Starting to Slither: Python for Beginners

by

Eric Manuel N. Pareja
Philippine Linux Users' Group

April 26, 2003
Asia Pacific College

Objectives

- Introduce the Python Programming Language
- Teach the Basics of Python
- Demonstrate Some Python Applications

Outline

- Introduction
  - What is Python / History of Python
  - Why Python / Features
- Starting to Slither
  - Variables and Arithmetic Expressions
  - Conditionals
  - File I/O
  - Strings
  - Lists and Tuples
  - Loops
  - Dictionaries
  - Functions
  - Classes
  - Exceptions
  - Modules
- Demonstration of some Python programs
- Books and Websites

Introduction: What is Python?

python, (Gr. Myth. An enormous serpent that lurked in the cave of Mount Parnassus and was slain by Apollo) 1. any of a genus of large, non-poisonous snakes of Asia, Africa and Australia that suffocate thei prey to death 2. popularly, any large snake that crushes its prey 3. totally awesome, bitchin' language tha will someday crush the $'s out of certain other so-called VHLL's ;-)

Python is an interpreted, interactive, object-oriented programming language.

It is often compared to Tcl, Perl, Scheme or Java.

Introduction: History of Python

Written by Guido van Rossum

Started work in 1990

First release in 1991

Minor number release every 6 months

Named after Monty Python

Current version is 2.3a2 (February 19, 2003)

Introduction: Why Python

- Simple yet powerful syntax
- Multi-platform
  - UNIX, Windows, Mac, BeOS, VMS, Amiga, OS/2, PalmOS, Java
  - Some modules are platform bound
- Speed of development
- Wealth of standard and contributed modules
- Strong community involvement, 10th annual python conference

Introduction: Features of Python

- Extensible in Python, C and other programming languages
- Object Oriented without being Object-centric
- White-space is significant
- Great for rapid prototyping
- Good for scripting
- Great for readability
- Restricted execution environment
- Exceptions

Perl is executable line noise. Python is executable pseudo-code.

Variables and Arithmetic Expressions

Python is a dynamically typed language in which names can represent values of different types during the execution of the program.

The assignment operator = associates a name and a value. Names or identifiers must begin with a non-numeric character or underscore but may contain both numeric and non-numeric characters.

```
principal = 1000
```

Different from C where a name represents a fixed size and location in memory into which results are placed.

Variables and Arithmetic Expressions

Example:

```
principal = 1000        # initial amount
rate = 0.05             # interest rate
numyears = 5            # number of years
year = 1
while year <= numyears:
    principal = principal*(1+rate)
    print year, principal
    year += 1
```

Conditionals

The if and else statements can perform simple tests.

Example:

```
#Compute the maximum (z) of a and b
if a < b:
    z = b
else:
    z = a
```

The bodies of if and else clauses are denoted by indentation. The else claus is optional.

Conditionals

To create an empty clause, use the pass statement.

Example:

```
if a < 0:
    pass # Do nothing
else:
    z = a
```

Conditionals

Boolean expressions can be formed using the or, and, and not keywords

Example:

```
if b >= a and b <= c:
    print "b is between a and c"
if not (b < a or b > c ):
    print "b is still between a and c"
```

Conditionals

To handle multiple-test cases, use the elif statement.

Example:

```
if a == '+':
    op = PLUS
elif a == '-':
    op = MINUS
elif a == '*':
    op = MULTIPLY
else:
    raise RuntimeError, "Unknown operator"
```

File Input and Output

The following program opens a file and reads its contents line by line.

```
f = open("foo.txt")    # Returns a file object
line = f.readline()    # Invokes the readline() method on file
while line:
    print line,        # trailin ',' omits newline character
f.close()
```

File Input and Output

Similarly, you can use the write() method

```
f = open("out", "w")   # Open a file for writing
while year <= numyears:
    principal = principal*(1+rate)
    f.write("%3d    %0.2f\n" % (year, principal)) # File output
f.close()
```

Strings

To create string literals, enclose them in single, double or triple quotes.

Examples:

```
a = "Hello World"
b = 'Python is groovy'
c = """Sino si Pepito Biglangliko?"""
```

The same type of quote used to start the string must be used to terminate it.

Strings

Triple-quoted strings capture -all- the text that appears before the terminating triple quote. Single and double quoted strings must be on one logical line.

Triple-quoted strings are useful when contents of the string span multiple lines of text.

Example:

```
print '''Content-type: text/html


<h1>Hello World</h1>
Click <a href="http://www.python.org">here</a>.
'''
```

Strings

Strings are sequences of characters indexed by integers starting at zero. To extract a single character, use the indexing operator s[i] like this:

Example:

```
a = "Hello World"
b = a[4]          % b = ’o’
```

Strings

To extract a substring, use the slicing operator string[i:j].

This extracts all elements from string whose index k is in the range i <= k < j.

If either index is omitted, the beginning or end of the string is assumed, respectively.

Examples:

```
c = a[0:6]    # c = "Hello"
d = a[7:]     # d = "World"
e = a[3:8]    # e = "lo Wo"
```

Strings

Strings are concatenated with the plus (+) operator:

Example:

```
g = a + " This is a test"
```

Strings

Other datatypes can be converted into a string using either str() or repr() functions or backquotes (‘), which are a shortcut notation for repr().

Example:

```
s = "The value of x is " + str(x)
s = "The value of y is " + repr(y)
s = "The value of y is " + ‘y‘
```

Lists and Tuples

Just as strings are sequences of characters, lists and tuples are sequences of arbitrary objects. You can create a list as follows.

Example:

```
names = [ "Eric", "Trixie", "Coley" ]
```

Lists and Tuples

Lists are indexed by integers starting with zero. Use the indexing operator to access and modify individual members of the list.

Example:

```
a = names[2]    # Returns the third element of the list "Coley"
names[0] = "pusakat"  # Changes the first element of the list to "pusakat"
```

Lists and Tuples

To append new members to a list, use the append() method.

Example:

```
names.append("Khamir")
```

Lists and Tuples

You can extract or reassign a portion of a list by using the slicing operator.

Example:

```
b = names[0:2]  # Returns [ "pusakat", "Trixie" ]
c = names[2:]   # Returns [ "Coley", "Khamir" ]
names[1] = 'Eric'  # Replace the 2nd item in names with 'Eric'
names[0:2] = [ 'Eric', 'Trixie', 'Coley' ] # Replace the first two elements
                    # of the list with the sublist on the right
```

Lists and Tuples

Use the plus (+) operator to concatenate lists.

Example:

```
a = [1,2,3] + [4,5]  # Result [1,2,3,4,5]
```

Lists and Tuples

Lists can contain any kind of Python object including other lists.

Example:

```
a = [1, "Dave", 3.14, ["Mark", 7, 9, [100, 101]], 10]
```

Nested lists are accessed as follows:

```
a[1] # returns "Dave"
a[3][2] # returns 9
a[3][3][1] # Returns 101
```

Lists and Tuples

Some advanced features of lists

Example:

```
import string                # load the string module
import sys                   # load the sys module
f = open(sys.argv[1])        # filename on the command line
svalues = f.readlines()      # read all lines into a list
f.close()

fvalues = map(string.atof, svalues)

print "The minimum value is ", min(fvalues)
print "The maximum value is ", max(fvalues)
```

Lists and Tuples

Closely related to lists is the tuple datatype. You create tuples by enclosing a group of values in parentheses or with a comma-separated list.

Examples:

```
a = (1,4,5,-9,10)
b = (7,)   # this is a singleton
person = (first_name, last_name, phone)
person = first_name, last_name, phone # same as previous line
```

Tuples support most of the same functions as a list except that you cannot modify the contents of a tuple after creation. (immutable object)

Loops

The simple loop shown earlier used the while statement. The other looping construct is the for statement, which iterates over the members of a sequence, such as a string, list or tuple.

Example:

```
for i in range(1, 10):
    print "2 to the %d power is %d" % (i, 2**i)
```

The range(i, j) function constructs a list of integers with values from i to j-1. If the starting value is omitte it's assumed to be zero. An optional stride or step size can be given as a third argument.

Loops

The range(i, j) function constructs a list of integers with values from i to j-1. If the starting value is omitted, it's assumed to be zero. An optional stride or step size can be given as a third argument.

Examples:

```
a = range(5)    # a = [0,1,2,3,4]
b = range(1,8)  # b = [1,2,3,4,5,6,7]
c = range(0,14,3) # c = [0,3,6,9,12]
d = range(8,1,-1) # d = [8,7,6,5,4,3,2]
```

Loops

The for statement can iterate over any sequence type and isn’t limited to sequences of integers.

Example:

```
a = "Hello World"
# Print out the characters in a
for c in a:
    print c


b = ["Eric", "Trixie", "Coley", "Khamir"]
# Print out the members of a list
for name in b:
    print name
```

Loops

range() works by constructing a list and populating it with values according to the starting, ending and stride values. For large ranges, this process is expensive in terms of both memory and runtime performance. To avoid this, you can use the xrange() function.

Example:

```
for i in xrange(1,10):
    print "2 to the %d power is %d" % (i, 2**i)

a = xrange(100000000)   # a = [0, ..., 100000000]
b = xrange(0,100000,5)   # b = [0,5,10,...,100000]
```

Instead of creating a sequence populated with values, the sequence returned by xrange() computes its values from the starting, ending and stride values everytime it’s accessed.

Dictionaries

A dictionary is an associative array or hash table that contains objects indexed by keys.

You create a dictionary by enclosing values in curly braces ({ }) like this:

```
a = {
        "username" : "xenos",
        "home" : "/home/xenos",
        "uid" : 500
    }
```

Dictionaries

To access members of a dictionary, use the key-indexing operator.

Example:

```
u = a["username"] # Returns "xenos"
d = a["home"]     # Returns "/home/xenos"
```

Dictionaries

To insert or modify objects, you assign a value to a key-indexed name.

Examples:

```
a["username"] = "trixie"
a["home"] = "/home/trixie"
a["shell"] = "/usr/bin/tcsh"
```

Dictionaries

Although strings are the most common type of key, you can use many other Python objects, including numbers and tuples. Some objects, including lists and dictionaries cannot be used as keys, because their contents are allowed to change.

Dictionaries

Dictionary membership is tested with the has_key() method.

Example:

```
if a.has_key("username"):
    username = a["username"]
else:
    username = "unknown user"
```

This can also be performed more compactly this way.

```
username = a.get("username", "unknown user")
```

Dictionaries

To obtain a list of dictionary keys, use the keys() method.

Example:

```
k = a.keys()  # k = ["username", "home", "uid", "shell" ]
```

Use the del statement to remove an element of a dictionary.

```
del a["username"]
```

Functions

You use the def statement to create a function.

Example:

```
def remainder(a,b):
    q = a/b
    r = a - q*b
    return r
```

To invoke the function, simply use the name of the function followed by its arguments enclosed in parenthesis.

Example:

```
result = remainder(37,15)
```

Functions

You can use a tuple to return multiple values from a function

Example:

```
def divide(a,b):
    q = a/b        # If a and b are integers, q is an integer.
    r = a - q*b
    return (q,r)
```

When returning multiple values in a tuple, it's often useful to invoke the function as follows:

```
quotient, remainder = divide(1456,33)
```

Functions

To assign a default value to a parameter, use assignment in the def statement.

Example:

```
def connect(hostname, port, timeout=300):
```

When default values are given in a function definition, they can be omitted from subsequent function calls.

Example:

```
connect('www.python.org', 80)
```

You can also invoke functions by using keyword arguments and supplying the arguments in arbitrary order.

Example:

```
connect(port=80,hostname="www.python.org")
```

Functions

When variables are created or assigned inside a function, their scope is local. To modify the value of a global variable from inside a function, use the global statement.

Example:

```
a = 4.5
def foo():
    global a
    a = 8.8    # Changes the global variable a
```

Classes

Exceptions

Modules

Books and Websites

http://www.python.org
http://www.diveintopython.org
http://www.vex.net
http://www.pygame.org
http://www.upm.edu.ph/~xenos

"Programming Python" by Mark Lutz
"Learning Python"
"Core Python Programming" by Wesley Chun
"Teach Yourself Python in 24 Hours"
"Python Essential Reference" by David M. Beazley