

Kenny Pyatt's

Learning Python

The following information is Copyright Kenny Pyatt.
Please feel free to share this with as many people as you like.

© Kenny Pyatt. All Rights Reserved.

Forward

I created this to help several of the interns at our office. The goal was to shortcut their learning cycle and get them up to speed. I wrote it in the wiki at the office over the course of a weekend and a couple of week nights. Several of the interns completed the tutorials and helped me catch errors, typos, etc. I hope that no new errors were created in my copy paste from the wiki into Open Office.

I have noticed there is some wrapping issues in some of the code blocks. I have attempted to edit those and hopefully I have managed to correct them.

There are 9 lessons total:

1. Basic Python Program
2. Running the program
3. Variables
4. Basic Data Structures
5. Conditionals
6. Control Structures
7. File Handling
8. Regular Expressions
9. Functions

Each lessons starts on a new page. Most people take about two weeks to complete the lessons. I don't cover setting up the server or any of the software I use. You should be able to Google for that.

Starting Python

In order to learn Python we are going to teach you the basic programming concepts in the terminal. Our goal is to give you foundations in the **Python Basics** section so we can teach basic web application development in **Python Web Development**. After that you will need to learn advanced topics, on your own, in order to become a valuable programmer. This is just a basic overview of Python and web development.

Note from the Author

When I was 16 I stumbled across Nik Silver's Perl tutorial and started teaching myself Perl. Nik's tutorial helped teach me the basics of Perl very quickly and saved me a ton of research time. I have looked and struggled to find something as simple as that for Python. Programmers often struggle with simple things. I truly hope the following tutorials will help people use python to make their lives better.

I also hope Nik understands the first section of this tutorial is sort of a "port" of his amazing tutorials to Python. I did make some changes but I hope I caught the spirit of what he was doing. If I didn't, Nik, please don't sue me.

Also this is teaching Python 2.x and not Python 3.x. That is important because python 3 has a few changes that would make these tutorials not run correctly.

The last important thing I would like to say surrounds basic learning styles. This is a lot of information. If you are new to programming you will struggle to grasp some of the basics. It is best to have a mentor but if you don't/can't then try to review this material and do some basic practice exercises. Learning to program isn't easy. It takes time. My goal is to provide some shortcuts and help you learn faster than you would on your own.

Python Basics

- [Basic Python Program](#)
- [Running the program](#)
- [Variables](#)
- [Basic Data Structures](#)
- [Conditionals](#)
- [Control Structures](#)
- [File Handling](#)
- [Regular Expressions](#)
- [Functions](#)

Python Web Development

The following assignments should take you step by step through the learning process on basic web development. This class of programming is called [LAMP](#) programming.

- [Hello, World via HTTP](#)
- [HTML Template Hello World](#)
- [Dynamic HTML Form](#)
- [Database interaction](#)
- [BREAD Intro](#)
- [BREAD via one controller](#)
- [Multiple BREAD interaction](#)
- [A better class of program](#)
- [File management via HTTP](#)

Basic Python Program

For the first python program you will need one of two things. Either a python installation in Windows or any Linux distribution. For details visit python.org. For these tutorials I am going to assume you are using Linux.

Everyone's first python program is below. Open a text editor and type the following. You will save it as hello.py.

```
#!/usr/bin/python
# My first python program!

# Tell the world hello
print "Hello, World"
```

The first line

The very first line is always the first line in every python program.

```
#!/usr/bin/python
```

This tells the computer what to do when the file is run (executed).

Printing

In python when you print it automatically puts a new line (enter key) character at the end of your printed line. This can be a tad annoying if you don't want it you can print with a , at the end of the line

```
print "Hello, World!",
```

This will keep the cursor from moving down in the terminal if that is what you want.

Comments

Python has two types of comments.

```
# This is a comment

"""
    This is a comment too
"""
```

Simple Printing

Python has the **print** command that simply prints any string after it.

Running the program

To run the program you are going to have to tell Linux that it is meant to be run. For the remainder of these tutorials the \$ will be used to show commands run in a terminal.

```
$chmod a+x ProgramName
```

Where *ProgramName* is the name of your program (i.e. hello.py).

Now at the Linux prompt you can run the program by saying

```
$/ProgramName
```

You could have also run it by specifying python.

```
$python ProgramName
```

But that seems a little silly.

Variables

The most basic types of variables in python are general variables. Perl would call them Scalars. They can hold strings, characters, integers (int), floating point numbers (floats) and several other types. Once you assign data to a variable Python will define it's type.

There are two types of variable **casting** in programming. There is strong casting and weak casting. This is sometimes called Type casting (yes programming steals its phrases from every other industry). Python is generally considered a weak casting language although in some cases it is kinda in the middle. I will try to explain. If you set a variable as a float it normally stays a float unless you change it. If you set it as an int it normally says that way.

Let's say you make a variable called priority.

```
priority = 9
```

You made priority an int. You can assign a string to the exact same variable.

```
priority = "high"
```

That works fairly well. The difference with python and some other languages is using the wrong operator won't convert an object's type. It will generate an error.

```
priority = priority + 4  
TypeError: cannot concatenate 'str' and 'int' objects
```

Because we set the type of priority to a string on the line **priority = "high"** we cast it as a string. Trying to add to it doesn't make sense.

In general variable names consists of numbers, letters and underscores, but they should not start with a number and the `__` (two underscores) is special, as we'll see later. Also, Python is case sensitive, so 'a' and 'A' are different.

Basic Data Structures

Python has three basic data structures. The list, tuple, and dictionary.

Lists First

The list is fairly obvious to most new programmers.

```
myList = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
myList = ["Apples", "Oranges", "Pears"]
```

This makes a simple structure that holds lists. In the second one if I wanted to reference the Pears I would say `myList[2]` since we start counting from 0. The 2 in this case is called a list index (or indices). In several other programming languages lists are called arrays.

To get the length of a list you can use the `len()` function.

```
myList = [1, 2, 3]
print len(myList)
```

The above program will print out a 3. There are 3 things in my list.

You can do some neat things with lists. Appending things to the back of a list is handled via the `append()` function.

```
myList.append(4)
```

`myList` will now look like `[1, 2, 3, 4]` since we added the list on the back.

You can also `pop()` things off the front or back of a list.

```
# Get something off the back of the list
listVar = myList.pop()
```

```
# Get something off the front
listVar = myList.pop(0)
```

`pop()` takes an index for a list and returns (and removes) that element from the list. `listVar` would equal 4 in the first line of the above code and it would equal 1 in the second line. `myList` would now look like `[2, 3]` since the 4 and 1 were removed.

To display a list to the screen you just print it.

```
print myList
[2, 3]
```

Tuple

I have a friend that started the facebook club 'Tuple Haters' after a few issues he had with tuples. Tuples are read only lists. It is the same thing as a list in every way except you cannot change the tuple. Once it is frozen (set) you have to leave it alone.

You can 'thaw' a tuple by using the `list()` function. You can freeze a list by using the `tuple()` function.

```
myList = [1, 2, 3]
myList = tuple(myList)
(1, 2, 3)
myList = list(myList)
```

```
[1, 2, 3]
```

You may notice that python uses [] for lists and () for tuples.

Dictionaries

The dictionary is a very useful tool in Python. It allows you to store data in a list format but instead of using index numbers you can assign names to the fields. We call these Key/Value pairs. You will also notice we use the {} to represent the dictionaries. Except when we index them.

```
myDict = {'key': 'value'} myDict = {'name': 'Kenny Pyatt', 'age': 31, 'job': 'Programmer'}
```

Now if I want to get someones name I can ask for it directly from the data structure.

```
myDict['name']  
'Kenny Pyatt'
```

When you index a dictionary you use the [] instead of the {}. This is a mistake a lot of new Python programmers make.

You can get a list of all the keys in by using the keys() function.

```
myDict.keys()  
['job', 'age', 'name']
```

Data Structures having babies

It is also powerful to combine structures in order to form complex data structures.

```
myList = []  
myList.append({'name': 'Kenny'})  
myList.append({'name': 'Josh'})  
myList.append({'name': 'Chris'})  
myList.append({'name': 'Julie'})  
  
print myList  
[{'name': 'Kenny'}, {'name': 'Josh'}, {'name': 'Julie'}, {'name': 'Chris'}]
```

This makes a list of dictionaries. The title of this section refers to having babies. The idea is that a list has several smaller structures inside it. Like a woman has a baby inside when she is pregnant.

Conditionals

If you speak English then If, Else, and Then make sense to you. Python has a structure to represent that logic.

```
if myVar:
    print "myVar was true"
else:
    print "myVar was NOT true"
```

This is the first point where I can mention indentation. In python you have to indent the code properly or it won't work. For example, in the above code the print statements are indented. The indentions are spaces. The stuff spaced in below the if keyword tells Python how many lines belong inside the if statement if it is true.

You can include additional tests with **elif**

```
if myVar:
    print "myVar was true"
elif anotherVar:
    print "anotherVar was true"
else:
    print "myVar and anotherVar were NOT true"
```

There are some other ways to compare than simply true/false.

```
myVar = 1

# The == compares for the exact value
if myVar == 1:
    print "It was 1!"

# The ! means NOT
if myVar != 2:
    print "myVar was not 2!"
```

Control Structures

Python has a couple of very powerful loop control structures. The for and while loops will make your life much easier.

for loops

The most common is the for loop. You set it up like this

```
for variable in list:
```

In the above example variable will get set with the next item in the list as the program loops.

```
# An example loop
myList = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
for i in myList:
    print i
```

This will print 0 through 9 to the screen. You could have used range to save yourself some typing.

```
for i in range(0, 10):
    print i
```

This does the same thing.

while

Python also has the while loop.

```
i = 0
while (i != 10):
    i += 1
```

This while loop will continue to add one to i until it equals 10 and then it will stop.

Scope

Variables only work in their level of indention or any level deeper than them. For example a variable declared in a loop will only work inside that loop and then they will cease to exist. The variable goes out of scope.

File Handling

If you write a program odds are you want to save the results or possible read in data. You can do that with files.

To open a file

```
open(filename, mode)
```

Example

```
myfile = open('mydata.dat', 'r')
entireFile = myfile.read()
myfile.close()
```

that would open a file in read only mode (the r). The second line reads the entire file into the variable entireFile. The readline() will read one line at a time. Using readlines() will return a list of all the lines in the file. I normally use the readlines() and then do a for loop.

```
# Get the lines of the file in a list
myfile = open('mydata.dat', 'r')
entireFile = myfile.readlines()
myfile.close()

# Loop through the list and print the lines
for line in entireFile:
    print line
```

Will print out every line in the file one line at a time.

To write to a file you can

```
myfile = open("mydata.dat", "w")
myfile.write("this goes in the file")
myfile.close()
```

To append to the end of a file use the 'a' mode.

```
myfile = open("mydata.dat", "a")
myfile.write("this goes at the END of the file")
myfile.close()
```

There is also a binary mode.

Regular Expressions

Python comes with the `re` module for handling regular expressions.

Regular Expressions

From Wikipedia, "In computing, regular expressions, also referred to as regex or regexp, provide a concise and flexible means for matching strings of text, such as particular characters, words, or patterns of characters. A regular expression is written in a formal language that can be interpreted by a regular expression processor, a program that either serves as a parser generator or examines text and identifies parts that match the provided specification."

There are entire books written on this subject so I will just give an overview. If you need more you will need google.com.

```
#!/usr/bin/python

# Let's load the RE object
import re

testString = 'happy'

# See if the testString matches 'happy' exactly
if re.match('happy', testString):
    print "found happy"

# That would print 'found happy'

testString = 'magic happy socks'
if re.search('happy', testString):
    print "found happy"

# This would also print found happy because it looks for the pattern anywhere in this
string.

# This won't work now...
if re.match('happy', testString):
    print "found happy"
```

`re.search()` matches anywhere in the string and `re.match()` only matches if it is exactly the same at the beginning of the string.

More advanced searching

There are some special characters you can use in matching. Kinda like the `*.py` in windows and linux represents any file that ends with the `.py` extension. There are dozens of characters that can help you search for exactly what you want to find in a string.

Some basics

```
.    # Any single character except a newline
^    # The beginning of the line or string
$    # The end of the line or string
*    # Zero or more of the last character
+    # One or more of the last character
?    # Zero or one of the last character
```

Here are some examples (taken from directly from Nik Silver's examples in his wonderful [Perl tutorial](#))

```
t.e    # t followed by anything followed by e
        # This will match the
        #             tre
        #             tle
        # but not te
        #             tale
^f     # f at the beginning of a line
^ftp   # ftp at the beginning of a line
e$     # e at the end of a line
tle$   # tle at the end of a line
und*   # un followed by zero or more d characters
        # This will match un
        #             und
        #             undd
        #             unddd (etc)
.*     # Any string without a newline. This is because
        # the . matches anything except a newline and
        # the * means zero or more of these.
^$     # A line with nothing in it.
```

Here are some more special characters:

```
\n     # A newline
\t     # A tab
\w     # Any alphanumeric (word) character.
        # The same as [a-zA-Z0-9_]
\W     # Any non-word character.
        # The same as [^a-zA-Z0-9_]
\d     # Any digit. The same as [0-9]
\D     # Any non-digit. The same as [^0-9]
\s     # Any whitespace character: space,
        # tab, newline, etc
\S     # Any non-whitespace character
\b     # A word boundary, outside [] only
\B     # No word boundary
```

Clearly characters like \$, |, [,), \, / and so on are peculiar cases in regular expressions. If you want to match for one of those then you have to precede it by a backslash. So:

```
\|     # Vertical bar
\[     # An open square bracket
\]     # A closing parenthesis
\*     # An asterisk
\^     # A carat symbol
\/     # A slash
\\     # A backslash
```

The python documentation offers some [detailed regular expression material](#).

Functions

Python has functions (aka subroutines, sub functions, helper subs). Functions make your life better by creating a simple block of code that you can run by name.

They have the following format

```
def MyFunction():  
    print "This is my boring function"
```

Parameters

You can pass variables to a function called parameters (or sometimes arguments). A parameter can be anything you want.

```
# No arguments  
MyFunction()  
  
# One arguments  
MyFunction(myVar)  
  
# Two arguments  
MyFunction(myVar, anotherVar)
```

Returning values

When you run a function you normally want it to do something useful. To get the value back from the function you use the **return** keyword.

```
def AddThree(startValue):  
    " Return the startValue + 3"  
  
    return startValue + 3
```

This function gets one parameter **startValue** and then adds 3 to it and sends it back. Normally you would make this a tad more complex but it shows a simple return situation.

To call this function you would do the following

```
myVar = AddThree(19)
```

This would make myVar equal 22.

Local Variables and Scope

Just like in loops the variables you declare inside your functions will cease to exist when the function exists (unless you return/save the value). It is normally considered good practice to declare your local (vars inside the function) at the top of the function's code.