Beginning Python: Data Types

1. Introduction

As I said in the tutorial "How a Computer Looks At Your Program", computer programs consist of **data** and **code**. In order to make code able to be used for more than a single set of data, computer programs use variables, or containers for data. These containers are technically called **data structures** because they allow the data to be structured in different ways. Just like you would probably be disinclined from cooking pasta in a coffee cup or drinking tea from a vat, so Python offers a variety of containers for different kinds of data.

In Python, there are five basic data structures or types of variables: integers, strings, lists, tuples, and dictionaries. In the following pages, we will look at each in turn.

2. Integers

An integer in Python, also known as a 'numeric literal', is simply a name used for a numeric value. For this reason, these values are commonly called integers even if they are assigned the value of a real number in the form of an integer and a decimal value. Any numeric value can be assigned to it at any time, overwriting the previous value.

To assign a value to an integer variable, one writes a statement after the following template:

```
<numeric literal name> = <numeric literal value>
```

Some examples are:

```
x = 1500
pi = 3.14
```

3. Strings - Part 1

A string literal, or **string**, holds any combination of letters and numbers you would like it to hold. Any number it holds, however, is not treated like a numerical value but is preserved as if it were a word.

One way to think about the difference between a number as a string and a number as a numeric literal is to consider a credit card. The credit card numbers themselves are an example of an integer value; you can add to them and subtract from their value. The credit card, however, preserves the numbers in a set format, complete with spaces. A string does the same thing. While you can multiply and divide the number of the credit card as an integer, you cannot break up the value by the way it is written. To do that you need a string. In this way,*an **integer** represents a quantity, a **string** represents a quality.*

4. Strings - Part 2

Because a string represents words and numbers in the format they are assigned, Python uses quotes to indicate their value. Trying to assign a string value without quotes looks like you are trying to assign the value of one string to another. If the string does not yet exist, Python will tell you so. To assign a value to a string, one uses the following template:

```
<name of string> = '<value of string>'
```

Examples of variable assignments are:

```
gettysburg = "four score and seven years ago..."
writ = 'habeas corpus'
Catch22 = """no win situation"""
postcode = '''91101-2509'''
```

"Wait a minute!", I hear you say, "You said that Python uses quotes to offset string values. You used single quotes in the template and then used single, double and triple quotes in the examples. Which is right?" The fact of the matter is that Python allows all of them. You get to choose which you want to use. Python does not care. *All Python asks is that you close the value with the same kind that you opened it* (otherwise, Python will not know where you mean the quotes to be part of the value or to signify the end of the string). If you begin a value with double quotes, you must end it with double quotes. The following are therefore **WRONG**:

```
name = 'Joanna Sebastian Bach"
body = "Sargasso See'
city = """Ed in burgh'''
```

5. Strings - Part 3

"But what do I do if I need to use a single quote when I already started with a single quote? Won't Python get confused?" It certainly will. If you use single quotes around the value, double quotes in the value are fine. Similarly, if you use double quotes around the value, single quotes are allowed in the value itself. However, if the value contains the same mark as the one used around it, you must use **escape** that part of the value.

Escaping is a fancy way of saying that you break out of the norm. It happens within quotation-like arguments when one places a backslash in front of a character like: "\n". When certain characters are 'escaped' they take on a different meaning. So, while a single quote may indicate a string's value, an escaped single quote within that value means something else: an actual single quote! The same is true for double quotes.

For more escaping, see the page "Escape Sequences".

6. Accessing Variables

To access the value of a variable within the Python shell, simply type it.

```
>>> x = 1500
>>> x
1500
```

Within a program, one can use the 'print' command.

```
cat = 'dog'
print cat
```

output:
dog

If the variable is a string, you can also access its parts. Each character of the string is indexed, starting with the first character at '0'. By accessing the string with an **indexing operator**, you can access only part of the string and leave the rest untouched. An example:

```
cat = 'dog'
print cat[0]
print cat[1]
print cat[2]
```

output:
d
o
g

7. Working With Variables

Working with Integer Variables

As you might imagine, you can perform any mathematical operation you like on a numeric literal.

```
d = 300
pi = 3.14159265
circumference = pi * d
r = d/2
c = 2 * pi * r
```

For more operators, see the page "Python Operators".

Working with Strings

With a string, one concatenates instead of adding. One can concatenate strings or their values. The result is a string that shows no indication that the two parts were ever separate. So if you need a space between the parts, you must be sure to include it when the two strings are joined.

```
a = 'big'
b = 'baboons'
phrase = a + b
full_phrase = b + " with " + a + " bahoochies"
print phrase
print full_phrase
output:
bigbaboons
baboons with big bahoochies
```

If you want more than one letter but still less than the whole string, you can cull out a substring by using the slicing operator:

```
>>> b = 'baboons'
>>> z = b[2:7]
>>> print z
boons
```

You can, of course, combine that substring with another string value and assign both to a single string literal.

```
>>> b = 'baboons'
>>> word = 'ball' + b[3:7]
>>> print word
balloons
```

8. Lists

Assigning Values to a List

A **list** is, as the name suggests, a series of values. In Python, these values are assigned by placing them within square braces and separating them by commas like this:

```
<name of list> = [ <value>, <value>, <value> ]
girls = ['sugar', 'spice', 'everything nice']
lotto = ['26', '12', '23']
addends = [4, 34, 7]
```

Note that 'addends' is comprised of integer variables while the others are comprised of strings, as the quotes suggest.

A list can contain any type of Python object -- even other lists.

```
>>> lotto[2] = addends
>>> print lotto
['26', '12', [4, 34, 7]]
```

Accessing the Values of List

To access a part of a list, one uses the same kind of phrase as one used for a string literal:

```
<name of list>[<index number>]
```

A few examples:

```
ingredient1 = girls[0]
print girls[1]
print 'brown ' + ingredient1
print girls[0] + " is not a " + girls[1]
```

output:

```
spice
brown sugar
sugar is not a spice
```

Combining Lists

Lists can be concatenated in a way similar to strings by using the plus operator ('+'):

```
primes = [1, 3, 5, 7] + [9, 11]
mishmash = girls + lotto + addends
print primes
print mishmash
```

output:
[1, 3, 5, 7, 9, 11]

['sugar', 'spice', 'everything nice', '26', '12', '23', 4, 34, 7]

9. Tuples

In Python, a **tuple** may be defined as a finite, static list of literals (numeric or string). For our purposes, a tuple is very similar to a list in that it contains a sequence of items. It differs from a list in that it cannot be changed once it is created. One can index, slice and concatenate, but one cannot append or alter the values of the tuple after it has been initialized.

To initialize a tuple, one encloses the values in parentheses and separates them by commas.

```
directions = ('north', 'south', 'east', 'west')
coordinates = (45, 34, 48, 32)
print directions[3]
print coordinates[1]
```

output:
west
34

10. Dictionaries

Defining a Dictionary

Dictionary is the Python term for an associative array. An array is, like a list, a series of values in two dimensions. An associative array gives one a way of accessing the values by a **key**, a term associated with the **value** instead of an item's index number.

Initializing a dictionary, one offsets the keys and values in curly braces. Each key-value pair is separated from the others by a comma. For each pair, the key and value are separated by a colon. The key of each member is offset in quotes. A sample dictionary is as follows:

```
my_dictionary = { "author" : "Andrew Melville",
"title" : "Moby Dick",
"year" : "1851",
"copies" : 5000000
}
```

Accessing a Dictionary

One accesses a dictionary member by its key:

```
>>> a = my_dictionary["author"]
>>> print a
Andrew Melville
```

To insert or modify a member, one simply assigns the value:

```
>>> my_dictionary["publisher"] = 'Harper and Brothers'
>>> my_dictionary["author"] = 'Herman Melville'
>>> print my_dictionary
{'publisher': 'Harper and Brothers', 'title': 'Moby Dick', 'year': '1851', 'copies': 5000000, 'author':
'Herman Melville'}
```

Beginning Python: Operators

Introduction

In any form of programming, one inevitably needs to add, subtract, combine, split, and otherwise manipulate data. To do this, small symbols and reserved words are used to tell the computer exactly what needs to be done. In this brief tutorial, we will look at the operators and how to use the most common of them.

Common Operators

Python uses the following operators:

+ addition (unary plus)
- subtraction (unary minus)
* multiplication
**
/ division
% modulo
< less than
> greater than
<= less than or equal to
>= greater than or equal to
== equal to
!= not equal to
<> an obsolete form of 'not equal to'

For strings, however, Python has just one operator: **+** for concatenation. All other operations are performed using modules like string or re.

Using Operators

Numerical operations (aka, 'Arithmetic')

As you might imagine, you can perform any mathematical operation you like on a numeric literal.

```
d = 300
pi = 3.14159265
circumference = pi * d
r = d/2
c = 2 * pi * r
```

String operations

As discussed under strings, one concatenates a string instead of adding it. One can concatenate strings or their values. The result is a string that shows no indication that the two parts were ever separate. So if you need a space between the parts, you must be sure to include it when the two strings are joined.

```
a = 'big'
b = 'baboons'
phrase = a + b
full_phrase = b + " with " + a + " bahoochies"
print phrase
print full_phrase
output:
bigbaboons
baboons with big bahoochies
```

One can also test a string for equality or inequality relative to another string:

```
if computer == "laptop"
if brand != "generic"
```

Beginning Python: Controlling the Flow

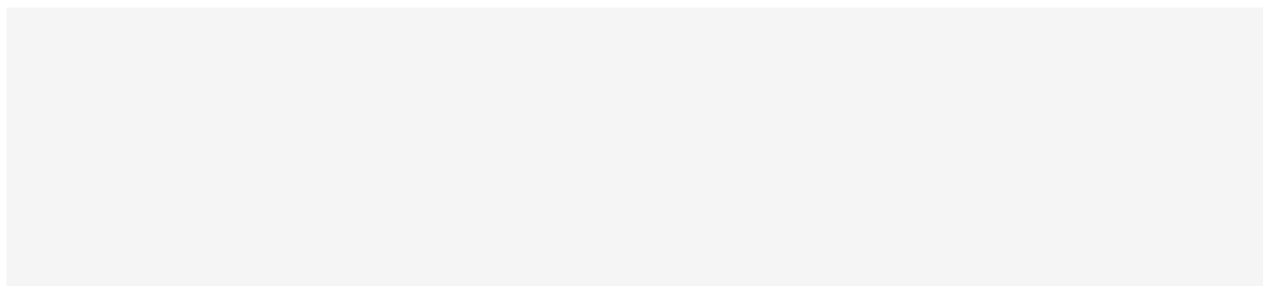
As I mentioned in the pre-tutorial tutorial, loops tell the computer when to go and when to stop. In this way, they are sometimes called tools for "control flow".

Every form of program control has the same two parts: condition and action. If the condition is met, the action is taken; otherwise, the entire loop is passed over by the interpreter. The execution of the loop, whether once or more, is called iteration. It is important to remember that the condition is only evaluated at the beginning of each pass; changes in the middle or at the end will only effect the next cycle of the loop.

In Python, there are three kinds of loops: while, if, and for. Each of the three precedes the action to be taken and ends its condition with a colon (':').

When you use a loop, you must remember to indent the body of the loop. Unlike some languages, Python does not use braces to indicate the beginning and ending of a loop. It relies upon the indentation level of the line. There is no set amount that you must indent, though four spaces for each level is a popular convention. However, you must be consistent about how far you indent.

Any condition one might pose in Python must be represented in the appropriate way, otherwise the interpreter will not understand it. So, before discussing the format for loops, let's take a look at how to express comparison and equality in Python.



Forming Conditions

Conditions in Python are expressed differently for numeric and string literals. Comparisons of numeric literals are very similar to standard algebraic statements:

Greater than: a > c
Less than: b < 6

Tests for equality, however, are slightly different. In Python, assignment of equality is done with a single equal operator ('='), but testing for equality is done with a double equal operator ('=='). Conversely, testing for inequality is done with a an exclamation point followed by an equal operator. Tests for greater-than-or-equal-to and less-than-or-equal-to are expressed analogously.

Assignment of equality: c = 6
Testing of equality: c == 6
Testing of inequality: c != 6
Greater than or equal to: c >= 6
Less than or equal to: c <= 6

For string literals, on the other hand, there is no greater or less than. It is obviously illogical to test whether 'cat' is greater than 'dog'. Rather, string literals are tested for equality and inequality as follows:

Testing of equality: cat == 'dog'
Testing of inequality: cat != 'dog'

There are ways to test whether parts of strings match a given value, but this requires the Python Standard Library modules 're' or 'string', something we will discuss later.

WHILE Loops

The while loop simply tells the computer to do something as long as the condition is met. Its syntax looks like this:

```
while <condition>:
<action to be taken>
```

That's it. As long as the condition is met, the action is taken. For example:

```
a = 6
while a > 5:
print a
a = a - 1
```

This loop will execute exactly one time because, on the second pass, 'a' will not be greater than 5. If, however, the last line was omitted, an endless loop would result. If the condition is met and the criterion never changes, the action will be taken repeatedly like this:

```
a = 6
while a > 5:
print "This is an endless loop!"
```

IF...ELIF...ELSE Loops - Part 1

Another means of controlling the flow is to say "If <some circumstance exists>: <do this>." Obviously, if the circumstance does not exist, the computer will ignore the lot. Sometimes, however, it is nice to have a "default setting" for the program's flow, an "else": If <a particular circumstance exists>: <do this> else: <do that>. The following templates and examples illustrate how these two loops are written:

```
if <condition>:
<action to be taken>
```

or

```
if <condition>:
<action to be taken>
else:
<default action>
if c < 0:
print c
if animal == dog:
print "bow-wow"
else:
print "meow"
```

You might think it a bit cludgy to have to write 'if...else' statements for every possible option. You would be right, and this is why Python has an additional, optional part of the 'if' loop: 'elif'. 'elif' is for the various options that fit neither the 'if' nor the else. The template and an example are as follows:

```
if <1st condition>:
<action to be taken>
elif <2nd condition>:
<other action to be taken>
else:
<default action>
if animal == 'dog':
print "bow-wow"
elif animal == 'cat':
print "meow"
else:
print "cockadoodledoo!"
```

IF...ELIF...ELSE Loops - Part 2

If one writes a series of 'if' conditions, the computer will work its way through each of them, performing the action of all that match. But when one uses 'if...elif...else', the computer treats the entire collection of conditions as a unit and only does the action of the first condition that matches.

In the series of 'if' statements below, all of the given conditions are true, so the program would perform each of the actions in turn. In the 'if...elif...else' series of statements, the computer will perform only the action associated with the first true condition, ignoring the rest.

```
c = 100

if c > 99:
print "c is greater than 99."
if c < 200:
print "c is less than 200."
if c <= 100:
print "c is less than or equal to 100."
if c >= 100:
print "c is greater than or equal to 100."
```

output:
c is greater than 99.
c is less than 200.
c is less than or equal to 100.
c is greater than or equal to 100.

```
c = 100

if c > 99:
print "c is greater than 99."
elif c < 200:
print "c is less than 200."
elif c <= 100:
print "c is less than or equal to 100."
elif c >= 100:
print "c is greater than or equal to 100."
else:
print "The value of c is unclear."
```

output:
c is greater than 99.

FOR Loops - Part 1

We saw in our discussion of variables and datatypes that Python, and programming in general, does a lot with series and sequences of data. Often it is faster or easier if the computer can perform a certain action a set number of times, perhaps based on a certain number of items. The 'for' loop allows this kind of iteration.

The syntax of the 'for' loop follows this template:

```
for <counter variable> in <range list>:
<action to be performed>
```

The counter variable can be any variable. It simply serves as a variable into which Python can put each item of the range list in turn. It can be, and often is, referenced in the command part of the loop.

The **range list** can be anything: numeric literal, string literal, list, tuple, dictionary, or any combination thereof. What matters is that you access the items in the correct way. Some examples follow below.

```
prime = [1,3,5,7,9,11]
for n in prime:
print n
```

output: 1
3
5
7
9
11

```
vornamen = ['Bill', 'John', 'Luke', 'Ricky']
for name in vornamen:
print name
```

output:
Bill
John
Luke
Ricky

```
info = {
"producer" : "Kubrick",
"title" : "Dr Strangelove",
"phone call" : "Hello, Dmitri"
}

for key in info:
print key
print info[key]
print "\n"
```

output:
phone call

Hello, Dmitri

producer
Kubrick
title
Dr Strangelove

FOR Loops - Part 2: range() and xrange()

Sometimes it is helpful to be able to just tell the computer to do something a certain number of times without having to type in every item of a range. This is why Python has a special function called range. **range()** takes two numbers and tells Python to count from the one to the other. By default, it does this in increments of one but can be told to do it in increments of any positive integer imaginable. The basic syntax of range() is as follows:

```
range(start, end)

range(0, 10)
```

When the increment element is used, it goes after the end number:

```
range(start, end, increment)
range(0, 10, 2)
```

You can use range() wherever you would use a list. Whether you assign the range to a variable or use it directly in a for condition, the results are the same.

```
    a = range(1, 10)
    for i in a:
    print i
# (Alternatively:)
    for i in range(1, 10):
    print i
output:
1
2
3
4
5
6
7
8
9
```

A word should be said here about large ranges. range() works by forming the range list when it is called. The full value of that list is then passed around the program whenever the range is needed. This consumes memory and CPU cycles, especially when range() is given a large number of items to amass. Therefore, Python has another range function called **xrange()**. The syntax of xrange() is exactly the same as range(), but xrange() populates its range list whenever it is accessed, allowing the memory to be freed when the list is not actively used.

Beginning Python: Putting It All Together With Syntax

Introduction

In programming, it is not enough to know the basic parts of the language. In order to construct a program, one must know how the parts work together to make a whole. This tutorial discusses some rudimentary aspects of Python syntax like input, output, classes and the use of modules. Other parts of syntax are addressed on the pages about the Python Standard Library and the blog for this site.

Indentation

Indentation in Python counts. This may seem odd at first, but it goes a long way to making Python programs some of the easiest programs to maintain.

Every block structure -- control structures, classes, functions, etc. -- must be indented the same number of times for their level in the program. Convention is to use four spaces (without tabs) for each level of indentation. So, a skeletal structure of a Python program might look like this:

```
import os, re, string

def test_function1():

def test_function2():

class CartoonCharacter(object):

def main():

a = 8
string = "sunshine"

if x == 1:

if x != 1:

if a > 0:

while b < 100:

elif a == 0:

else:

else:

if __init__ == "__main__":
main()
```

Comments and Remarks

Any good program should have comments from the programmer written amidst the code in order to explain the flow of the program to others who may read the program. Obviously, these comments are not meant to be executed; they are ignored when the program executes. In order to offset them from the rest of the program and to tell the computer to ignore them, Python uses the hash symbol, or number sign, at the beginning of the comment. Some examples:

```
# This is a comment
print "This is code."
print "This is code." # But this is a comment.
# Note that comments can be on a line by themselves
# or at the end of a line. But there is no way for
# comments to precede code on the same line.
```

File Input and Output

As with everything in Python, files are objects. To open a file object, you must assign it to a variable as follows:

```
a = open("sample.txt")
```

By default, the open command takes a minimum of one and possibly two arguments: the name of the file to open and the control, or permission, you would like the program to exercise over that file. The two basic ways to open a file are for reading ('r') and writing ('w'). If the file is a binary file, instead of text, you can follow the 'r' or the 'w' with a 'b' to tell Python to deal with the file accordingly.

The following example opens a file for reading. It then reads every line of the file, printing it to the screen, and closes the file at the end.

```
a = open("sample.txt", "r")
line = a.readline()
while line:
print line
line = a.readline() # Note that the content of line changes
# here, resetting the loop

a.close()
```

Writing to a file is even easier. The following code opens a file, 'output.txt', for writing and writes a sample line to it, complete with a newline at the end.

```
a = open("output.txt", "w")
a.write("This is a sample line.\n")
```

Functions

Functions in Python are declared using the reserved word 'def'. If the function takes anyarguments, these are enclosed in parentheses after the function name. As with other languages, the names given to the functions arguments are those used in the function code. In the rest of the program, they may be known by other names. What matters is not the precise variable name but the data type of the variable passed.

The code below illustrates how a function is defined and later called by the main program. Unlike C, Python requires that all functions be prototyped (i.e., be declared before being called in the main program).

```
def multiply(a,b):
c = a*b
return c

product = multiply (x,y)
```

Classes - Part 1

Classes describe a collection of variables and methods which are encapsulated (i.e., protected from the rest of the program) and which are related in some significant way. Classes exist in the flow of the program as instances. These instances are called objects. So, the class reflects the theoretical ideal of a set of objects. Classes do not take arguments, but their methods may.

Classes in Python are defined by using the reserved word 'class' and the class name. The following definition of a class Stack is found in David Beazley's excellent book, *Python Essential Reference*, (p.16):

```
class Stack:
def __init__(self):
self.stack = [ ]
def push(self,object):
self.stack.append(object)
def pop(self, object):
return self.stack.pop()
def length(self):
return len(self.stack)
```

A computer stack is a temporary abstract data structure which operates on the principle of 'last in first out' (more information on it may be found athttp://en.wikipedia.org/wiki/Stack_data_structure).

Classes - Part 2

A computer stack is a temporary abstract data structure which operates on the principle of 'last in first out' (more information on it may be found athttp://en.wikipedia.org/wiki/Stack_data_structure). In this class definition, the first function is for the class itself. This is a required part of every class; it must have a function pertaining to itself. As the activities around a stack are essentially two, two functions are set up. The first pushes data on the stack. The second pops off the last piece of data. Finally, a very helpful piece of information to know is how many items are in the stack. For this, the class has a length function.

To initate a class, simply assign an instance to a variable name as shown in the first line below. One accesses the methods of the class by prefixing the name of the instance to the name of the method.

```
heap = Stack() # Create an instance of the class
heap.push('Belteshazar') # Push items on the instance heap of class Stack
heap.push(['Battle of Hastings', 1066])
heap.push('1945')
a = heap.pop() # Assigns '1945' to a
b = heap.pop() # Assigns '[Battle of Hastings, 1066]' to b
c = heap.pop() # Assigns 'Belteshazar' to c
del heap # Destroy the instance heap of class Stack
```

Exceptions and Errors

In writing a Python program, you should always include some form of error checking. Doing so allows the computer to fail softly instead of giving the user an ugly error message. It also means that if one thing goes wrong the entire program does not grind to a halt.

To provide this safety net, use 'try...except'. 'try' tells the computer to attempt to do something. 'except' tells it what to do on failure. The syntax goes like this:

```
try:
a = open("sample.txt", "r")
except IOError, error:
print error
```

Modules - Part 1: Importing Modules From the Python Library

Modules are pieces of Python code that you import into your programs. When taken from the Python Standard Library, these are boiler-plate functions that are optimized for a particular task.

Some important Python modules are 're' for pattern matching (i.e., regular expressions), 'string' for string handling, 'sys' for system-specific functions, and 'os' for other operating system functions. To import one of these, simply write 'import' and the library you want to import:

```
import re
import string
import sys
import os
```

There is no specific order in which you must import them, but you must import them before you use any functions which they contain. If you would rather import all of them on a single line, you can:

```
import re, string, sys, os
```

Sometimes the module itself is so large that importing the whole thing can bog down the program upon execution. This is particularly true of the posix module. In this case, it is better to import specific functions as follows:

```
from posix import environ
```

For most operating system services, however, it is best to use the os module, which is platform independent.

Once a module is imported, preface the function calls with the name of that module:

```
new1 = re.match('\t', variable)
new2 = string.sub('\t', '_|_|_|_|', varibale)
```

From the Python shell, if you do not know what functions are supported by the module, call dir() to receive a list of them:

```
>>> import string
>>> dir(string)
['Template', '_TemplateMetaclass', '__builtins__', '__doc__', '__file__', '__name__', '_float', '_idmap', '_idmapL', '_int', '_long', '_multimap', '_re', 'ascii_letters', 'ascii_lowercase', 'ascii_uppercase', 'atof', 'atof_error', 'atoi', 'atoi_error', 'atol', 'atol_error', 'capitalize', 'capwords', 'center', 'count', 'digits', 'expandtabs', 'find', 'hexdigits', 'index', 'index_error', 'join', 'joinfields', 'letters', 'ljust', 'lower', 'lowercase', 'lstrip', 'maketrans', 'octdigits', 'printable', 'punctuation', 'replace', 'rfind', 'rindex', 'rjust', 'rsplit', 'rstrip', 'split', 'splitfields', 'strip', 'swapcase', 'translate', 'upper', 'uppercase', 'whitespace', 'zfill']
```

Obviously, this is just to refresh your memory and not to replace the reference documentation.

Modules - Part 2: Importing Functions From Your Own Programs

Whenever you write a program that contains functions (which is almost any program of substantial size and purpose), you can call on the functions of that program from others. This assumes that the programs do not have the same names as Python modules. When the Python interpreter is told to import a module, it checks both the location(s) of the Python library *and* the current directory in order to find the requested module.

The precise order of this checking is determined by the import search path, a list of directories contained in sys.path. You can view the path in the Python shell as follows:

```
>>> import sys
>>> sys.path
['', '/usr/lib/python2.4/site-packages/TurboGears-0.8.9-py2.4.egg', '/usr/lib/python2.4/site-packages/TestGears-0.2-py2.4.egg', '/usr/lib/python2.4/site-packages/FormEncode-0.4-py2.4.egg', '/usr/lib/python2.4/site-packages/cElementTree-1.0.5_20051216-py2.4-linux-i686.egg', '/usr/lib/python2.4/site-packages/elementtree-1.2.6-py2.4.egg', '/usr/lib/python2.4/site-packages/json_py-3.4-py2.4.egg', '/usr/lib/python2.4/site-packages/SQLObject-0.7.1dev_r1457-py2.4.egg', '/usr/lib/python2.4/site-packages/CherryPy-2.1.1-py2.4.egg', '/usr/lib/python2.4/site-packages/kid-0.8-py2.4.egg', '/usr/lib/python2.4/site-packages/setuptools-0.6a11-py2.4.egg', '/usr/lib/python2.4/site-packages/Django-0.91-py2.4.egg', '/usr/lib/python24.zip', '/usr/lib/python2.4', '/usr/lib/python2.4/plat-linux2', '/usr/lib/python2.4/lib-tk', '/usr/lib/python2.4/lib-dynload', '/usr/local/lib/python2.4/site-packages', '/usr/lib/python2.4/site-packages', '/usr/lib/python2.4/site-packages/HTMLgen', '/usr/lib/python2.4/site-packages/Numeric', '/usr/lib/python2.4/site-packages/PIL', '/usr/lib/python2.4/site-packages/gtk-2.0', '/usr/lib/site-python']
```

Note the first empty item in the list, ''. This equates to the directory in which Python is executing the program. With this first, Python will search the current directory before all others.

Beginning Python: Exceptions, Errors, and Warnings

What is an Exception?

More often than not, a program is written to address a certain problem. It may be as simple as converting certain kinds of British spelling to American spelling, or vice versa. Or it may be working with hundreds of internet spiders throughout the Web or tracking and processing millions of financial transactions. Of the latter two, Google uses Python for the first and the New York Stock Exchange (NYSE) does the second *every day*.

Any publisher who has authors in Australia, New Zealand, Canada or elsewhere in the former British Commonwealth must do the first task whenever they bring a book to the US markets. If their conversion program expects only plain English (ASCII) it will be in trouble. The program will inevitably encounter an foreign phrase or other unexpected piece of text. It will then throw an error [read: *crash*].

Whenever Python throws an error on a program that is programmed with proper Python syntax, it has hit upon an exception. While your data may vary, you can program Python to expect variability in the data. Using the information presented in this tutorial, your Python programs will cope quite nicely and deal with failures in the way that you decide.

Exception Exemplified

If reading the previous page has caused you to wonder: "Gee, what *does* Python do with unexpected variations in data?", try this in your Python shell:

```
Python 2.5 (r25:51908, Dec 8 2006, 15:53:50)
[GCC 4.0.3 (Ubuntu 4.0.3-1ubuntu5)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print a
```

You will immediately receive Python's response: a **NameError**.

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

If you would like another example, try this one from "Beginning Python":

```
>>> 1/0
```

If you have come far enough in life to be reading this, there is a good chance that you know that division by zero is a no-no in the world of non-imaginary numbers. Python may not have been around as long as you have, but it knows that, too, and it has a special error just for it:

```
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

You will notice that this error is called a **ZeroDivisionError**. Python has many kinds of exceptions. Descriptions of each can be found on the last page of this tutorial. Alternatively, if you just want to get a feel for what kinds of exceptions can be thrown, type the following in your Python shell:

```
>>> import exceptions
>>> dir(exceptions)
```

If at First You Don't Succeed

As mentioned earlier, Python throws an error or exception whenever things do not go to plan. Fortunately, when this happens, a well-coded program knows how to handle these errors and can "catch" the error. This is sometimes called "trapping". To catch the error, one must be a bit less forceful with one's commands and use the **try...except** program structure.

To use the first exception from the previous page, we know that we get a NameError if we try

```
print a
```

If we want to ensure that the whole house of code does not fall down on us, we can couch the **print** statement in a **try** clause and handle the exception with **except**:

```
try:
print a
except:
print "Doh, you need to define 'a' before you can print it!"
```

If you plug this into your Python shell, being sure to indent the arguments of **try** and **except**, you will get the following output:

Doh, you need to define 'a' before you can print it!

Of course, this except clause handles *every* exception. What if you only want to process the **NameError** with that statement? Then tell Python to take exception at the NameError to the implicit exclusion of other types:

```
try:
print a
except NameError:
print "Doh, you need to define 'a' before you can print it!"
```

The output will be the same.

You can **except** any error in the module **exceptions**. In the section on the Python Library, there are lists of all exceptions and warnings.

Try, Try Again

Now that you can catch errors and turn them into simple exceptions. The question begs asking: "How do you catch more than one kind of error?" Simple: Use multiple **except**statements.

If, for example, we take Hetland's example of dividing by 0 and turn the division equation into variables instead of values, we should check for both attribution of value and zero division. In which case, we want a program like that below.

```
a = input("Please enter the dividend (the number to be divided):")
b = input("Please enter the divisor (the number by which to divide):")

try:
print a/b
except NameError:
print "Both values must be integers!"
except ZeroDivisionError:
print "The divisor must not be zero. Division by null is not allowed."
```

Similarly, if you need to do more calculations with the two values (e.g., monetary conversion for multiple currencies), you can lob all of the calculations into the single **try** statement. If anything goes wrong, the same **except** statements for one calculation will pick up the mathematical debris of the others.

If you would like to perform the same action on multiple errors, you can combine the errors into a single argument for **except**:

```
a = input("Please enter the dividend (the number to be divided):")
b = input("Please enter the divisor (the number by which to divide):")

try:
print a/b
except (NameError, ZeroDivisionError):
print "The divisor must not be zero. Null division is not allowed."
```

Taking Exception

While it is all well and good to handle an exception and do something predictable when it happens, what if you want to process the error as data, perhaps printing it in a bold HTML typeface? Simply pass to **except** the variable by which you want to reference the error data. Using the last example from the previous page, it would look like this:

```
a = input("Please enter the dividend (the number to be divided):")
b = input("Please enter the divisor (the number by which to divide):")

try:
print a/b
except (NameError, ZeroDivisionError), error:
print "The divisor must not be zero. When the divisor is zero, Python hiccups like every good
calculator should. The error passed was:"
print "<div><b>%s</b><div>" %(error)
```

If All Else Fails...

Sometimes you may want Python to behave one way if there is an exception and another way if no error occurred. In this case, you will want to use a structure of **try...except...else**. To nuance the same previous example, we can add a third clause to affirm the user in not dividing by zero.

```
a = input("Please enter the dividend (the number to be divided):")
b = input("Please enter the divisor (the number by which to divide):")

try:
print a/b
except (NameError, ZeroDivisionError), error:
print "The divisor must not be zero. When the divisor is zero, Python hiccups like every good calculator should. The error passed was:"
print "<div><b>%s</b><div>" %(error)
else:
print "Not dividing by zero is a good thing."
```

When run, this program can produce the following output:

Please enter the dividend (the number to be divided): 6

Please enter the divisor (the number by which to divide): 3

2

Not dividing by zero is a good thing.

Raising Your Own Exceptions With raise - Part 1

The standard way by which Python "handles" exceptions is by throwing an error of some sort. If the program is not coded to "catch" the error and do something with it, the program usually crashes, as we saw earlier. However, even if the program works just as you intended, you can program in your own exceptions using the **raise** statement.

At a Python shell, type the following:

```
raise NameError
```

You will get feedback from the Python interpreter that looks like this:

```
>>> raise NameError
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError
```

You can use any kind of error that is included in the module **exceptions** (for a full list of exceptions, see the page of exceptions). You can also raise your own exceptions (see below). However, the exception raised must be a class or an instance of a class. Anything else will result in a **TypeError**. If you want to see what would happen, type the following at a Python shell prompt:

```
>>> raise
```

Whenever an exception is raised in the code, Python will raise that error -- even if it sees nothing wrong. In this way, your programs can be stricter than the Python interpreter. Say, for example, you have a set of options and would like a NameError error to be thrown if the user chooses an option not listed. Python will do that as part of the exception handling process, instead of throwing a KeyError or something similar.

Raising Your Own Exceptions With raise - Part 2

But what if you do not want to give the user something obtuse like **NameError** or **IOError**. Python also provides a way to present customised error messages by using the built-in class**Exception**. To use it, simply plug "Exception" as the argument of **raise**.

```
>>> raise Exception
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
Exception
```

"Where is the error message?" Well, I wanted you to see that you could raise a simple error by not giving any further arguments than "Exception". If you want to print something beyond this, enclose your message in quotes and separate it from "Exception" by a comma:

```
>>> raise Exception, "This is my customised error message."
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
Exception: This is my customised error message.
```

That summarises Python's exceptions, if you would like a list of them, you will find a page each of Python's exceptions and warnings in the section on the Python Library.

Beginning Python: Python Encodings

The Reason for Encodings in Python

Anyone who lives on this planet long enough knows that more than one langauge is spoken on it. Anyone who programs long enough knows that data comes in more forms than simple, English ASCII. It was not always so. Thirty years ago, anyone who worked on computers had only the basic English character range to use. If they needed to talk about a piÃƒÂ±ata, they would type it "pinata" and need to disambiguate it as a part of a Mexican celebration by other means.

Eventually, different sets of characters came to be used to represent the sundry language communities in the world. They were, however, mapped over the same key values as their American English counterparts. Key values come in four part combinations like "0xdf". For American English, this could be mapped, say, to the lowercase letter 'a'. But in French, that value is mapped to the letter 'q' because French keyboards conventionally have 'q' where American keyboards have 'a' and 'a' where American keyboards have 'q'.

The trouble in multilingual programming comes because, when processing text, the computer operates on these values, not the letters. By default, Python uses North American English. However, it readily uses any other alphabet in the world; you simply need to tell it do so. This tutorial shows you how to save character strings as Unicode, how to convert Unicode strings to standard string format, and how to convert standard strings to Unicode.

How to Work with Unicode in Python

The mapping of "0x40" for the letter "g" is called an encoding. The value is encoded as the letter. Depending on the encoding, "0x40" could be the letter "g" (as in many North American and European encodings) or the Bangladeshi "Ã™â€ž" or the Georgian "Ã¡Æ’Å¾".

For multilingual encoding, Python uses Unicode. Unicode is a system that provides a unique number for every character of a language, no matter what the language.

In order to print a Unicode character, Python requires you to do two things:

1. Define the character string to be printed as Unicode, and
2. Declare the type of encoding you would like Python to use in the output.

The next couple pages of this tutorial will look at these two requirements in-depth. We will then look at how to convert a standard string into Unicode.

Python's Way of Defining a Unicode String

To define a string as Unicode, one simply prefixes a 'u' to the opening quotation mark of the assignment. So, for example, to define variable **x** as holding the Hebrew letter *shin*, we simply write:

```
x = u"Ã—Â©"
```

The value within quotation marks can be literally anything. The basic structure of the statement remains the same:

```
y = u"Ã™â€ž"
z = u"Ã¡Æ’Å¾"
```

One important aspect of working in multiple languages and Unicode is that just because you do not see the letter does not mean that it is not there. If the editor in which you are writing these pieces of code does not support the encoding or does not have a font to reflect the alphabet being used, you will see either blank space, rectangular boxes, or aberrant output that looks a bit like comic strip profanity.

Nonetheless, the assignment is made. But a value saved is not worth very much unless you do something with it. While Python easily handles the variable with the Unicode value, we still need to produce some output. Go to the next page to see how.

How Python Prints Unicode

To output a Unicode string, we must first encode the Unicode string as a standard string. To do this, we use Python's **encode** function in order to tell Python which encoding to use. For the Unicode strings of the previous page, we could write:

```
a = x.encode("utf-8")
b = y.encode("utf-8")
c = z.encode("utf-8")
```

Then we can print them as usual or otherwise write them to a file-like object (e.g., local file, web page, etc.). Now that the values are assigned to other variables, we can reference those handles like any other variable:

```
>>> converted = (a, b, c)
>>> for i in converted:
... Ã‚Â Ã‚Â Ã‚Â Ã‚Â print i
...
Ã—Â©
Ã™â€ž
Ã¡Æ’Å¾
```

This is one way of converting Unicode strings to standard strings in Python. There is, however, a much easier way. Go to the next page of this tutorial to find out how.

Python's Way of Encoding Unicode Strings "En Passant"

Another way to use the encode function is to convert the value on-the-fly, including it in the print statement.

```
print x.encode("utf-8")
print y.encode("utf-8")
print z.encode("utf-8")
```

If this is still too much typing, you will be glad to know that Python will take any object for the encode function, even an iterator. Therefore, one can also feed the encoding argument to a series of Unicode strings like this:

```
>>> convert_on_output = (x, y, z)
>>> for j in convert_on_output:
... Ã‚Â Ã‚Â Ã‚Â Ã‚Â print j.encode("utf-8")
...
Ã—Â©
Ã™â€ž
Ã¡Æ’Å¾
```

Encoding ASCII as Unicode with Python

But what if we want to encode a standard string as Unicode? While there are some who think that, when it is spoken slowly enough and loudly enough, everyone understands English. This method of communication works even less for computers than it does for humans.

As mentioned earlier, Python's default encoding is usually ASCII. It can, however, be set to other encodings. To find out which encoding a given Python installation is using, use the method **sys.getdefaultencoding()**. In the Python shell, it will look like this:

```
>>> import sys >>> sys.getdefaultencoding() 'ascii'
```

To convert a standard, ASCII string to Unicode, we simply use Python's built-in **unicode**function:

```
t = "Hello"
x = unicode(t)
```

Now, **x** contains the Unicode form of **t**. You might ask: *What's the difference?* On your local system, probably none. If, however, you are a consultant programming in Missoula, Montana for a Greek-speaking client in Athens, there is a substantial difference. See the next page for a more detailed explanation.

Python Makes ASCII "Hello" Read as "Hello" in Unicode

In order to show the difference more clearly, let's get a bit more technical. ASCII key values bridge a range of 128 hexadecimal values from 0x00 to 0x7f ("0x00" is the hexadecimal form of "0"; "0x7f" is the hexadecimal equivalent of "127"; the integers from 0 to 127 are 128 in number). As I stated earlier, non-English keybindings cannot use this range nor this set. For many years, different encodings were used as a foil. This is whence the ISO 8859 series developed. However, many of these encodings could not "talk" to each other. This caused programming to become incredibly complex and subsequently impinged upon the ability of users to interact internationally.

Then the Unicode Consortium developed a system whereby every possible alphabet (save for Old Chinese) had its own unique identifier. So, the ASCII set became the beginning of a series that now includes over 100,000 possible characters. The ASCII set is reflected in addresses U+0000 through U+007F of that series. So, when "Hello" is converted from ASCII to Unicode, the computer stops "seeing" ASCII (that range from 0x00 to 0x7f) and starts reading Unicode. This offers greater interoperability with other languages and protects the program itself from premature obsolescence. Instead of multiple encodings, Unicode effectively offers different parts of one giant encoding.