LEARN
PYTHON
in 24 Hours!

Python For Beginners

PYTHON
Learn Python in 24 Hours!
Python for Beginners

--UpSkill Learning

Copyright:

<u>Dedication:</u>

Dedicated to the ones who look at the world from a different perspective, the ones who are restless, the ones who strive for change, the ones who see things differently, the ones who don't accept the status quo, the ones who challenge current thinking patterns, the ones who break down existing barriers, the ones who make the impossible possible, the ones who build new things.....

Table Of Contents

<u>Welcome to Python for Beginners!</u>

Delving into the world of coding can be intimidating. With so many complex languages and implementation possibilities, it's easy to become overwhelmed. By starting off with Python programming, you'll learn a simple, versatile and highly readable code that you can execute on a wide variety of systems quickly and easily.

Do you want to become a programmer? Is coding your new passion? Do you want to be able to create games, parse the web and much more?

Let's get started learning one of the easiest coding languages out there right now. There's no need to fret if you haven't coded before. By the time you finish this book, you'll be a pro at Python!

Python is a great and friendly language to use and learn. It's fun, and can be adapted to both small and large projects. Python will cut your development time greatly and overall it's much faster to write Python than other languages. This book "Python for Beginners" will be a quick way to understand all the major concepts of Python programming. If you want to be a python wizard in no time, this is the book for you!

This book is a one-stop-shop for everything you'll need to know to get started with Python, along with a few incentives. We'll begin with the basics of Python, learning about strings, variables, and getting to know the data types. We'll soon move on to the loops and conditions in Python. Once we're done with that, we'll learn about functions and modules used in Python. We'll also learn some real life examples and applications and how to code in Python.

If you've never written a single line of code or if you're well-versed in multiple program languages, Python Programming for Beginners

will enable you to better understand programming concepts.

Widely regarded as one of the most simple and versatile programming languages out there, Python is used for web programming, video game building, microchip testing, desktop apps, and so much more.

Used by programmers, developers, designers and everyone in between, it's one of the easiest programming languages to learn, and definitely the best starting point for new coders. This book will not only give you an understanding of the code, but will enable you to create and run real world Python programs too.

Master one of the most popular programming languages in the world

-
  Understand and implement basic Python code
-
  Create and run a real-world Python program
-
  Gain a knowledge of basic programming concepts
-
  Learn a simple, streamlined coding language quickly and easily

We hope you're excited to dive into the World of Python. Well, what are you waiting for? Let's get started!

What are the requirements?

-

Macintosh (OSX)/ Windows(Vista and higher) Machine

-

Internet Connection

What am I going to get from this course?

-

Create your own Python Programs

-

Become an experienced Python Programmer

-

Parse the Web and Create your own Games

Target audience:

Even if you haven't touched coding before, it won't matter. The easy step-to-step course material will quickly guide you through everything you'll need to know about coding, mainly Python. This course is here for you to get accustomed and familiar with Python and its syntax. And above all, Python is one of the easiest coding languages to learn, and there's a lot you can do with it.

What You'll Learn From This Book?

You'll have the opportunity to put your knowledge to practical use by working with files and classes, importing syntax and making modules, and most importantly, by building your own Python program from scratch.

You'll walk away with detailed knowledge of one of the most widely

used programming languages in the world. You'd have gained a foundation of skills that will enable you to progress to more complex coding languages, as well as understanding the underlying principles of all programming languages. In short, you'll have everything you need to become a proficient programmer.

Chapter 1
Introduction To Python

Python is a high-level language. Python is interpreted, interactive and also object oriented scripting language. Python was designed to be highly reusable which uses English keywords frequently. It also uses almost same punctuations which are used in the most programming languages and it has fewer syntactical constructions than other languages.

• **Python is interpreted** : This means that it is processed at runtime by the interpreter and you do not need to compile your program before executing it. This is similar to PERL and PHP.

• **Python is Interactive** : This means that you can actually sit at a Python prompt and interact with the interpreter directly to write your programs.

• **Python is a Beginner's Language:** Python is a great language for the beginner programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

So far Python is matured with two major versions; they are Python 2.x and Python 3.x. While getting started, you may get confused which language to use for learning and developing an application. To

put it in nutshell, Python 2.x is legacy and Python 3.x is future. Whichever version you opt to learn or use, it is upto you because both are almost same programmatically.

The last Python 2.x version was Python 2.7 released in mid-2010 but quickly after that Python officially stopped support for that version.

History Of Python:

Python was developed by Guido van Rossum in the late eighties and early nineties at the National Research Institute for Mathematics and Computer Science in the Netherlands.

Python is derived from many other languages, including ABC, Modula-3, C, C++, Algol-68, SmallTalk and UNIX shell and other scripting languages. **Python is copyrighted** . Like Perl, Python source code is now available under the **GNU General Public License (GPL)** . Python is now maintained by a core development team at the institute, although Guido van Rossum still holds a vital role in directing its progress.

Chapter 2
Python – Features

Python's feature highlights include:

• **Easy-to-learn** : Python has relatively few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language in a relatively short period of time.

• **Easy-to-read:** Python code is much more clearly defined and visible to the eyes. Structure of codes is easy to understand and implement.

• **Easy-to-maintain:** Python's success is that its source code is fairly easy-to-maintain.

• **A broad standard library:** One of Python's greatest strengths is, the bulk of the library is portable and importantly cross-platform compatibility on UNIX, Windows and Macintosh.

• **Interactive Mode:** Support for an interactive mode in which you can enter results from a terminal right to the language, allowing interactive testing and debugging of snippets of code.

• **Portable:** Python can run on a wide variety of hardware platforms and has the same interface on all platforms.

• **Extendable:** You can add low-level modules to the Python interpreter. These modules enable programmers to add or customize their tools to be more efficient.

• **Databases:** Python provides interfaces to all major commercial

databases.

• **GUI Programming:** Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh and the X Window system of UNIX.

• **Scalable:** Python provides a better structure and support for large programs than shell scripting.

<u>Python has some special silent features:</u>

• Support for functional and structured programming methods as well as OOP.

• It can be used as a scripting language or can be compiled to byte-code for building large applications.

• Very high-level dynamic data types and supports dynamic type checking.

• Supports automatic garbage collection.

• It can be easily integrated with C, C++, COM, ActiveX, CORBA and Java.

"Jython is the java implementation of python programming language. But it is frequently called as JPython. The most popular one is the C implementation of Python. You can call it CPython or Python."

　　　　If you are trying to learn Python, we presume that you are already familiar with JAVA or C programming. If you are accustomed to Java / C programming, then you know the pain of leaving a semi-colon ";" at the end of the line. It is the programmer's

nightmare and remember that IDE also won't put the semicolon automatically at the end of line. If you had this experience frequently, then Python is a boon for you.

We've seen the complete history and introduction for Python in the previous chapters, now we will move to setting up the environment for Python.

Chapter 3
Setting Up The Environment

Python is an interpreted, object-oriented, high-level programming language and it is a great place for beginners to start learning how to program. Python comes installed on Macs and with Linux, but you'll need to install it yourself if you're using Windows. If you're using Mac or Linux computer, you can install the latest version to ensure you have access to the latest features.

FOR WINDOWS:

To download the setup file, you need to go to the following link.,

https://www.python.org/downloads/

From this website, You can download every detail pertaining to Python - like codes, snippets, plug-ins and you can read blog articles and documents related to python.

STEP 1:

Once you open this website, it automatically detects that you are using windows and it will go to the links of windows installer.

STEP 2:

As we've already discussed that there are two major versions of python, you can download any version to engage with it. Currently available versions of Python are: 3.x.x and 2.7.10. Python's both versions are available to download, but we advise new users to choose the 3.x.x version. Download the 2.7.10 if you are going to be working with legacy Python code or with programs and libraries that haven't adopted 3.x.x yet.
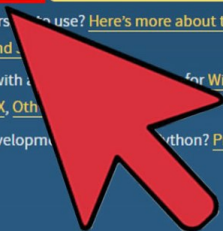
STEP 3:

Now you have to run the installer, clicking the button for the version you want. Run this installer after it has finished downloading. Make sure you have checked the "ADD PYTHON 3.5 TO PATH" button before proceeding. By checking this you can run python through command prompt itself.

STEP 4:

Now click install. This will install Python with all of its default settings, which should be fine for most users. If you want to disable certain functions, change the installation directory, or install the debugger, click "Customize installation" instead and then check or uncheck the boxes.

STEP 5:

It's always good to check whether all went correctly, as a programmer it is the important characteristic you need to have. To verify Python is installed and working correctly, open the newly-installed interpreter. Click the Start button and type "python" to quickly open it. You will get something like this.

Python will open to a command line. Type the following command and press ↵ Enter to display "Hello world!" on the screen:

print ('Hello world!')

Open the IDLE development environment. Python comes with a development environment called IDLE. This allows you to run, test, and debug scripts. You can quickly launch IDLE by opening the Start menu and searching for "idle".
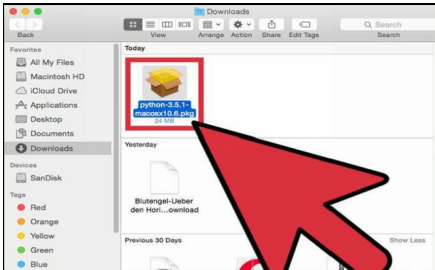
Now we are done! You are ready to start exploring the world of programming with Python.

Ste 1 and 2 are same as above (like windows).

STEP 3:

Visit www.python.org/download website and it will automatically detects that you are using Mac OS or else just select the "Mac OS X" link.
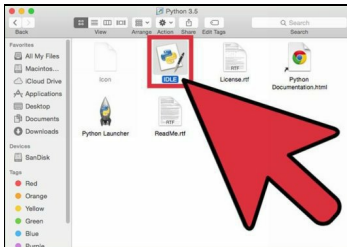

STEP 4:

Double-click the downloaded PKG file to start installing Python. Follow the prompts to install Python. We recommend using the default settings.

STEP 5:

Launch Python in the terminal. To verify the installation went correctly launch the terminal and type python3. This should start the Python 3.x.x interface, and display the version.

STEP 6:

Open the IDLE development environment. This program allows you to write and test Python scripts. You can find it in the

Applications folder.

STEP 7:

Try out a test script. IDLE will open an environment similar to a terminal screen. Type the following command and press ↵ Enter to display "Hello world!":

print ('Hello world!')

Now everything is set. Its time for your experimentation with python!

TEXT EDITOR:

        Python shell is being used as a text editor for python. Python Shell is an in-bulit feature available in the python package but for a beginner we recommand **Notepad++** text editor.

You can download it the below link:
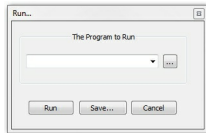https://notepad-plus-plus.org/download/v6.9.2.html

To run python program, we need to set the path and select .bat or .py ile in python package. To select that follow the following steps.

Step 1:

After typing the program save it and press f5 key or click Run in menu bar.

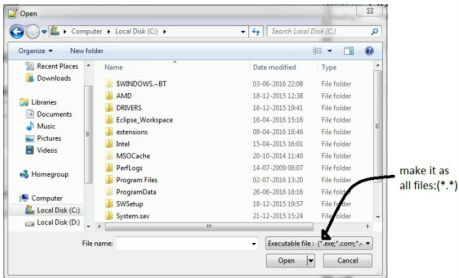A small dialog box will appear as shown below:

Step 2:

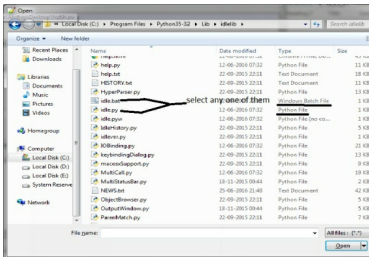Now click the **...** button. Open dialog box will appear.

Step 3:

Go to the python folder in the program files under C drive or the file location you select while installing the python in your computer.

After opening the python file (name will be with python versions), click **Lib** folder.

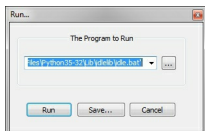Under Lib folder you will find the folder name **idlelib,** click that.

We recommand you to try both and fix the one with which you are comfortable. There is also idle.pyw i.e python file(no console) don't select that.

Now click Open button.

Step 4:

      Once you selected the Open Button, it will look as shown below:

If you save it and run the program, it will run and open python shell but you may need to again do the program coding there or had to paste. It is a double task, to avoid that add the following text along the address.

-r"$(FULL_CURRENT_PATH)"
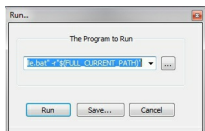
"C:\Program Files\Python35-32\Lib\idlelib\idle.bat" -r"$(FULL_CURRENT_PATH)"

The above one is the address path in our system, it may vary in yours.

We recommend you to use idle.bat file because it is handy and comfortable compared to the idle.py

Chapter 4
Identifiers

If you have experience with programming then you should be familiar with Identifies. It is the same in Python too.

Python identifiers is the name used to identify everythings like variable , function,class,module, and any other objects. actually it will be in alphanumeric i.e. with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9). Python does not allow punctuation characters such as @, $ and % within identifiers.

Important things to care about identifiers:

Python is a case sensitive programming language. Thus, **USA** and **usa** are two different identifiers in Python. So you should handle the identifiers carefully.

Identifier naming convention for Python:

• Class names start with an uppercase letter and all other identifiers with a lowercase letter.

• Starting an identifier with a single leading underscore indicates by convention that the identifier is meant to be private.

• Starting an identifier with two leading underscores indicates a strongly private identifier.

• If the identifier also ends with two trailing underscores, the identifier is a language-defined special name.

Every programming language have some reserved keywords i.e. it

can't be used as **variables** or as **identifiers** . Following are some of
the reserved keywords:

- and
- assert
- break
- class
- continue
- def
  - del
  - elif
  - else
  - except
- exec
- finally
- for
- from
- global
- if
  - import
  - in

is
-

lambda
- not
- or
- pass
- print
- raise
- return
-

try
-

while
-

with
-

yield

Chapter 5
Variables

Programiming is not done with assigning the value. Variables will have reserved memory location to store value. That is, when we create a variable, we are reserving some space for it in our system memory.

With respect to the datatype of variable ,the interpreter allocates memory and will decide to be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals or characters in these variables.

How to assign values to variables?

Python variables do not need explicit declaration to reserve memory space. The declaration happens automatically, when you assign a value to a variable. The equal sign (=) is used to assign values to variables.

The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable.

For example:

my_variable = 9

If you notice the above example, we've given variable name and assigned value to it. You may think it is wrong because we didn't delcare the datatype of variable but it is another advantage of Python. You don't need to assign variable type.

Lets look at the following example:

Eg., **my_var_int = 9**

     my_var_float = 9.09

     my_var_string = "nine"

Like other programming languages, we can do multiple assignment in python too.

Eg., **a = b = c = 9**

We are done with assigning a variables now, at later stages we will discuss how effectively we can use it in our program.

```
In python you can assign [] = () but not () = []

>>> [] = ()
>>> () = []

File "<stdin>", line 1

SyntaxError: can't assign to ()
```
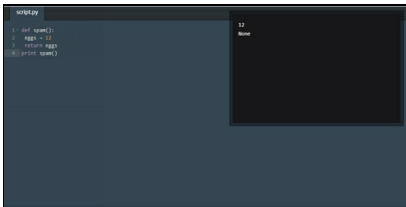
Chapter 6

Whitespaces

Whitespaces play a vital role in pyhton. In python we don't put curvely braces to make code blocks instead we use whitespaces.

Let's give you an example. Take a look at the following snipets.,

1-> def spam():

2->   eggs = 12

3->   return eggs

4-> print spam()

If you don't understand the above snipet, don't worry. Just look how the snipet is written. The function spam() is defined and the block of codes inside those i.e line 2 and 3 are moved some spaces. These spaces are called whitespaces.

The above code is correctly done with whitespaces. So what will happen if the code is incorrect? see the below example:

```
script.py
1  def spam():
2      eggs = 12
3      return eggs
4
5  print spam()
```

```
File "python", line 2
    eggs = 12
           ^
IndentationError: expected an indented block
```

You will get the following error.,

IndentationError: excepted an indented block

Funfact:

Python's name is derived from Monty Python's Flying Circus and has nothing to do with the species of snakes!
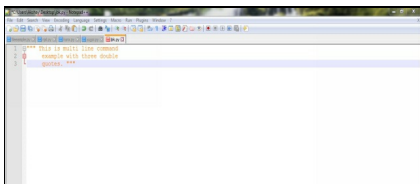
Common variables in Python *spam* and *eggs* are also taken from Monty Python's Flying Circus!

Chapter 7
Comments

It is always good practice to add comments in the program. In most of the programming languages " // " and " /* */ " are used. The things present inside the // symbol will be ignored by the compiler or interpretor. In python programming, " # " symbol is used for single line comment and """" """" symbol is used for multiline comments. Comments make your program easier to understand and read. Python won't try to run those and it is for human undersatnding.

Eg.,

> **ⓘ Hint**
>
> Your multiline comment is just a regular phrase or sentence starting with `"""` and ending with `"""`. No `#` needed at all!

GETTING OUTPUT:

The end result of the program obtained must have to be

visible as a result. How to show that result in the screen? For that we use **print function** . In the Python 2.6+ version, to make print effective for using flush etc, it is need to be imported in your program.

from __future__ import print_function from this package we can do more operation in print function.

But in Python 3 there is no need for that, as print becomes function in this version.

print "Hello Programmers!"

We learned variable assignment right? Now is the time to print the value of variable.

Following snipets.,

$$var1 = 9$$

$$print\ var1$$

So what will be the output?

The " **%s** " is used to insert or append the value of variable to the string in the printing statement and " **%** " symbol is used to tell which variable is needed to be placed.

Eg.,

$$Var1 = 10$$

print(" Messi jersey number is %s" %

(Var1))

    output will be.,

      Messi jersey number is 10

Now we will see few simple tests:

> **var1 = "India"**
>
> **var2 = "Pakistan"**
>
> **print (' The final match is between %s and %s ' %(var1,var2))**

guess the output for the above code., I will give three options for it.,

Option A:

The final match is between %s and %s

Option B:

'The final match is between %s and %s' %(var1,var2)

Option C:

The final match is between India and Pakistan

What is your answer? If you have guessed option **c** then you are awesome.

If you guessed any other option, revise the above chapters once again.

Chapter 8
Strings

Handling the strings is always important in Python prorgamming language. String is the collection of the characters. Strings in Python are identified as a contiguous set of characters represented in the quotation marks either single **' '** or double **" "** . Python allows for either pairs of single or double quotes.

Subsets of strings can be taken using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end. The plus (+) sign is the string concatenation operator and the asterisk (*) is the repetition operator.

String can be anything – words, numbers, symbols...

Eg., **var_1 = "World"**

var_2 = "09"

Var_1 = "!!!!"

Even white spaces inside quations will be considered as strings. For cancatenation we use plus(+) symbol.

Eg ., **var_1="hello"**

Var_1=" world"

print var_1 + Var_1

output :

hello world

Symbols & Functionality:

Symbol
Functionality
*
argument specifies width or precision
-
left justification
+
display the sign
<sp>
leave a blank space before a positive number
#
add the octal leading zero ('0') or hexadecimal leading '0x' or '0X', depending on whether 'x' or 'X' were used.
0
pad from left with zeros (instead of spaces)
%
'%%' leaves you with a single literal '%'
(var)
mapping variable (dictionary arguments)
m.n.
m is the minimum total width and n is the number of digits to display after the decimal point (if appl.)

Format Symbol & Conversion

Do you remember we have used the ==%s== during print section? Well there are some other format symbols also available, they are

Format Symbol
Conversion
%c

	character
%s	
	string conversion via str() prior to formatting
%i	
	signed decimal integer
%d	
	signed decimal integer
%u	
	unsigned decimal integer
%o	
	octal integer
%x	
	hexadecimal integer (lowercase letters)
%X	
	hexadecimal integer (UPPERcase letters)
%e	
	exponential notation (with lowercase 'e')
%E	
	exponential notation (with UPPERcase 'E')
%f	
	floating point real number
%g	
	the shorter of %f and %e
%G	
	the shorter of %f and %E

Now try to print this statement:

> **print (' That is Tom's car ')**

The expected output is **That is Tom's car** but python may have thrown some error or the output may be - **That is Tom** . This is

because Python thinks apostrophe in Rama's end as the code. So how to overcome this problem?

To overcome the above mentioned problem, we have to use escape sequence. The escape sequence is the backslash \ symbol. You should use it before apostrophe.

print(' That is Tom\'s car '

now run it. The output will be as we've expected i.e. **That is Tom's car .**

Try to print this now:

print ('D:\\Movies')

The output will be D:\Movies. This is because the python takes first backslash as escape sequence. For that we need to do **print r('D:\\Movies')** now you will get output as you've expected. You've got desired result because raw strings do not treat the backslash as a special character.

STRING INDEX:

The string is a collection of characters. Its index i.e. position of character stating from 0. See the below given example program:

```
1   """
2   The string "PYTHON" has six characters,
3   numbered 0 to 5, as shown below:
4
5   +---+---+---+---+---+---+
6   | P | Y | T | H | O | N |
7   +---+---+---+---+---+---+
8     0   1   2   3   4   5
9
10  So if you wanted "Y", you could just type
11  "PYTHON"[1] (always start counting from 0!)
12  """
13  fifth_letter = "MONTY"[4]
14
15  print fifth_letter
```

```
Y
None
```

The above example clearly explains the string index concept.

Let's try to write a code for obtaining ninth letter from word "happiness".

Code for above question is.,

> **A= "happiness"[8]**
>
> **print A**

output: s

Well sometimes we need to add something to the String which is not a string content. To do that we need to change non-string to string.

For this we use str() function which is the easiest way.

Try to concatenate the integer with string. Just print the following code.,

> **print 2 + "hello" + 3 + "world"**

If you think **2hello3world** is the output, you are wrong; we can't concatenate the integer with the string. It will throw error

```
script.py
1 print 2 + "hello" + 3 + "world"
```

```
Traceback (most recent call last):
  File "python", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int'
and 'str'
```

To overcome that we use str() function. Now try this code

print str(2) + "hello" + str(3) + "world" , this time we will get the expected output.


```
script.py
1 print str(2) + "hello" + str(3) + "world"
```

```
2hello3world
None
```
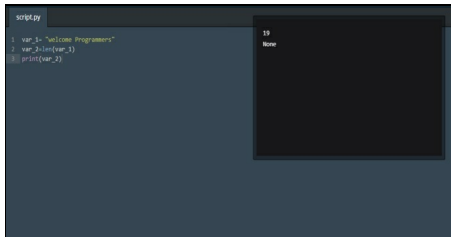
You may think instead of using str() function we can use double quotes. Right?

Assign two variables with any integer value then concatenate them with string. Try with and without str() functions. You will get its usage. Try this on your own, answer will be revealed in next page.

FIND LENGTH OF STRING:

We can find the total length of the string using len() function. Let's give an example so you can figure it out by yourself.,
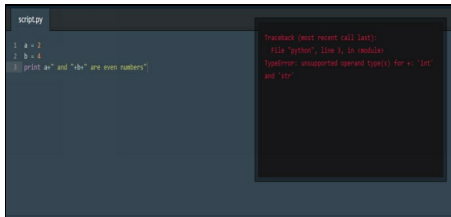
```
script.py
1  var_1= "welcome Programmers"
2  var_2=len(var_1)
3  print(var_3)
```

```
19
None
```

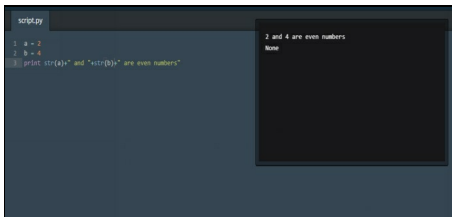Answer for the above question is puzzled by following examples, just scan the following codes.,

First without using str() function. Let's figure out what will be the output.

```
script.py
1  a = 2
2  b = 4
3  print a+" and "+b+" are even numbers")
```

```
Traceback (most recent call last):
  File "python", line 3, in <module>
TypeError: unsupported operand type(s) for +: 'int'
and 'str'
```

Now by using str() function.

```
script.py

1  a = 2
2  b = 4
3  print str(a)+" and "+str(b)+" are even numbers"
```

```
2 and 4 are even numbers
None
```

Got the answer? It's that simple!

HOW TO GET INPUT FROM USER:

So far we've assigned value and printed that value in screen and did some operation from the assigned value. But what is the motive of application development? It has to be user interactive. Inorder for the program to be user interactive, the very first step is we need to get inputs from user. How to do that? Python provides an effective solution for this by the pre-defined function called **raw_input()** and **input().**

PYTHON 2:

In python 2 raw_input() function takes exactly what user types in an string format. input() function takes the raw_input() and performs an eval() on it as well.
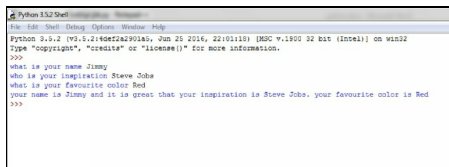
PYTHON 3:

In python 3 raw_input() is renamed as input() and old input() function. But if you want to use the old input() in python 3, you can use eval(input()).

Let's look into some examples:

a=input("what is your name")
b=input("who is your inspiration")
c=input("what is your favourite color")
print ("your name is %s and it is great that your inspiration is %s. your favourite color is %s" %(a,b,c))

Check out the above code and figure out what will be the output!.

First Python asks for the user input. Like args[0] in java. You can tell what is needed to give as input through message i.e. string between brackets ("what is your name"). When user press the enter key python moves to next command in program. Then it moves to print statement which we've already discussed.
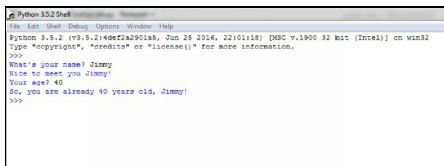

Whatever is entered by the user is taken as string by Python.

For input check out the following example, try this code and notice the output you are getting.

name = input("What's your name? ")
print("Nice to meet you " + name + "!")
age = input("Your age? ")

print("So, you are already " + age + " years old, " + name + "!")

If you are using python 2.x version while entering the input to python interpreter key-in " " symbol while entering or giving input to the python, well that's what the difference between raw_input() and input().

So what will happen if you missed " " while giving input. It will throw an error message.

Chapter 9
Types Of Operations

Python has the following Operations:

- Arithmetic Operations,
- Relation operations,
- Assignment operations,
- Logical Operations,
- Bitwise Operations,
- Membership Operations,
- Identity operations.

These operations are implemented using the operators. These operators have the same name as of their operation's name.

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

ARITHMETIC OPERATORS:

Arithmetic operators perform arithmetic operations like addition, subtraction, multiplication, divisions, exponent and modulus among two operands.

Operator	Description	Example

+ Addition
Adds values of the operands.
06 + 03 = 09
- Subtraction
Subtracts right hand operand from left hand operand.
12 – 03 = 09
*Multiplication
Multiplies values on either side of the operator
20 * 10 = 200

/ Division
Divides left hand operand by right hand operand.
10 / 5 = 2
// Floor Division
The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, simply rounded away from zero.
11//2 = 5and11.0//2.0 = 5.0, -11//3 = -4, -11.0//3 = -4.0
% Modulus
Divides left hand operand by right hand operand and returns remainder
10%3=1
** Exponent
Performs exponential (power) calculation on operators
5**2= 25 i.e. 5 * 5

Example:

```
script.py
1  print 2+5
2  print 7-2
3  print 5*5
4  print 9/3
5  print 7**7
6  print 30%4
```
```
7
5
25
3
823543
2
None
```

The above example prints the values just like a calculator does. In the next example we will a constant value to the variable.

```
script.py
1  a = 10
2  b = 2.0
3  print a+b
4  print a-b
5  print a*b
6  print a/b
7  print a**b
8  print a%b
```
```
12.0
8.0
20.0
5.0
100.0
0.0
None
```

RELATIONAL OPERATOR:

These operators compare the values on either sides of them and decide the relation among them. They are also called comparison operators.

Operator		
	Description	
Example		

==
If the values of two operands are equal, then the condition becomes
(8 == 7) is false.

!=
If values of two operands are not equal, then condition becomes true
(8 != 8) is false

<>
If values of two operands are not equal, then condition becomes true
(8 <> 7) is true. This is similar to != operator.

>
If the value of left operand is greater than the value of right operand,
(8 > 7) is true.

<
If the value of left operand is less than the value of right operand, th
(8 < 7) is false.

>=
If the value of left operand is greater than or equal to the value
becomes true.
(8 >= 7) is true.

<=
If the value of left operand is less than or equal to the value of righ
true.
(8 <= 7) is false.

Example for relational operator is given below.,

```
script.py
1  print 9==9
2  print 9>8
3  print 9<8
4  print 9!=8
5  print 9>=9
6  print 9<=9
7  print 7<>7
```

```
True
True
False
True
True
True
False
None
```

ASSIGNMENT OPERATORS:

Assignment operators are the operators which will assign values to the variable by implementing some operations. Assume a=5, b=6

Operator
Description
Example
=
Assigns values from right side operands to left side operand
c = a + b assigns value of a + b into c.
Ans: 11
+= Add AND
It adds right operand to the left operand and assign the result to left o
c += a is equivalent to c = c + a. (initially c=11)

Ans: 16

-= Subtract AND

It subtracts right operand from the left operand and assign the result

c -= a is equivalent to c = c – a

Ans= 11

*= Multiply AND

It multiplies right operand with the left operand and assign the result

c *= a is equivalent to c = c * a

Ans:55

/= Divide AND

It divides left operand with the right operand and assign the result to

c /= a is equivalent to c = c / a

c /= a is equivalent to c = c / a

%= Modulus AND

It takes modulus using two operands and assign the result to left ope

c %= a is equivalent to c = c % a

**= Exponent AND

Performs exponential (power) calculation on operators and assign va

c **= a is equivalent to c = c ** a

//= Floor Division

It performs floor division on operators and assign value to the left op

c //= a is equivalent to c = c // a

```
script.py
1  a = 5
2  b = 6
3  c = a + b
4  print c
5  c += a
6  print c
7  c -= a
8  print c
9  c *= a
10 print c
11 c /= a
12 print c
13 c %= a
14 print c
15 c **= a
16 print c
17 c //= a
18 print c
```

```
11
16
11
55
11
1
1
0
```

Bitwise Operators

Bitwise operator works on bits and performs bit by bit operation. Assume if a = 60; and b = 13; Now in binary format they will be as follows −

a = 0011 1100

b = 0000 1101

----

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a = 1100 0011

Operator
Description
Example
& Binary AND
Operator copies a bit to the result if it exists in both operands
(a & b) (means 0000 1100)

| Binary OR

It copies a bit if it exists in either operand.

(a | b) = 61 (means 0011 1101)

^ Binary XOR

It copies the bit if it is set in one operand but not both.

(a ^ b) = 49 (means 0011 0001)

~ Binary Ones Complement

It is unary and has the effect of 'flipping' bits.

(~a) = -61 (means 1100 0011 in 2's complement form due to a signe

<< Binary Left Shift

The left operands value is moved left by the number of bits specified

a << = 240 (means 1111 0000)

>> Binary Right Shift

The left operands value is moved right by the number of bits specifi

a >> = 15 (means 0000 1111)

LOGICAL OPERATOR:

Logical operators are the operators which perform
the logical operations on variables either side.

Operator
Descr
Example
and Logical AND

If both the operands are true then condition becomes true.

(a and b) is true.

or Logical OR

If any of the two operands are non-zero then condition becomes true

(a or b) is true.

not Logical NOT

Used to reverse the logical state of its operand.

Not (a and b) is false.

TRUTH TABLE for NOT , AND, OR is given below.,

NOT		NAND			NOR			AND			OR		
A	X	A	B	X	A	B	X	A	B	X	A	B	X
0	1	0	0	1	0	0	1	0	0	0	0	0	0
1	0	0	1	1	0	1	0	0	1	0	0	1	1
		1	0	1	1	0	0	1	0	0	1	0	1
		1	1	0	1	1	0	1	1	1	1	1	1

MEMBERSHIP OPERATOR:

Python's membership operators test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators, they are **in** and **not in.**

Operator

	Desc
Example	
in	
Evaluates to true if it finds a variable in the specified sequence or fa	
x in y, here in results in a 1 if x is a member of sequence y.	
not in	
Evaluates to true if it does not finds a variable in the specified seque	
x not in y, here not in results in a 1 if x is not a member of sequence	

Chapter 10
Data Types

We've already discussed in previous chapters that in Python there is no need to define a datatype of variable while assigning values. Python will take that load for you. The common datatypes are:

- ✓ INTEGERS
- ✓ FLOAT
- ✓ STRING
- ✓ BOOLEAN

Data types are having two major divisions, they are
- ❖ **Mutable**
- ❖ **Immutable**

MUTABLE:

Mutable type: The values in the objects can be changed.

IMMUTABLE:

Immutable type: The values in the objects are not changeable. The content in the object under immutable couldn't be changed once they are created.

It is important to understand that variables in Python are really just references to objects in memory. Feeling stuck? You will definitely learn and understand this concept in this and upcoming chapters.

Now let's learn few concepts in data types.

Some mutable types:

- byte array
- list
- set
- dict
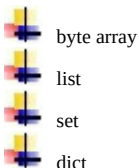
LIST:

  A list is a container which holds comma separated values (items or elements) between square brackets where Items or elements need not to have the same type.

SET:

    A set is an unordered collection of unique elements. Basic uses include dealing with set theory (which support mathematical operations like union, intersection, difference, and symmetric difference) or eliminating duplicate entries.

DICT: (DICTIONARY)

    Python dictionary is a container of unordered set of objects like lists. The objects are surrounded by curly braces { }. The items in a dictionary are comma-separated list of key:value pairs where keys and values are Python data type. Each object or value accessed by key and keys are unique in the dictionary. As keys are used for indexing, they must be immutable type (string, number, or tuple).

You can create an empty dictionary using empty curly braces.

Some IMMUTABLE types are

- Numbers-int, float, long, complex
- str
- bytes
- tuples
- frozen set

Numbers:

Numbers are created by numeric literals. Numeric objects are immutable, which means when an object is created its value cannot be changed.

Python has three numeric types:

- **integers** ,
- **floating point numbers**
- **complex numbers** .

Integers represent negative and positive integers without fractional parts. (eg. 3, 8, 9)   Floating point numbers represents negative and positive numbers with fractional parts. (eg.3.0) Complex numbers are combination of the real and imaginary numbers representing both positive and negative numbers. (eg. 3-5j,

2+4j). first one is real number and the number with the j is imaginary number.

Str:

Str is string. We have already dedicated a chapter for discussing this topic.

Tuples:

A tuple is container which holds a series of comma separated values (items or elements) between parentheses. Tuples are immutable (i.e. you cannot change its content once created and can hold mix data types).

Example:

Tup1 = ("social", "science", 1997, 2000);

Tup2 = (1, 2, 3, 4, 5);

To create the empty tuple, just put tuple name with empty parentheses,

Example:

Tupe = ()

While writing a single value tuple, you need to add comma after the single value inside the parentheses.

Example:

Tups = (1,)

There are some built-in functions available, they are given below.

S.no	Function name	Description
1	cmp(tuple1, tuple2)	Compares elements of both tuples.
2	len(tuple)	Returns total length of the tuple.
3	max(tuple)	Returns maximum value item from the tuple.
4	min(tuple)	Returns minimum value item from the tuple.
5	tuple(seq)	Converts list into tuple.
6	del tuple	Deletes the tuple.

Indices in tuple is same as the indices in the string. Starting position is zero then last position will be (n-1)th position.[n is total number of values]

Example,

Tup1 = ('rick' , 'danny' , 'maddy')

To make sure you learned indices concept thoroughly try to answer the following questions on your own, the answer will be revealed a bit later.

Tup1[1] returns which value?

Tup1[3] returns which value?

Tup1[-3] returns which value?

Tup1[-1] returns which value?

Some basic operations in Tuple.

There are some basic tuple operations available - like in strings /concatenation and length also other operations like repetition, membership, iteration with symbols like +, *.

Expression
Results
Description
len((1, 2, 3, 4, 5, 5))
6

Length of tuple
(1, 2, 3,4) + (5, 6, 7, 8)
(1, 2, 3, 4, 5, 6, 7, 8)
Concatenation
('YES!',) * 5
('YES!', 'YES!', 'YES!', 'YES!', 'YES!')
Repetition
1 in (1, 2, 3,4)
True
Membership
for x in (1, 2, 3, 4, 5):   print x
1 2 3 4 5
Iteration

Tup1 = ('rick' , 'danny' , 'maddy')
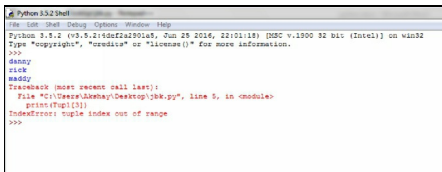
Tup1[1] returns which value?
Tup1[3] returns which value?
Tup1[-3] returns which value?
Tup1[-1] returns which value?
Answers
Tup1[1] -> returns danny
Tup1[3] -> error out of bound, because index starts from 0 in this tuple ending index is 2
Tup1[-3] -> returns rick negative means starts from right side means -1,-2,-3
Tup1[-1] -> returns maddy

```
Python 3.5.2 Shell
File Edit Shell Debug Options Window Help
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 25 2016, 22:01:18) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
danny
rick
maddy
Traceback (most recent call last):
  File "C:\Users\Akshay\Desktop\jbk.py", line 5, in <module>
    print(Tup1[5])
IndexError: tuple index out of range
>>>
```

To learn about tuple more thoroughly, let's try an exercise: Create a tuple and put four random values to it. Then print it. Now print the first and last value only. Then delete the tuple.

Answers:

Tup1 = ("rick" , "danny" , "maddy" , "jimmy")
print(Tup1[0])
print(Tup1[-1])
print("after printing this, now delete")
del Tup1

List:

The list is a most versatile datatype available in Python like tuples which can be written as a list of comma-separated values (items) between square brackets. There is no need to have same data type values in the list.

Creating a list is very simple by putting different comma-separated values between square brackets.

Example:

List1 = [1, 2, 3, 'a', 'b', 'c']

List2 = ['red' , 'black', 'blue']

The built-in functions and operations in the list are as same as the tuple. There is no difference between them.

Indices in list is also like tuple and string. Accessing the values in the list can be done by matrix.

Example:

To retrieve value 2 from list1 expression is **List1[1]** .

To retrieve value a,b,c only from list1 expression is **List1[3:5]** .

UPDATING LIST:

Updating the single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator, and you can add to elements in a list by using the append() method.

List1 = [1, 2, 3, 'a', 'b', 'c']

print("Value available at index 2 : %s" % (List1[2]))
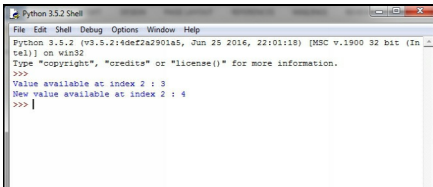
List1[2] = 4

print("New value available at index 2 : %s " % (List1[2]))

Output:

Value available at index 2 : 3

New value available at index 2 : 4

```
Python 3.5.2 Shell
File Edit Shell Debug Options Window Help
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 25 2016, 22:01:18) [MSC v.1900 32 bit (In
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
Value available at index 2 : 3
New value available at index 2 : 4
>>>
```

It seems both tuple and list looks similar, but both are different. **List** is mutable and **tuples** is immutable. The main **difference between** mutable and immutable is memory usage when you are trying to append an item. When you create a variable, some fixed memory is assigned to the variable. If it is a **list**, more memory is assigned than actually used.

Now you have learned operations and data types in Python, let's move on to learn a new concept called Typecasting.

What is Type casting?

Typecasting is nothing but changing one variable's datatype into another datatype. Consider a scenario where you want to add the two variables that is user entered. Can you add two strings? No you can only concatenate two variables. So how to do it? Typecasting comes handy here!

Think a scenario: You are getting two inputs from user and that is in string data type. Now you need to add it. How will you add it?

If the following program comes to your mind, then you have learned the concept but didn't observe it properly. The following method is a wrong approach, let's look into it.

```
a = input('enter the value of a')
```

```
        b = input('enter the value of b')
        c = a + b
        print (c)
```

If you think this is the correct program for above scenario then you have to revisit the String chapter and study the string concatenation.

Let's see why you should use Typecasting:

```
        a = int (input('enter the value of a'))
        b = int (input('enter the value of b'))
        c = a + b
        print (c)
```

The input function returns string, so the **c = a + b** line concatenates by the symbol "+". After typecasting the string into int(integer) the third line of code will perform the addition operation.

```
        a = float(input('enter the value of a'))
        b = float(input('enter the value of b'))
        c = a + b
        print (c)
```

It changes the string into float. Do you remember the str() in the string section. It changes or typecast the integer or float to the string.

Chapter 11
Decision Making

Decision making, it is the most important topic in both life and programming. The right decision at the right time will make the life better, it rings true in programming concepts too. While making a decision the most common possible answer is either true (yes) or false (no).
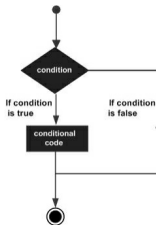
The above diagram is the basic representation of the decision making.

The statements used for the decision making are:

- if statement
- if….else statement
- nested if statement

IF STATEMENT:

The **if** statement of Python is similar to that of other languages. The **if** statement contains a logical expression using which data is compared and a decision is made based on the result of the comparison.

Syntax:

```
if (expression):
        statement(s)
```
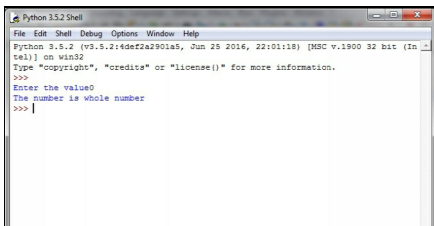
If the boolean **expression** evaluates to **true,** then the block of statements inside the if statement will be executed. If boolean expression evaluates to **false,** then the first set of code after the end of the if statements will be executed.

Python programming language assumes any **non-zero** and **non-null** values as **true,** and if it is either **zero** or **null,** then it is assumed as **false** value.

We've already discussed whitespaces and its usage and its needs in python. So let's get ahead and directly see an example program.

Write a program to find whether the given number is whole number or not.

```
A = int(input('Enter the value'))
if A == 0:
        print ("The number is whole
number")
```

Like other languages in Python too we can combine **else** statement with **if** statement. An **else** statement contains the block of code that executes if the conditional expression in **if** statement resolves to 0 or a false value.
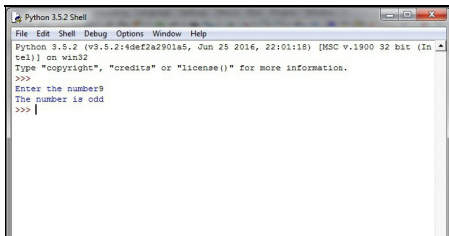
The *else* statement is an optional statement and there could be at most only one **else** statement following **if** .

Syntax:
```
if (expression):
    statement(s)
else:
    statement(s)
```

Let's move to an example: Get an input from a user and check whether it is even or odd.

```
A = int(input("Enter the number"))
if  A %2==0:
        print "The number is even"
else:
        print "The number is odd"
```

elif **Statement**

The **elif** statement allows you to check multiple expressions for truth value and execute a block of code as soon as one of the conditions evaluates to true.

Like **else** , the **elif** statement is optional. However, unlike **else** , for which there can be at most one statement, there can be an arbitrary number of **elif** statements following **if** .

SYNTAX:

```
if expression1:
    statement(s)
elif expression2:
    statement(s)
elif expression3:
    statement(s)
else:
    statement(s)
```

The main reason to use the elif statement is, in python we don't have switch case like we have in other object oriented programming languages like java or c++. So instead of using some confusing user-defined functions to implement switch case you can use elif statement. *{This advice is only for Python beginners}*

There may be a situation when you want to check for another condition after a condition resolves to true. In such situation, you can use the nested **if** construct.

In a nested **if** construct, you can have **if...elif...else** construct inside another **if...elif...else** construct.

Syntax:

if expression1:

   statement(s)

if expression2:

    statement(s)

elif expression3:

     statement(s)

else:
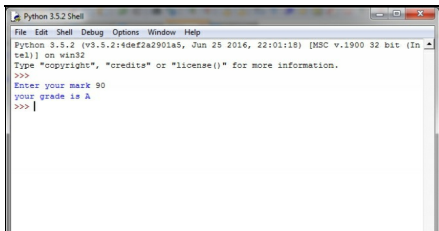
    statement(s)

elif expression4:

    statement(s)

else:

     statement(s)

Example:

Get the user mark and return the grade for that mark.

```python
score = int(input('Enter your mark'))
if score >= 90:
        print( 'your grade is A')
else:
                # grade must be B, C, D or F
if score >= 80:
          print ('your grade is B')
else:
                        # grade must be C, D or F
  if score >= 70:
              print ('your grade is C')
  else:
                                    # grade must D or F
   if score >= 60:
                  print ('your grade is D')
    else:
                  print ('your grade is F')
```
output:

```
Python 3.5.2 Shell
File  Edit  Shell  Debug  Options  Window  Help
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 25 2016, 22:01:18) [MSC v.1900 32 bit (In
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
Enter your mark 90
your grade is A
>>>
```

Chapter 12
Loops In Python

There are more chances for a programmer to face implementation of same expression or operation repeatedly for n number of times. To handle that problem effectively we have loops in programming languages. In Python too we have loop concepts. The 2 loops in Python are listed below:

1. **while loop**

Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.

2. **for loop**

Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.

3. **nested loops**

You can use one or more loop inside any other loop.

A **while** loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true.

Syntax:
while expression:
    statement(s)

Here, **statement(s)** may be a single statement or block of statements.

The **condition** may be any expression, and true is any non-zero value. The loop iterates while the condition is true.

When the condition becomes false, program comes out of loops.

Example:

count = 0

while (count < 5):

    print ('The count is:', count)

    count = count + 1

print("LOOP EXPIRED")

Output:
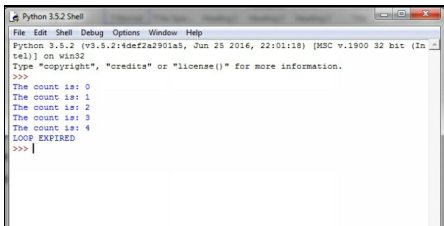
The count is: 0
The count is: 1
The count is: 2
The count is: 3
The count is: 4
LOOP EXPIRED

There is one unique facility available in Python which is not available in the other programming languages i.e. you can use **else** statement in the looping statement.

If the **else** statement is used with **for** loop, the **else** statement is executed when the loop has exhausted iterating the list.

If the **else** statement is used with a **while** loop, the **else** statement is executed when the condition becomes false.

Example:

```
count = 0
while count < 3:
        print( count, " is less than 3")
        count = count + 1
else:
        print( count, " is not less than 3")
```
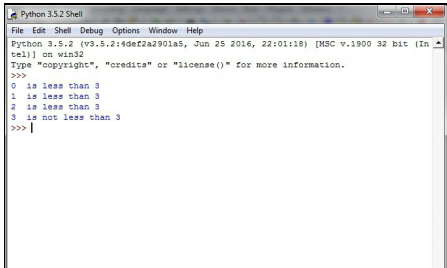
Output:

0 is less than 3

1 is less than 3

2 is less than 3

4 is not less than 3

```
Python 3.5.2 Shell

File  Edit  Shell  Debug  Options  Window  Help

Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 25 2016, 22:01:18) [MSC v.1900 32 bit (In
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
0  is less than 3
1  is less than 3
2  is less than 3
3  is not less than 3
>>>
```

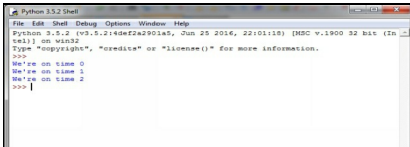The **for** loop in Python has ability to iterate over the items of any sequence, such as a list or a string until the condition is true.

SYNTAX:

for iterating_var in sequence :

    statements (**s**)

simple example is
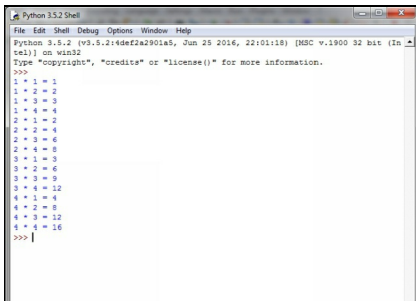
for x in range(0, 3):

    print("We're on time %d" % (x))

Nested for loop example,

for x in range(1, 5):

   for y in range(1, 5):

    print('%d * %d = %d' % (x, y, x*y))

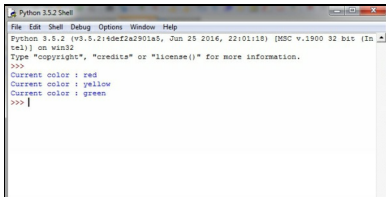**For loop** are used in the list and tuple to print the content one by one.

Example,

colors = ['red', 'yellow', 'green']

for color in colors:

    print('Current color :', color)

Fun fact:

Python has interesting **for loop**

for i **in foo:**

    if bar(i):

        break

    else:

        baz()

the else is executed after the for, but only if the for terminates normally (not by a break).

Else statement in for loop it is unique.

Loop Control Statements:

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

Python supports the following control statements.

- **BREAK STATEMENT**
- **CONTINUE STATEMENT**
- **PASS STATEMENT**

1. break statement

Terminates the **loop** statement and transfers execution to the statement immediately following the loop.

2. continue statement

Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

3. pass statement

The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.

Break is used to exit **for loop** or a **while loop** , whereas continue is

used to skip the current block, and return to the "for" or "while" statement.

Example for break statement:
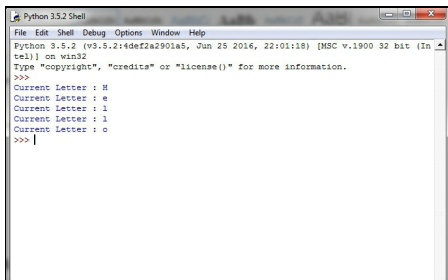
```
for letter in 'Helloworld':
    print ('Current Letter :', letter)
    if letter == 'o':
        break
```

OUTPUT:

EXAMPLE FOR CONTINUE STATEMENT:
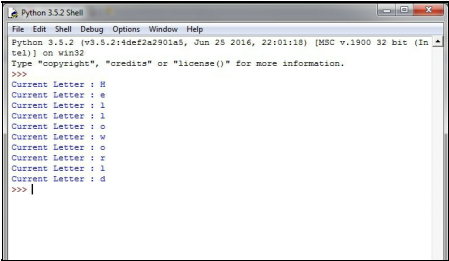
```
for letter in 'Helloworld':
    print( 'Current Letter :', letter)
    if letter == 'o':
        continue
    print( 'Current Letter :', letter)
```

OUTPUT:

Chapter 13
Functions

A **function** is a block of organized, reusable code that is used to perform a single and many related action. **Functions** provide better modularity for your application and a high degree of code reusing. We have already seen many built-in **functions** like print(), str(), input() etc in python in previous sections and also you can create your own **functions** .

In all the programming languages, functions need to be defined, declared with or without parameters with set of codes. Actually, functions are a convenient way to divide your code into useful blocks, allowing us to order our code, make it more readable, reuse it and save some time. Also functions are a key way to define interfaces so programmers can share their code.

<mark>**Defining a Function**</mark>

The very basic and first step is defining the functions. We can define functions to provide the required functionality. There are some simple rules to define a function in Python.

1) Function blocks always begin with the keyword **def** followed by the function name and parentheses (( )).

2) Any input parameters or arguments should be placed within these parentheses. We can also define parameters inside these parentheses. The first statement of a function can be an optional statement - the documentation string of the function or docstring.

3) The code block within every function starts with a colon (:) and is indented. [Again remember white spaces needs and usage].

4) The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no

arguments is the same as return none or null.

{Remember you don't have to use the keywords i.e. reserved words as functions name}

Example:

def my_function():

    print ("Hello This is printing from my_function!")

*) function with arguments or parentheses:

def my_function_with_args(name, color):

    print ("Hello, %s , your favourite color is %s"%(username, color))

Here name and color are the arguments or parentheses where the variables passed from the user by either programmer or user.

*) Functions may return a value to the caller, using the keyword-'return'. For example:

def sum_two_numbers(a, b):

    print a + b

    return a + b

HOW TO CALL THE FUNCTION:

        We already saw that function need to be called in the program to implement it or to use or to take actions in the program. So how to implement it?

Let's see few examples.

def my_function():

    print("Hello This is printing from my_function!")

```python
def my_function_with_args(name, color):
    print( "Hello, %s , your favourite color is %s"%(name, color))
def sum_two_numbers(a, b):
    print a + b
    return a + b

my_function()
my_function_with_args("Jimmy", "red")
x = sum_two_numbers(5,4)
```
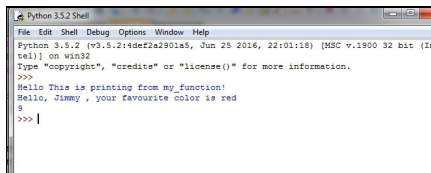
OUTPUT:

Hello This is printing from my_function!

Hello, Jimmy , your favourite color is red

9

The above picture showcases clear example of all the function / facilities in one roof. But it is necessary for a programmer and especially for a beginner to learn about each and every important concept so let's look into some more definitions and pertaining examples. You also have to solve some logical problem in this section so it will be useful for you to understand the concepts.

<mark>**Arguments in Functions:**</mark>

You can call a function by using the following types of formal arguments:
• Required arguments
• Keyword arguments
• Default arguments
• Variable-length arguments

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition. To call the function *myfunction()*, you definitely need to pass one argument, otherwise it would give a syntax error as follows:

```
def myfunction( str ):
"This prints a passed string into this function"
  print ( str)
   return

myfunction("hello world")
```

OUTPUT:
   hello world

Keyword arguments:

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided

to match the values with parameters.

```
def myfunction( str ):
"This prints a passed string into this function"
   print (str)
   return

myfunction( str = "hello world")
```

output:

        hello world

In the above program we passed the argument defining variable itself.

To understand this we are providing another example which may give you a clear understanding about it.

```
def myfunction( str, str1 ):
"This prints a passed string into this function"
  print (str)
  print (str1)
  return

myfunction( str = "hello world", str1="programmer")
```

output:

        hello world
      programmer

Default arguments:

      A default argument is an argument that assumes a default

value if a value is not provided in the function call for that argument. Following example gives an idea on default arguments, it would print default age if it is not passed:

```python
def myfunction( name, age = 30 ):
    "This prints a passed info into this function"
    print( "Name: ", name)
    print ("Age ", age)
    return
myfunction( age=50, name="Jim" );
myfunction( name="carry" );
```

OUTPUT:

Name: Jim
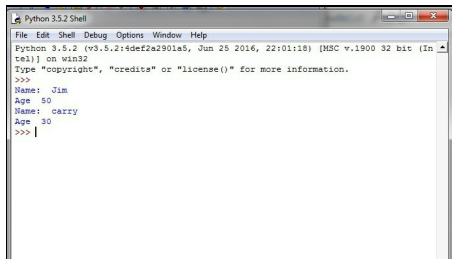
Age: 50

Name: carry

Age: 30

The *return* Statement:

The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

def sum_two_numbers(a, b):

    return a + b

x = sum_two_numbers(5,4)

print("Sum= " , a + b)

OUTPUT:

Sum= 9

Scope of Variables:

All variables in a program may not be accessible at all locations in that program. This depends on where we have declared a variable. The scope of a variable determines the portion of the program where you can access a particular identifier.

There are two basic scopes of variables in Python:
• Global variables
• Local variables

Variables that are defined inside a function body have a local scope, and those defined outside have a global scope. This means that local variables can be accessed only inside the function in which they are declared, whereas global variables can be accessed throughout the program body by all functions. When you call a function, the variables declared inside it are brought into scope.

Example:

We'll give you a same program in two ways, check out how the output differs:

PROGRAM 1:

```
i = 0
count = 0
#global variable
while ( i < 5):
        print ('count=', count)
        count= count + 1
        i = i + 1
else:
        print ('i greater than 5, out of loop' )
```

Output:

PROGRAM 2:

```
i = 0
while (i < 5):
```

```
        #local variable
            count = 0
            print ('count=',count)
            count = count+1
            i = i + 1
else:
        print ('i greater than 5, out of loop')
```
output:

Analyzing the output of both the programs, you may get some clear picture about the scope of the variable. For your better understanding, let's look into the following example that gives even more detailed picture.

```
var1 = 1
def my_function1():
        var2 = 5
        print (var1 + var2)
def my_function2():
        var2 = 5
```
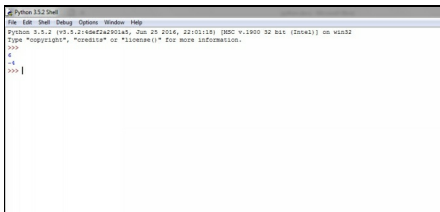
```
            print (var1 - var2)
```

my_function1()

my_function2()

output:

Chapter 14
Modules

Grouping the related codes into modules make the program easier, understandable and also paves way to the reusability of program. Simply put, a module is a file consisting of Python code. A module can define functions, classes and variables. A module can also include runnable code.

Import Statement:

You can use any Python source file as a module by executing an import statement in some other Python source file.

syntax:

import module1[, module2[,... moduleN]

When the interpreter encounters an import statement, it imports the module if the module is present in the search path. A search path is a list of directories that the interpreter searches before importing a module.

Example:

def my_function(name):

        print ("value passed and printed by import" , name)

        return

save the above program as printimport .py

Now to use it in another program, just include import statement in the next program.

import printimport

printimport .my_function("Jimmy")

While you execute the second program, the output will be

value passed and printed by import Jimmy

Let's explain what happened here, while python interpreter encounters the import statement it will bring all functions, classes, variables into its stack and in the second line we call my_function() function by object then all happened like actual pass by value argumented function.

Well before going to the next statements in the modules, we will elaborate the previous example program for understanding.

Program 1:

```
def my_function( name ):
            print ("value passed and your name is" , name)
            return
def my_function1( num ):
            print ("value passed and your age is", num)
             return
def my_function2( color ):
            print ("vaue passed and your favourite color is",
color)
             return
```

save it has printimport.py

Program 2:

```
import printimport
printimport .my_function("Jimmy")

output :
```

value passed and your name is Jimmy

Python's **from statement** lets you import specific attributes from a module into the current namespace.

Example:

    from printimport import my_function1

    By this program, the my_function1() from program 1 only imported to the second program.

        from…. import* statement:

           symbol * means ALL - it will import all modules into the current workspace.

Chapter 15
File Handling

FILE HANDLING:

So far you've learned giving input to the program by getting user input or by passing values in program. There is also another way i.e. by connecting the files to the program. The output of the program also can be printed in the output. In java it is simple and in python it is even simpler because mostly there is no need to import any python file. File can be any type - text, audio, video or image. It can open, write, read, alter or delete.

File opening

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

Modes are like

- r -> open with read only, you can read the file but can't edit / delete anything in the file.
- w -> open with write power, i.e. if the file exists then we can delete all content, delete certain contents and open it to write.
- a -> open in append mode, i.e. opening the file in write mode and adding content at the end of the file.

If you didn't provide any mode, it will automatically open the file as read only. Which means read mode is the default mode.

Example:

File1 = open("world.txt")

File closing

To close a file we use **close() function.** It is necessary to close the file after its use because you can only be able to open some amount of file, if it reached its limit; there is a high chance of crashing the application or program. So make sure you properly close the file after its use.

Example:

File1.close()

Reading a file:

To read a file, **read()** function is used. This function reads the whole file. We can read the file line by line, for that we use **readline()** function and **readlines()** function. The readline() function read one line after another whereas readlines() function is used to read multiple lines.

Example:

File1.read()

Writing a file:

To write a content in the file, use **write()** function. We can write any number of lines in the file. To use write function don't forget to open the file in either write mode or append mode.

Example:

File1.write("anything")

Thinks to remember

If you opened the file in write mode, you can't use read() function. If you opened the file in read mode, you can't use write() function.

While opening a file by giving full path or address of the file, don't forget to put directory.

Also use \\ because \ inside the double quotes or single quotes will make it as escaping tag.

Always properly/fully close the file after its use. Don't forget to do that, because being a programmer it is your duty to take care of all things and avoid the things that may mess the application or program.

To learn the file handling in depth, try for yourself the following exercise:

Task: Write a code to do the following tasks.

Create a file, read its content and print it. Now update the file and again print it.

Sounds easy right, just try by yourself first then check out the below code.

The answer for the above task is,

```
file = open('C:\\Users\\Desktop\\world.txt')
print("real file content\n")
s=file.read()
print(s)
file = open('C:\\Users\\Desktop\\world.txt','a')
file.write('\n earth\n')
file.write('mars\n')
print("updated content\n")
```

```
file = open('C:\\Users\\Desktop\\world.txt')
d=file.read()
print(d)
file.close()
```

The output for this program is:

Now replace read() function with readline() function, readlines() function in the above program to see the varying output inorder to thoroughly learn those functions.

Output when using readline() function

Output when using readlines() function

Chapter 16
Exception Handling

EXCEPTION HANDLING:

As per dictionary definition, exception means abnormal condition. Handling exception is one of the most powerful features that every programming language gives and python is no exception for it. So far you have faced some syntax errors while coding, or you might face some truncation error or white space error.

When your program / coding is wrong, it is okay to get error but you will get some error even when the program is syntactically correct. These errors can be solved by yourself with little more practice but there are some errors which occurs unexpectedly like less memory, connection failure, resource unavailability or may be due to the error occurred by the operating system. How these errors can be solved while your client or customer is using the application or program that are not programmed to handle the situation by you. This exception or error can't be solved by you at the time of occurrence. For tackling this, **exception handling** is used.

There are many type of error occurs. Some of them are:

EXCEPTION NAME	
DESCRIPTION	
Exception	
Base class for all exceptions	
StopIteration	
Raised when the next() method of an iterator does not point to any o	

SystemExit

Raised by the sys.exit() function.

StandardError

Base class for all built-in exceptions except StopIteration and System

ArithmeticError

Base class for all errors that occur for numeric calculation.

OverflowError

Raised when a calculation exceeds maximum limit for a numeric typ

FloatingPointError

Raised when a floating point calculation fails.

ZeroDivisonError

Raised when division or modulo by zero takes place for all numeric

AssertionError

Raised in case of failure of the Assert statement.

AttributeError

Raised in case of failure of attribute reference or assignment.

EOFError

Raised when there is no input from either the raw_input() or input()
file is reached.

ImportError

Raised when an import statement fails.

KeyboardInterrupt

Raised when the user interrupts program execution, usually by press

LookupError

Base class for all lookup errors.

IndexError

Raised when an index is not found in a sequence.

KeyError

Raised when the specified key is not found in the dictionary.

NameError

Raised when an identifier is not found in the local or global namespa

UnboundLocalError

Raised when trying to access a local variable in a function or method
assigned to it.

EnvironmentError

Base class for all exceptions that occur outside the Python environm

IOError

Raised when an input/ output operation fails, such as the print st
function when trying to open a file that does not exist.

OSError

Raised for operating system-related errors.

SyntaxError

Raised when there is an error in Python syntax.

IndentationError

Raised when indentation is not specified properly.

SystemError

Raised when the interpreter finds an internal problem, but when th
the Python interpreter does not exit.

SystemExit

Raised when Python interpreter is quit by using the sys.exit() function
code, causes the interpreter to exit.

TypeError

Raised when an operation or function is attempted that is invalid for

ValueError

Raised when the built-in function for a data type has the valid type
arguments have invalid values specified.

RuntimeError

Raised when a generated error does not fall into any category.

NotImplementedError

Raised when an abstract method that needs to be implemented in a
actually implemented.

NameError

When one starts writing code, this will be one of the
most common exception they will find. NameError happens when
someone tries to access a variable which is not defined.

>>> print (var)

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

NameError: name 'var' is not defined

The last line contains details of the error message, the rest of the lines show the details of how it happened or what caused that exception and where it happened.

TypeError

TypeError is also one of the most found exception. This happens when someone tries to do an operation with different kinds of incompatible data types. A common example is to do addition of Integers and a string.

```
>>> print (1 + "a")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Syntax Errors

Syntax errors are also known as parsing errors, and they are the most common error you may incur while learning Python:

```
>>>
>>> while True print('Hello world')
  File "<stdin>", line 1, in ?
    while True print('Hello world')
                   ^
SyntaxError: invalid syntax
```

The parser repeats the offending line and displays a little 'arrow' pointing at the earliest point in the line where the error was detected. The error is caused by (or at least detected at) the token preceding the arrow: in the example, the error is detected at the

function print(), since a colon (':') is missing before it. File name and line number are printed so you know where to look in case the input came from a script.

Exceptions

Exception means abnormal condition - which is also called as error or bug.

The following are some of the exception and the reply message from python

>>> 9 * (9/0)

Traceback (most recent call last):

  File "<stdin>", line 1, in ?

ZeroDivisionError: division by zero

"""anything divided by zero is infinity by mathematical rule. So python can't answer it which is turned as ZeroDivisionError"""

>>> 4 + var*3

Traceback (most recent call last):

  File "<stdin>", line 1, in ?

NameError: name 'var' is not defined

>>> '3' + 3

Traceback (most recent call last):

  File "<stdin>", line 1, in ?

TypeError: Can't convert 'int' object to str implicitly

How to handle exceptions?

We use ***try...except*** **blocks** to handle any exception. The basic syntax looks like:

try:

    statements to be inside try clause

    statement2

    statement3

    ...

except ExceptionName:

    statements to be evaluated in case of ExceptionName happens

It works in the following way,

- First all lines between **try** and **except** statements will be considered and **try** block will be executed.

- If **ExceptionName** happens during execution of the statements then **except** clause statements execute

- If no exception happens then the statements inside **except** clause does not execute.

- If the **Exception** is not handled in **except** block then it goes out of **try** block.

The above points explain the step by step execution of exception handling mechanism. The exception handling in python is almost similar as other OOPs languages like JAVA, C++. So understanding it is easy, only the keywords are different here.

Now we will see some example in exception handling by solving a scenario.

You need to take the integer input from user and throw or ping him,

if any other datatype input like string or float is entered by user.

```python
def int_number():
    "Returns int number"
    number = int(input("Enter any number: "))
    return number
while True:
    try:
        print(int_number())
        print("this is from the try block")
    except ValueError:
            print("You entered a wrong value. this is from except
block")
```

Output:

In the above example, 9 is integer value so python takes it to execute in **try** block. While user enters 7.0 (a float value) python takes it to **except** block.

The **try ... except** statement has an optional **else** clause, which, when present, must follow all **except** clauses.

Raising Exceptions

The raise statement allows the programmer to force a specified exception to occur. For example:

raise NameError('this is sparta')

Traceback (most recent call last):

  File "<stdin>", line 1, in ?

NameError: this is sparta

       The sole argument to **raise** indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from Exception). If you need to determine whether an exception was raised but don't intend to handle it, a simpler form of the raise statement allows you to re-raise the exception:

try:

    raise NameError('this is sparta')

except NameError:

  print('An exception came by')

   raise

Output:

An exception came by

Traceback (most recent call last):

  File "<stdin>", line 2, in ?

NameError: this is sparta

FINALLY:

This is being used in the socket, file programs. Commonly they will be used while closing the file or socket connections.

A finally clause is always executed before leaving the try statement,

whether an exception has occurred or not. When an exception has occurred in the try clause and has not been handled by except clause (or it has occurred in except or else clause), it is re-raised after the **finally clause** has been executed. The finally clause is also executed "on the way out" when any other clause of the try statement is left via a break, continue or return statement.

Example:

```
try:
    file1 = open("world.txt", "w")
    result = 9/ 0
except ZeroDivisionError:
    print("We have an error in division")
finally:
    file1.close()
    print("Closing the file object.")
```

We have an error in division
Closing the file object.

In this example we are making sure that the file object we open, gets closed in the finally clause.

USER-DEFINED EXCEPTION:

A programmer can't always encounter built in errors, sometimes he may also face some exception which are not predefined by any language. To overcome that, the programmer will create some user defined exception which will help him solving the problem – this is usually done by the exception class. Programs may name their own exceptions by creating a new exception class. Exceptions should typically be derived from the Exception class, either directly or

indirectly.

For example:

```
class MyError(Exception):
    def __init__(self, value):
        self.value = value
    def __str__(self):
        return repr(self.value)

try:
    raise MyError(2*2)
except MyError as e:
    print('My exception occurred, value:', e.value)
```

OUTPUT:

My exception occurred, value: 4

raise MyError('oops!')

Traceback (most recent call last):

  File "<stdin>", line 1, in ?

__main__.MyError: 'oops!'

In this example, the default __init__() of Exception has been overridden. The new behaviour simply creates the value attribute. This replaces the default behaviour of creating the **args** attribute.

Exception classes can be defined as which does anything that other classes can do, but they are usually kept simple, often only offering a number of attributes that allow information about the error to be extracted by handlers for the exception. When creating a module that can raise several distinct errors, a common practice is to create a base class for exceptions defined by that module, and subclass to create

specific exception classes for different error conditions.

```python
class Error(Exception):
    """Base class for exceptions in this module."""
    pass
class InputError(Error):
    """Exception raised for errors in the input.
    Attributes:
        expression -- input expression in which the error occurred
        message -- explanation of the error """
    def __init__(self, expression, message):
        self.expression = expression
        self.message = message
class TransitionError(Error):
    """Raised when an operation attempts a state transition that's not
allowed.
    Attributes:
        previous -- state at beginning of transition
        next -- attempted new state
        message -- explanation of why the specific transition is not
allowed """
    def __init__(self, previous, next, message):
        self.previous = previous
        self.next = next
        self.message = message
```

Most exceptions are defined with names that end in "Error," similar
to the naming of the standard exceptions. Many standard modules

define their own exceptions to report errors that may occur in functions they define.

Below given are some of the exceptions available in Python and its hierarchy.

The class hierarchy for built-in exceptions is:

BaseException

+-- SystemExit

+-- KeyboardInterrupt

+-- GeneratorExit

+-- Exception

    +-- StopIteration

    +-- StopAsyncIteration

    +-- ArithmeticError

    | +-- FloatingPointError

    | +-- OverflowError

    | +-- ZeroDivisionError

    +-- AssertionError

    +-- AttributeError

    +-- BufferError

    +-- EOFError

    +-- ImportError

    +-- LookupError

    | +-- IndexError

    | +-- KeyError

```
+-- MemoryError
+-- NameError
|    +-- UnboundLocalError
+-- OSError
|    +-- BlockingIOError
|    +-- ChildProcessError
|    +-- ConnectionError
|    |    +-- BrokenPipeError
|    |    +-- ConnectionAbortedError
|    |    +-- ConnectionRefusedError
|    |    +-- ConnectionResetError
|    +-- FileExistsError
|    +-- FileNotFoundError
|    +-- InterruptedError
|    +-- IsADirectoryError
|    +-- NotADirectoryError
|    +-- PermissionError
|    +-- ProcessLookupError
|    +-- TimeoutError
+-- ReferenceError
+-- RuntimeError
|    +-- NotImplementedError
|    +-- RecursionError
+-- SyntaxError
|    +-- IndentationError
|         +-- TabError
```

```
+-- SystemError
+-- TypeError
+-- ValueError
|   +-- UnicodeError
|       +-- UnicodeDecodeError
|       +-- UnicodeEncodeError
|       +-- UnicodeTranslateError
+-- Warning
    +-- DeprecationWarning
    +-- PendingDeprecationWarning
    +-- RuntimeWarning
    +-- SyntaxWarning
    +-- UserWarning
    +-- FutureWarning
    +-- ImportWarning
    +-- UnicodeWarning
    +-- BytesWarning
    +-- ResourceWarning
```

The total number of exceptions is being updated in each and every version update of Python. It is somehow not important to know all these exceptions as a beginner, but for the advanced level of programming and debugging, knowing these exceptions will help you in getting better programming results.

We have covered all the important concepts of Python programming language till now. Let's get ahead to learn one of the most important concepts in almost every programming language which evolved from the normal procedural language to the superpower object oriented

programming language.

Enter "from __future__ import braces". You'll get "not a chance" syntax error.

1. >>> **from** __future__ **import** braces
2. File "<stdin>", line 1
3. SyntaxError: **not** a chance

Chapter 17
Classes In Python

Python is an object oriented programming language. Unlike procedure oriented programming or procedural language, in which the main emphasis is on functions, object oriented programming stresses on **objects** . Object is simply a collection of data (variables) and methods (functions) that act on those data. Python's class mechanism adds classes with a minimum of new syntax and semantics. It is a mixture of the class mechanisms found in C++ and Modula-3.

Python classes provide all the standard features of Object Oriented Programming: the class inheritance mechanism allows multiple base classes - a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name.

In python the class structure is simple and compared to the other languages it has less number of line of codes.

Syntax:

class nameoftheclass(parent_class):

    statement1

    statement2

    statement3

Defining a Class in Python

As function definitions begin with the keyword **def** , in Python, we define a class using the keyword **class** . The first string is called docstring and has a brief description about the class. Although not mandatory, this is recommended. Here is a simple class definition.

```
class Firstclass:
    "'This is a docstring. We have created a new class'"
    pass
```

A class creates a new local namespace where all its attributes are defined. Attributes may be data or functions. There are also special attributes in it that begins with double underscores (__). For example, __doc__ gives us the docstring of that class.

Now defining object for a class is another important concept to learn. We hope you do remember the file object we used during while learning the chapter **file handling** .

Example:

```
class Secondclass(object):
# This is the second class.
    a = 90 #attribute
    b = 88 #attribute
p = Secondclass()   #p is object
print(p.a)
```

In the above program, class name is **Secondclass** , **a** and **b** are the class variables. **p** is class object.

After initialising the class object, in the next line we are printing the variable value of **a** by having class object.

Output for the above program:

Here's another example. Try to get an explanation for it by yourself.

```python
class ThirdClass:
    "This is my third class"
    a = 9
    def func1(self):
        print('Hello')
m = ThirdClass()
print(m.a)
m.func1()
print(m.__doc__)
```

Output for the above program is

__init__ METHOD:

       __init__ is a special method in Python classes, it is the constructor method for a class.

In the following example you can learn how to use it.

```python
class contact(object):
    """ Returns a ```contact``` object with the given name, number
and address."""

    def __init__(self, name, number, address):
        self.name = name
        self.number = number
        self.address = address
        print("A Contact object is created.")

    def print_details(self):
        """Prints the details of the student."""
        print("Name:", self.name)
        print("Number:", self.number)
        print("Address:", self.address)
```

__init__ is called whenever an object of the class is constructed. That means whenever we create contact object we will see the message **"A Contact object is created "** in the prompt. You can see the first argument to the method is self. It is a special variable which points to the current object (like this in C++). The object is passed implicitly to every method available in it, but we have to get it properly in every method while writing the methods.

Example:

std1 = contact()

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: __init__() takes exactly 4 arguments (1 given)

con1 = contact('Jim',98888,'walls street')

A Contact object is created

In this example, at first we tried to create a Contact object without passing any argument and Python interpreter complained that it takes exactly 4 arguments but received only one (self). Then we created an object with proper argument values and from the message printed, one can easily understand that __init__ method was called as the constructor method.

Now we are going to call print_details() method.

con1.print_details()

Name: Jim

Number: 98888

Address: walls street

```
Python 3.5.2 Shell
File Edit Shell Debug Options Window Help
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 25 2016, 22:01:18) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
>>> con1=contact('Jim',98888,'walls street')
A Contact object is created.
>>> con1.print_details()
Name: Jim
Number: 98888
Address: walls street
>>>
```

As we already learned how to create an object, now we are going to see how to delete Python object. We use del for this.

del con1

If you try to see object you will get an error:

Example

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

NameError: name 'con1' is not defined

INHERITANCE:

Inheritance is a powerful feature in object oriented programming. It refers to defining a new class with little or no modification to an existing class. The new class is called derived (or child) class and the one from which it inherits is called the base (or

parent) class. Derived class inherits features from the base class, adding new features to it. This helps in re-usability of code.

Classes can inherit from other classes. A class can inherit attributes and behaviour methods from another class, called the superclass. A class which inherits from a superclass is called a subclass, also called heir class or child class.

Python Inheritance Syntax

class DerivedClass(BaseClass):

    body_of_derived_class

```
class Car(object):
    """Returns a car name. """
    def __init__(self, name):
        self.name = name

    def get_details(self):
        "Returns a string containing name of the car"
        return self.name

class model(Car):
    """Returns a model object, takes name, modelid, price"""
    def __init__(self, name, modelid, price):
        Car.__init__(self,name)
        self.modelid = modelid
        self.price = price

    def get_details(self):
```

```
        "Returns a string containing model deatils."
        return "%s , %s  value is %s." % (self.name, self.modelid,
self.price)
```

OUTPUT:

Above example is a single level inheritance example, where **Car** is super class and **model** is derived class.

We are inheriting the name attribute i.e. car name and made it available in the derived class.

We will now see syntax for multiple inheritance:

class Base1:

   pass

class Base2:

   pass

class MultiDerived(Base1, Base2):

   pass

syntax for multi-level inheritance is

class Base:

```
    pass
class Derived1(Base):
    pass
class Derived2(Derived1):
    pass
```
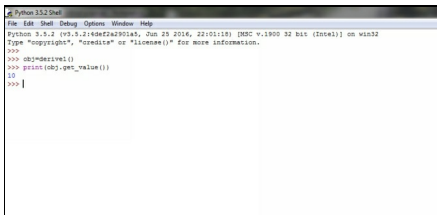
Method overriding

In Python **method overriding** occurs when defining in the child class a method with the same name of a method in the parent class. When you define a method in the object you make this latter able to satisfy that method call, so the implementations of its ancestors do not come into play i.e. base class method will not be implemented.

```
class base1(object):

    def __init__(self):
        self.value = 9

    def get_value(self):
        return self.value

class derive1(base1):

    def get_value(self):
        return self.value + 1
```

Output:

```
Python 3.5.2 Shell
File Edit Shell Debug Options Window Help
Python 3.5.2 (v3.5.2:4def2a901a5, Jun 25 2016, 22:01:18) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
>>> obj=derive1()
>>> print(obj.get_value())
10
>>>
```

Inheritance delegation occurs automatically, but if a method is overridden the implementation of the ancestors i.e. base class is not considered at all. So, if you want to run the implementation of one or more of the ancestors of your class, you have to call them properly/clearly.
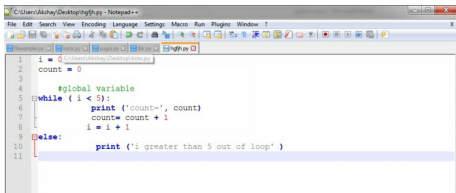
Chapter 18
Tips For Beginners

Now we've almost come to the end and for a beginner level you have learned all the important concepts in Python. What we've taught you in this tutorial is the fundamental of Python Programming and it is the base for the higher concepts like Implementing Python in networking, CGI, Application development and Server scripting etc., .

While debugging the program definitely you will have errors. Here we are talking about syntax error, it may be because of syntax missing or IndentationError problem. We advise you not to give up and loose hope – keep learning and keep trying, you will get ahead with Python Programming very shortly.

IndentationError is an error which can be easily solved if you focus on the left most corner of the text editor.

Check out the line 5 to 6, new sub line occurs in line 7 right? so if you adjust that you will get the correct output. It is very common that while typing program quickly we may press **enter** . If you press that in between block or module, sub branch will be created inside that. It will show like in line 5 and 9. This also creates IndentationError.

If you can't solve the error, simply copy the error and paste it in google and find solution. Stackoverflow members are experts in it so try that website.

ADVANTAGES OF LEARNING PYTHON:

- ➢ Easy to learn
- ➢ High efficiency compared to other languages
- ➢ Faster than c, c++, java and slightly slower than perl, php, which makes it an average speed language. Though the speed is lesser comparatively, Python is highly efficient with latest hardware which makes it a favourite language.
- ➢ Python gained quick publicity and many communities are developing the language.
- ➢ It is an open source
- ➢ Whatever the field may be, Python can be used.
- ➢ Less lines of code compared to others.
- ➢ Best suitable for start-ups looking for quick profit.

****

Dear Reader, if you liked what you read, please leave an honest review in Amazon.

****

****

If you want to tell us about the quality or improvement areas in this book, please write to upskillpublishing@gmail.com

We read all your comments, feedback and inputs and ensure to make reading this book a pleasant experience by constantly updating it.

This guide is developed to help you to get started with Python Programming. If we served this purpose, we consider it a success.

****

---The End---