Python for you and me

0.1

Kushal Das

ISBN:

Publication date:

This book is for the python newbies

Python for you and me: Python for you and me

Author Kushal Das <kushal@fedoraproject.org>

1. Document Conventions

Certain words in this manual are represented in different fonts, styles, and weights. This highlighting indicates that the word is part of a specific category. The categories include the following:

`Courier font`
Courier font represents `commands`, `file names and paths`, and `prompts` .

When shown as below, it indicates computer output:

```
Desktop       about.html       logs      paulwesterberg.png
Mail          backupfiles      mail      reports
```

`bold Courier font`
Bold Courier font represents text that you are to type, such as: **`service jonas start`**

If you have to run a command as root, the root prompt (`#`) precedes the command:

```
# gconftool-2
```

`italic Courier font`
Italic Courier font represents a variable, such as an installation directory: *`install_dir`*`/bin/`

bold font
Bold font represents **application programs** and **text found on a graphical interface**.

When shown like this: **OK** , it indicates a button on a graphical application interface.

Additionally, the manual uses different strategies to draw your attention to pieces of information. In order of how critical the information is to you, these items are marked as follows:

Note

A note is typically information that you need to understand the behavior of the system.

Tip

A tip is typically an alternative way of performing a task.

Important

Important information is necessary, but possibly unexpected, such as a configuration change that will not persist after a reboot.

Caution

A caution indicates an act that would violate your support agreement, such as recompiling the kernel.

Warning

A warning indicates potential data loss, as may happen when tuning hardware for maximum performance.

2. We Need Feedback!

You should over ride this by creating your own local Feedback.xml file.

Installation

In this chapter you will learn how to install python

1.1. On Windows

You have to download the latest Windows(TM) installer from the python site http://python.org/ftp/python/2.5.2/python-2.5.2.msi . Install it just as any other Windows software.

1.2. On GNU/Linux

Generally all GNU/Linux distributions come with Python, so no need to worry about that :) If you don't have it then you can install it by either downloading from the python website or from your distribution's repository.

For Fedora

```
#yum install python
```

For Debian

```
#apt-get install python
```

The Beginning

So we are going to look at our first code. As python is an interpreted language , you can directly write the code into the python interpreter or write in a file and then run the file. First we are going to do that using the interpreter, to start type python in the command prompt (shell or terminal).

```
[kd@kdlappy ~]$ python
Python 2.5.1 (r251:54863, Oct 30 2007, 13:54:11)
[GCC 4.1.2 20070925 (Red Hat 4.1.2-33)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

In our first code we are going to print "Hello World!" , so do it as below,

```
>>> print "Hello World!"
Hello World!
```

2.1. helloworld.py

Now as a serious programmer you may want to write the above code into a source file. We will create a helloworld.py. Use any text editor you like to create the file. I used vi, you can even use GUI based tools like Kate, gedit too.

```
#!/usr/bin/env python
print "Hello World!"
```

To run the code first you have to make the file executable, in GNU/Linux you can do that by giving the command in a shell or terminal

```
$ chmod +x helloworld.py
```

Then

```
$ ./helloworld.py
Hello World!
```

On the first line you can *#!* , we call it sha-bang. Using this we are telling that use python interpreter to run this code. In the next line we are printing a text message. In python we call all the line of texts as strings.

2.2. Whitespaces and indentation

In Python whitespace is an important thing. We divide different identifiers using spaces.Whitespace in the beginning of the line is known as indentation, but if you give wrong identation it will throw an error. Examples are given below:

```
>>> a = 12
>>>  a = 12
  File "<stdin>", line 1
      a = 12
          ^
          IndentationError: unexpected indent
```

Caution

There is an extra space in the beginning of the second line which is causing the error, so always look for the proper indentation.

2.3. Comments

Comments are some piece of English text which explains what this code does, we write comments in the code so that is easier for others to understand. A comment line starts with # , everything after that is ignored as comment, that means they don't effect on the program.

```
>>> #this is a comment
>>> #the next line will add two numbers
>>> a = 12 + 34
>>> print c #this is a comment too :)
```

2.4. Modules

Modules are python files which contain different function definitions , variables which we can reuse, it should always end with a .py extension.. Python itself is having a vast module library with the default installation. We are going to use some of them. To use a module you have to import it first.

```
>>> import math
>>> print math.e
2.71828182846
```

We are going to learn more about modules on the Modules chapter.

Variables and Datatypes

Every programming language is having own grammar rules just like the other languages we speak.

3.1. Keywords and Identifiers

Python codes can be divided into identifiers. Identifiers (also referred to as names) are described by the following lexical definitions:

```
identifier ::= (letter|"_") (letter | digit | "_")*
letter ::= lowercase | uppercase
lowercase ::= "a"..."z"
uppercase ::= "A"..."Z"
digit ::= "0"..."9"
```

This means *_abcd* is a valid identifier where as *1sd* is not. The following identifiers are used as reserved words, or keywords of the language, and cannot be used as ordinary identifiers. They must be spelled exactly as written here:

```
and       del      from      not   while
as        elif     global    or    with
assert    else     if        pass  yield
break     except   import    print
class     exec     in        raise
continue  finally  is        return
def       for      lambda    try
```

In Python we don't specify what kind of data we are going to put in a variable. So you can directly write abc = 1 and abc will become an integer datatype. If you write abc = 1.0 abc will become of floating type. Here is a small program to add two given numbers

```
>>> a = 13
>>> b = 23
>>> a + b
36
```

From the above example you can understand that to declare a variable in python , what you need is just to type the name and the value. Python can also manipulate strings They can be enclosed in single quotes or double quotes like

```
>>> 'India'
'India'
>>> 'India\'s best'
"India's best"
>>> "Hello World!"
'Hello World!'
```

3.2. Reading input from the Keyboard

Generally the real life python codes do not need to read input from the keyboard. In python we use raw_input function to do input. *raw_input("String to show")* , this will return a string as output. Let us write a program to read a number from the keyboard and check if it is less than 100 or not. Name of the program is testhundred.py

```
#!/usr/bin/env python
number = int(raw_input("Enter an integer: "))
if number < 100:
    print "Your number is smaller than 100"
else:
    print "Your number is greater than 100"
```

The output

```
[kd@kdlappy book]$ ./testhundred.py
Enter an integer: 13
Your number is smaller than 100
[kd@kdlappy book]$ ./testhundred.py
Enter an integer: 123
Your number is greater than 100
```

In the next program we are going to calculate investments.

```
#!/usr/bin/env python
amount = float(raw_input("Enter amount: "))
inrate = float(raw_input("Enter Interest rate: "))
period = int(raw_input("Enter period: "))
value = 0
year = 1
while year <= period:
```

```
    value = amount + (inrate * amount)
    print "Year %d Rs. %.2f" %(year, value)
    amount = value
    year = year + 1
```

The output

```
[kd@kdlappy book]$ ./investment.py
Enter amount: 10000
Enter Interest rate: 0.14
Enter period: 5
Year 1 Rs. 11400.00
Year 2 Rs. 12996.00
Year 3 Rs. 14815.44
Year 4 Rs. 16889.60
Year 5 Rs. 19254.15
```

3.3. Some Examples

Some examples of variables and datatypes:

3.3.1. Average of N numbers

In the next program we will do an average of N numbers.

```
#!/usr/bin/env python
N = 10
sum = 0
count = 0
while count < N:
    number = float(raw_input(""))
    sum = sum + number
    count = count + 1
average = float(sum)/N
print "N = %d , Sum = %f" % (N, sum)
print "Average = %f" % average
```

The output

```
[kd@kdlappy book]$ ./averagen.py
1
```

```
2.3
4.67
1.42
7
3.67
4.08
2.2
4.25
8.21
N = 10 , Sum = 38.800000
Average = 3.880000
```

3.3.2. Temperature conversion

In this program we will convert the given temperature to Celsius from Fahrenheit by using the formula C=(F-32)/1.8

```
#!/usr/bin/env python
fahrenhite = 0.0
print "Fahrenheit Celsius"
while fahrenheit <= 250:
    celsius = ( fahrenheit - 32.0 ) / 1.8 #Here we calculate the fahrenhite value
    print "%5.1f %7.2f" % (fahrenheit , celsius)
    fahrenheit = fahrenheit + 25
```

The output

```
[kd@kdlappy book]$ ./temperature.py
Fahrenheit Celsius
  0.0  -17.78
 25.0   -3.89
 50.0   10.00
 75.0   23.89
100.0   37.78
125.0   51.67
150.0   65.56
175.0   79.44
200.0   93.33
225.0  107.22
250.0  121.11
```

3.4. Multiple assignments in a single line

You can even assign values to multiple variables in a single line, like

```
>>> a , b = 45, 54
>>> a
45
>>> b
54
```

Using this swapping two numbers becomes very easy

```
>>> a, b = b , a
>>> a
54
>>> b
45
```

Operators and expressions

In python most of the lines you will write will be expressions. Expressions are made of operators and operands. An expression is like *2 + 3* .

4.1. Operators

Operators are the symbols which tells the python interpreter to do some mathematical or logical operation. Few basic examples of mathematical operators are given below:

```
>>> 2 + 3
5
>>> 23 - 3
20
>>> 22.0 / 12
1.8333333333333333
```

To get floating result you need to the division using any of operand as floating number. To do modulo operation use % operator

```
>>> 14 % 3
2
```

4.2. Example of integer arithmetic

The code

```
#!/usr/bin/env python
days = int(raw_input("Enter days: "))
months = days / 30
days = days % 30
print "Months = %d Days = %d" % (months, days)
```

The output

```
[kd@kdlappy book]$ ./integer.py
Enter days: 265
```

```
Months = 8 Days = 25
```

In the first line I am taking the input of days, then getting the months and days and at last printing them. You can do it in a easy way

```
#!/usr/bin/env python
days = int(raw_input("Enter days: "))
print "Months = %d Days = %d" % (divmod(days, 30))
```

The divmod(num1, num2) function returns two values , first is the division of num1 and num2 and in second the modulo of num1 and num2.

4.3. Relational Operators

You can use the following operators as relational operators

Relational Operators

Operator	Meaning
<	Is less than
<=	Is less than or equal to
>	Is greater than
>=	Is greater than or equal to
==	Is equal to
!=	Is not equal to

Some examples

```
>>> 1 < 2
True
>>> 3 > 34
False
>>> 23 == 45
False
>>> 34 != 323
True
```

// operator gives the floor division result

```
>>> 4.0 // 3
1.0
```

```
>>> 4.0 / 3
1.3333333333333333
```

4.4. Logical Operators

To do logical AND , OR we use *and* , *or* keywords. *x and y* returns *False* if *x* is *False* else it returns evaluation of *y*. If *x* is *True*, it returns *True*.

```
>>> 1 and 4
4
>>> 1 or 4
1
>>> -1 or 4
-1
>>> 0 or 4
4
```

4.5. Shorthand Operator

x op = expression is the syntax for shorthand operators. It will be evaluated like *x = x op expression* , Few examples are

```
>>> a = 12
>>> a += 13
>>> a
25
>>> a /= 3
>>> a
8
>>> a += (26* 32)
>>> a
840
```

shorthand.py example

```
#!/usr/bin/env python
N = 100
a = 2
while a < N:
    print "%d" % a
    a *= a
```

The output

```
[kd@kdlappy book]$ ./shorthand.py
2
4
16
```

4.6. Expressions

Generally while writing expressions we put spaces before and after every operator so that the code becomes clearer to read, like

```
a = 234 * (45 - 56.0 / 34)
```

One example code used to show expressions

```
#!/usr/bin/env python
a = 9
b = 12
c = 3
x = a -b / 3 + c * 2 -1
y = a -b / (3 + c) * (2 -1)
z = a - (b / (3 +c) * 2) -1
print "X = ", x
print "Y = ", y
print "Z = ", z
```

The output

```
[kd@kdlappy book]$ ./evaluationexp.py
X =  10
Y =  7
Z =  4
```

At first *x* is being calculated. The steps are like this

```
9 - 12 / 3 + 3 * 2 -1
```

```
9 - 4 + 3 * 2 - 1
9 - 4 + 6 - 1
5 + 6 -1
11 - 1
10
```

Now for *y* and *z* we have parentheses, so the expressions evaluated in different way. Do the calculation yourself to check them.

4.7. Type Conversions

We have to do the type conversions manually. Like

```
float(string) -> float value
int(string) -> integer value
str(integer) or str(float) -> string representation
>>> a = 8.126768
>>> str(a)
'8.126768'
```

4.8. evaluateequ.py

This is a program to evaluate 1/x+1/(x+1)+1/(x+2)+ ... +1/n series upto n, in our case x = 1 and n =10

```
#!/usr/bin/env python
sum = 0.0
for i in range(1, 11):
    sum += 1.0 / i
    print "%2d %6.4f" % (i , sum)
```

The output

```
[kd@kdlappy book]$ ./evaluateequ.py
 1 1.0000
 2 1.5000
 3 1.8333
 4 2.0833
 5 2.2833
 6 2.4500
 7 2.5929
 8 2.7179
```

```
 9 2.8290
10 2.9290
```

In the line *sum += 1.0 / i* what is actually happening is *sum = sum + 1.0 / i*.

4.9. quadraticequation.py

This is a program to evaluate the quadratic equation

```
#!/usr/bin/env python
import math
a = int(raw_input("Enter value of a: "))
b = int(raw_input("Enter value of b: "))
c = int(raw_input("Enter value of c: "))
d = b * b - 4 * a * c
if d < 0:
    print "ROOTS are imaginary"
else:
    root1 = (-b + math.sqrt(d)) / (2.0 * a)
    root2 = (-b - math.sqrt(d)) / (2.0 * a)
print "Root 1 = ", root1
print "Root 2 = ", root2
```

4.10. salesmansalary.py

In this example we are going to calculate the salary of a camera salesman. His basic salary is 1500, for every camera he will sell he will get 200 and the commission on the month's sale is 2 %. The input will be number of cameras sold and total price of the cameras.

```
#!/usr/bin/env python
import math
a = int(raw_input("Enter value of a: "))
b = int(raw_input("Enter value of b: "))
c = int(raw_input("Enter value of c: "))
d = b * b - 4 * a * c
if d < 0:
    print "ROOTS are imaginary"
else:
    root1 = (-b + math.sqrt(d)) / (2.0 * a)
    root2 = (-b - math.sqrt(d)) / (2.0 * a)
print "Root 1 = ", root1
print "Root 2 = ", root2
```

The output

```
[kd@kdlappy book]$ ./salesmansalary.py
Enter the number of inputs sold: 5
Enter the total prices: 20450
Bonus        = 1000.00
Commision    = 2045.00
Gross salary = 4545.00
```

If-else , the control flow

While working on real life of problems we have to make decisions. Decisions like which camera to buy or which cricket bat is better. At the time of writing a computer program we do the same. We make the decisions using if-else statements, we change the flow of control in the program by using them.

5.1. If statement

The syntax looks like

```
if expression:
    do this
```

If the value of *expression* is true (anything than zero), do the what is written below under indentation. Please remember to give proper indentation, all the lines indented will be evaluated on the True value of the expression. One simple example is to take some number as input and check if the number is less than 100 or not.

```
#!/usr/bin/env python
number = int(raw_input("Enter a number: "))
if number < 100:
    print "The number is less than 100"
```

Then we run it

```
[kd@kdlappy book]$ ./number100.py
Enter a number: 12
The number is less than 100
```

5.2. Else statement

Now in the above example we want to print "Greater than" if the number is greater than 100. For that we have to use the *else* statement. This works when the *if*statement is not fulfilled.

```
#!/usr/bin/env python
number = int(raw_input("Enter a number: "))
if number < 100:
```

```
    print "The number is less than 100"
else:
    print "The number is greater than 100"
```

The output

```
[kd@kdlappy book]$ ./number100.py
Enter a number: 345
The number is greater than 100
```

Another very basic example

```
>>> x = int(raw_input("Please enter an integer: "))
>>> if x < 0:
...      x = 0
...      print 'Negative changed to zero'
... elif x == 0:
...      print 'Zero'
... elif x == 1:
...      print 'Single'
... else:
...      print 'More'
```

Looping

In the examples we used before , sometimes it was required to do the same work couple of times. We use a counter to check how many times the code needs to be executed. This technique is known as looping. First we are going to look into while statement for looping.

6.1. While loop

The syntax for *while* statement is like

```
while condition:
    statement1
    statement2
```

The code we want to reuse must be indented properly under the while statement. They will be executed if the *condition* is true. Again like in *if-else* statement any non zero value is true. Let us write a simple code to print numbers 0 to 10

```
>>> n = 0
>>> while n < 11:
...     print n
...     n += 1
...
0
1
2
3
4
5
6
7
8
9
10
```

In the first line we are setting *n = 0*, then in the while statement the condition is *n < 11* , that means what ever line indented below that will execute until n becomes same or greater than 11. Inside the loop we are just printing the value of n and then increasing it by one.

6.2. Fibonacci Series

Let us try to solve *Fibonacci* series. In this series we get the next number by adding the previous two numbers. So the series looks like *1,1,2,3,5,8,13 .....*

```
#!/usr/bin/env python
a, b = 0, 1
while b < 100:
    print b
    a, b = b, a + b
```

The output

```
[kd@kdlappy book]$ ./fibonacci1.py
1
1
2
3
5
8
13
21
34
55
89
```

In the first line of the code we are initializing *a* and *b*, then looping while b's value is less than 100. Inside the loop first we are printing the value of *b* and then in the next line putting the value of *b* to *a* and *a* + *b* to *b* in the same line.

If you put a trailing comma in the *print* statement , then it will print in the same line

```
#!/usr/bin/env python
a, b = 0, 1
while b < 100:
    print b,
    a, b = b, a + b
```

The output

```
[kd@kdlappy book]$ ./fibonacci2.py
1 1 2 3 5 8 13 21 34 55 89
```

6.3. Power Series

Let us write a program to evaluate the power series. The series looks like e**x =1+x+x**2/2! +x**3/3! +....+ x**n/n! where 0 < x < 1

```
#!/usr/bin/env python
x = float(raw_input("Enter the value of x: "))
n = term = num = 1
sum = 1.0
while n <= 100:
    term *= x / n
    sum += terrm
    n += 1
    if term < 0.0001:
        break
print "No of Times= %d and Sum= %f" % (n, sum)
```

The output

```
kd@kdlappy book]$ ./powerseries.py
Enter the value of x: 0
No of Times= 2 and Sum= 1.000000
[kd@kdlappy book]$ ./powerseries.py
Enter the value of x: 0.1
No of Times= 5 and Sum= 1.105171
[kd@kdlappy book]$ ./powerseries.py
Enter the value of x: 0.5
No of Times= 7 and Sum= 1.648720
```

In this program we introduced a new keyword called *break*. What break does is stop the innermost loop. In this example we are using *break* under the *if* statement

```
if term < 0.0001:
        break
```

This means if the value of *term* is less than *0.0001* then get out of the loop.

6.4. Multiplication Table

In this example we are going to print the multiplication table up to 10.

```
#!/usr/bin/env python
i = 1
print "-" * 50
while i < 11:
    n = 1
    while n <= 10:
        print "%4d" % (i * n),
        n += 1
    print ""
    i += 1
print "-" * 50
```

The output

```
[kd@kdlappy book]$ ./multiplication.py
--------------------------------------------------
   1    2    3    4    5    6    7    8    9   10
   2    4    6    8   10   12   14   16   18   20
   3    6    9   12   15   18   21   24   27   30
   4    8   12   16   20   24   28   32   36   40
   5   10   15   20   25   30   35   40   45   50
   6   12   18   24   30   36   42   48   54   60
   7   14   21   28   35   42   49   56   63   70
   8   16   24   32   40   48   56   64   72   80
   9   18   27   36   45   54   63   72   81   90
  10   20   30   40   50   60   70   80   90  100
--------------------------------------------------
```

Here we used one while loop inside another loop, this is known as nested looping. You can also see one interesting statement here

```
print "-" * 50
```

In a *print* statement if we multiply the string with an integer *n* , the string will be printed *n*many times. Some examples

```
>>> print "*" * 10
**********
>>> print "#" * 20
####################
>>> print "--" * 20
----------------------------------------
```

```
>>> print "-" * 40
----------------------------------------
```

6.5. Some printing * examples

Here are some examples which you can find very often in college lab reports

Design 1

```
#!/usr/bin/env python
row = int(raw_input("Enter the number of rows: "))
n = row
while n >= 0:
    x =  "*" * n
    print x
    n -= 1
```

The output

```
[kd@kdlappy book]$ ./design1.py
Enter the number of rows: 5
*****
****
***
**
*
```

Design 2

```
#!/usr/bin/env python
n = int(raw_input("Enter the number of rows: "))
i = 1
while i <= n:
    print "*" * i
    i += 1
```

The output

```
[kd@kdlappy book]$ ./design2.py
Enter the number of rows: 5
```

```
*
**
***
****
*****
```

Design 3

```
#!/usr/bin/env python
row = int(raw_input("Enter the number of rows: "))
n = row
while n >= 0:
    x =  "*" * n
    y = " " * (row - n)
    print y + x
    n -= 1
```

The output

```
[kd@kdlappy book]$ ./design3.py
Enter the number of rows: 5
*****
 ****
  ***
   **
    *
```

6.6. Lists

We are going to learn a data structure called list before we go ahead to learn more on looping. Lists an be written as a list of comma-separated values (items) between square brackets.

```
>>> a = [ 1 , 342, 2233423, 'India', 'Fedora']
>>> a
[1, 342, 2233423, 'India', 'Fedora']
```

Lists can keep any other data inside it. It works as a sequence too, that means

```
>>> a[0]
1
```

```
>>> a[4]
'Fedora'
```

You can even slice it into different pieces, examples are given below

```
>>> a[4]
'Fedora'
>>> a[-1]
'Fedora'
>>> a[-2]
'India'
>>> a[0:-1]
[1, 342, 2233423, 'India']
>>> a[2:-2]
[2233423]
>>> a[:-2]
[1, 342, 2233423]
>>> a[0::2]
[1, 2233423, 'Fedora']
```

In the last example we used two :(s) , the last value inside the third brackets indicates step. *s[i:j:k]* means slice of *s* from *i* to *j* with step *k*.

To check if any value exists within the list or not you can do

```
>>> a = ['Fedora', 'is', 'cool']
>>> 'cool' in a
True
>>> 'Linux' in a
False
```

That means we can use the above statement as *if* clause expression. The built-in function *len()* can tell the length of a list.

```
>>> len(a)
3
```

6.7. For loop

There is another to loop by using *for* statement. In python the *for* statement is different from the way it works in *C*. Here for statement iterates over the items of any sequence (a list or a string). Example given below

```
>>> a = ['Fedora', 'is', 'powerfull']
>>> for x in a:
...     print x,
...
Fedora is powerfull
```

We can also do things like

```
>>> a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> for x in a[::2]:
...     print x
...
1
3
5
7
9
```

6.8. range() function

range() is a buitin function. From the help document

```
range(...)
    range([start,] stop[, step]) -> list of integers
    Return a list containing an arithmetic progression of integers.
    range(i, j) returns [i, i+1, i+2, ..., j-1]; start (!) defaults to 0.
    When step is given, it specifies the increment (or decrement).
    For example, range(4) returns [0, 1, 2, 3].  The end point is omitted!
    These are exactly the valid indices for a list of 4 elements.
```

Now if you want to see this help message on your system type *help(range)* in the python interpreter. *help(s)* will return help message on the object *s*. Examples of *range* function

```
>>> range(1,5)
[1, 2, 3, 4]
>>> range(1,15,3)
[1, 4, 7, 10, 13]
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

6.9. Continue statement

Just like *break* we have another statement, *continue*, which skips the execution of the code after itself and goes back to the start of the loop. That means it will help you to skip a part of the loop. In the below example we will ask the user to input an integer, if the input is negative then we will ask again, if positive then we will square the number. To get out of the infinite loop user must input 0.

```
#!/usr/bin/env python
while True:
    n = int(raw_input("Please enter an Integer: "))
    if n < 0:
        continue #this will take the execution back to the starting of the loop
    elif n == 0:
        break
    print "Square is ", n ** 2
print "Goodbye"
```

The output

```
[kd@kdlappy book]$ ./continue.py
Please enter an Integer: 34
Square is 1156
Please enter an Integer: 4
Square is 16
Please enter an Integer: -9
Please enter an Integer: 0
Goodbye
```

6.10. Else loop

We can have an optional *else* statement after any loop. It will be executed after the loop unless a *break* statement stopped the loop.

```
>>> for i in range(0,5):
...     print i
... else:
...     print "Bye bye"
...
0
1
2
3
```

```
  4
Bye bye
```

We will see more example of *break* and *continue* later in the book.

Data Structures

Python is having a few built-in data structure. If you are still wondering what is a data structure, then it is nothing a but a way to store data and the having particular methods to retrieve or manipulate it. We already saw lists before, now we will go in depth.

7.1. Lists

```
>>> a = [23, 45, 1, -3434, 43624356, 234]
>>> a.append(45)
>>> a
[23, 45, 1, -3434, 43624356, 234, 45]
```

At first we created a list *a*. Then to add *45* at the end of the list we call *a.append(45)* method. You can see that *45* added at the end of the list. Sometimes it may require to insert data at any place within the list, for that we have *insert()* method.

```
>>> a.insert(0, 1) # 1 added at the 0th position of the list
>>> a
[1, 23, 45, 1, -3434, 43624356, 234, 45]
>>> a.insert(0, 111)
>>> a
[111, 1, 23, 45, 1, -3434, 43624356, 234, 45]
```

count(s) will return you number of times *s* is in the list. Here we are going to check how many times *45* is there in the list.

```
>>> a.count(45)
2
```

If you want to any particular value from the list you have to use *remove()* method.

```
>>> a.remove(234)
>>> a
[111, 1, 23, 45, 1, -3434, 43624356, 45]
```

Now to reverse the whole list

```
>>> a.reverse()
>>> a
[45, 43624356, -3434, 1, 45, 23, 1, 111]
```

We can store anything in the list, so first we are going to add another list *b* in *a* , then we will learn how to add the values of *b* into *a* .

```
>>> b = [45, 56, 90]
>>> a.append(b)
>>> a
[45, 43624356, -3434, 1, 45, 23, 1, 111, [45, 56, 90]]
>>> a[-1]
[45, 56, 90]
>>> a.extend(b) #To add the values of b not the b itself
>>> a
[45, 43624356, -3434, 1, 45, 23, 1, 111, [45, 56, 90], 45, 56, 90]
>>> a[-1]
90
```

Above you can see how we used *a.extend()* method to extend the list. To sort any list we have *sort()* method.

```
>>> a.sort()
>>> a
[-3434, 1, 1, 23, 45, 45, 45, 56, 90, 111, 43624356, [45, 56, 90]]
```

You can also delete element at any particular position of the list using the del keyword.

```
>>> del a[-1]
>>> a
[-3434, 1, 1, 23, 45, 45, 45, 56, 90, 111, 43624356]
```

7.2. Using lists as stack and queue

Stacks are often known as LIFO (Last In First Out) structure. It means the data will enter into it at the end , and the last data will come out first. The easiest example can be of couple of marbles in an one side closed pipe. So if you want to take the marbles out of it you have to do that from the end where you entered the last marble. To achieve the same in code

```
>>> a
[1, 2, 3, 4, 5, 6]
>>> a.pop()
6
>>> a.pop()
5
>>> a.pop()
4
>>> a.pop()
3
>>> a
[1, 2]
>>> a.append(34)
>>> a
[1, 2, 34)
```

We learned a new method above *pop()*. *pop(i)* will take out the ith data from the list.

In our daily life we have to encounter queues many times, like in ticket counters or in library or in the billing section of any supermarket. Queue is the data structure where you can append more data at the end and take out data from the beginning. That is why it is known as FIFO (First In First Out).

```
>>> a = [1, 2, 3, 4, 5]
>>> a.append(1)
>>> a
[1, 2, 3, 4, 5, 1]
>>> a.pop(0)
1
>>> a.pop(0)
2
>>> a
[3, 4, 5, 1]
```

To take out the first element of the list we are using *a.pop(0)*.

7.3. List Comprehensions

List comprehensions provide a concise way to create lists. Each list comprehension consists of an expression followed by a for clause, then zero or more for or if clauses. The result will be a list resulting from evaluating the expression in the context of the for and if clauses which follow it.

For example if we want to make a list out of the square values of another list, then

```
>>> a = [1, 2, 3]
>>> [x ** 2 for x in a]
[1, 4, 9]
>>> z = [x + 1 for x in [x ** 2 for x in a]]
>>> z
[2, 5, 10]
```

Above in the second case we used two list comprehensions in a same line.

7.4. Tuples

Tuples are data separated by comma.

```
>>> a = 'Fedora', 'Debian', 'Kubuntu', 'Pardus'
>>> a
('Fedora', 'Debian', 'Kubuntu', 'Pardus')
>>> a[1]
'Debian'
>>> for x in a:
...     print x,
...
Fedora Debian Kubuntu Pardus
```

You can also unpack values of any tuple in to variables, like

```
>>> divmod(15,2)
(7, 1)
>>> x, y = divmod(15,2)
>>> x
7
>>> y
1
```

Tuples are immutable, that means you can not del/add/edit any value inside the tuple. Here is another example

```
>>> a = (1, 2, 3, 4)
>>> del a[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object doesn't support item deletion
```

Above you can see python is giving error when we are trying to delete a value in the tuple.

To create a tuple which contains only one value you have to type a trailing comma.

```
>>> a = (123)
>>> a
123
>>> type(a)
<type 'int'>
>>> a = (123, ) #Look at the trailing comma
>>> a
(123,)
>>> type(a)
<type 'tuple'>
```

Using the buitin function *type()* you can know the data type of any variable. Remember the *len()* function we used to find the length of any sequence ?

```
>>> type(len)
<type 'bulletin_function_or_method'>
```

7.5. Sets

Sets are another type of data structure with no duplicate items. We can also mathematical set operations on sets.

```
>>> a = set('abcthabcjwethddda')
>>> a
set(['a', 'c', 'b', 'e', 'd', 'h', 'j', 't', 'w'])
```

And some examples of the set operations

```
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # unique letters in a
set(['a', 'r', 'b', 'c', 'd'])
>>> a - b                              # letters in a but not in b
set(['r', 'd', 'b'])
>>> a | b                              # letters in either a or b
set(['a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'])
```

```
>>> a & b                           # letters in both a and b
set(['a', 'c'])
>>> a ^ b                           # letters in a or b but not both
set(['r', 'd', 'b', 'm', 'z', 'l'])
```

To add or pop values from a set

```
>>> a
set(['a', 'c', 'b', 'e', 'd', 'h', 'j', 'q', 't', 'w'])
>>> a.add('p')
>>> a
set(['a', 'c', 'b', 'e', 'd', 'h', 'j', 'q', 'p', 't', 'w'])
```

7.6. Dictionaries

Dictionaries are unordered set of *key: value* pairs where keys are unique. We declare dictionaries using {} braces. We use dictionaries to store data for any particular key and then retrieve them.

```
>>> data = {'kushal':'Fedora', 'kart_':'Debian', 'Jace':'Mac'}
>>> data
{'kushal': 'Fedora', 'Jace': 'Mac', 'kart_': 'Debian'}
>>> data['kart_']
'Debian'
```

We can add more data to it by simply

```
>>> data['parthan'] = 'Ubuntu'
>>> data
{'kushal': 'Fedora', 'Jace': 'Mac', 'kart_': 'Debian', 'parthan': 'Ubuntu'}
```

To delete any particular *key:value* pair

```
>>> del data['kushal']
>>> data
{'Jace': 'Mac', 'kart_': 'Debian', 'parthan': 'Ubuntu'
```

To check if any *key* is there in the dictionary or not you can use *has_key()* or in keyword.

```
```

```
>>> data.has_key('Soumya')
False
>>> 'Soumya' in data
False
```

You must remember that no mutable object can be a *key*, that means you can not use a *list* as a *key*.

dict() can create dictionaries from tuples of *key,value* pair.

```
>>> dict((('Indian','Delhi'),('Bangladesh','Dhaka')))
{'Indian': 'Delhi', 'Bangladesh': 'Dhaka'}
```

If you want to loop through a dict use *iteritems()* method.

```
>>> data
{'Kushal': 'Fedora', 'Jace': 'Mac', 'kart_': 'Debian', 'parthan': 'Ubuntu'}
>>> for x, y in data.iteritems():
...     print "%s uses %s" % (x, y)
...
Kushal uses Fedora
Jace uses Mac
kart_ uses Debian
parthan uses Ubuntu
```

If you want to loop through a list (or any sequence) and get iteration number at the same time you have to use *enumerate()*.

```
>>> for i, j in enumerate(['a', 'b', 'c']):
...     print i, j
...
0 a
1 b
2 c
```

You may also need to iterate through two sequences same time, for that use *zip()* function.

```
>>> a = ['Pradeepto', 'Kushal']
>>> b = ['OpenSUSE', 'Fedora']
>>> for x, y in zip(a, b):
```

```
...     print "%s uses %s" % (x, y)
...
Pradeepto uses OpenSUSE
Kushal uses Fedora
```

7.7. students.py

In this example , you have to take number of students as input , then ask marks for three subjects as 'Physics', 'Maths', 'History', if the total number for any student is less 120 then print he failed, or else say passed.

```
#!/usr/bin/env python
n = int(raw_input("Enter the number of students:"))
data = {} # here we will store the data
languages = ('Physics', 'Maths', 'History') #all languages
for i in range(0, n): #for the n number of students
    name = raw_input('Enter the name of the student %d: ' % (i + 1)) #Get the
 name of the student
    marks = []
    for x in languages:
        marks.append(int(raw_input('Enter marks of %s: ' % x))) #Get the marks
 for  languages
    data[name] = marks
for x, y in data.iteritems():
    total =  sum(y)
    print "%s 's  total marks %d" % (x, total)
    if total < 120:
        print "%s failed :(" % x
    else:
        print "%s passed :)" % y
```

The output

```
[kd@kdlappy book]$ ./students.py
Enter the number of students:2
Enter the name of the student 1: Babai
Enter marks of Physics: 12
Enter marks of Maths: 45
Enter marks of History: 40
Enter the name of the student 2: Ria
Enter marks of Physics: 89
Enter marks of Maths: 98
Enter marks of History: 40
Babai 's  total marks 97
```

```
Babai failed :(
Ria 's  total marks 227
Ria passed :)
```

7.8. matrixmul.py

In this example we will multiply two matrix's. First we will take input the number of rows/columns in the matrix (here we assume we are using n x n matrix). Then values of the matrix's.

```
#!/usr/bin/env python
n = int(raw_input("Enter the value of n: "))
print "Enter values for the Matrix A"
a = []
for i in range(0, n):
    a.append([int(x) for x in raw_input("").split(" ")])
print "Enter values for the Matrix B"
b = []
for i in range(0, n):
    b.append([int(x) for x in raw_input("").split(" ")])
c = []
for i in range(0, n):
    c.append([a[i][j] * b[j][i] for j in range(0,n)])
print "After matrix multiplication"
print "-" * 10 * n
for x in c:
    for y in x:
        print "%5d" % y,
    print ""
print "-" * 10 * n
```

The output

```
[kd@kdlappy book]$ ./matrixmul.py
Enter the value of n: 3
Enter values for the Matrix A
1 2 3
4 5 6
7 8 9
Enter values for the Matrix B
9 8 7
6 5 4
3 2 1
After matrix multiplication
------------------------------
```

```
     9    12     9
    32    25    12
    49    32     9
------------------------------
```

Here we have used list comprehensions couple of times. *[int(x) for x in raw_input("").split(" ")]* here first it takes the input as string by *raw_input()*, then split the result by " ", then for each value create one int. We are also using *[a[i][j] * b[j][i] for j in range(0,n)]* to get the resultant row in a single line.

Strings

Strings are nothing but simple text. In python we declare strings in between "" or '' or ''' ''' or """ """. The examples below will help you to understand sting in a better way.

```
>>> s = "I am Indian"
>>> s
'I am Indian'
>>> s = 'I am Indian'
>>> s = "Here is a line \
... splitted in two lines"
>>> s
'Here is a line splitted in two lines'
>>> s = "Here is a line \n splitted in two lines"
>>> s
'Here is a line \n splitted in two lines'
>>> print s
Here is a line
 splitted in two lines
```

Now if you want to multiline strings you have to use triple single/double quotes.

```
>>> s = """ This is a
... multiline string, so you can
... write many lines"""
>>> print s
 This is a
multiline string, so you can
write many lines
```

8.1. Different methods available for Strings

Every string object is having couple of buildin methods available, we already saw some of them like *s.split(" ")*.

```
>>> s = "kushal das"
>>> s.title()
'Kushal Das'
```

title() method returns a titlecased version of the string, words start with uppercase characters, all remaining cased characters are lowercase.

```
>>> z = s.upper()
>>> z
'KUSHAL DAS'
>>> z.lower()
'kushal das'
```

upper() returns a total uppercase version whereas *lower()* returns a lower case version of the string.

```
>>> s = "I am A pRoGraMMer"
>> s.swapcase()
'i AM a PrOgRAmmER'
```

swapcase() returns the string with case swapped :)

```
>>> s = "jdwb 2323bjb"
>>> s.isalnum()
False
>>> s = "jdwb2323bjb"
>>> s.isalnum()
True
```

Because of the space in the first line *isalnum()* returned *False* , it checks for all charecters are alpha numeric or not.

```
>>> s = "SankarshanSir"
>>> s.isalpha()
True
>>> s = "Sankarshan Sir"
>>> s.isalpha()
False
```

isalpha() checkes for only alphabets.

```
>>> s = "1234"
>>> s.isdigit() #To check if all the characters are digits or not
True
>>> s = "Fedora9 is coming"
>>> s.islower() # To check if all chracters are lower case or not
```

```
False
>>> s = "Fedora9 Is Coming"
>>> s.istitle() # To check if it is a title or not
True
>>> s = "INDIA"
>>> s.isupper() # To check if characters are in upper case or not
True
```

To split any string we have *split()*. It takes a string as an argument , depending on that it will split the main string and returns a list containing splitted strings.

```
>>> s = "We all love Python"
>>> s.split(" ")
['We', 'all', 'love', 'Python']
>>> x = "Nishant:is:waiting"
>>> x.split(':')
['Nishant', 'is', 'waiting']
```

The opposite method for *split()* is *join()*. It takes a list contains strings as input and join them.

```
>>> "-".join("GNU/Linux is great".split(" "))
'GNU/Linux-is-great'
```

In the above example first we are splitting the string "GNU/Linux is great" based on the white space, then joining them with "-".

8.2. String the strings

Strings do have few methods to do striping. The simplest one is *strip(chars)*. If you provide the chars argument then it will strip any combination of them. By default it strips only whitespace or newline characters.

```
>>> s = "  abc\n "
>>> s.strip()
'abc'
```

You can particularly strip from the left hand or right hand side also using *lstrip(chars)* or *rstrip(chars)*.

```
>>> s = "www.foss.in"
```

```
>>> s.lstrip("cwsd.")
'foss.in'
>>> s.rstrip("cnwdi.")
'www.foss'
```

8.3. Finding text

Stings have some methods which will help you in finding text/substring in a string. Examples are given below:

```
>>> s.find("for")
7
>>> s.find("fora")
-1
>>> s.startswith("fa") #To check if the string startswith fa or not
True
>>> s.endswith("reason") #
True
```

find() helps to find the first occurrence of the substring given, if not found it returns -1.

8.4. Palindrome checking

Palindrome are the kind of strings which are same from left or right whichever way you read them. Example "madam". In this example we will take the word as input from the user and say if it is palindrome or not.

```
#!/usr/bin/env python
s = raw_input("Please enter a string: ")
z = [x for x in s]
z.reverse()
if s == "".join(z):
    print "The string is a palindrome"
else:
    print "The string is not a palindrome"
```

The output

```
[kd@kdlappy book]$ ./palindrome.py
Please enter a string: madam1
The string is not a palindrome
```

```
[kd@kdlappy book]$ ./palindrome.py
Please enter a string: madam
The string is a palindrome
```

8.5. Number of words

In this example we will count the number of words in a given line

```
#!/usr/bin/env python
s = raw_input("Enter a line: ")
print "The number of words in the line are %d" % (len(s.split(" ")))
```

The output

```
[kd@kdlappy book]$ ./countwords.py
Enter a line: Sayamindu is a great programmer
The number of words in the line are 5
```

Functions

Reusing the same code is required many times within a same program. Functions help us to do so. We write the things we have to do repeatedly in a function then call it where ever required. We already saw build in functions like *len()*, *divmod()*.

9.1. Defining a function

We use *def* keyword to define a function. general syntax is like

```
def functionname(params):
    statement1
    statement2
```

Let us write a function which will take two integers as input and then return the sum.

```
>>> def sum(a, b):
...     return a + b
```

In the second line with the *return* keyword, we are sending back the value of *a* + *b* to the caller. You must call it like

```
>>> res = sum(234234, 34453546464)
>>> res
34453780698L
```

Remember the palindrome program we wrote in the last chapter. Let us write a function which will check if a given string is palindrome or not, then return *True* or *False*.

```
#!/usr/bin/env python
def palindrome(s):
    z = s
    z = [x for x in z]
    z.reverse()
    if s == "".join(z):
        return True
    else:
        return False
```

```
s = raw_input("Enter a string: ")
if palindrome(s):
    print "Yay a palindrome"
else:
    print "Oh no, not a palindrome"
```

Now run the code :)

9.2. Local and global variables

To understand local and global variables we will go through two examples.

```


#!/usr/bin/env python
def change(b):
    a = 90
    print a
a = 9
print "Before the function call ", a
print "inside change function",
change(a)
print "After the function call ", a
```

The output

```


[kd@kdlappy book]$ ./local.py
Before the function call  9
inside change function 90
After the function call  9
```

First we are assigning *9* to *a*, then calling change function, inside of that we are assigning *90* to *a* and printing *a*. After the function call we are again printing the value of *a*. When we are writing *a = 90* inside the function, it is actually creating a new variable called *a*, which is only available inside the function and will be destroyed after the function finished. So though the name is same for the variable *a* but they are different in and out side of the function.

```


#!/usr/bin/env python
def chvariable ange(b):
    global a
    a = 90
    print a
a = 9
print "Before the function call ", a
```

```
print "inside change function",
change(a)
print "After the function call ", a
```

Here by using global keyword we are telling that *a* is globally defined, so when we are changing a's value inside the function it is actually changing for the *a* outside of the function also.

9.3. Default argument value

In a function variables may have default argument values, that means if we don't give any value for that particular variable it will assigned automatically.

```
>>> def test(a , b = -99):
...     if a > b:
...         return True
...     else:
...         return False
```

In the above example we have written *b = -99* in the function parameter list. That means of no value for *b* is given then b's value is *-99*. This is a very simple example of default arguments. You can test the code by

```
>>> test(12, 23)
False
>>> test(12)
True
```

Important

Remember that you can not have an argument without default argument if you already have one argument with default values before it. Like *f(a, b=90, c)* is illegal as *b* is having a default value but after that *c* is not having any default value.

Also remember that default value is evaluated only once, so if you have any mutable object like list it will make a difference. See the next example

```
>>> def f(a, data=[]):
...     data.append(a)
...     return data
...
```

```
>>> print f(1)
[1]
>>> print f(2)
[1, 2]
>>> print f(3)
[1, 2, 3]
```

9.4. Keyword arguments

```
>>> def func(a, b=5, c=10):
...     print 'a is', a, 'and b is', b, 'and c is', c
...
>>> func(12, 24)
a is 12 and b is 24 and c is 10
>>> func(12, c = 24)
a is 12 and b is 5 and c is 24
>>> func(b=12, c = 24, a = -1)
a is -1 and b is 12 and c is 24
```

In the above example you can see we are calling the function with variable names, like *func(12, c = 24)*, by that we are assigning *24* to *c* and *b* is getting its default value. Also remember that you can not have without keyword based argument after a keyword based argument. like

```
>>> def func(a, b=13, v):
...     print a, b, v
...
  File "<stdin>", line 1
SyntaxError: non-default argument follows default argument
```

File handling

A file is some information or data which stays in the computer storage devices. You already know about different kinds of file , like your music files, video files, text files. Python gives you easy ways to manipulate these files. Generally we divide files in two categories, text file and binary file. Text files are simple text where as the binary files contain binary data which is only readable by computer.

10.1. File opening

To open a file we use *open()* function. It requires two arguments, first the file path or file name, second which mode it should open. Modes are like

"r" -> open read only, you can read the file but can not edit / delete anything inside
"w" -> open with write power, means if the file exists then delete all content and open it to write
"a" -> open in append mode

The default mode is read only, ie if you do not provide any mode it will open the file as read only. Let us open a file

```
>>> f = open("love.txt")
>>> f
<open file 'love.txt', mode 'r' at 0xb7f2d968>
```

10.2. Reading a file

To read the whole file at once use the *read()* method.

```
>>> f = open("sample.txt")
>>> f.read()
'I love Python\nPradeepto loves KDE\nSankarshan loves Openoffice\n'
```

If you call read() again it will return empty string as it already read the whole file. readline() can help you to read one line each time from the file.

```
>>> f = open("sample.txt")
>>> f.readline()
'I love Python\n'
>>> f.readline()
'Pradeepto loves KDE\n'
```

To read all the all the lines in a list we use *readlines()* method.

```
>>> f = open("sample.txt")
>>> f.readlines()
['I love Python\n', 'Pradeepto loves KDE\n', 'Sankarshan loves Openoffice\n']
```

You can even loop through the lines in a file object.

```
>>> f = open("sample.txt")
>>> for x in f:
...     print x,
...
I love Python
Pradeepto loves KDE
Sankarshan loves Openoffice
```

Let us write a program which will take the file name as the input from the user and show the content of the file in the console.

```
#!/usr/bin/env python
name = raw_input("Enter the file name: ")
f = open(name)
print f.read()
f.close()
```

In the last line you can see that we closed the file object with the help of close() method.

The output

```
[kd@kdlappy book]$ ./showfile.py
Enter the filename: sample.txt
I love Python
Pradeepto loves KDE
Sankarshan loves Openoffice
```

10.3. Writing in a file

Let us open a file then we will write some random text into it by using the write() method.

```
```

```
>>> f = open("list.txt", 'w')
>>> f.write('powerpork\n')
>>> f.write('indrag\n')
>>> f.write('mishti\n')
>>> f.write('sm|CPU')
>>> f.close()
```

Now read the file we just created

```
>>> f = open('ircnicks.txt')
>>> s = f.read()
>>> print s
powerpork
indrag
mishti
sm|CPU
```

10.4. copyfile.py

In this example we will copy a given file to another file.

```
#!/usr/bin/env python
import sys
if len(sys.argv) < 3:
    print "Wrong parameter"
    print "./copyfile.py file1 file2"
    sys.exit(1)
f1 = open(sys.argv[1])
s = f1.read()
f1.close()
f2 = open(sys.argv[2], 'w')
f2.write(s)
f2.close()
```

You can see we used a new module here *sys*. *sys.argv* contains all command line parameters. Remember *cp* command in shell, after *cp* we type first the file to be copied and then the new file name.

The first value in *sys.argv* is the name of the command itself.

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

```
print "First value", sys.argv[0]
print "All values"
for i, x  in enumerate(sys.argv):
    print i, x
```

The output

```
[kd@kdlappy book]$ ./argvtest.py Hi there
First value ./argvtest.py
All values
0 ./argvtest.py
1 Hi
2 there
```

Here we used a new function *enumerate(iterableobject)*, which returns the index number and the value from the iterable object.

10.5. Random seeking in a file

You can also randomly move around inside a file using *seek()* method. It takes two arguments , offset and whence. To know more about it let us read what python help tells us

seek(...) seek(offset[, whence]) -> None. Move to new file position. Argument offset is a byte count. Optional argument whence defaults to 0 (offset from start of file, offset should be >= 0); other values are 1 (move relative to current position, positive or negative), and 2 (move relative to end of file, usually negative, although many platforms allow seeking beyond the end of a file). If the file is opened in text mode, only offsets returned by tell() are legal. Use of other offsets causes undefined behavior. Note that not all file objects are speakable.

Let us see one example

```
>>> f = open('tempfile', 'w')
>>> f.write('0123456789abcdef')
>>> f.close()
>>> f = open('tempfile')
>>> f.tell()    #tell us the offset position
0L
>>> f.seek(5) # Goto 5th byte
>>> f.tell()
5L
>>> f.read(1) #Read 1 byte
'5'
>>> f.seek(-3, 2) # goto 3rd byte from the end
>>> f.read() #Read till the end of the file
```

```
'def'
```

Acknowledgment

I would like to thank the following people who have helped me make it through this book. Specially #fedora-docs in irc.freenode.net. Names are in alphabetic order:

- Jared Smith
- Paul W. Frields
- Pradeepto K Bhattacharya
- Sankarshan Mukhopadhyay
- Sayamindu Dasgupta
- Stephanie Whiting

I am missing some names in the above list, will add them soon

I also took help from the following sites

- http://docs.python.org
- http://en.wikipedia.org

Few books or sites I would recommend to read

- *Byte of Python*[1]
- *Dive into Python*[2]
- *Python Tutorial*[3]

Appendix A. Revision History

Revision History

Revision 0.1 Kushal Das

First release