Getting Started with Python

Introduction

This is designed to be a short guide to help you learn Python. Python is an open source, interpreted programming language with a simple syntax that's rather nice to work with, without any insane conventions or odd thoughts behind it.

Python doesn't enforce any specific paradigm (way of thinking). So, you can write things in many different ways (e.g.: procedurally, object orientated, functional). I recommend you become comfortable with the first two.

This guide is based on Python 2.4 and up. But not 3, as there's quite a few changes (and it's not in wide usage yet.)

It's also designed to supplement a talk, or if you're already pretty comfortable with programming, just the language itself. Extra points are gained by noticing the Monty Python references.

Syntax Basics

The print Function

The first bit of syntax you'll see is the print. Here we'll do the usual "Hello World!".

```
print 'Hello World!'
```

You can also easily concatenate strings...

```
a = 'Hello'
b = 'World!"

print a + b
```

As an aside, you can print any object which has a write() function. You'll find that lists and other data structures can be printed nicely to the console using print.

Commenting

In Python, commenting is done using the # character. This applies to both block and inline comments.

You'll often see three """ followed by """ for multiline comments, especially as code level documentation.

Indentation

Python is whitespace sensitive. This means that you need to observe the correct amount of space between the syntax. Which means that you need to indent things properly, otherwise you'll be thrown a ton of errors.

It is suggested that you use spaces to indent, and that you use 4 of them. You can use tabs but these might throw up a warning. You certainly shouldn't mix tabs and spaces.

Whilst it's a bit odd (among programming languages), it enforces good style, which once you start working with other people's code, you'll be thankful for.

Numerical Operations

Most of the mathematical operations are exactly the same in other languages. The table below explains a select few of them:

Function	Operator
Addition	`+`
Subtraction	`-`
Multiplication	`*`
Division	`/`
Modulo	`%`

Boolean Operators

In Python, true and false are declared as `True` and `False`. As with other languages, they default to 1 and 0 respectively.

However, when printing, they return their text versions.

Comparison Operators

As with numerical operators, all of the standard ones are here. Here's the one's you'll likely need most:

Function	Operator
Equal To	`==`
Not Equal To	`!=`
Greater Than	`>`
Less Than	`<`
Greater Than or Equal To	`>=`
Less Than or Equal To	`<=`

String Operations

Strings are defined by a group of characters wrapped in either "" or ''. They both work exactly the same, and can escape each other, so things like below are possible:

```
"S't'ring"
'S"t"ring'
```

You can replace inside a string by using the % operator, this is much like C's printf, e.g.:

```
'The Knights who said %s!' % 'Ni'
```

Other methods available on strings are:

```
S.capitalize()
S.replace(old, new)
```

You can specify Unicode strings by prefixing a 'u' on the "". (e.g.: u"this is ünicode"). You'll need this if you want to put in any special characters (as in, anything outside the usual latin alphabet.)

Flow Control

Flow statements allow you to descend through a block of code given different sets of criteria, making decisions according to values or other defined rules. The examples here cover the if, while and for statements.

if

Due to the whitespace requirements, you'll notice that we don't need to use any form of braces around the relevant code.

The if statement looks like this:

```
if test:
      // do stuff
```

The 'test' value is the comparison operator (above).

If you wish to have a fallback option (else), that would look like:

```
if test:
      // do stuff
else:
      // do stuff
```

while

The while loop is similar to the if statement. It looks like:

```
while something:
      // do stuff
```

You can also include an else on a while loop.

for

```
for target in sequence:
      // do stuff
```

The target can mean anything which could usually go before an = statement, so (x, y) would be valid.

Functions

Functions are snippets of code that you'll likely need to use again. This allows you to save yourself writing it over and over again, or, even better share it with others. Functions help underly procedural programming (as the program works it way through, it executes in the same order it was written, and down the code.)

```
def name():
      // do stuff
```

Functions are the same as methods (see below.) But, functions are not attached to a class.

Objects

Object-Oriented Programming, or OOP is a different style of programming. It's based around the idea of things existing as an object. Each object has a set of functions that it supports (known as methods), and properties to hold data (known as members, or instance or class variables.)

OOP fits modeling real-world things pretty well. You'll probably hear of various analogies when people try to explain it (animals and cars are popular.) But, it's a bit beyond the scope of this guide.

Like most Object-Orientated languages, these declared using the class operator.

A few points on objects:

- all methods in python are public
- methods are declared using the function operator (as above)
- they support multiple-inheritance (a class a acquire the methods of multiple others)

Example:

```
class MyClass:
      def f(self):
            print 'hello world'

x = MyClass()

x.f()
```

Note: Methods are hidden by using underscores. In Python however, methods are usually only made pseudo-private to stop name clashes. The convention is for two underscores.

Class & Instance Variables, Initializers and Other Special Bits

Class Variables are defined in the class definition itself. Instance variables are defined inside the initialiser. Like so:

```
class MyClass:
      classVariable = 0

      def __init__:
            instanceVariable = 0
```

Just like Java has the convention of toString() to provide a string representation of an object, Python has something similar. As mentioned above, Python's print operator looks for a method called write() to use for formatting.

Python supports multiple inheritance. The classes which another class inherits from are listed in brackets after the class definition, like so:

```
class MyClass(MyOtherClass)
```

Multiple inheritance is handled with commas:

```
class MyClass(MyOtherClass, OhGodNotMoreClasses)
```

Data Structures

Data Structures are ways of storing information. Some are more appropriate than others. It's up to you to work out the best one to use.

Python sports a nice set of built in data structures, the most common are lists and dictionaries:

Lists (Arrays)

Lists (known as Arrays in other languages) are ordered bits of data. In Python, lists are mutable (can change) and can hold any type of object.

```
x = [0, 'Me']
```

The code above is a simple list. The elements are accessed using an index (which starts with 0, if you were wondering).

Dictionaries

Dictionaries are key-value stores. Sometimes these can be a better way to store data, especially if it is well defined (like a person's details).

```
y = {'number' => 0, 'word' => 'Me'}
```

They are accessed using the key. So:

```
y['number'] // gives us 0
```

Note: As both lists and dictionaries are objects, they can be contained within one another.

The standard library has support for several more too and SciPy & NumPy provide a much wider range of data structures for the storage of more complex data, e.g.: sparse matrices. These libraries also provide very quick ways to handle matrix algebra, et. al. But, unless you're doing that kind of thing, you don't need to worry about it (it's quite specialised.)

The Packaging System

The final bit of the main parts of Python that you *need* to know about is the packaging system. Python has a wide range of packages covering everything from serial device access to web services in the "Python Packaging Index" or PyPI. This is what makes Python shine. There's lots out there to allow you to easily build cool stuff.

You can browse PyPI from: http://pypi.python.org/pypi

There are two ways to install packages, `easy_install` and `pip`. You should install `pip` once using `easy_install` and then use `pip`, it's much better.

```
$ pip install <package_name>
```

These will install packages for you. They are then available inside your environment (see the environments heading below.)

Packages from PyPI follow a standardised convention. Regardless of what the package is designed to do, accessing it will be done in a similar manner. But often, packages will be made up of multiple "modules" (the same applies to the Python standard library, too), these contain a collection of the relevant methods and data for a given task, nicely compartmentalised.

There's two concepts to get to grips with here. 1. Packages are slightly special modules. 2. Python uses modules to split up code where it can. So, you'll see the same operators for several things.

```
# import everything from a module (very bad.)
from <module> import *
# import a specific part of a package
from <module> import <function>
# or, just allow referencing outside code
import <module> # then, you'll call <module>.<function>
```

You shouldn't use the first because you don't quite know what you'll be importing and then using. It's possible that you'll reference something that later on you won't realise you've used, causing a more specific import to break.

Just importing the module keeps the module under it's own namespace. This keeps the imported code more separate at the code level. It's worth pointing out here: "import" isn't like C's "<include file.h>", the imported code is isolated when it's executed, not just added to the file on runtime.

More Python

Now, it's going to get a bit more complicated. Don't worry if you don't understand, there's a lot less explanation. But, hopefully it'll tickle the fancy of people who have programmed before and for those who want to plough ahead.

Environments with Virtualenv

Virtualenv allows you to define an environment to work within. This ensures that projects cannot clash with each other, and you don't add in dependencies without first defining them. It also makes it much easier to depend on a specific version of a package in one project, and in another use something different. Virtualenv is a very useful tool.

```
$ pip install virtualenv
$ virtualenv --distribute venv
$ source venv/bin/activate
```

This will create an environment inside the current directory (say, the root of your project repository) called "venv" and then activate it. Now, when you run "pip install …" packages will be installed here rather than to the whole system. Python will also only use packages from here.

Once you've finished using a specific environment, deactivate it:

```
$ deactivate
```

I create a new environment (often just called venv) at the root of each project. I then add it to the project's .gitignore so it's never committed. This ensures that if I ever hand the project over to someone else, or someone else collaborates it will work without much difficulty.

Exception Handling

Like most languages, the try is used as the main statement for exceptions. Most of the built in classes have quite a few levels of possible exceptions. It'll be up to you to look up those, though.

```
try:
    # your statements
except [clause]
```

The except underneath your statements will be called if an exception is raised. You need to replace "`[clause]`" with whichever expression you are trying to catch. You can catch as many exceptions as you like.

If you omit an except clause, it will act as a default value (especially useful in a block of exceptions).

You can add an "else:" statement to the end of a try/except block. The best way to show this being used is in reading a file:

```
try:
    f = open(arg, 'r')
except IOError:
    print 'cannot open', arg
else:
    print arg, 'has', len(f.readlines()), 'lines'
    f.close()
```

The "raise" statement is used to fire off an exception. To create a NameError with a message, you'd do the following:

raise NameError('HiThere')

To learn about creating your own exceptions, and more read the Errors section of the Python tutorial.

Load Arguments

If you are writing console applications, or using Python mostly in the shell, you'll want to understand how the load arguments work. If you have used any C-based languages, it's pretty much the same.

Firstly, you'll need to import the sys module. Then you can access the `sys.argv` array. The first element (0) is the file's name, then followed by any arguments. For example:

```
import sys

print sys.argv[0]
print sys.argv[1]


$ python argv.py hello
   argv.py
   hello
```

Handling Files

Like the load arguments, if you've done any C, you'll recognise this quite well. The process for dealing with a file goes something like this: "open the file, and specify what you want to do with it, then read the contents, or write down to it".

However, when you read in a file it is an object.

Opening File Objects

Firstly, you will need to open up a file object. You will need to do this for read, writing or otherwise.

```
f = open('file', 'mode')
```

This will return a file object. The table below described the modes:

Mode	Function
r	Opens the file for reading.
w	Opens the file for writing.
r+	Opens the file for reading and writing.
a	Appends data to the bottom of the file.

If you exclude the mode, it will assume you wish to read.

Basic Reading and Writing

There are several functions for reading in files.

- `read()` loads the entire file.
- `readline()` reads each of the lines (this has a memory, so several stacked readline() calls will work your way through the file).
- `readlines()` will read each of the lines to a list.

However, you can also loop through the file object, which is likely the best way:

```
for line in f:
    print line,
```

`write()` is used to write a string to a file.

Once you've finished with the file, call `close()` on the object to get rid of it.

Sidenote: Persistence and Logging

If you're interested in implementing persistence (that is saving the state between running the program) you'll want to look at Pickle.

Python also has a nice logging library. It's a little bit out of the scope of this guide, but it's pretty awesome, so I figured it needed a mention.

Where to Go Next

This is only supposed to be a brief introduction which gets you up and running. In 10 pages, I can't go through everything you need to know about Python. I haven't even touched on how to style your code.

So, the best thing to go and do is make something! Someone will help if you get stuck.

Because of that, here's a list of things you should go away and read.

The Hitchhiker's Guide to Python – this is the best guide to getting going and where to look for getting started with other aspects of Python.
PEP008: Style Guide for Python Code – this explains the conventions for laying out code.
The Python Tutorial – this is the official introductory tutorial. You might find it better than this.
Python HOTOs – if you want to build something specific (say using curses) this is probably a good place to look. Also has stuff on sockets, functional programming and regular expressions.
What the Heck does 'Pythonic' mean? – A good guide to building things the Python way.

About / Changes

The original along with other resources (slides, notes, etc.) is available here: http://github.com/nickcharlton/python-tutorial

You can read more about it here: http://nickcharlton.net/posts/python-tutorial.html

Date	Changes
01/10/2012	Rephrasing and clarification. Adds virtualenv, emphasises pip and explains packaging better, mentions SciPy/NumPi, and more resource links.
??/??/2012	Removes tuples, sphinx. Adds an introduction, better explanation of the basics and comparison operators.
??/??/2011	First Version