Today's flavour is raspberry
```

The "global" keyword tells Python that changes made to the variable should be applied globally, and should not remain localized to the function space alone.

Note, however, that the "global" keyword is only used when you plan to alter a global variable; if all you need to do is read a global variable, Python can access its value without any requirement to first declare it global.

```
#!/usr/bin/python

# set a variable outside the function
flavour = "tangerine"

# use the variable within the function
def whatFlavour():
print "Today's flavour is", flavour

# invoke function
whatFlavour()
```

In this case, even though the variable is declared outside the function, Python can still access (though not change) its value.

```
Today's flavour is tangerine
```

Hip To Be Square

Python comes with a couple of built−in functions to help you work with your own user−defined functions. And yes, I realize that that sentence didn't make all that much sense...

Perhaps an example will make it clearer. Consider the Python apply() function. The sole raison d'etre of apply() is to run another function with a set of arguments. Consequently, if I had a simple function named whoAmI(),

```
>>> def whoAmI(name):
... print "Your name is", name
...
>>>
```

the following statements would be equivalent:

```
>>> apply(whoAmI, ["Batman, Dark Knight"])
Your name is Batman, Dark Knight
>>> whoAmI("Batman, Dark Knight")
Your name is Batman, Dark Knight
>>>
```

This might seem pointless at first glance; however, apply() can come in handy if functions or function arguments are generated dynamically by your program, and you are unsure of their values.

Note that the second argument to apply() must always be a sequence − list, tuple et al − containing a list of function arguments.

The map() function runs a function on every element of a list, and places the results in another list. Consider the following function, which squares the argument passed to it.

```
>>> def hipToBeSquare(num):
... return num**2
...
>>>
```

And here's a list of numbers

```
>>> numList = [2, 5, 9, 11, 13, 20]
>>>
```

Look what happens when I put them both together with map().

```
>>> resultList = map(hipToBeSquare, numList)
>>> resultList
[4, 25, 81, 121, 169, 400]
>>>
```

Similar to map() is the filter() function; it runs a function on a list, and creates a subset of those items which returned true. So, if I had a function which checked whether or not a number was prime (perhaps you remember this from a previous article?) and a mixed list of prime and non−prime numbers, filter() would come in handy to separate the two.

```
#!/usr/bin/python

# returns 1 if prime, 0 if non−prime
def isPrime(num):
for count in range(2,num):
if num % count == 0:
return 0
break
else:
return 1

# list of numbers
numList = [4, 3, 11, 18, 200, 171, 67, 5]

# filter and print result list
print filter(isPrime, numList)
```

And the output is

```
[3, 11, 67, 5]
```

And that just about concludes this article. This time, you've taken a big step towards better software design, by learning how to abstract out parts of your Python code into reusable functions. You've learned how to add flexibility to your functions by allowing them to accept different arguments, and how to obtain return values from them. Finally, you've learned how Python treats variables inside and outside functions, and tried out three of the most common built−in helper functions available in Python.

In the next article, I'll be expanding on this theme by discussing Python modules, constructs which add a whole new level of reusability to this powerful programming language. See you then!