Python CheatSheet (For Linux)
Beginning Python: From Novice to Professional
Chapter 1

- In most Linux and UNIX installations (including Mac OS X), a **Python interpreter will already be present.** You can check whether this is the case for you by running the python command at the prompt, as follows:
  $ python
  Running this command should start the **interactive Python interpreter**, with output similar to the following(Using **Ubunutu 14.04**):
  'Python 2.7.6 (default, Mar 22 2014, 22:59:56)
  [GCC 4.8.2] on linux2
  Type "help", "copyright", "credits" or "license" for more information.
  >>>
  If there is no Python interpreter installed, you will probably get an error message similar to the following:
  bash: python: command not found
  In that case, you need to install Python yourself.
- The >>> thingy is the prompt.
- Print anything : >>>**print "Write whatever you want to print"**
- No Semicolon.

Numbers & Expersion

- The interactive Python interpreter can be used as a **powerful calculator**.
- All the **usual arithmetic operators work as expected—almost**. In Terminal,

Like : type **20+30-10** then enter and get the result : **40**

- One **potential trap** here, and that is integer

division (in Python versions prior to 3.0):

```
>>> 1/2
0
```

Cause: One integer (a nonfractional number) was divided by another, and the result was **rounded down to give an integer result**. This behavior can be useful at times, but often (if not most of the time), you need ordinary division.

Solution: There are two possible solutions: **use real numbers (numbers with decimal points) rather than integers, or tell Python to change how division works.**

Different ways like bellow:

```
>>> 1.0 / 2.0
0.5
>>> 1/2.0
0.5
>>> 1.0/2
0.5
>>> 1/2.
0.5
```

Another two ways to solve this problem:

One: If you would rather have Python do proper division, you could add the following statement to the beginning of your program (writing full programs is described later) or simply execute it in the interactive interpreter:

```
>>> from __future__ import division
```

Two: if you're running Python from the command line (e.g., on a Linux machine), is to supply the

command-line switch **-Qnew**. In either case, division will suddenly make a bit more sense:

```
>>> 1 / 2
0.5
```

Of course, the single slash can no longer be used for the kind of integer division shown earlier. A separate operator will do this for you—the double slash:

```
>>> 1 // 2
0
```

The double slash consistently performs integer division, even with floats:

```
>>> 1.0 // 2.0
0.0
```

- Real numbers are called **floats (or floating-point numbers)** in Python.
- Basic arithmetic operators (addition +, subtraction -, multiplication *, division / and mod %) do same operations.
- The last operator is the exponentiation (or power) operator:

```
>>> 2 ** 3
8
>>> -3 ** 2
-9
>>> (-3) ** 2
9
```

Note that the exponentiation operator binds tighter than the negation (unary minus), so -3**2 is in fact the same as -(3**2). If you want to calculate (-3)**2, you must say so explicitly.

- **nan** is simply a special value meaning “not a number.”

Large Integers

- Python can handle really large integers:

```
>>> 1000000000000000000
1000000000000000000L
```

Ordinary integers can’t be larger than 2147483647 (or smaller than –2147483648). If you want really big numbers, you must use longs. A long (or long integer) is written just like an ordinary integer but with an **L at the end. (Important: Consideration of long and short integer will be described later)**

Hexadecimals and Octals

Hexadecimal numbers are written like this:

```
>>> 0xAF
175
```

and octal numbers like this:

```
>>> 010
8
```

Complex Numbers:

There is no separate type for imaginary numbers in Python. They are treated as complex numbers whose real component is zero.

Example1:

```
>>> import cmath
>>> cmath.sqrt(-1)
1j
```

Example2:

```
>>> (1+3j) * (9+4j)
(-3+31j)
```

Variables:

- A variable is basically a name that represents (or refers to) some value.(As Like others

programing languages)

- Execute the following: **>>> x = 3**

This is called an **assignment.** We assign the value 3 to the variable x. Another way ofputting this is to say that we bind the variable x to the value (or object) 3. After you've assigned a value to a variable, you can use the variable in expressions:

```
>>> x * 2
6
```

Getting Input from the User

Something like that:

```
>>> x = input("x: ")
x: 34
>>> y = input("y: ")
y: 42
>>> print x * y
1428
```

Functions:

You can use some **built in** functions like : pow(2,3) , abs(-10), round(1/2) and so on.
Integer division always rounds
down, whereas **round** rounds to the nearest integer.

Modules

You may think of **modules as extensions** that can be imported into Python to extend its capabilities. You import modules with a special command called (naturally enough) **import**.

```
>>> import math
>>> math.floor(32.9)
32.0
```

Saving and Executing Your Programs

In a UNIX environment, you might use a directory like ~/python. Give your file any reasonable name, such as hello.py. The **.py**ending is important.

Running Your Python Scripts from a Command Prompt:

UNIX:

```
$ python test.py
```

In UNIX, there is a standard way of doing this: have the first line of your script begin with the character sequence **#!** (**called pound bang or shebang**) followed by the **absolute path** to the program that interprets the script (in our case Python).

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

This should run the script, regardless of where the Python binary is located.
Before you can actually run your script, you must make it executable:

```
$ chmod a+x test.py
```

Now it can be run like this (assuming that you have the current directory in your path):

```
$ test.py
```

If this doesn't work, try using **./test.py** instead, which will work even if the current directory (.) is not part of your execution path.

There is one problem with running your program like this, however. Once you've entered your name, the program window closes before you can read the result. The window closes
when the program is finished. Try changing the script by adding the following line at the end:

```
raw_input("Press <enter>")
```

Comments:

The hash sign (#) is a bit special in Python. When you put it in your code, everything to the right of it is ignored (which is why the Python interpreter didn't choke on the /usr/bin/env stuff used

earlier). Here is an example:
 # Print the circumference of the circle

Strings:

Strings are values, just as numbers using single or double quotes:

```
>>> "Hello, world!"
'Hello, world!'
```

same as

```
>>> 'Hello, world!'
'Hello, world!'
>>> "Let's go!"
"Let's go!"
>>> '"Hello, world!" she said'
'"Hello, world!" she said'
>>> 'Let's go!'
SyntaxError: invalid syntax
```

Solution: An alternative is to use the backslash character (\) to escape the quotes in the string, like this:

```
>>> 'Let\'s go!'
"Let's go!"
```

Concatenating:

```
>>> x = "Hello, "
>>> y = "world!"
>>> x y
```

SyntaxError: invalid syntax

In other words, this is just a special way of writing strings, not a general method of concatenating them. How, then, do you concatenate strings? Just like you add numbers:

```
>>> "Hello, " + "world!"
'Hello, world!'
>>> x = "Hello, "
>>> y = "world!"
>>> x + y
'Hello, world!'
```

String Representations, str and repr:

```
>>> "Hello, world!"
'Hello, world!'
>>> 10000L
10000L
>>> print "Hello, world!"
Hello, world!
>>> print 10000L
10000
```

Values are converted to strings through two different mechanisms. You can use both mechanisms yourself, through the functions str and repr. Str simply converts a value into a string in some reasonable fashion that will probably be understood by a user, for example. repr creates a string that is a representation of the value as a legal Python expression. Here are a few examples:

```
>>> print repr("Hello, world!")
'Hello, world!'
>>> print repr(10000L)
10000L
>>> print str("Hello, world!")
```

Hello, world!
>>> print str(10000L)
10000
A synonym for repr(x) is `x` (here, you use backticks, not single quotes). This can be useful when you want to print out a sentence containing a number:
>>> temp = 42
>>> print "The temperature is " + temp
Traceback (most recent call last):
File "<pyshell#61>", line 1, in ?
print "The temperature is " + temp
TypeError: cannot add type "int" to string
>>> print "The temperature is " + `temp`
The temperature is 42

Backticks are removed in Python 3.0, so even though you may find backticks in old code, you shouldprobably stick with repr yourself.

In above example, The first print statement doesn't work because you can't add a string to a number. The second one, however, works because I have converted temp to the string "42" by using the backticks. (I could have just as well used repr, which means the same thing, but may be a bit clearer. Actually, in this case, I could also have used str. Don't worry too much about this right now.)
In short, str, repr, and backticks are three ways of converting a Python value to a string. **The function str makes it look good, while repr (and the backticks) tries to make the resulting string a legal Python expression.**
Actually, **str is a type, just like int and long. repr, however, is simply a function.**

input vs. raw_input

Now you know what "Hello, " + name + "!" means. But what about raw_input? Isn't input good enough? Let's try it. Enter the following in a separate script file:
name = input("What is your name? ")
print "Hello, " + name + "!"
This is a perfectly valid program, but as you will soon see, it's a bit impractical. Let's try to run it:
What is your name? Gumby
Traceback (most recent call last):
File "C:/python/test.py", line 2, in ?
name = input("What is your name? ")
File "<string>", line 0, in ?
NameError: name 'Gumby' is not defined
The problem is that input assumes that what you enter is a valid Python expression (it's more or less the inverse of repr). If you write your name as a string, that's no problem:
What is your name? "Gumby"
Hello, Gumby!
However, it's just a bit too much to ask that users write their name in quotes like this.
Therefore, we use raw_input, which treats all input as raw data and puts it into a string:
>>> input("Enter a number: ")
Enter a number: 3
3
>>> raw_input("Enter a number: ")
Enter a number: 3
'3'
Unless you have a special need for input, you should probably use raw_input.

Long Strings, Raw Strings, and Unicode:

Long Strings:

If you want to write a really long string, one that spans several lines, you can use triple quotes instead of ordinary quotes:

```
print '''This is a very long string.
It continues here.
And it's not over yet.
"Hello, world!"
```

You can also use triple double quotes, """like this""". Note that because of the distinctive enclosing quotes, both single and double quotes are allowed inside, without being backslash escaped. Ordinary strings can also span several lines. If the last character on a line is a backslash, the line break itself is “escaped” and ignored. For example:

```
print "Hello, \
world!"
```

would print out Hello, world! .

Raw String:

Raw strings aren’t too picky about backslashes, which can be very useful sometimes. 12 In ordinary strings, the backslash has a special role: it escapes things, letting you put things into your string that you couldn’t normally write directly. For example, a new line is written \n, and can be put into a string like this:

```
>>> print 'Hello,\nworld!'
Hello,
world!
```

Raw strings are useful in such cases. They don’t treat the backslash as a special character at all. Every character you put into a raw string stays the way you wrote it:

```
>>> print r'C:\nowhere'
C:\nowhere
```

The one thing you can’t have in a raw string is a lone, final backslash. In other words, the last character in a raw string cannot be a backslash unless you escape it (and then the backslash you use to escape it will be part of the string, too).

Unicode String:

Normal strings in Python are stored internally as 8-bit ASCII, while Unicode strings are stored as 16-bit Unicode. This allows for a more varied set of characters, including special characters from most languages in the world. I’ll restrict my treatment of Unicode strings to the following:

```
>>> u'Hello, world!'
u'Hello, world!'
```

As you can see, Unicode strings use the prefix u, just as raw strings use the prefix r.