PYTHON

"To describe something as clever is NOT considered a compliment in the Python culture." – Alex Martelli, *Python Cookbook*.

History:

Python was invented by Guido van Rossum in the late 1980s. It was named after Monty Python's Flying Circus. Its main influence was ABC, a learning language he was part of designing in the 80s. It was aimed at being more similar to Unix "without being Unix-bound." Besides ABC, it was heavily influenced by the Modula-3 language. It also borrowed some of C's "least controversial features."

Python uses indentation as a syntactic feature. This was borrowed from ABC because van Rossum liked how it improved uniform program style and made it easier to read someone else's code. Van Rossum wanted Python to encourage modularity and reuse of code. One of the many ways in which it achieves this goal is by limiting the ways a given task can be accomplished. One of the main nineteen philosophical credos python holds is: "There should be one-- and preferably only one --obvious way to do it." This means that code with similar function will look like other code like it and easy to understand. Without unnecessary variability it is also easier to maintain code because other programmers can modify it easily. It also caters to many different styles of programming. According the Wiki about Python it is meant to be "multi-paradigm," working in Object Oriented, Structured, and Functional styles equally easily.

Python's philosophy has earned itself quite a following. It is easy to learn and is especially comfortable for programmers who have already learned another language. Because it is designed for ease of use it maintains that "Correctness and clarity [come] before speed." And that is one of the biggest criticisms of python. The core libraries are written in a very straightforward style: the developers are not worried about speed and have made choices that have hurt performance in order to preserve the readability and maintainability of the code.

[References: www.python.org/dev/culture, http://python-history.blogspot.com, http://www.python.org/dev/peps/pep-0020/, for an easy to reach version of the last reference run your Python interpreter and enter: "import this"]

Basic Concepts:

Dynamic Typing

The most important feature of Python to grasp is its Dynamic Typing. What this means, is that you don't have to declare a variable. When you need one, you just use it. For example:

```
A = 15
```

creates a variable named "A" and loads it with the value 15. Using the notion of a variable as a four tuple, the following tuple is created: [A] - [int] - [<memory location> [15]]. It is now an integer until you make it otherwise by a legitimate assignment statement. Python is strongly

typed, however, which means that string operations cannot be applied to integers (so no automatic concatenation between int and str objects).

A consequence of dynamic typing is that most python programs will load just fine as long as there are no obvious syntax errors. Until a line is executed, the interpreter has no way of knowing whether a semantic error is present, so testing must be done on a line-by-line basis.

Simple Syntax

Python enthusiasts pride their language for its simple to learn syntax and its ability to enforce the production of readable code. It uses white space as an important syntactic element. In place of curly braces, code of the same indentation is considered "blocked" together, so compound statements in "if," "while," "for," and "try/catch" blocks are indented (typically 4 spaces, NOT a TAB character) more than their conditional statements are. Even more intuitively, colons ":" are used to mark the end of a condition for "if" "while" and "for" blocks.

Interpreted

Python is an "interpreted" language. This is related to it being a dynamic language, in that it means that each line of code is converted to machine code as the program is running, each time a new line is encountered. This means that when an error occurs in a python program you are privy to a detailed look at the code which violated a rule. It also means that you can run the programming language in interpreter mode: you just enter the code you want run and the code gets executed.

This means that testing is easy because you can import a file you've been working on and run methods or create instances of classes you've been creating and then look at the variables as you mess with the instances.

Another consequence of the interpreted nature of Python is that there are some "introspective" functions especially useful to you in interpreted mode: dir() lists all objects in existence and all global functions or variables available to you. dir() with an object as a parameter lists all the methods and variables of that parameter. help() with an object as a parameter functions similarly to the man file in Linux, giving usage information and documentation.

Object Orientation

Python claims to be completely object oriented, and in many ways does behave much like an object oriented language. Everything is an object in Python, from integers to sockets. But Python does not support Encapsulation.

Program Organization

Python is designed to be easy to modularize. Each file can contain as much or little as you want as long as it's syntactically correct. Files that contain other files (called packages) can be used to pull together multiple files and objects into a package. When you import the file or package, any un-indented lines in the code get executed.

Other organizational features of python include functions, classes, methods of classes, and lambda functions. All these help to organize and break down a program. It is part of its "multi-paradigm nature.

Indentation is an important part of the organization of code on a lower level. This leads to an important fact: *All lines of code must be on one line* unless the '\' character is used to indicate that it will be continued on the next line. When that character is used, indentation for the remainder of the line is ignored.

Identifiers

Python identifiers are case sensitive and unlimited in length.

Identifiers must:

-Start with a letter or underscore (Good: `level`, `_counter`. Bad: `4level`, `^counter`)

-Contain only letters, digits, or underscores

The "_" identifier is reserved for use in the interpreter.

Names beginning and ending with "__" (double underscore) are reserved by convention for the Python interpreter and standard library functions.

Names beginning with "__" are "mangled" and treated as private variables of a class.

[References: http://docs.python.org/reference/lexical_analysis.html]

Random Picture before a table because word formats things weird.

Reserved Words (The following assumes a degree of familiarity with Java. For more detailed explanations of each reserved word see the Python documentation):

`and`	Short circuit Boolean "and" operator, like Java "&&"
`as`	Used in importing modules to allow module aliasing
`assert`	Raises an error when the condition given is false with the message given: assert <condition>, <message>
`break`	Same as java "break" statement
`class`	Use to define a class: def class <class name>: <statement list>
`continue`	Same as java continue statement
`def`	Used for defining functions and classes (the following defines a function): def <function name>: <statement list>
`del`	Used to remove an element from a collection, also used to "remove the binding of that name from the local or global namespace." Can delete portions or the entirety of a list without error
`elif`	Same as java "else if"
`else`	Same as java "else"
`except`	Same as java "catch"
`exec`	Allows for dynamic library code execution: takes a string (of correct python code), an open file, or a filename and executes the code

`finally`	Same as java “finally”
`for`	Similar to java “for” except closer to “foreach” in C# or C++. Iterates over each element of a collection or each element in a range: for i in range(0, 100): <statement list>
`from`	Used in import statements: from <module name> import (<the functions or objects you want> | *)
`global`	Used to explicitly refer to something in the global namespace, especially needed when assigning to something global while you are in a tighter scope (which would usually just create a new variable)
`if`	Same as java “if”
`import`	Import keyword for importing modules
`in`	Membership test, to see if an element is in the collection in question: <object> in <collection> #returns true if the #object is in the collection
`is`	Determines if the objects you’re comparing are the same object (object identity test).
`lambda`	Defines an anonymous function, much like you can in Lisp.
`not`	Same as java “!”
`or`	Short circuit Boolean “or” statement, like java “||”
`pass`	A “no-op” line of code. Useful where syntax demands a legitimate line of code, like an empty class definition (can be used like a struct). Also allows for busy wait:

	while true: pass
`print`	Like C++'s "cout" or "System.out.println()" in Java, but requires no parentheses. Concatenation of different types is done with commas: Print <expression>, <diff-type-expr>, value, etc…. Print can take formatting instructions like FORTRAN's
`raise`	Same as java "throw"
`return`	Same as java "return"
`try`	Same as java "try"
`while`	Same as java "while"
`with`	Used in controlling objects which require special setup and tear down (see documentation for details)
`yield`	Very complicated keyword, used for making functions have memory, allowing them to behave like a collection so you can use the function in a for loop. Vaguely related to the return statement. Don't hurt your brain doing this.

Globally defined Keywords:

`none`: Similar to "null" in C++, C#, or Java, etc.

The "none" keyword cannot be overwritten, although it is not considered a reserved word.

True: the value "true"

False: the value "false"

These variables are globally defined, and while you can use them as you would any other legitimate variable name you should NOT do so.

[References: Explore www.python.org for details.]

Data Types and Variables:

Basics

Because of dynamic typing, variables themselves do not have a type, the *value* has a type.

Primitives

Boolean – evaluates either true or false

bool: Boolean operators: ‘or’ ‘and’ ‘not’ – ‘or’ , ‘and’ both short-circuit

Numeric:

int – 32 bit precision. Ex: 1, -234

long – integer with unlimited length/precision.

float – real number, precision determined by system. Ex: 4.0

complex: a tuple of doubles. Each one has the field, “re” and “im” for access. Ex: 4+5

String – str() can be denoted either with single (‘) or double (“) quotes. Ex:

‘foo’, “hello”

Comparison

The comparison operators are usable on all objects:

< - strictly less than

<= - less than or equal

> - strictly greater than

>= - greater than or equal

== - equal to

!= - not equal to

is - tests object identity

is not - negated object identity

Structured Data

`tuple` – immutable sequence of an arbitrary number of objects (can be different types). Items can be identified by position.

```
x = (1, "String", 2)
```

```
x[1] = "String"
x[0:1] # returns (1, "String")
```

`list` – mutable sequence of an arbitrary number of objects (can be different types), separated by commas and enclosed in square brackets. Items can be identified by position.

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

`dict` – mutable map of objects; groups keys and values, enclosed in brackets. Ex:

```
{'Hello': 1, 'World': 2}
```

`set` - unordered object that contains an arbitrary number of objects (can be different types), with no duplicates. Ex:

```
{1,4.0, 2}
```

`frozenset` – same as a set, but immutable

Assignment Statements:

Simple Assignment

Assignment is done using the "=" operator like in Java. No type needs to be assigned to the variable because Python will automatically assign each variable a type when you assign a value to the type. Examples:

```
A = 15
TempMethod = lambda(…) #For more on Lambda Functions see below…
local_list = [1, 2, 3, 6, 4, 18]
B = "Hello my friends!"
```

Lambda Functions

Lambda functions can be created on the fly and used wherever a method is expected. They are "syntactically restricted to a single expression," although this has not prevented creative Python programmers from finding ways around this. One of their most useful functions is in filters, maps, and reducing functions.

The `filter()` function acts on each element of the list or string it is applied to and returns a list without the elements that fail whatever method you hand it as its first parameter. `map()` takes a collection and returns a new list with each element mapped to whatever the function returns when given that element of the list as a parameter. `reduce()` works like a

summation or production: it takes a function with a binary parameter list and applies it to the first two elements, and then to the result of that function and the next element until the end of the list.

Lambda functions are very useful for defining such actions because instead of defining an entire method for the sole purpose of using it once, you can simply define the function right in the parameter list. The structure is:

lambda_form ::= "lambda" [parameter_list]: expression

You can assign them to variables and use them that way, or you can simply use them within the parameter list of a function requiring a function name in place of a standard method name. Examples:

#The following creates an incrementor

```
incrementor = lambda x: x + 1
```

#The following uses a lambda function to filter a list of strings of all names containing '2' or '3':

```
list_of_strings = ['Anthony', 'Chris', 'R2-D2', 'Neal',
'William', 'C3PO']

filter(lambda x: (x.find('2') > -1 or x.find('3') > -1),
list_of_strings)
```

[References: http://www.secnetix.de/olli/Python/lambda_functions.hawk, http://docs.python.org/tutorial/controlflow.html#lambda-forms, http://docs.python.org/tutorial/datastructures.html]

Control Statements:

Conditional Statements

Example of the basic "If":

```
if x<5:
        x = 5
else:
   x = 0
```

Example of a more complicated "If":

```
if (x<5 and y<5):
   x = 5
elif (x>=5 or y>=5):
   y = 5
else:
   x = 0
   y = 0
```

Indefinite Looping

Example of the basic “While”:

```
while x < 5:
   x + 1
```

Another example:

```
while condition == true:
   #Do the following
   …
   condition = false.
```

Infinite “While” Loop:

```
while 1:
   x + 1
   print x
```

Another Infinite “While” Loop:

```
while True:
   x + 1
   print x
```

Definite Looping

Python’s “for” loops behave differently from loops seen in C in that they iterate over lists instead of just incrementing integers.

Example of a basic “For” loop:

```
for item in list:
   doSomething(item)
```

Here are two example loops, first in C then in Python:

C example of a for loop:

```
for(int i = 1; i < test.length; i++)
{
   #statements using “i”
}
```

Python example of the same loop:

```
for x in range(0, test.length):
```

```
   #statements using "x"
```

Program Flow Modification

Python, like C/C++ uses "pass," "continue," and "break":

Example use of "continue":

```
for x in range(0, 100):
   if x < test.length
      continue
   # doSomething to x
```

Example use of "break":

```
while 1:
   if x = 0:
      break
   else:
      print x
      x - 1
```

Example use of "pass":

```
while 1:
   if x < 10:
      pass
   elif x = 10:
      # do something with x
   else:
      # do something else with x
```

Case Structures in Python

In order to have a case structure set up like in C/C++, you must use the if, elif, and else statements. No direct equivalent exists, although the following structure will behave identically to a switch statement.

Example of case structure in python:

```
number = raw_input("Enter a number:  ")
if number < 10:
   #do something
elif (number >= 10 and number <100):
   # do something else
else:
   # perform the default case
```

Modularity:

Modules

Python supports the creation of different modules. A module has a distinct namespace, and any number of classes and functions. A user-defined module resides in its own file, something like modulename.py and may be imported by entering "import modulename" into the interpreter. Classes and functions in the module may then be accessed through the module namespace: modulename.ClassName. Module namespaces can be reassigned on import by use of the reserved word "as." "import modulename as m" allows "m.ClassName" instead of the original approach. In addition, module elements may be added to the current namespace by using the reserved word from. For instance, entering "from modulename import *" allows access to all the elements of modulename in the current namespace. You can import particular elements by replacing * with the desired element.

Callable objects

The specification for Python states that "A call calls a callable object (e.g., a function) with a possibly empty series of arguments" Python supports callable user-defined functions, built in functions, class objects, class instance methods, and class instances. A call to a callable object always raises some value which includes the value None unless the call raises an exception.

User-defined functions follow the syntax:

```
def function_name(argumentlist):
```

{executable statements}

{return statement}

Classes follow the syntax:

```
class ClassName:
```

{Class method declarations}

where method declarations look exactly like those for user-defined functions with one addition: the first argument must be "self." When calling class methods however, the self argument is supplied by the object itself, and is thus not explicitly stated.

Consider the following examples:

User-Defined functions follow the syntax:

```
def demo_function(message):
```

```
    print message
    return 1
```

Classes follow the syntax:

```
class MyClass:
    def __init__(self, width):
        self.width = width
        self.inputExpected = True
    def to_string(self):
        return "Input is expected " + str(inputExpected)
```

All calls in python work by using a parameter list. The parameters are copied for use by the code before the first statement of the code body is executed. Python supports default parameters as well as keyword parameters. Default parameters are calculated once, when the callable is defined. For this reason mutable objects such as lists and dictionaries should not be used for default parameter values. A Python call may be used anywhere where an expression is expected.

IO Functionality:

Console I/O

Input from the console is done through the raw_input function. raw_input takes a prompt as its parameter, and returns whatever the user types before pressing enter.

Output from the console is produced using the print keyword or the print function. Print accepts an arbitrary number of arguments. A new line is written after the last argument unless the print statement ends with a comma.

The following is a very simple echo program:

```
def echo:
    data = raw_input("Please enter something: ")
    print data
```

File I/O

File objects are opened using the `open` function. `open` has two parameters, the first is the filename of the file to open, and the second is the mode. 'r' opens the file for reading, 'w' for writing, 'a' for appending, 'r+' for both reading and writing. b may be appended to each of the

modes to specify binary mode on windows machines. It does not hurt to add the b on Unix systems, and ensures the portability of your project.

File objects support the operations `read`, `readline`, `readlines`, `write`, `tell`, `seek`, and `close`. `read` takes an optional size parameter which specifies the number of bytes to read. When the end of file is reached, “” is returned.

`readline` reads a single line from the file and the new line character is left on all lines except the last line of the file.

`readlines` returns a list containing all the lines in the file. An optional parameter `sizehint` causes the method to read that many bytes and enough more to finish off a line.

`write` takes a string parameter and returns `None`.

`tell` returns the file objects current position in the file measured in bytes from the beginning of the file.

`seek` changes the file objects current position. `seek` has two parameters, `offset` and `from_what`. The new position is computed by adding the offset to the choice indicated by `from_what` using 0 as the file beginning, 1 as the current file position, and 2 as the file end.

`close` releases any system resources being used by the file object. Any attempts to use the file object after calling close will automatically fail. You should always close a file when you are done with it.

Please consider the following example:

```
filename = raw_input(“File to open: “)

infile = open(filename, r+)

lines = infile.readlines()

infile.close()

for line in lines:

     print line
```