Python 101 (part 2): If Wishes Were Pythons

By Vikram Vaswani

Table of Contents

Riding The Snake

You must be tired of hearing this by now − but, since Python is based on an object−oriented framework, it's only natural that the language contain a few...well, objects. And over the course of the next few pages, I'm going to introduce you to two of the most commonly−used basic object types in Python: numbers and strings.

That's not all, though − I'll also be demonstrating how to assign values to Python variables on the basis of user input, taking a long, hard look at the different operations possible with each object type, and explaining some basic conditional statements. Hang on to your hats, kids...this is going to be one wild ride!

Tax Evasion

The first − and simplest − object type you're likely to encounter is the number. I've already shown you a few examples of how Python treats numbers last time − but here's a quick refresher anyway:

```
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
>>> print 24+1
25
>>> print 12*10
120
>>> print 65/5
13
>>> print 15−9
6
>>>
```

There are four basic number types available in Python:

Integers: These are plain−vanilla numbers like 23, 5748, −947 and 3; they are also the most commonly−used type.

Floats: These are typically decimal numbers, although Python also allows you to use the scientific notation to represent them − for example, 3.67, 98.573 or 5.347e−17.

Long integers: These are integer values which are too large to be handled by the regular Integer type − for example, 45734893498L or 53267282982L. Note the uppercase "L" appended to the end of a long integer, which makes this type easier to identify.

Complex numbers: You may remember these from algebra class (I don't, but that's probably because I slept my way through college) − 2.5+0.3j, 4.1−03.j and so on. These numbers can be broken up into "real" and "imaginary" parts...the "imaginary" part is what you bank, while the "real" part is what the government takes away in taxes.

Putting Two And Two Together

Most of the time, you'll be working with integers and floats; the other two types are typically used for specialized applications. And as the example above demonstrates, Python allows you to add, multiply, divide and otherwise mess with these numbers via simple arithmetic operators. Here's an example which demonstrates the important ones:

```
>>> alpha = 10
>>> beta = 2
>>> # standard stuff
... sum = alpha+beta
>>> sum
12
>>> difference = alpha-beta
>>> difference
8
>>> product = alpha*beta
>>> product
20
>>> quotient = alpha/beta
>>> quotient
5
>>> # non-standard stuff
... remainder = alpha%beta
>>> remainder
0
>>> exponent = alpha**beta
>>> exponent
100
>>>
```

As with all other programming languages, division and multiplication take precedence over addition and subtraction, although parentheses can be used to give a particular operation greater precedence. For example,

```
#!/usr/bin/python
print(10 + 2 * 4)
```

returns 18, while

```
#!/usr/bin/python
print((10 + 2) * 4)
```

returns 48.

It should be noted at this point that Python does not support the popular auto−increment (++) and auto−decrement (−−) operators you may have used in Perl, PHP and JavaScript.

There are also some built−in functions you can use with Python numbers − the most useful are abs() and pow(), which return the absolute value of a number and raise one number to the power of another. Take a look.

```
>>> alpha = −89
>>> abs(alpha)
89
>>> pow(3,2)
9
>>>
```

Q &A

Thus far, you've been assigning values to your variables at the time of writing the program itself (referred to in geek lingo as "design time"). In the real world, however, many variables are assigned only after the program is executed (referred to as "run time"). And so, before moving on to strings, I'd like to take some time out to demonstrate a Python program which allows you to ask the user for input, assign this input to a variable, and then perform specific actions on that variable.

```
#!/usr/bin/python

# ask a question...
number = input("Gimme a number! ")

# process the answer...
square = number ** 2

# display the result
print "The square of", number, "is", square
```

If you try it out, you'll see something like this:

```
$
Gimme a number! 4
The square of 4 is 16
$
```

The first line of the program prints a prompt, asking the user to enter a number; this is the purpose of the input() function call. Once the user enters a number, it is raised to the power 2 and the result is assigned to the a new variable. Finally, the print() statement is used to output the result to the terminal.

Playing It Again...And Again...And Again...

Next up, strings. String values can be assigned to variables in the same manner as numeric values, except that strings are always surrounded by single or double quotes.

```
>>> name = "popeye"
>>> snack = 'spinach'
```

Note that if your string contains quotes, it's necessary to escape them with a backslash.

```
>>> string = "...and so he said, \"backslash me, knave!\""
>>> print string
...and so he said, "backslash me, knave!"
>>>
```

Python also allows you to create strings which span multiple lines, by enclosing them in triple quotes (""""). The original formatting of the string, including newlines and whitespace, is retained when such a string is printed.

```
>>> welcome = """Hello, and how
... are you today? My name is Vikram
... and I will be your guide to the wonderful and
... mysterious world of Python."""
>>> print welcome
Hello, and how
are you today? My name is Vikram
and I will be your guide to the wonderful and
mysterious world of Python.
>>>
```

Strings can be concatenated with the + operator, in much the same way as numbers are added. Note that although the operator is the same, Python possesses enough intelligence to decide whether the operation is addition or concatenation on the basis of the object type.

```
>>> a = "ding"
>>> b = "dong"
>>> c = "bell"
>>> abc = a + b + c
>>> abc
'dingdongbell'
>>>
```

Strings placed next to each other are automatically concatenated.

```
>>> print "ding" "dong" "bell"
dingdongbell
>>> # this is equivalent
>>> print "ding" + "dong" + "bell"
dingdongbell
>>>
```

A quick aside on the print() function call here: typically, if arguments passed to print() are separated with commas, Python automatically prints them one after the other (separated by a single space) and adds a line break at the end.

```
>>> print "ding", "dong", "bell"
ding dong bell
>>>
```

Python also includes a repetition operator, the asterisk (*), used to repeat a string a number of times.

```
>>> famousoldmovieline = "Play it again, Sam!\n"
>>> print famousoldmovieline * 4
Play it again, Sam!
Play it again, Sam!
Play it again, Sam!
Play it again, Sam!
>>>
```

Sliced And Diced

In Python−lingo, strings are an "immutable" object type, which means that they cannot be changed in place; the only way to alter a string is to create a new string with the changes. Python comes with powerful built−in operators designed to make it easier to extract subsections of a string, and thereby create new strings.

For example, to extract the letter "b" from the string "hobgoblin", I could use

```
>>> str = "hobgoblin"
>>> str[2]
'b'
>>>
```

This is referred to as "slicing"; the square braces [and] specify the starting and ending location (or "index") for the slice. Note that the first character is referred to by index 0.

You can extract a substring by specifying two indices within the square braces...

```
>>> str = "hobgoblin"
>>> str[3:9]
'goblin'
>>>
```

...and watch what happens when I omit either one of the two indices.

```
>>> str = "hobgoblin"
>>> str[3:]
'goblin'
>>> str[:7]
'hobgobl'
>>>
```

You can also use negative indices, to force Python to begin counting from the right instead of the left.

```
>>> str = "hobgoblin"
>>> str[-6]
'g'
>>> str[-6:]
'goblin'
>>>
```

The built-in len() function can be used to calculate the number of characters in a string,

```
>>> str = "hobgoblin"
>>> len(str)
9
>>>
```

while the "in" and "not in" operators can be used to test for the presence of a particular character in a string. A match returns 1 (true), while a failure returns 0 (false).

```
>>> str = "hobgoblin"
>>> "g" in str
1
>>> "x" in str
0
```

Comparing Apples And Oranges

The examples you've just seen are very rudimentary. To really add some punch, you need to know how to construct what the geeks call a conditional statement. And the very basis of a conditional statement is comparison − for example, "if this is equal to that, do thus and such".

Python comes with a bunch of useful operators designed specifically for use in conditional statements. Here's a list:

```
Assume delta = 12 and omega = 9

Operator What It Means Expression Evaluates To
-------------------------------------------------------------------------

== is equal to delta == omega False

!= is not equal to delta != omega True

> is greater than delta > omega True

< is less than delta < omega False

>= is greater than or equal to delta >= omega True

<= is less than or equal to delta <= omega False
```

These comparison operators can be used for both strings and numbers. A positive result returns true (1), while a negative result returns false (0).

An important point to note − and one which many novice programmers fall foul of − is the difference between the assignment operator (=) and the equality operator (==). The former is used to assign a value to a variable, while the latter is used to test for equality in a conditional expression. So

```
a = 47;
```

assigns the value 47 to the variable a, while

```
a == 47
```

tests whether the value of a is equal to 47.

If Only...

Why do you need to know all this? Well, comparison operators come in very useful when building conditional expressions − and conditional expressions come in very useful when adding control routines to your code. Control routines check for the existence of certain conditions, and execute appropriate program code depending on what they find.

The first − and simplest − decision−making routine is the "if" statement, which looks like this:

```
if condition:
do this!
```

The "condition" here refers to a conditional expression, which evaluates to either true or false. For example,

```
if hearing spooky noises:
call Ghostbusters!
```

or, in Python lingo:

```
if spooky_noises == 1:
callGhostbusters()
```

If the conditional expression evaluates as true, all statements within the indented code block are executed. If the conditional expression evaluates as false, all statements within the indented block will be ignored, and the lines of code following the "if" block will be executed.

One of the unique things about Python is that it does not require you to enclose conditional statement blocks within curly braces, like most other languages. Instead, Python identifies code blocks according to indentation; all Python statements with the same amount of indentation are treated as though they belong to the same code block.

If, however, the conditional block consists of a single statement, Python also allows you to place it on the same line as the "if" statement. Consequently, the example above could also be written as

```
if spooky_noises == 1: callGhostbusters()
```

Here's a simple program that illustrates the basics of the "if" statement.

```
#!/usr/bin/python

# ask for a number
alpha = input("Gimme a number! ")

# ask for another number
beta = input("Gimme another number! ")

# check
if alpha == beta:
print ("Can't you read, moron? I need two *different*
numbers!")

print ("Go away now!")
```

And they say that the popular conception of programmers as rude, uncouth hooligans is untrue!

Tying Up The Loose Ends

In addition to the "if" statement, Python also allows you a couple of variants − the first is the "if−else" statement, which allows you to execute different blocks of code depending on whether the expression is evaluated as true or false.

The structure of an "if−else" statement looks like this:

```
if condition:
do this!
else:
do this!
```

In this case, if the conditional expression evaluates as false, all statements within the indented "else" block will be executed. Modifying the example above, we have

```
#!/usr/bin/python

# ask for a number
alpha = input("Gimme a number! ")

# ask for another number
beta = input("Gimme another number! ")

# check
if alpha == beta:
print ("Can't you read, moron? I need two *different*
numbers!")
else:
print ("Finally! A user with active brain cells!");
```

Cookie−Cutter Code

The "if−else" construct certainly offers a smidgen more flexibility than the basic "if" construct, but still limits you to only two possible courses of action. If your Python script needs to be capable of handling more than two possibilities, you should reach for the "if−elif−else" construct, which is a happy combination of the two constructs you've just been reading about.

```
if first condition is true:
do this!
elif second condition is true:
do this!
elif third condition is true:
do this!

... and so on ...

else:
do this!
```

There's one important point to be noted here − as soon as one of the "if" statements within the block is found to be true, Python will execute the corresponding code, skip the remaining "if" statements in the block, and jump immediately to the lines following the entire "if−elif−else" block.

Take a look at it in action:

```
#!/usr/bin/python

print "Welcome to the Amazing Fortune Cookie Generator";

# get input
day = raw_input("Pick a day, any day: ");

# check day and assign fortune
if day == "Monday":
fortune = "Never make anything simple and efficient when a way
can
be found to make it complex and wonderful."
elif day == "Tuesday":
fortune = "Life is a game of bridge −− and you've just been
finessed."
elif day == "Wednesday":
```

```
fortune = "What sane person could live in this world and not
be
crazy?"
elif day == "Thursday":
fortune = "Don't get mad, get interest."
elif day == "Friday":
fortune = "Just go with the flow control, roll with the
crunches,
and, when you get a prompt, type like hell."
else:
fortune = "Sorry, closed on the weekend"

# output
print fortune
```

You'll notice that in this case, I've used the raw_input() function instead of the regular input() function. The difference between the two is minor: raw_input() stores user input as is, while input() attempts to convert user input into a Python expression.

Since Python does not support PHP− or JSP−style "switch/case" conditional statements, the "if−elif−else" construct is the only way to route program control to multiple code blocks.

Time For Lunch

Python also allows you to combine multiple conditional tests with the "and", "or" and "not" logical operators. If you consider the following code snippet,

```
if weekday == "Thursday":
if time == "12":
if place == "Italy":
lunch = "pasta"
```

you'll agree that is both complex and frightening. And so, in addition to the comparison operators we've used so liberally thus far, Python also provides the "and", "or" and "not" logical operators which allow you to group conditional expressions together. The following table should make this clearer.

```
Assume delta = 12, gamma = 12 and omega = 9

delta == gamma and delta > omega
True

delta == gamma and delta < omega
False

delta == gamma or delta < omega
True

delta > gamma or delta < omega
False

not delta
False
```

Given this knowledge, it's a simple matter to rewrite the example above in terms of logical operators:

```
if weekday == "Thursday" and time == "12" and place ==
"Italy":
lunch = "pasta"
```

Simple and elegant, wot?

And that's about it for the moment. You've learned a lot today − you now know how to manipulate strings and numbers, obtain user input from the command line, and use the "if" family of conditional statements to control the flow of your program.

In the next article, I'll be discussing some of Python's loops, together with a close look at another very useful and interesting Python data structure − the list. You make sure that you come on back for that one!