Python notes

Introduction

Python is an extremely powerful and dynamic object-oriented programming. There are several reasons to use Python one of the biggest draws being portability. Python is able to run on almost operating system. In addition python is fairly easy to pick up

Invoking Python

On windows the python interpreter is invoked by launching the `cmd` utility and entering in `python` which should give you a prompt similar to the one below.

```
Python 2.6.6 (r266:84297, Aug 24 2010, 18:46:32) [MSC v.1500 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information
>>>
```

Functions

One of the most basic building blocks of any programming language are functions. In python functions are defined with the `def` keyword. Functions consist of two main parts, the function declaration and the function body.

```
>>> def functionName():       #Decleration
>>>    print "Hello World!"   #Function body
>>>
>>> functionName()
Hello World!
```

One job of functions is to accept arguments and optionally return a value.

```
>>> def argFuction(a, b=3):
>>>     return a + b
>>>
>>> print argFuction(1,2)
3
>>> print argFuction(1)
4
```

Something that you may have noticed in the preceding example is that we set the second argument `b` to equal 3. This is what is called a `default argument` which allows us to leave out the second argument in the last function call.

Variables

Another basic building block of any programming language is variables. Variables consist of three components `type`, `identifier`, and a `value`. Python is what is known as a `dynamic language`. What this means is that you are not required to include a type declaration, in contrast to strongly typed languages such as *Java* or *C++*.

Java

```
>>> int number = 5;
>>>
>>> System.out.print(number);
5
```

Python

```
>>> number = 5
>>>
>>> print(5)
5
```

Conditional

Python supports conditional statements in the form of `if`, `else` and `elif`. The way if statements work is the code below the if statement is executed if the condition is true.

```
>>> x = True
>>> if x == True:
>>>     print "x is true"
>>> elif x == False:
>>>     print "x is false"
>>> else:
>>>     print "x is neither"
x is true
```

In the example above we used the equals operator `(==)` to test if x was true. In reality we wouldn't need to compare x to the `boolean` True since `if` checks for a true statement to begin with.
In addition to the `==` operator there are also a number of other boolean and comparison operators available in python which we'll show in the tables below.

Boolean Operators

x `or` y	Evaluates to true if either x or y are true
x `and` y	Evaluates to true if x and y are true
`not` x	Negates the true false value of x

Comparison Operators

x `>` y	Evaluates to true if x is greater than y
x `<=` y	Evaluates to true if x is less than or equal y
x `<` y	Evaluates to true if x is less than y
x `>=` y	Evaluates to true if x is greater than or equal to y
x `==` y	Evaluates to true if x and y are equal
x `!=` y	Evaluates to true if x and y are not equal
x in [1,2,3]	Used to test if x is in a list, tuple, string or dictionary

Looping Constructs

Python features several constructs which you can use to repeat a statement. The first of these expressions is the `while` loop. The `while` loop repeats the code below the loop while the condition is true.

```
>>> x = 1
>>> while x < 10:
>>>     print(str(x) + ","),
>>>     x += 1
>>>
1,2,3,4,5,6,7,8,9,
```

Another type of loop in Python is the `for` loop. The syntax of the for loop is `for item in sequence: block`. Each loop item is set to the next item in the sequence and the block of code is executed.

```
>>> word = "Python"
>>> for ch in word:
>>>     print(ch + "-"),
>>>
P-y-t-h-o-n-
```

There are times when your working with loops when you may need to break out of the loop. Two common ways of doing this is through the keywords `continue` and `break`. The first keyword `break` will like the name implies break you out of the loop.
You may notice in our previous example the hyphenated word 'Python' has a trailing dash, one way to deal with this is to use the break keyword in combination with an `if` statement.

```
>>> word = "Python"
>>> for ch in word:
>>>     if(ch == word[-1])
>>>         print(ch),
>>>         break
>>>     print(ch + "-"),
>>>
P-y-t-h-o-n
```

The `continue` keyword is another programming construct that will allow you to skip an iteration of a loop. Lets take a look at an example in which we skip the 4th element in a list.

```
>>> a = [1,2,3,4,5,6,7]
>>> for x in a:
>>>     if x == a[3]
>>>         continue
>>>     print x,
>>>
1 2 3 5 6 7
```

Python Classes

Object oriented programming *(O.O.P.)* is a programming paradigm that is used to model real world objects and situations. Two of the main concepts involved in *O.O.P.* are classes and objects. To make an analogy to the real world, classes are like a blue print for a house and objects would be instances of that house.

Classes in Python are basically a collection of variables and functions. In the context of object oriented programming functions are referred to as methods and variables are referred to as fields, or attributes. Lets take a look at an example below.

```
>>> class WordPrinter():
>>>    word = "Hello World"
>>>    def printWord(self):
>>>       print self.word
>>>
>>> wp = test()
>>> wp.printWord()
>>>
Hello World
```

In the example above the class declaration starts with the keyword `class` followed by an identifier and a pair of parentheses. Following the declaration we have the variable definition `word` which is attached to the `WordPrinter` class. Similarly we have a function attached to the test class called `printWord` which will print the contents of the variable word.

Objects

In our analogy we compared objects to houses built from a set of blueprints. Like our real world example, we sometimes need to make adjustments to a class when creating it. This can be done through the use of a special method call the `constructor`.

Constructors in object oriented programming are special methods that are called when we create an instance of an object from a class. In order to define a constructor in a Python class we use the special function `init` function. Lets take the previous example and change it so that we can set it to hold a different message.

```
>>> class WordPrinter():
>>>     word = "Hello World"
>>>     def __init__(self, msg):
>>>         self.word = msg
>>>
>>>     def printWord(self):
>>>         print self.word
>>> wp = WordPrinter("Hello Jeorge")
>>> wp.printWord()
>>>
>>> Hello Jeorge
```

Inheritance

One important concept in object oriented programming is inheritance. Inheritance allows you to reuse parts of your code by deriving and extending a new class from a preexisting class. Lets take a look at how this is done in the example below.

```
>>> class Mammal:
>>>     weight = 0
>>>     name = 4
>>>
>>>     def __init__(self, weight, name):
>>>         self.weight = weight
>>>         self.name = name
>>>
>>>     def talk():
>>>         pass
>>>
>>> class Dog(Mammal):
>>>
>>>     def__init__(self,weight=50,name="Dog"):
>>>         self.weight = weight
>>>         self.name = name
>>>
>>>     def talk():
>>>         print "Woof Woof"
>>>
```

You may noticed in the *Mammal* class the function `talk` just has the word `pass` in the body. Since the python language relies on indentation to separate code it is necessary to place a statement in the body of a function. In python the keyword `pass` serves as a place holder for a function body.

Types

In programming it is important to differentiate between different data types you may use. In Python there are five basic data types:

- Numbers
- String
- List
- Tuple
- Dictionary

Numbers

Numbers in Python come in several different varieties including floating point, complex, integer and long types.

```
>>> numA = 3.2           # Floating point
>>> numB = 2             # Integer
>>> numC = 535633629843L # Long
>>> print(numA + numB)
>>>
5.2
```

In Python unlike other languages numbers are converted internally in an expression containing two types to a common type. However there are times when you will need to coerce a number explicitly from one type to another. For this python offers a number of built in conversion functions.

Numerical Conversion Functions

int(x)	converts x to a plain integer
long(x)	converts x into a long integer
float(x)	converts x into a floating-point number

In addition Python also includes a variety of functions and numerical operators that perform standard mathematical calculations.

Common Math Functions

abs(x)	Returns the absolute value of x
ceil(x)	Returns the ceiling of x
cmp(x,y)	Returns -1 if x < y, 0 if x == y and 1 if x > y
exp(x)	Returns the exponential of x: e^x
log10(x)	Returns the base-10 logarithm of x
pow(x,y)	Returns the value of x^y
sqrt(x)	Returns the square root of x
floor(x)	Returns the floor of x
log(x)	Returns the natural logarithm of x

Common Math Functions

a + b	Adds two numbers together.
a - b	Subtracts b from a.
a * b	Multiplies the variables a and b together.
a / b	Divides b by a.
a ** b	Raises a to the b power. a^b
a % b	Returns the remainder of b divided by a.

Strings

Strings are used in almost all languages to represent things like names, conversation and anything else that you could write using normal language. To create a `string` in Python you simply enclose a list of characters in quotes. Python treats single quotes the same as double quotes.

```
>>> var1 = "Hello World"
>>> var2 = 'Python is cool!'
```

String concatination

One common programming task is combining two strings. Combining strings in programming is referred to as string concatenation. In Python we combine strings by using the + operator

```
>>> var1 = "Hello"
>>> var2 = var1 + " Jeorge"
>>> print(var2)
>>>
Hello Jeorge
```

Another neat trick that you might not use as often is using the * operator with a string, for example.

```
>>> var1 = "Hello "
>>> var2 = var1 * 4
>>> print var2
>>>
Hello Hello Hello Hello
```

List

One of the most basic data structures in Python is `list`. Each element in a list is assigned a number, its position or index. In Python and most programming languages the indexing starts at zero.
List in Python are written as a sequence of comma-separated values between square brackets. These list are not required to be made up of all the same types.

```
>>> list1 = ['physics', 'chemistry', 1997, 2001]
```

You can access access values in list through their index. To access values in list, use the square brackets for slicing along with the index or indices to obtain the value at that index.

```
>>> list1 = [1, 2, 3, 4, 5, 6, 7]
>>> varA = list[0]
>>> varB = list[1:5]
>>>
>>> print(varA)
>>> print(varB)
>>>
1
[2, 3, 4, 5]
```

Updating List
List can be updated in a variety of differing ways. One common way of updating a list is to assign a new value to a particular index using bracket notation.

```
>>> list1 = [1, 2, 3, 4, 5, 6, 7]
>>> list1[3] = A
>>>
>>> print(list1)
>>>
[1, 2, 3, A, 5, 6, 7]
```

You can also use the extend function on list to concatenate to list to each other.

```
>>> list1 = [1, 2, 3]
>>> list2 = [4, 5, 6]
>>> list1.extend(list2)
>>> print(list1)
>>>
[1, 2, 3, 4, 5, 6]
```

Since list are used so heavily in Python list come with a number of built in methods that can be called on list to help you work with them.

Python List Methods

list.append(obj)	Appends object obj to list
list.count(obj)	Returns a count of how many times obj occurs in the list
list.extend(seq)	Appends the contents of seq to list
list.index(obj)	Returns the lowest index in list that obj appears.
list.insert(index,obj)	Inserts object obj into list at offset index
list.remove(obj)	Removes object obj from list
list.reverse()	Reverses objects of list in place.
list.sort([func])	Sorts objects of list, uses compare func if given.

In addition Python also have a number of higher order functions that can be called on list.

Python List Functions

cmp(list1,list2)	Compares elements of both list.
len(list)	Gives the total length of the list.
max(list)	Returns the item from the list with the max value.
min(list)	Returns the item from the list with the min value.
list(seq)	Converts a tuple into a list

Tuples

A tuple in Python is a sequence of immutable Python objects. Immutable simply means that once a value is set it can't be changed. Tuples are constructed similar to list except for a coma separated list between brackets we use parentheses.

```
>>> tup1 = ('physics', 'chemistry', 1997, 2000)
```

You can access values in tuples in the same way in which we access list.

```
>>> tup1 = ('physics', 'chemistry', 1997, 2000)
>>>
>>> print(tup1[2])
>>>
1997
```

p **Python List Functions**

cmp(tuple1,tuple2)	Compares elements of both tuples.
len(tuple)	Gives the total length of the tuple.
max(tuple)	Returns the item from the tuple with the max value.
min(tuple)	Returns the item from the tuple with the min value.
tuple(seq)	Converts a list into a tuple.

Dictionaries

A `dictionary` in Python is a type of data structure that can store any number of Python objects, including other container types such as list. Dictionaries consist of pairs of `keys` and their corresponding `values`.

```
>>> dict = {'Alice': 2341, 'Beth': 9102}
```

Each key is separated from its value by a colon, the items are separated by comas and the entire data structure is enclosed by brackets. To access elements in a dictionary you can use the familiar square brackets along with the key to obtain the value.

```
>>> dict = {'Name': 'Zara', 'Age': 7}
>>> print dict['Name']
>>> print dict['Age']
>>>
Zara
7
```

You can update a dictionary by adding a new entry or item, modifying an existing entry, or deleting an entry as shown in the following example.

```
>>> dict = {'Name': 'Zara', 'Age': 7}
>>> dict['Age'] = 8
>>> dict['Weight'] = 125
>>>
>>> print dict['Age']
>>> print dict['Weight']
>>>
8
125
```

Python Dictionary Functions

cmp(dict1,dict2)	Compares elements of both dictionaries.
len(dict)	Gives the total length of the dictionary.
str(dict)	Produces a printable string representation of a dictionary.

Python Dictionary Methods

dict.clear()	Removes all elements of dictionary dict.
dict.copy()	Returns a shallow copy of dictionary dict.
dict.fromKeys()	Create a new dictionary with keys from seq and values set to a value.
dict.get(key,default=None)	For key key, returns value or default if the key is not in the dictionary.
dict.has_key(key)	Returns true if key is in the dictionary dict, false otherwise.
dict.items()	Returns a list of dicts (key, value) tuple pairs.
dict.keys()	Returns list of dictionary dict's keys.
dict.update(dict2)	Adds dictionary dict2's key-value pairs to dict.
dict.values()	Returns list of dictionary dict's values.

Slicing Syntax

Python has an interesting feature for accessing elements in `list`, `tuples`, and `strings`. Slicing syntax is similar to accessing values in a list with bracket notation. Slicing is divided into three optional sections separated by a colon:

- start: Index you want to start your selection with.
- end : Index you want to end your selection with.
- step : Number of steps you want to take. (a value of 2 would return every other element)

Perhaps the best way to explain Pythons slicing syntax would be to go through some examples. One thing to note however is even though the following examples are done using a list, the same syntax can be applied to `strings` and `tuples`.

```
>>> a = ['A','B','C','D','E','F','G']
>>>
>>> a[:3]   # ['A','B','C']
>>> a[:4]   # ['E','F','G']
>>> a[3:5]  # ['D','E']
>>> a[3]    # D
>>> a[-1]   # G
>>> a[::-1] # ['G','F','E','D','C','B','A']
>>> a[::2]  # ['A','C','E','G']
```

Exception Handling

Something that is guaranteed as a programmer is that you'll eventually run into errors. One way of mitigating the problems that may stem from these programming errors is to use what is called exception handling. Handling exceptions involves the use of four main constructs; `try`, `except`, `finally` and `raise`.

One simple example where you might use exception handling is creating a function that takes two arguments and divides them.

```
>>> def reciprocal(a,b):
>>>     try:
>>>         r = a/b
>>>     except:
>>>         print('Exception caught')
>>>         return
>>>     return r
>>>
>>> print reciprocal(1,0)
>>> print reciprocal(8,2)
>>>
Exception caught
4
```

Another way of writing the preceding example is to raise an exception yourself. In the preceding

example python raises an exception when you try to divide by zero. In the example below we'll raise an exception ourselves using the `raise` keyword if the values are not numbers.

```
>>> def reciprocal(a,b):
>>>     if isinstance(a,str) or isinstance(b,str) == True:
>>>         raise ValueError
>>>     print a/b
>>>
>>> try:
>>>     reciprocal('a','b')
>>> except ZeroDivisionError as e:
>>>     print "Zero Division Error has been caught"
>>> except ValueError:
>>>     print "Value Error has been caught"
>>> finally:
>>>     print "Enter a non-zero term for b"
Value Error has been caught
Enter a non-zero term for b
```

The preceding example will still catch `ZeroDivision` errors if any should occur. However this time instead we pass the function values of the wrong type instead and a `ValueError` exception is raised by the `reciprocal` function and caught in the `try/except` block. The last branch of the `try/except` block `finally` is called whether or not an exception is raised or not.