

```
dl<<"iterations #"<<" "<<" X(1)"<<"X(2)"<<" X(3)";
0"<<setw(17)<<j1<<setw(15)<<j2<<setw(14)<<j3;
float temp[3];
long float j1,j2,j3;
cout<<endl<<"Form
cout<<endl<<"X(1)
cin>>j1;
cout<<endl<<"X(2)
cin>>j2;
cout<<endl<<"X(3)
cin>>j3;
temp[1]=j2;temp[2]=j3;
]*temp[1]-a[2]*temp[2])/a[0];
0)*temp[0]-b[2]*temp[2])/b[1];
0)*temp[1])/c[2];
<<s< w(17) 1<<setw(1  <j2 tw(14)<<j3<<endl;
p[0] ==te 3=
{
i<3;
tion Of
float b[]
function d finition */
cout<<"b(1) =";
cin>>a[3];
cout<<endl;
for(int j=0;j<3;j++)
{
cout<<"a(2"<<j+1<<")=";
cin>>b[j];
cout<<endl;
}
cout<<"b(2) =";
cin>>b[3];
cout<<endl;
for(int k=0;k<3;k++)
{
cout<<"a(3"<<k+1<<")=";
0
g-- "<<endl<<endl;
ncode"<<a[0]<<"X(1) + "<<a[1]<<"X(2) +
cout<<"b(3) =";
cin>>c[3];
cout<<endl;
}
```

# Python

## FOR BEGINNERS

LEARN PYTHON QUICKLY AND EASILY

# **PYTHON FOR BEGINNERS**

**Learn Python Quickly and Easily**

# **Disclaimer**

The entire rights of the content found in this e-book are reserved with the publisher. The replication or republication of any part of this content in any possible form is strictly prohibited without the author's consent. Any such action is punishable by law.

This e-book is solely for educational purposes and should be taken as such. The author takes no responsibility for any misappropriation of the contents stated herein and thus cannot and will not be held liable for any damages incurred because of it.

# **Thank You!**

Hey,

First, I wanted to thank you for choosing to download this book. I am very excited to share this book with you and I sincerely hope that you will like it and find it useful.

If you enjoy this book, I would be very grateful if you posted a short review on Amazon and if you don't just ask for a refund and feel free to contact me if you have any questions, comments or suggestions for my next books. I am very grateful for your support; it really makes my work worthwhile. I read every review personally, so I can receive your feedback and make my books even better.

Enjoy!

# Table of Contents

[Chapter 1: Getting Started](#)

[Chapter 2: Introducing Variables](#)

[Chapter 3: Operators](#)

[Chapter 4: Strings](#)

[Chapter 5: User Input](#)

[Chapter 6: Lists](#)

[Chapter 7: Dictionaries](#)

[Chapter 8: Conditionals If-Else](#)

[Chapter 9: Loops](#)

[Chapter 10: Functions](#)

[Chapter 11: Introducing Object Oriented Programming](#)

[Conclusion](#)

## Chapter One

# Getting Started

This chapter will walk you through the steps of Python installation, as well as the basics of Python coding. We will be using Windows 7 PC throughout the book, but you can still follow along if you are using later Windows versions or if your computer is a Macintosh or Linux.



## Python Installation

Most modern systems today come with Python pre-installed. Before proceeding, check if you already have Python installed.

### *Pre-Installation Check*

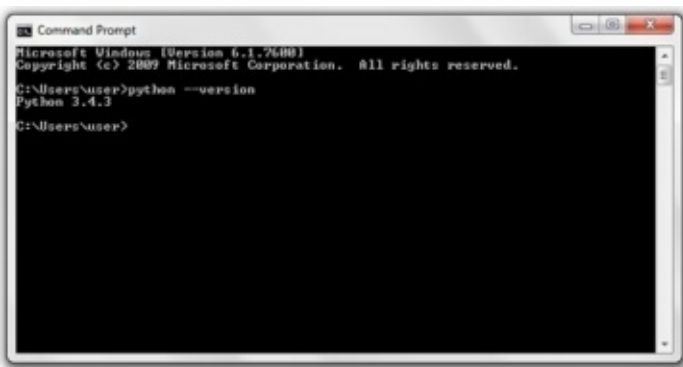
Use the command prompt to determine if you already have Python, as well as which version.

1. Open the command prompt in Windows by going to **Start -> Run**, type *cmd* then hit Enter (see Figure 1).
2. On the command prompt, type *python --version* and hit Enter. If it reports an error such as “python is an unrecognized command,” it means you do not have Python yet and you will have to download and install it. If you already have Python, you will see the version (see Figure 2).

Figure 1



Figure 2



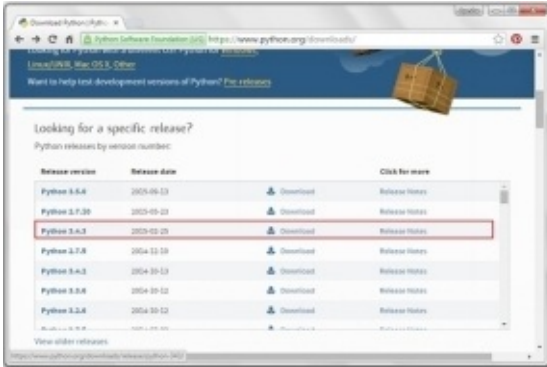
### *Downloading Python*

We will use Python version 3.4.3 in this book. Follow these steps:

1. Go to the official Python website: <https://www.python.org/>
2. From the menu, go to **Downloads -> All releases**. Scroll down to the list of Python releases and select version 3.4.3 (see Figure 3).

3. Scroll down to the **Files** list and click on your system specific file to start downloading.
4. Click to open the downloaded file, then click **Run** in the dialog box. Next, follow the prompts.

Figure 3



Congratulations! You have installed Python. Now you can create your first program.



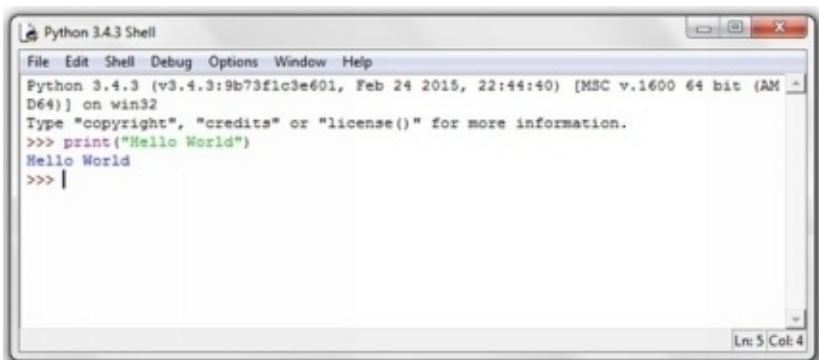
## Your First Program

Python installation comes with an integrated development environment (IDLE), which is the tools set you will need to write, run, and debug code. As we proceed in this chapter, we will investigate a number of different options for running Python code.

Following tradition, we will create the “Hello World” program.

1. From the start menu, go to **All Programs** -> **Python 3.4** folder. Expand to view contents.
2. Launch **IDLE (Python 3.4 GUI -64 bit)**. This launches a window called “Python Shell,” which is part of Python’s IDLE, and we will use it to experiment on our code. Python Shell prompts you with triple chevrons (`>>>`) to enter your code, which indicates that the shell is ready to take your instructions.
3. Type in the command `print(“Hello World”)` at the prompt, then hit Enter to run the code.
4. If you entered the above code correctly, you will see this output: *Hello World*.

Figure 4



Congratulations! You have created your very first program in Python. With that said, you probably have more questions than previously. Let us see what is happening behind the scenes.

## **The Basic Terminologies**

### ***What is a Computer Program?***

Computers can help us with many tasks, from fetching information from the internet to connecting people, but they are nothing more than intelligently-instructed machines—they can only help if given clear instructions. If asked to do something that is not included in their instructions set, they will be confused. These set of instructions are called computer programs.

Computer programs are not tangible parts of the computer, like the hardisk, monitor, or CPU. Programs comprise the software, which controls the computer's behavior. No matter how simple or complex a program may be, it is always composed of a set of instructions. Writing these instructions is called “programming,” or sometimes simply “coding.”

In the previous section, we instructed the computer to print the statement “Hello World.” Simply put, we created a program to have the computer print that statement.

### ***What is a Computer Language?***

If you are in a foreign country and do not speak its language well, you will need to at least know some basic words and phrases so the locals would understand you sufficiently. Likewise, when you are instructing a computer, you will need to know a language comprehensible to it.

Unfortunately, computers only understand and run programs written in machine or low-level language. Now, there are number of disadvantages to low-level languages. First, these are difficult to write; the code is difficult to read, understand, and maintain; and these are often lengthy. Second, programs written in low-level languages are machine type specific—they cannot be ported to other computers.

In contrast, there are high-level languages that are close to natural English, and these are how we can communicate with and instruct computers. Programs written in high-level languages are readable, short, and can be ported with little or no modification onto different computer types. Since they can understand only low-level languages, programs written in high-level languages need to be processed before computers can run them. Python is an example of high-level language.

### ***What is Syntax?***

“Ball get me.” This sentence is not written in proper English, but you can still make sense of it. As we have pointed out, computers are not smart enough to decipher ambiguous instructions. When writing programs, we need to follow specific rules to make the instructions comprehensible to computers. The syntax of a computer language defines the

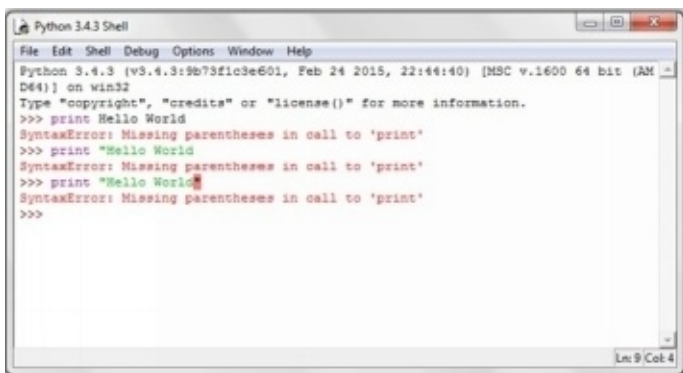
rules and structure. If we make coding mistakes, the computer will report a syntax error and stop working. You can think of syntax as the rules of grammar for computers.

Previously, we typed `print ("Hello World")` to instruct the computer to print a line. To get it to print a line, we need to use special syntax or code:

1. Use `print ( )` command
2. Enclose the statement you want to print in quote marks

If we do not follow this syntax (or rules to code) and instruct the computer to print something, it will report an error.

Figure 5



```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> print Hello World
SyntaxError: Missing parentheses in call to 'print'
>>> print "Hello World
SyntaxError: Missing parentheses in call to 'print'
>>> print "Hello World"
SyntaxError: Missing parentheses in call to 'print'
>>>
```

In the above figure I tried to print “Hello World.” Every time I did not follow the syntax, I got an error. You will need to memorize Python syntax rules to communicate with your computer.

## The Development Environment

In the previous section we used IDLE (Integrated DeveLopment Environment) to write our code. The program was run in IDLE's Python Shell which printed *Hello World*. IDLE also features a full-featured text editor, which provides color-code highlighting and helps you indent text according to python syntax specifications. If this does not make much sense to you right now, do not worry; you will get the hang of it soon enough.

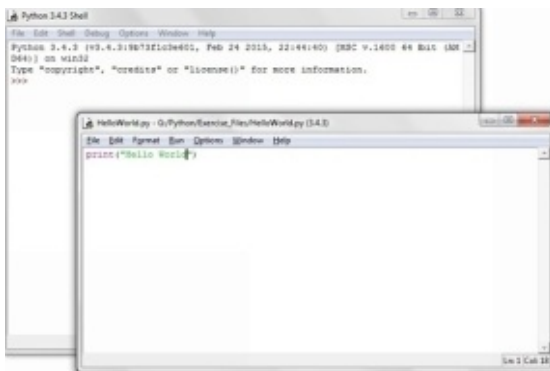
### *Saving and Running Files in IDLE*

We coded a simple program which prints a statement, but programs are usually much more complex. Suppose a program computes a complex mathematical operation. We cannot re-write all the code every time; we need to save them for later use. To do this, follow these steps:

#### 1. Launch IDLE

2. Go to **File -> New**. This opens a new, untitled window
3. Type your code again: `print ("Hello World")` (see Figure 6)
4. Go to **File -> Save**
5. Name the code file *HelloWorld.py* and save it.
6. From IDLE menu, go to **Run -> Run Module**. You should see the result of the code you wrote and saved displayed in Python Shell (see Figure 6).

Figure 6



## Commenting Your Code

Comments are text notes you add to your program to give explanatory information, which can be used to increase the readability of your code or for self-reference. Comments are ignored by the computer when executing a program. You can think of comments like notes in your presentation; the audience cannot view your notes, but these notes provide you information for your reference. Also, when working in large organizations, programs are usually maintained by a number of individuals, so adding comments help peers understand the purpose of the codes.

In python you can add comments by using the character #

Let us try it out

1. Launch IDLE Text Editor. In Python Shell, got to **File -> New**
2. In the text editor type `print ("Hello World") # This is a comment`
3. Save this new file
4. From the menu, go to **Run -> Run Module**
5. Note that the output is the same—the computer ignored the comment.

Note: Most high-level language support multiline (or block) comments. In python you may use `''' Block comment'''`. However, Python style guide suggests using multiple single line comments using #.

## Chapter Two

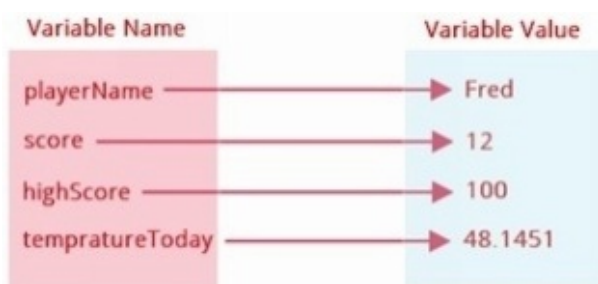
# Introducing Variables

## What are Variables?

Variables in programming provide a way to store and retrieve data. Technically, variables are named locations in the computer's memory space (when you define a variable, you secure a location in the computer's memory). Any value assigned to the variable will be stored in the acquired location. You can then access these values and manipulate them using variables. In other words, variables are filenames you assign to a memory location that contains values.

You can think of variables as empty labelled boxes. You can store your items in these boxes, take out old items, replace them with new ones, and so on. You identify and access each box by its label. Each variable stores only one piece of information (see Figure 7).

Figure 7

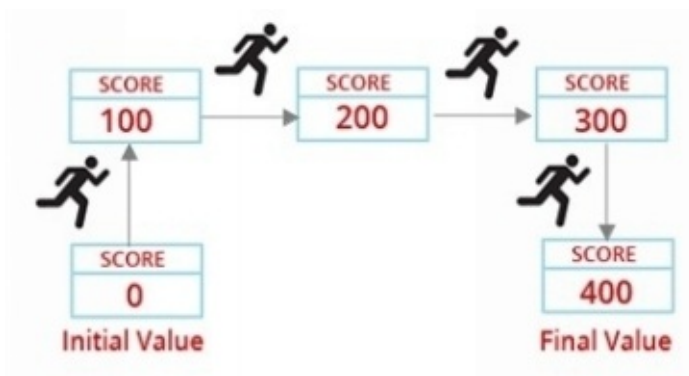


A variable, as the name implies, is something that can change (as opposed to a constant). A simple example is a “score” variable in a game. A player’s score starts at zero, but it increases as the game progresses. In this case, a program defines a variable called “score.” At the start of the game, the value stored in “score” is 0. Each time the player progresses, the value in “score” is updated.

Now, let us assume 100 is added to the “score” each time the player progresses. The following figure shows the state diagram of “score.” It shows that the variable “score” is used to save different values at different times in game play.

Figure 8





## Common Variable Types in Python

You can use variables to store different kind of data items. Some values are numbers while others could be letters, words, or even phrases. The “type” of variable refers to the type of data/value it stores.

Python data types		Types of data	Examples
Numeric Data type	int	Stores integers	-2,-1,0,1,2... From our discussion above, “score” is a type of int.
	float	Stores numbers with floating point or decimal numbers	3.1315 1.00 -5.10
	str	Stores string of letters, numbers, or phrases. Strings are identified by single or double quotes around the string.	“Hello World”, “a”, ‘22’, ‘ ’. From our discussion above, “playerName” is a type of string.

## Defining and Assigning Variables

Defining variables is simply creating a variable and storing a value in it. To assign a value in a variable, we use the assignment operator =.

<variable name> = <value to store>

Python evaluates the right-hand side of the expressions and binds (or assigns) it to the variable on the left. The type of variable is set when it is defined—you do not specify the type of variable. When you define a variable and assign a value to it, python infers the type of variable according to the type of data stored therein. Once defined, we can use the variable in our program; this is called variable referencing.

In strongly typed languages, such as C++, you need to declare (or define) variables before you use them. When you declare a variable, you bind it to a data type (you decide which type of data to store). During the life of a program, you may change the value of the variable, but you cannot change its type. For example, if you declare a variable of type int, then you may use it to save any integer, but you cannot use it to save a string. Python does not require this type of declaration.

### Exercise # 01: Defining Variables

Launch IDLE text editor and type the following code. Save the file as Variables.py

```
customerFirstName = "Fred"
customerLastName = 'Robert'
customerID = "4312-11"
booksPurchased = 10
subtotal = 350
taxAmount = 20
```

From Run->Run Module to open the Python Shell. Type the following:

```
>>> customerFirstName
>>> customerLastName

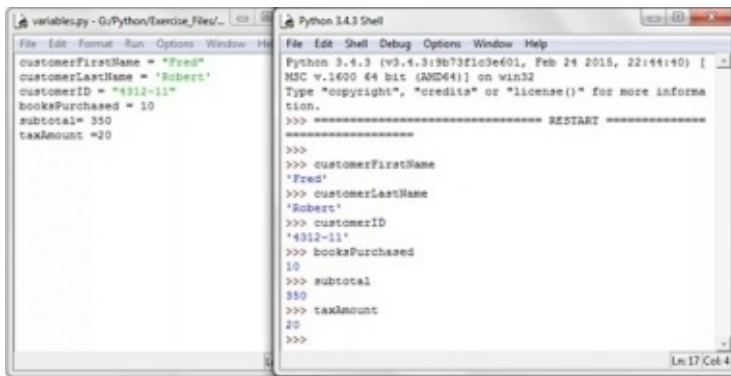
>>> customerID
>>> booksPurchased

>>> subtotal

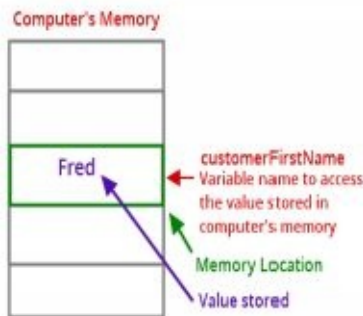
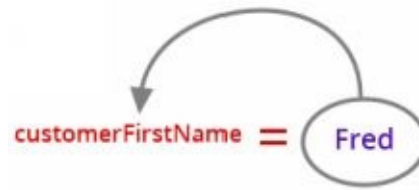
>>> taxAmount
```

Figure 9

In the above example, we have created a variable named *customerFirstName* and stored the string value "Fred" in it. We have used the assignment operator = to bind (assign) the



value at the right to the variable at the left. Note that strings are enclosed in quotes.



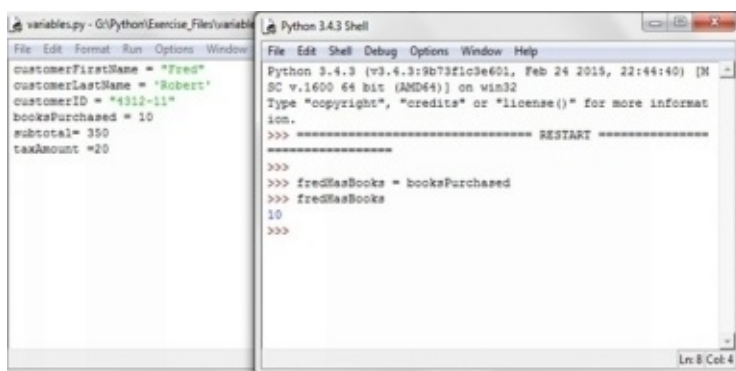
## Exercise # 02: Creating and Assigning new variables

Open the Variables.py and go to Run -> Run Module to launch the Python shell. At prompt >>> type the following, then hit Enter

```
>>> FredhasBooks = booksPurchased
>>> FredhasBooks
```

In the above example, we create a new variable, *fredHasBooks*, and assigned it the value stored in *booksPurchased*, then we checked the value for *fredHasBooks*.

Figure 10



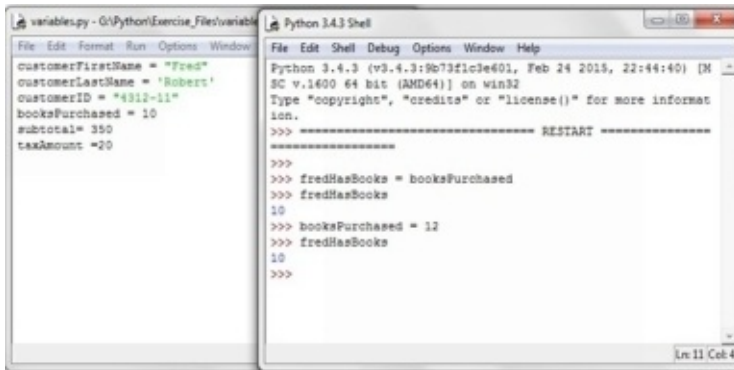
The output is 10.

## Exercise # 03: Changing Variables

Continuing from the last program, add the following to the Python Shell and hit Enter.

```
>>> booksPurchased = 12
>>> fredHasBooks
```

Figure 11



Let us investigate the code line by line:

1. Created a new variable, *fredHasBooks*, and assigned it the value in *booksPurchased*
2. Updated variable *booksPurchased* and set its value at 12
3. Sought the value of *fredHasBooks*.

Can you guess why the value did not change?

In Step 1, we set the value of *fredHasBooks* to whatever was stored in *booksPurchased* (the value was 10). In step 2, we changed the value of *booksPurchased*, but it had no effect on *fredHasBooks*.

Technically, when we set *fredHasBooks* to the value of *booksPurchased*, the variable *fredHasBooks* did not acquire a new memory location to store value, so it can simply point to variable *bookPurchased*. When we store new value in *booksPurchased*, it acquires a new memory location, but *fredHasBooks* retains the old value.

## Finding the Variable Type

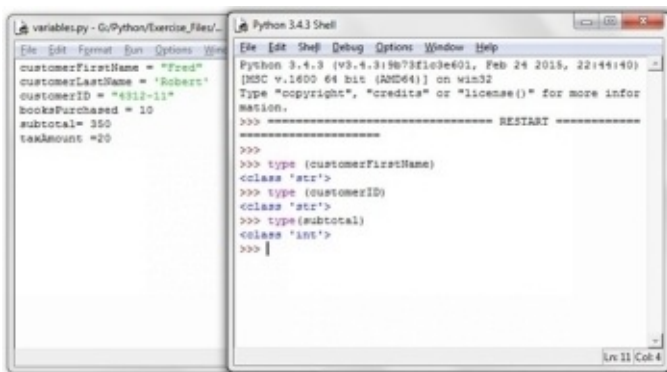
You can identify the variable type by using the `type (variable_name)` command.

Exercise # 04: Find the type of variables

Open Variables.py and go to Run ->Run Module. At prompt >>> type the following

```
>>> type (customerFirstName)
>>> type (customerID)
>>> type (subtotal)
```

Figure 12

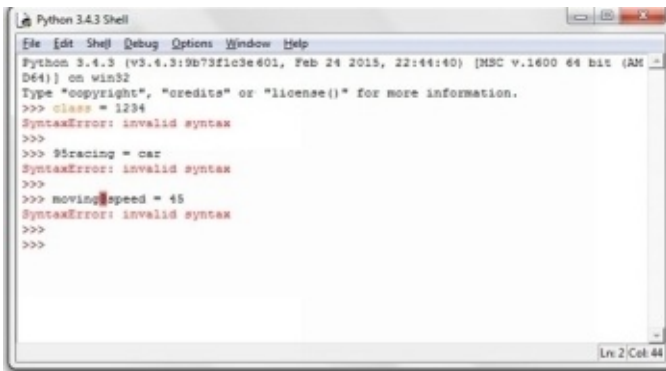


## Variable Naming Convention

Programming is all about defining and working with variables. When naming variables, the following are key points:

1. Use descriptive variable (generally telling its purpose).
2. Python is case sensitivity language. This means a variable called “total” is different than a variable called “TOTAL”.
3. Variables must begin with letters, so 214zip is an invalid variable name.
4. It is better to start variable names with lower case letters because programmers use upper case letters for variables that contain values that will not change during the course of program.
5. If a variable’s name contains more than one word, it may be separated with an underscore (*example, my\_score*). Alternatively, you can use camel case (*example, myScore*).
6. You cannot use keywords reserved by Python for certain tasks. For example, since Python uses class as a keyword to define a class, you cannot create a variable and name it “class”.

Figure 13



## Chapter Three



# Operators

If a computer program is a recipe, then operators are its key ingredients. Operators are special characters that enable you to manipulate, assign, and compare values in a program. Operators work on operands, and Python provides a number of operators to do computations and comparisons.

We have already seen the assignment operator `=` in our last section. We used the assignment operator to assign values to variables. In this chapter, we will see and use more operators to perform computations and manipulate values stored in variables.

# Operators in Python

## Mathematical Operators

Python provides operators to perform mathematical operations on values such as multiplication, division, addition, and subtraction.

Operators	Functions
+	Addition
-	Subtraction
*	Multiplication
/	Division
//	Truncated Division (or floor division) Quotient
**	Exponential
%	Modulo

All of the above are called binary operators as they take two operands (for example,  $2 + 3$ ; here 2 and 3 are operators).

Besides binary operators, there are unary operators, which accept only one operand. Negation (-) is an example of unary operator (more on this later).

## Comparison Operators

Operators	Functions
+	Addition
-	Subtraction
*	Multiplication
/	Division
//	Truncated Division (or floor division) Quotient
**	Exponential
%	Modulo

We will look more closely at comparison operators later when we discuss conditionals and

loops.

***Logical Operators***

Operator	Description
and	Boolean and
or	Boolean or
not	Boolean not

## Computations with Python

Now, let us use mathematical operators to manipulate values stored in variables.

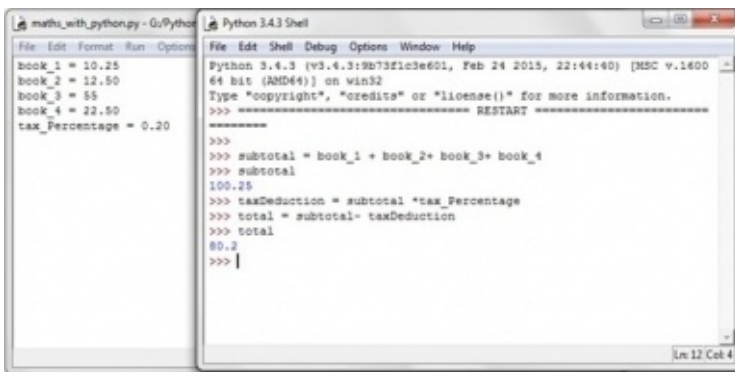
### Exercise #05: Doing Math with Python

For this exercise, we will calculate the total cost of four books purchased by Fred. Open IDLE Text file and type the following code. Name the file as `maths_with_python.py` and save.

```
book_1 = 10.25
book_2 = 12.50
book_3 = 55
book_4 = 22.50
tax_Percentage = 0.20
```

Run the python file by Run-> Run Module this will open the Python Shell. Type the following code at the prompt `>>>`

```
1 >>> subtotal = book_1 + book_2 + book_3 + book_4
2
3 >>> subtotal
4
5 >>>
6 >>> taxDeduction = subtotal * tax_Percentage
7
>>> total = subtotal- taxDeduction
>>>total
>>>
```



In “`maths_with_python.py`”, we defined a few variables and called them `Book_1`, `Book_2`, `Book_3`, and `Book_4`. Each variable holds the price for a book. Note that `Book_1`, `Book_2` and `Book_4` are variables of type float,

while `Book_3` is type int. We also defined another variable to hold the tax percentage, `tax_Percentage`. Let us investigate line-by-line what is happening.

1. We added the price of all the books using the `+` operator and assigned it using the assignment operator `=` to a new variable called “`subtotal`”.
2. Next, we typed `subtotal` at the prompt to show its value, which is 100.25.
3. Python shell returned the value of `subtotal`.
4. We then calculated for the tax deduction by multiplying `subtotal` with `tax_Percentage` using the `*` multiplication operator and assigned it to another variable, “`taxDeduction`”.
5. Finally, we calculated the total amount Fred had to pay by subtracting the tax deduction from `subtotal` using the `-` subtraction operator. The answer is stored in a new variable called “`total`”.
6. We found the value of `total` and Python returned its value.

## Note:

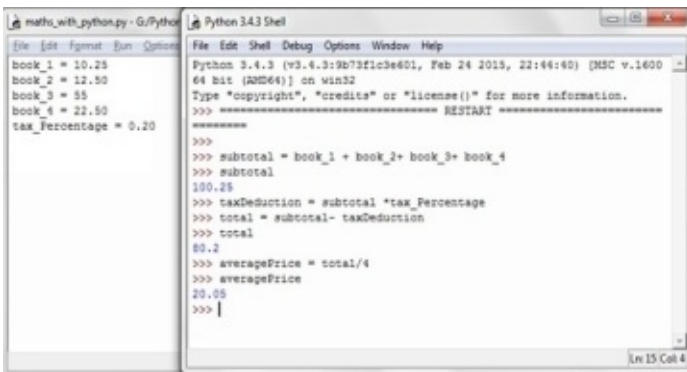
When we perform binary operations on operands of different types (such as int and float), the lesser general type (int) is converted to the more general type (float) automatically. So, Book\_3 is converted to float and its value becomes 55.0

**Exercise # 06:** Continuing from our last exercise, let us now find the average price of each book. In the Python shell, type the following code at the prompt >>>

```
1 >>>total
2
3 >>> averagePrice = total/4

>>>averagePrice
```

## Output

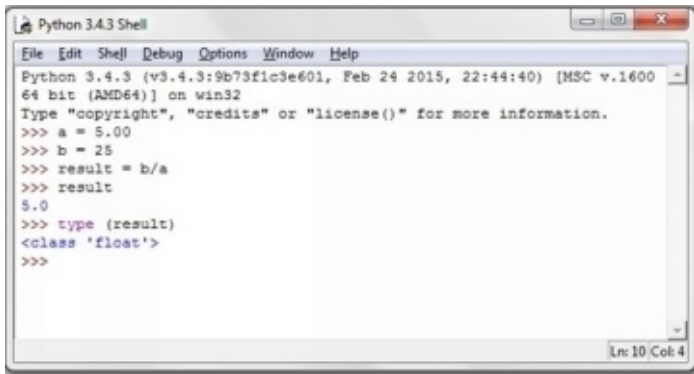


Here, we used the division / operator to find the average price of each book. If you find the type of averagePrice, you will see that it is of type float.

## Exercise # 07: Division and Variable types

Open the Python Shell and type the following code. Can you guess the type of result variable in the following code?

```
1 >>> a = 5.00
2
3 >>> b = 25
4
>>> result = b/a
>>> type (result)
```

A screenshot of a Python 3.4.3 Shell window. The window has a title bar that says "Python 3.4.3 Shell" and a menu bar with "File", "Edit", "Shell", "Debug", "Options", "Window", and "Help". The main text area shows the following code and output:

```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600  
64 bit (AMD64)] on win32  
Type "copyright", "credits" or "license()" for more information.  
>>> a = 5.00  
>>> b = 25  
>>> result = b/a  
>>> result  
5.0  
>>> type (result)  
<class 'float'>  
>>>
```

The status bar at the bottom right indicates "Ln: 10 Col: 4".

Also, the division operator / did not ignore the decimal part of the result. The dividend “b” (25) is an int, the divisor “a” (5.00) is a float, and the quotient is also a float.

## Floor Division

In floor division, or truncated division, the integer part of the result is retained; meaning, all digits after the decimal point in the result are removed—hence, truncated.

- If the dividend and divisor are int, the result is also int.
- If either the dividend or divisor is float, the result will be float truncating the digits after the decimal point.

Floor Division is done by the // operator.

### Exercise # 08: Floor Division

Open the python Shell and type the following code. Can you guess the result value and its type?

```
1 >>> a = 5.00
2
3 >>> b = 25
4
5 >>> result = b//a
6
7 >>> type (result)
8
9 # Example 2
10 >>> result = 10//3
11 >>> result
12 >>> type (result)
13
14 #Example 3
15 >>> result = 26.00//5
16 >>> result
17 >>> type (result)
```

In the first example, we see that the operators / or // do not make any difference since the result is a whole number (no decimals).

In example 2, 10 is not completely divisible by 3. Using the usual / operator, the result would have been  $10/3 = 3.333$ . However with the // operator, the digits after the decimal point are truncated and the integer value of the result is retained. And since the dividend and divisor are int, the result is also int.

The last example is particularly interesting. If we are to divide 26.00 by 5 using the / operator, the result would be 5.2. However, we used the // operator, so the truncated result is 5. Since the dividend is float, the result should also be float. Hence, the result shown is 5.0 (instead of simply 5).



```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
Type "copyright", "credits" or "license()" for more information.
>>> a = 5.00
>>> b = 25
>>> result = b/a
>>> result
5.0
>>> type(result)
<class 'float'>
>>> result = 10//3
>>> result
3
>>> type(result)
<class 'int'>
>>> result = 26.00/5
>>> result
5.0
>>> type(result)
<class 'float'>
>>> |
```

Ln: 20 Col: 4

## Using Parenthesis to specify Order

To control the operation sequence, we use parentheses. The computer then understands that values inside parentheses should be evaluated before the rest of expression. Note that Python normally performs multiplication or division before addition and subtraction.

### Exercise # 09: Computation Sequence

Open the Python Shell and type the following code.

```
>>>
>>> 10 + 2 * 10
>>>
>>> ( 10 +2) * 10

# multiple level of brackets

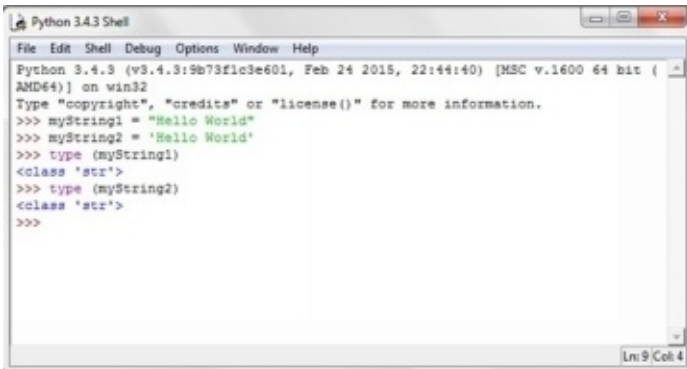
>>> (6 + 4) * (10 - 8)
>>>
>>> ((5 + 15)/ 2) + ((16-1)/3)
>>>>>> ((5 +10 ) *2)/ 5
```

## Chapter Four

# Strings

A string is a sequence of characters enclosed in quotes. In Python, we can use both single and double quotes to define a string, but it is important you use one type of quote (either single or double) to open and close the string. In the figure below, we define two strings, *myString1* and *myString2*.

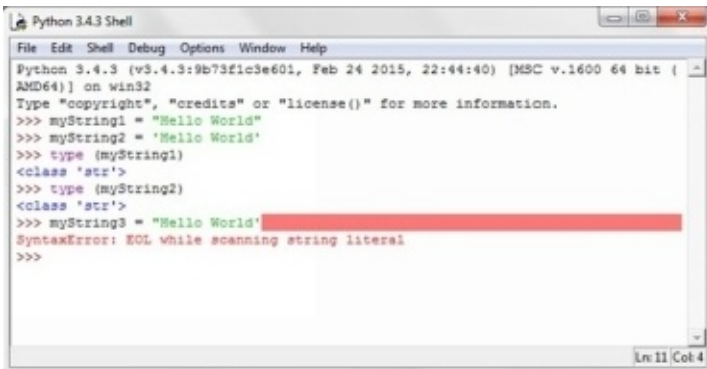
Figure 18



```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> myString1 = "Hello World"
>>> myString2 = 'Hello World'
>>> type(myString1)
<class 'str'>
>>> type(myString2)
<class 'str'>
>>>
```

Now consider the figure below:

Figure 19



```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> myString1 = "Hello World"
>>> myString2 = 'Hello World'
>>> type(myString1)
<class 'str'>
>>> type(myString2)
<class 'str'>
>>> myString3 = "Hello World"
SyntaxError: EOL while scanning string literal
>>>
```

In the above figure, the Python interpreter reported an error when we attempted to define a string, *myString3* and assign value “Hello World”. Now, take note of the error message. What do you think does it mean?

When we defined the new string, Python scanned the string literal for a matching closing quote (supposedly, an end quote ” symbol). Since Python could not find matching quotes, it reported an EOL—an “end-of-line” error. To correct the error, we will need to close the string with the matching end quote.

## Single, Double, and Three Single Quotes

### Exercise #10: Understanding String Problems

Launch Python Shell and type the following code. Can you figure out the problem?

#Case 1: enclosing the string in double quotes

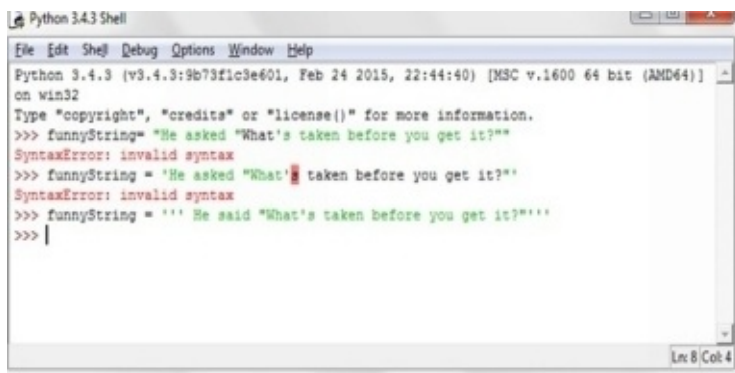
```
>>> funnyString = "He asked, "What's it going to take before you get it?" "
```

#Case 2: enclosing the string in single quotes

```
>>> funnyString = 'He asked, "What's it going to take before you get it?"'
```

In this exercise, we have defined the string *funnyString*. For Case1, we opened and closed the string with a pair of double quotes; for Case 2, we enclosed the string in single quotes. Now, the program has reported errors. What do you think happened?

Figure 20



In Case 1, we defined a string using double quotes. Python then scanned the string for a matching pair of double quotes and found it just before *What's*. Python marked this point as the end, thereby recognizing *"He asked, "* as a string, even while it is merely a segment of the string. Python then marked the rest as invalid and reported an error.

In Case 2, we defined a string using single quotes. Here, Python marked *'He asked, "What'* as the string and reported an error from *s*.

A simple solution is to use three single quotes to enclose the string because these will allow us to combine double and single quotes in one string.

A tidier solution is to use escape sequences.

## Escape Sequences

In the previous section, we saw that using three single quotes allows us to combine single and double quotes in one string. Another approach is to use the `\` back-slash symbol. In Python, a back-slash is used to express special characters called escape sequences.

Escape Sequence	Usage
<code>\'</code>	To print single quote
<code>\"</code>	To print double quote
<code>\n</code>	To print a new line

Let us see how escape sequences works.

**Exercise #11: Using Escape Sequence**  
Launch Python Shell and type the following code.

```
>>> funnyString = "He asked, \"What's it going to take before you get it?\"" "
```

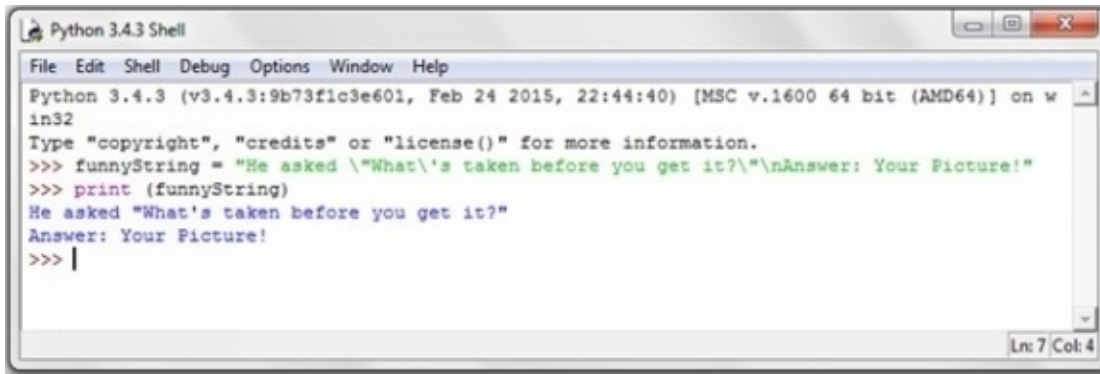
This time, we marked the quotes we want to print with a `\` before them. The program did not report any error. Whenever Python encounters the `\` character followed by a single or double quote, it ignores it until it finds the matching closing pair. This is called escaping. We can also use the `\n` character to start a new line.

The following exercise illustrates the use of `\n` new line escape sequence.

**Exercise #12: Using New Line Escape**  
Launch Python Shell and type the following code.

```
>>> funnyString = "He asked, \"What's it going to take before you get it?\" \nAnswer: Your picture!"
```

Figure 21



```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AMD64)] on w
in32
Type "copyright", "credits" or "license()" for more information.
>>> funnyString = "He asked \"What's taken before you get it?\"\\nAnswer: Your Picture!"
>>> print (funnyString)
He asked "What's taken before you get it?"
Answer: Your Picture!
>>> |
```

This time, *funnyString* contained the `\n` escape sequence in a string. When Python interpreter encounters `\` followed by `n`, it instantly knows that it has to start a new line when printing the string. Python supports a number of other escape sequences; feel free to practice using them on your own.



# String Mathematics

## String Concatenation

String concatenation is the combining of strings using the + operator. Let us try it.

### Exercise #13: Using String Concatenation

Launce Python Shell and type the following code. Guess the output for each part before proceeding!

#PART1

```
>>> thingsToDo1 = " appointment with vet"
>>> thingsToDo2 = "replace hiking joggers"
>>>thingsToDo3 = " call dentist"
>>> myToDoList = thingsToDo1 + thingsToDo2 + thingsToDo3
>>>print (myToDoList)
```

# PART 2

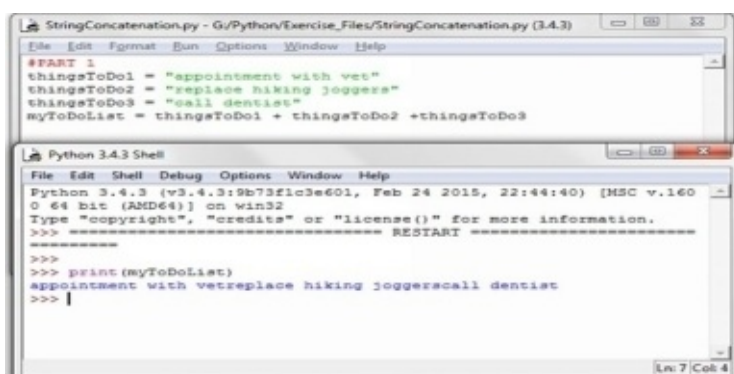
```
>>> myToDoList = thingsToDo1 + " , " + thingsToDo2 + " , " + thingsToDo3
```

# PART 3

```
>>> space = " , "
>>>>> myToDoList = thingsToDo1+ space + thingsToDo2+ space + thingsToDo3
```

Output: PART 1

Figure 22



Output: Part 2 and Part 3

Figure 23

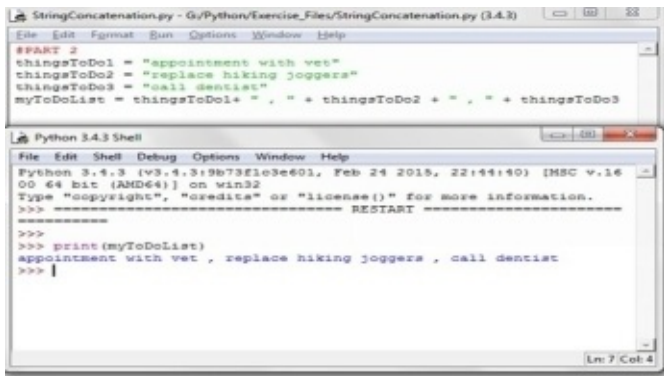
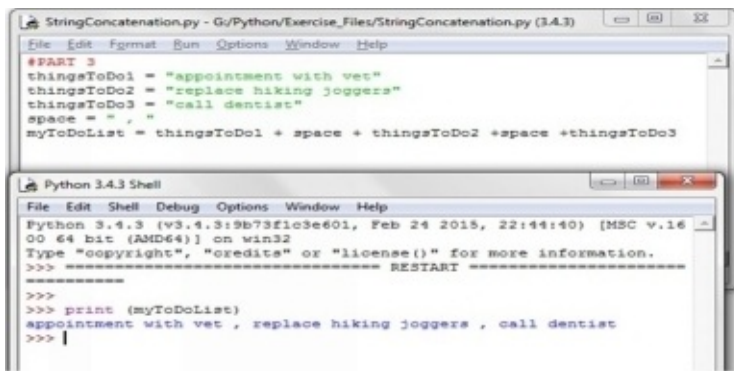


Figure 24



Straightforward, right? We took the contents of one string and combined them with that the other using the + operator. Here, space could also be an empty string with only spaces, but we used a comma to separate the individual strings instead.

#### Exercise #14: Using String Concatenation (B)

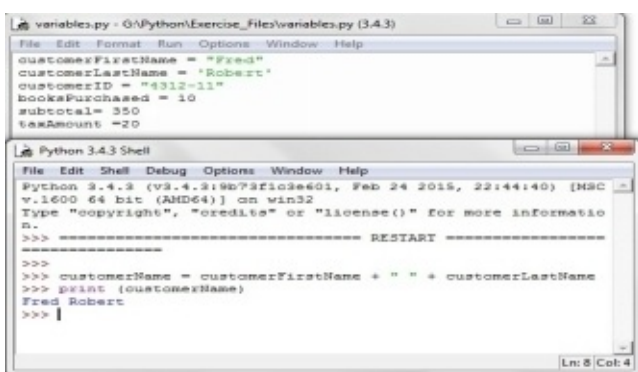
Open the file variables.py Run the Python Module to open Python shell and try to type the following code.

```

>>> customerName = customerFirstName + " " + customerLastName
>>> print(customerName)

```

Figure 25



Here, we used an empty “ ” string to insert a space in *customerName*.

## ***Multiplying Strings***

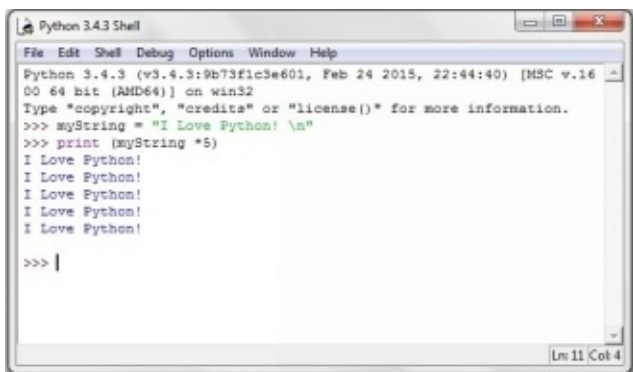
We can also perform multiplication on strings using the \* operator and it will repeat the string a certain number of times. Let us try it.

### **Exercise #15: Using Multiplying Strings**

Launch Python Shell, type the following code, and take note of how the escape character is used.

```
>>> myString = "I Love Python!\n"
>>> print (myString * 5)
```

Figure 26



Multiplying strings is also useful for formatting text.

### **Exercise #16: Using Multiplying Strings to Format Text**

Launch Python Shell, go to File -> New File, type the following code and save the file as **multiplyingString.py**

```
address1 = "Looney St, Brownwood"
address2 = "Texas, USA"
zipCode = "TX76802"
space = " " * 30
Message = "Greetings Sir,\nYour books have been dispatched at the given address.\nKind Regards,\nBookWorm Team."

print (space + address1)
print (space + address2)
print (space + "Post Code: " + zipCode)

print (Message)
```

Figure 27



The screenshot shows a Python IDE window titled "multiplyingString.py - G:/Python/Exercise\_Files/multiplyingString.py (3.4.3)". The code defines variables for an address, zip code, and a message, and then prints them. The status bar at the bottom right indicates "Ln: 4 Col: 16".

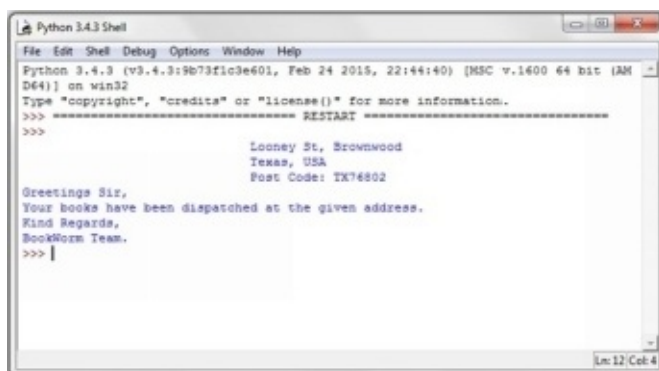
```
File Edit Format Run Options Window Help
address1 = "Looney St, Brownwood"
address2 = "Texas, USA"
zipCode = "TX76802"
space = " " * 30
Message = "Greetings Sir,\nYour books have been dispatched at the given address.\nKind Regards,\nBookWorm Team."

print (space + address1)
print (space + address2)
print (space + "Post Code: " + zipCode)

print (Message)
```

Ln: 4 Col: 16

Figure 28



The screenshot shows a Python 3.4.3 Shell window. It displays the output of the script, which includes the address, zip code, and a message. The status bar at the bottom right indicates "Ln: 12 Col: 4".

```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1400 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
                Looney St, Brownwood
                Texas, USA
                Post Code: TX76802

Greetings Sir,
Your books have been dispatched at the given address.
Kind Regards,
BookWorm Team.
>>> |
```

Ln: 12 Col: 4

## String as a Sequence

At the beginning of this chapter, we stated that a string is “a sequence of characters”. Let us refine that statement. String variables store alphanumeric value in a sequence where every character can be accessed by an integer. Recall the string type variable *customerFirstName*, which was assigned the value *FRED*. We can represent it graphically as:

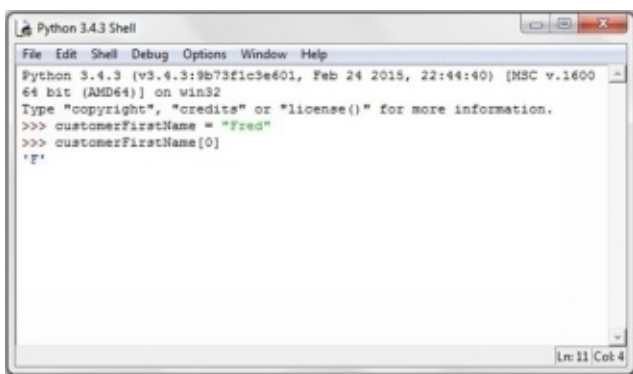
0	1	2	3
F	R	E	D

### Exercise #17: String as a Sequence

Launch Python Shell and type the following code.

```
>>> customerFirstName = "Fred"
>>> customerFirstName[0]
>>>
```

Figure 29



Note the following key points:

1. A string is a sequence of characters. We can access individual characters by pointing to their corresponding integers.
2. The first character is always stored at the “zero-th” index.
3. To access an individual character, enclose its corresponding integer in a pair of square brackets.

### ***Finding the Length of a String***

Python provides a number of built-in functions to assist us when working with strings. We

will discuss the concept of functions at length in later chapters; for now, we will learn to use the *len* (<*string name*>) function.

**Exercise # 18: Finding the Length of a String**  
Type the following into Python Shell:

```
>>> myFavoriteFood = "pizza"
>>> len (myFavoriteFood)
>>>
```

## Chapter Five

# User Input

Taking and working on user input are two of core features of any interactive program. In other words, these are some of the ways a programmer (or any other “user”) interacts with the running computer program. In this chapter, we will see how we can take user input and validate it.



## Taking User Input

Simply put, in taking user input, the program will ask the user to supply data. To do this, we must use the `input ()` function. Now, in you take user input, the data will naturally be stored in a variable. Let us see how this plays out in an actual code.

### Exercise #: 20 Taking User Input

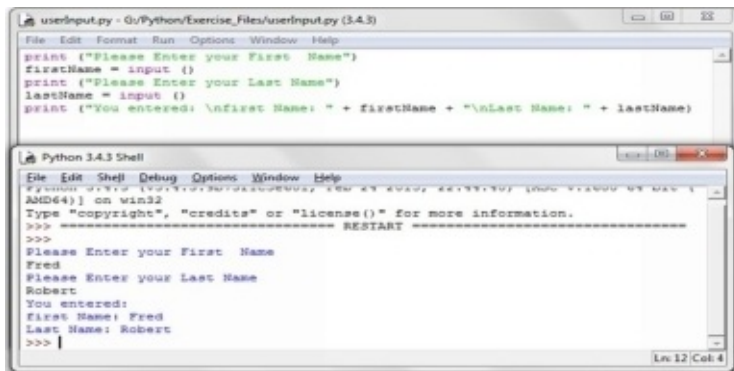
Launce Python Shell, go to File -> New File, type the below code, and save the file as `User_Input.py`

```
print ("Please Enter your First Name")
firstName = input ()

print ("Please Enter your Last Name")
lastName = input ()

print ("You entered: \n First Name: " + firstName + "\nLast Name: " + lastName )
```

Figure 30



## Formatting Strings with %s

When working with strings, there will be situations when you would need to insert variables within the string. This can be done using the following operators:

%s and %

%d and %

Note: %s is used to format type string, while %d is for type int

Essentially, %s is like telling Python to “insert string here,” then % specifies which string is needed. Let us see how this plays out.

### Exercise # 21: String Formatting with %s

Launce Python Shell, go to File -> New File, type the below code, and save the file as stringFormatting.py

```
print ("Please Enter your First Name")
firstName = input ()
print ("Please Enter your Last Name")
lastName = input ()
print ("Please Enter your Customer ID")
customerID = input ()

print (" You entered: ")
print ("First Name: %s " %firstName )
print ("Last Name: %s " %lastName )
print ("Customer ID: %s " %customerID )
```

Figure 31



Let us take a closer look at `print ("First Name: %s " %firstName)`.

1. First, we “took” a string from the user (user input) and stored it in a variable.
2. To print these values within a longer string, we placed the %s formatter at the point where we wanted to insert the variable.
3. We specified what variable was needed using the % operator.
4. In the above example, we wanted to place the variable *firstName* at the end of the string “FirstName: ”

We can also use the string formatter to insert multiple values using the same process. For now, however, let us enclose the values in parentheses—%(value1, value2, etc).

#### Exercise #: 22 Inserting Multiple Values

Launch Python Shell, go to File -> New File, type the below code, and save the file as weather.py

```
temperature = “17 degree Celsius”
```

```
precipitation = 0
```

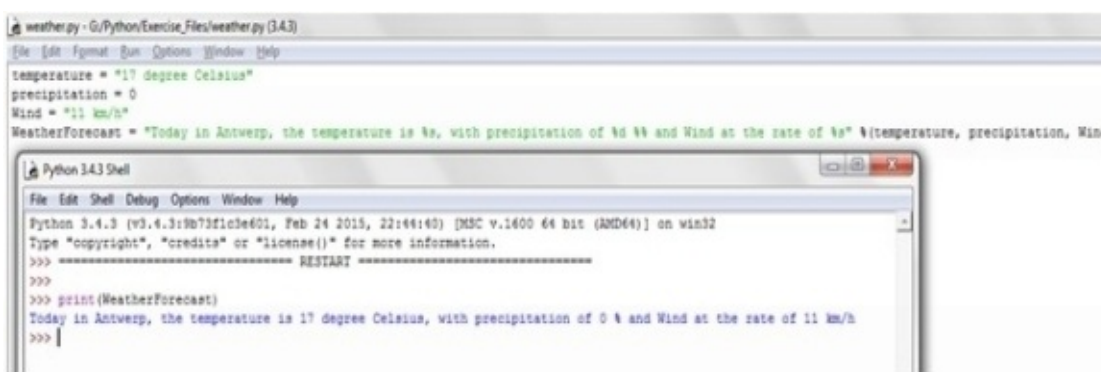
```
Wind = “11 km/h”
```

```
WeatherForecast = “Today in Antwerp, the temperature is %, with precipitation of %d %% and Wind at the rate of %s” %(temperature, precipitation, Wind)
```

**Run->Run Module**

**In Python shell type: print (WeatherForecast)**

Figure 32



Let us look at the above exercise line-by-line:

1. First, we defined the variables. Note that variable *precipitation* is type int, unlike *Wind* and *temperature*, which are type string.
2. We then defined the string to display. We pointed to locations for embedding values using the string formatter %s and number formatter %d. Also, note how we have used the % character to print a percentage symbol. Had we not done so, we would have received an error stating that the values to be embedded are less than the value

positions pointed at the string (recall escaping in the previous chapter).

3. Finally, we used the % operator to specify the variables we needed. Since we were embedding multiple variables, we had to enclose them in parentheses and list them in their appropriate sequence in the string.

## Prompting User

Previously, we used the `input ( )` function to input a value entered by the user, but the input question had to be printed first using the `print ( )` function. Alternatively, we can use the `input ( )` function to print the prompt question and simultaneously take the value the user enters. Here is how to do it: `<storage variable> = input (<prompt question enclosed in double quotes>)`

Let us do a quick exercise.

### Exercise # 23: Prompting User

Launch Python Shell, go to File -> New File, type the below code, and save the file as `promptingUser.py`

```
age = input ("How old are you? ")
print ( "you entered : %s" %age)
```

**Run -> Run Module**

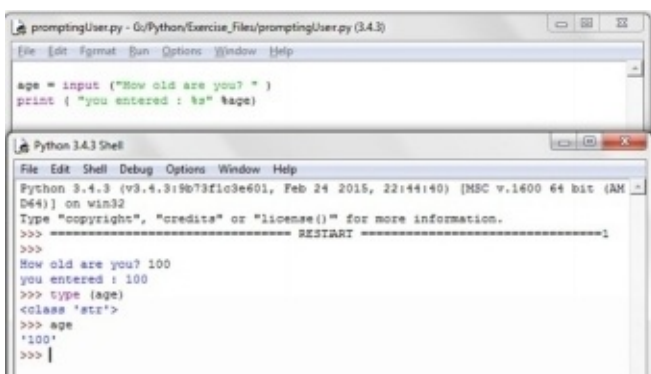
**Enter age**

```
>>>type (age)
```

```
>>> age
```

Note how the prompt question appears in the same line. If you check the type of `age` and print its value, you will see that it is type string. So, this `input` function works the same way, but using the `input` method with prompts makes the code shorter and more readable.

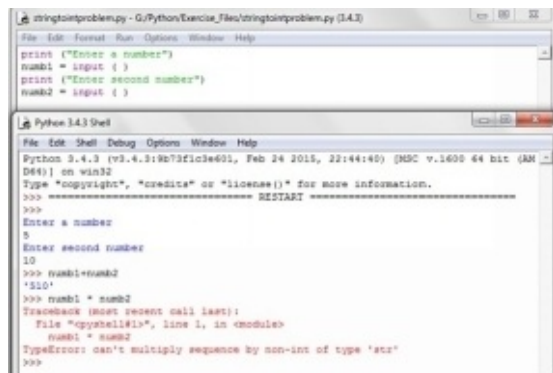
Figure 33



## String to Int Conversion

We know that the `input ( )` function returns a string variable. However, you will sometimes need the user to input numeric values. Now, if you input numeric values using `input ( )`, you will not be able to perform mathematical operations on them since the operations are defined only for numeric data types (see the example below).

Figure 34



To manipulate numeric values, we first need to convert string values to int, and this is how to do it: `int (<String variable>)`. The `int ( )` function takes a string parameter and converts it to int. Let us try it.

### Exercise #: 24 String to int

Launce Python Shell, go to File -> New File, type the below code, and save the file.

```
print ("Enter a number")
numb1 = input ( )
print ("Enter second number")
numb2 = input ( )
```

Run->Run Module, type the following in python shell

```
>>>Enter a number
>>>5
>>>Enter second number
>>>10
>>> numb_int = int(numb1)
>>> type (numb_int)
<class 'int'>
>>> numb_int
5
>>>
```

---

Here, we converted the string *numb1* using the *int ( )* function and stored it in variable *numb\_int*. We then checked the value and type of *numb\_int*, confirming that *int ( )* has indeed converted the string value to numeric. Note that *int ( )* only works on valid strings; if you try converting “fred” with *int ( )*, it will report an error.

## Validating User Input

Finally, when taking input, you need to validate if the user entered a correct data type. For example, if you want a user to input his birth year, you do not want to get a string that contains alphanumeric characters. Thankfully, easily validate the user's input using the following functions:

<string name>.isdigit( )

<string name>.isalpha( )

Note that both functions return Boolean true or false.

### Exercise #: 25 Checking Input String

Launce Python Shell, go to File -> New File, type the below code, and save the file.

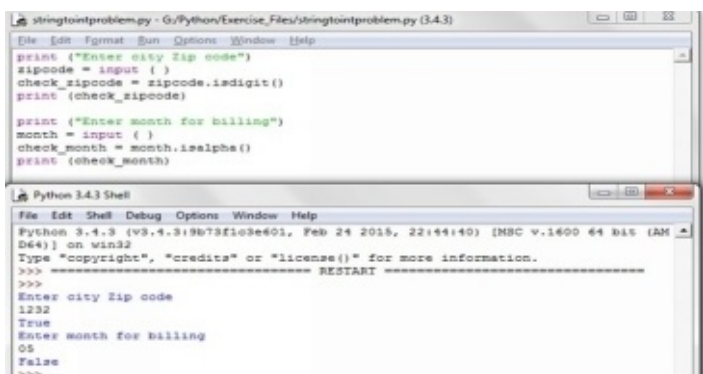
```
print ("Enter city Zip code")
zipcode = input ( )
check_zipcode = zipcode.isdigit()
print (check_zipcode)

print ("Enter month for billing")
month = input ( )
check_month = month.isalpha()
print (check_month)
```

### Run->Run Module

Enter inputs to check how the functions work.

Figure 35





## Chapter Six

# Lists

Every modern language utilizes some sort of data structure to combine data and process it. Two most common types of data structures used in Python are *Lists* and *Dictionaries*. We will focus on lists in this chapter.

A list is a sequence of data values called *items*. All items are placed in contiguous memory locations and can be accessed individually by index number. Like strings, lists have sequential data structures and their logical structures are comparable. However, while a string is a sequence of characters, we can use lists to store any kind of data item—and not necessarily of the same data. The index of the first item is zero.

# Creating and Manipulating Lists

## Defining Lists

You can use a set of square brackets to create a list, and each item within is separated by a comma— [*<listItem\_1>*, *<listItem\_2>*, *<listItem\_3>*, *<listItem\_4>*, *<listItem\_5>*]. To warm up, let us create a simple list of colors.

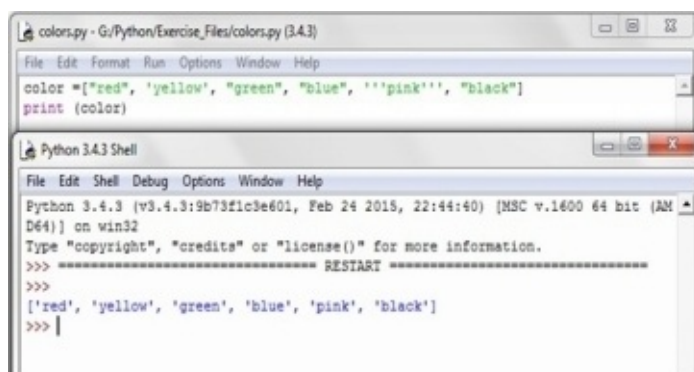
**Exercise # 26: Understanding Lists**  
Launce Python Shell, go to File -> New File, type the below code, and save the file as colors.py

color=["red", 'yellow', "green", "blue", "'pink'", "black"]

**Run-> Run Module**

>>>print (color)

Figure 36



Here is the indexing of the items in our list:

0	1	2	3	4	5
Red	Yellow	Green	Blue	pink	Black

Now, to make it a bit more complex, let us create a list of variables consisting of user-inputted values.

### Exercise # 27: Creating a List from User Input

Launch Python Shell, go to File -> New File, type the below code, and save the file.

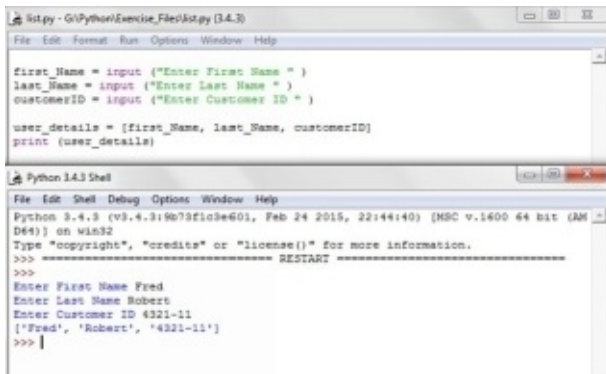
```
first_Name = input ("Enter First Name " )
last_Name = input ("Enter Last Name " )
customerID = input ("Enter Customer ID " )

user_details = [first_Name, last_Name, customerID]
print (user_details)
```

**Run->Run Module**

**Enter values when prompted.**

Figure 37



The screenshot shows two windows. The top window is a Python IDE with the following code:

```
first_Name = input ("Enter First Name " )
last_Name = input ("Enter Last Name " )
customerID = input ("Enter Customer ID " )

user_details = [first_Name, last_Name, customerID]
print (user_details)
```

The bottom window is a Python Shell showing the execution of the code. It prompts for input and displays the resulting list:

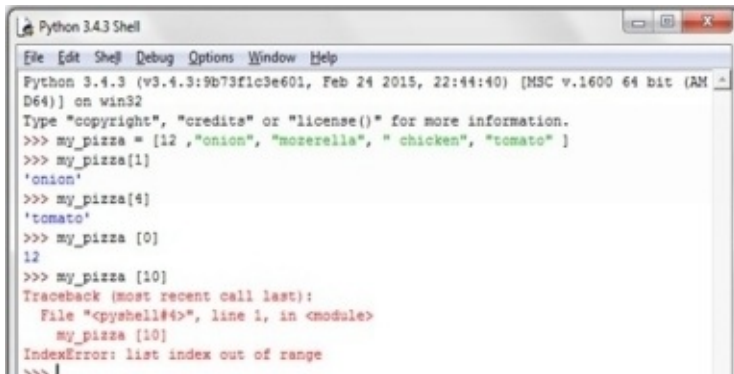
```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:987f103e601, Feb 24 2015, 22:44:40) [MSC v.1400 64 bit (AMD64)] on win32
Type "copyright", "credits() or "license()" for more information.
>>> ===== RESTART =====
>>>
Enter First Name Fred
Enter Last Name Robert
Enter Customer ID 4321-11
['Fred', 'Robert', '4321-11']
>>> |
```

### Accessing Item in a List

Like strings, you can access individual items in a list using their index numbers (remember, the first index is always 0), and here is how: `<list name> = [<index number>]`. Try it out for yourself. Launch Python Shell, create a list of arbitrary values, and try to access individual data items using their index numbers (like we did in Exercise 17 for strings).

Now, refer to the following sample code. Why do you suppose Python reported an error?

Figure 38

A screenshot of a Python 3.4.3 Shell window. The window has a menu bar with 'File', 'Edit', 'Shell', 'Debug', 'Options', 'Window', and 'Help'. The main text area shows the following code and output:

```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> my_pizza = [12, "onion", "mozzarella", "chicken", "tomato"]
>>> my_pizza[1]
'onion'
>>> my_pizza[4]
'tomato'
>>> my_pizza[0]
12
>>> my_pizza[10]
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    my_pizza[10]
IndexError: list index out of range
```

In the above example, we see that the list consists of five elements, which means the last valid index is 4. Apparently, `my_pizza [10]` is using an index that does not exist; hence, the error.

## List Slicing

Simply put, a *list slice* is a portion of a list specified by index. List slicing is done by using the code `<list name> [n:m]`. Now, in list slicing, note that the items from index *n* to index *m* will be selected, but the *m*th item will be excluded when printed. Let us see how it works.

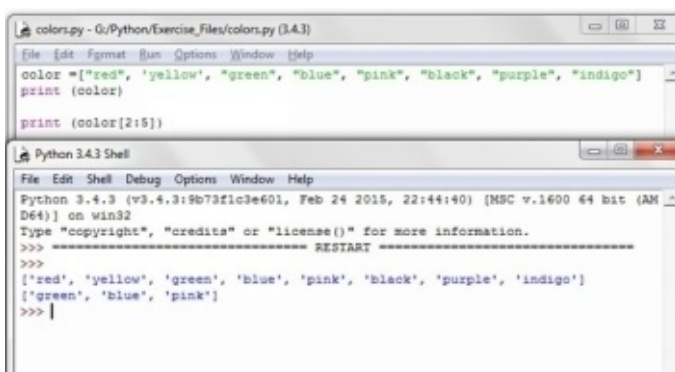
### Exercise #28

Open `colors.py` from Exercise 26 and edit the code as follows:

```
color=["red", 'yellow', "green", "blue", "pink", "black", "purple", "indigo"]  
print (color)  
print (color[2:5])
```

Run->Run Module

Figure 39



The below figure represents the indexing of the items in Exercise 28, as well its list slicing:



By the way, you can also store the list slice in another list. Let us try it out.

### Exercise #29

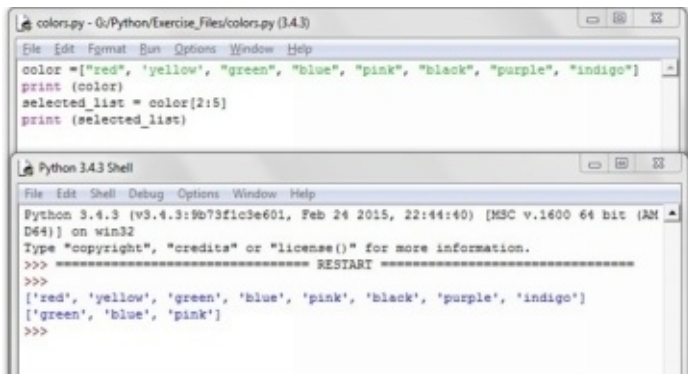
Open `colors.py` from Exercise 26 and edit the code as follows:

```
color=["red", 'yellow', "green", "blue", "pink", "black", "purple", "indigo"]
```

```
print (color)
selected_list = color[2:5]
print (selected_list)
```

**Run->Run Module**

Figure 40



The screenshot displays two windows from a Python IDE. The top window, titled 'color.py - G:\Python\Exercise\_Files\color.py (3.4.3)', contains the following code:

```
color = ["red", "yellow", "green", "blue", "pink", "black", "purple", "indigo"]
print (color)
selected_list = color[2:5]
print (selected_list)
```

The bottom window, titled 'Python 3.4.3 Shell', shows the output of the script:

```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
['red', 'yellow', 'green', 'blue', 'pink', 'black', 'purple', 'indigo']
['green', 'blue', 'pink']
>>>
```

## Getting Information About Lists

Python provides a number of built-in functions to get information about a list. This section discusses the most commonly used functions.

### *Finding the length of list*

To find the total number of data items stored in a list, use the *len ( )* function

`len ( <list_name> )`

The `len ( )` function returns an integer value.

#### Exercise #30

Open `colors.py` from exercise #26 and edit the code file as follows:

```
color=["red", 'yellow', "green", "blue", "pink", "black", "purple", "indigo"]
print (color)
selected_list = color[2:5]
print (selected_list)
# finding the length of lists
number_of_selected_colors = len (selected_list)
print (number_of_selected_colors)
print ( len (color))
```

**Run->Run Module**

### *Repeating items in a list*

To find out the number of times an item appears in list use the count function `count`.

`<list_name>.count (" item_to_count")`

#### Exercise #31

Open `colors.py` from exercise #26 and edit the code file as follows:

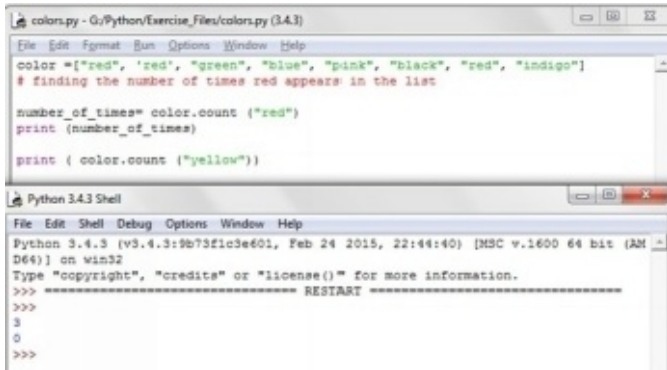
```
color=["red", 'red', "green", "blue", "pink", "black", "red", "indigo"]

number_of_times= color.count ("red")
print (number_of_times)
print ( color.count ("yellow"))
```

**Run->Run Module**



Figure 41



The screenshot shows a Python IDE with two windows. The top window, titled 'colors.py - G:\Python\Exercise\_Files\colors.py (3.4.3)', contains the following code:

```
color = ["red", 'red', "green", "blue", "pink", "black", "red", "indigo"]
# finding the number of times red appears in the list

number_of_times= color.count ("red")
print (number_of_times)

print ( color.count ("yellow"))
```

The bottom window, titled 'Python 3.4.3 Shell', shows the execution output:

```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
3
0
>>>
```

Note that the count function returns 0 as “yellow” does not exist in the list.

### ***Check if in list***

You may check for a certain item if it exists in a list by using the **in** command.

<item\_to\_check> **in** <list\_name>

**in** returns a Boolean true if the item to check is present in the list. If absent, the function returns false.

#### **Exercise #32**

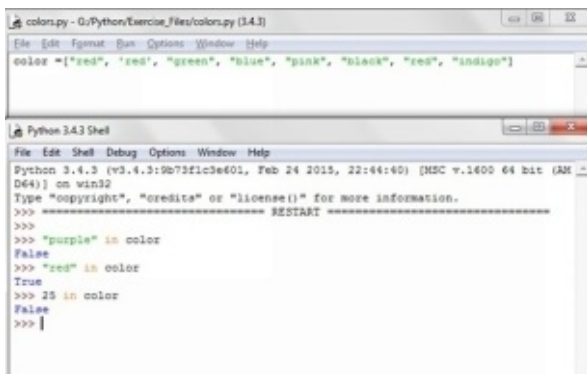
Open colors.py from exercise #26 and edit the code file as follows:

```
color=["red", 'red', "green", "blue", "pink", "black", "red", "indigo"]
# finding purple is in list
```

#### **Run->Run Module**

```
>>>"purple" in color
>>>
>>> "red" in color
>>>
>>> 25 in color
>>>
```

Figure 42



## *Finding Index of a Item*

Once you know that an item is present in a list you may find the index at which the item is present in the list using:

`<list_name>.index ( item )`

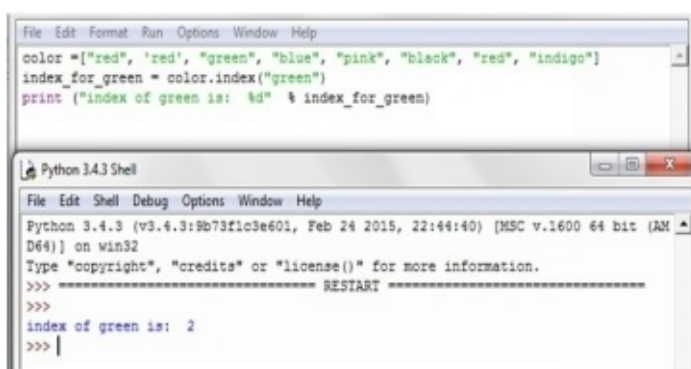
### Exercise #33

Open `colors.py` from exercise #26 and edit the code file as follows:

```
color=["red", "red", "green", "blue", "pink", "black", "red", "indigo"]
index_for_green = color.index("green")
print ("index of green is: %d" % index_for_green)
```

**Run->Run Module**

Figure 43



## Adding items to list

### *Append*

You can add items at the end of the list or append the items to a list using the append method as follows:

```
<list_name>.append (item)
```

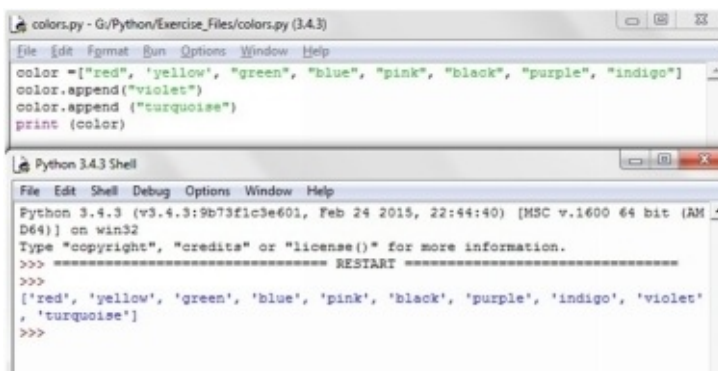
#### Exercise #34: Appending Values

Open colors.py from exercise #26 and edit the code file as follows:

```
color=["red", 'yellow', "green", "blue", "pink", "black", "purple", "indigo"]
color.append("violet")
color.append("turquoise")
print (color)
```

**Run->Run Module**

Figure 44



### *Extend*

The extend method appends a list at the end of another list.

```
list_1 = [1,2,3]
list_2 = [4,5]
list_1.extend(list_2)
list_1 = [1,2,3,4,5]
list_2 = [4,5]
```

#### Exercise #35

Open IDLE text editor and write the following code. Save the file as order.py

```
order_books1 = ["Black Beauty","The Hobbit","The Alchemist","the Little Prince"]
```

```
order_books2 = ['War and Peace','Heidi']
```

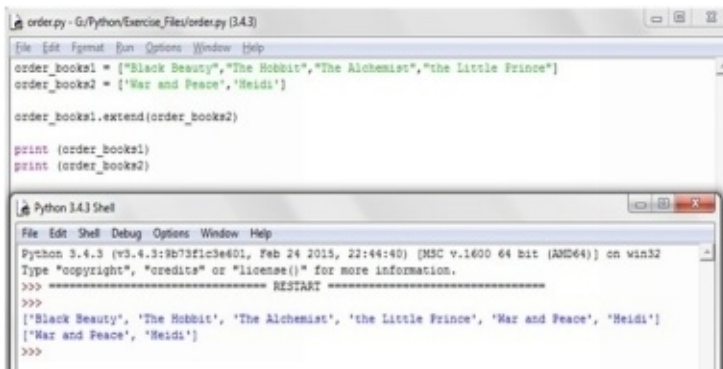
```
order_books1.extend(order_books2)
```

```
print (order_books1)
```

```
print (order_books2)
```

**Run->Run Module**

Figure 45



Note that the extend method does not return any value. Therefore if you try to do something like:

```
Final_order = order_books1.extend(order_books2)
```

It will report an error.

**Exercise # 36: Appending user input in list**  
Open IDLE the file list.py and edit as follows

```
user_details = [ ]
```

```
user_details.append(input ("Enter First Name " ))
```

```
user_details.append(input ("Enter Last Name " ))
```

```
user_details.append(input ("Enter Customer ID "))
```

```
print (user_details)
```

**Run->Run Module**

**Enter values when prompted.**

In the above example we first define an empty list called user\_details. Then we append

values in the list as the user inputs them.

## ***Insert***

You may also insert an item at a particular index using the insert method as shown below:

```
<list_name>.insert (index, value)
```

The insert function inserts the item at the given index in the list called list\_name. If the index is less than the length of list, the item will be added at the end of list.

All the other data items in the list are moved one step up to make room for the new data item.

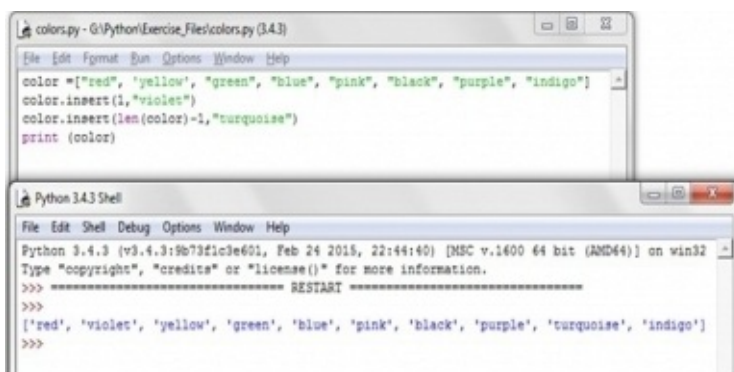
### **Exercise #37 Inserting values**

Open colors.py from exercise #26 and edit the code file as follows:

```
color=["red", 'yellow', "green", "blue", "pink", "black", "purple", "indigo"]  
#example 1  
color.insert(1,"violet")  
  
#example 2  
color.insert(len(color)-1,"turquoise")  
print (color)
```

**Run->Run Module**

Figure 46



In example 1, we are inserting string “violet” at the index 1. For example 2 we add string “turquoise” in the list; however this time the index is not a fixed integer. The index is given by len (color) -1. Here, the insert function will first computes the index and then insert the value at the computed index.



Investigate the code below which is just another implementation of the same exercise.

Figure 47

```
color.py - G:\Python\Exercise_Files\color.py (34.8)
File Edit Format Run Options Window Help
color = ["red", "yellow", "green", "blue", "pink", "black", "purple", "indigo"]
print (len(color))
color.insert(1, "violet")
print (color)
print (len(color))
color.insert(len(color)-1, "turquoise")
print (color)
```

```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====>>>
0
['red', 'violet', 'yellow', 'green', 'blue', 'pink', 'black', 'purple', 'indigo']
9
['red', 'violet', 'yellow', 'green', 'blue', 'pink', 'black', 'purple', 'turquoise', 'indigo']
>>> |
```

## Manipulating value of List

We have already pointed out that both strings and lists are sequential data types. However there is a noteworthy difference between the two data types. Strings are not mutable while lists are mutable. That is, when working with lists you can replace individual items of the list by the subscript operator which is not permissible with strings.

<list\_name> [<index\_number>] = <new value>

e.g.

color [0] = "yellow"

this replaces whatever was stored at index 0 of the list with "yellow"

### Exercise #38

Open order.py file from the exercise 35 and edit the code as follows"

```
order_books1 = ["Black Beauty", "The Hobbit", "The Alchemist", "the Little Prince"]
```

```
#print the list before
```

```
print (order_books1)
```

```
order_books1[1] = 'War and Peace'
```

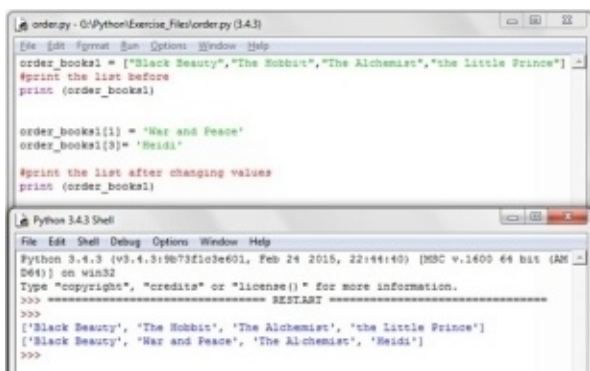
```
order_books1[3]= 'Heidi'
```

```
#print the list after changing values
```

```
print (order_books1)
```

**Run->Run Module**

Figure 48



## Removing items from the list

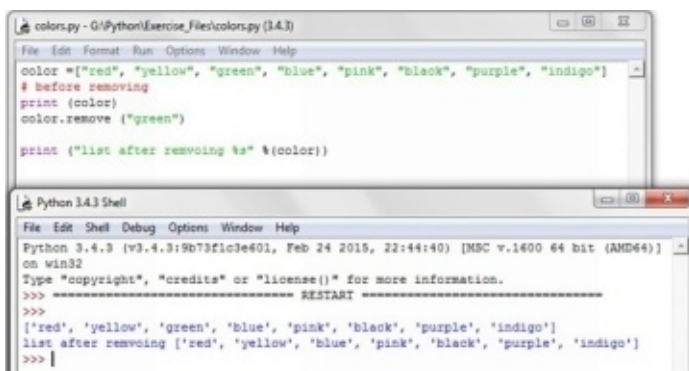
### *Remove*

If you want to remove a specific item in the list you can use the remove method.

#### Exercise #39: Removing items from list

```
color=["red", "yellow", "green", "blue", "pink", "black", "purple", "indigo"]  
# before removing  
print (color)  
color.remove ("green")  
  
print ("list after removing %s" %(color))
```

Figure 49



### *Delete*

If you want to remove an item on a particular index use the **del** command as follows:

```
del <list_name>[index]
```



this will delete the item at the provided index

#### Exercise #40: Deleting item at a given index

Open IDLE text editor and write the following code. Save the file

```
1. color=["red", "yellow", "green", "blue", "pink", "black", "purple", "indigo"]
2. print (color)

3.
4. index = input ("Enter index number for the item you want to delete: ")
   print ("The item at index %s is : %s" % (index, color [int(index)]))

5.
6. print ("Deleting %s..." %color [int(index)] )
7. del color [int(index)]
   print ("List is now: %s " %(color))
```

**Run->Run module**

**Enter the index for the item you want to delete.**

Let's investigate the exercise line by line:

1. In line 1 we define a list called colors.
2. In line 2 we print the contents of the list
3. At line 3 we prompt the user to enter the index of item he/she wants to remove from the list. We have stored the user input in a variable called index. Recall that the method input returns a string value so the variable index is of string type.
4. We then print the item at the index provided by the user. Here we use double string formatters to insert
  - The index submitted by the user.
  - The item stored at the provided index. Recall to access the item at an index x we use <list\_name>[x]. For the color list we will use color [index]; but wait!! Index is of string type therefore we first need to convert it to an integer using the function int (index).
5. Line 5 prints the item to be deleted.
6. Use the del method to delete
7. Print the remaining list.

Figure 50

```
color.py - C:\Python\Exercise_files\color.py (14.1)
File Edit Format Run Options Window Help
color = ["red", "yellow", "green", "blue", "pink", "black", "purple", "indigo"]
print (color)
index = input ("Enter index number for the item you want to delete: ")
print ("The item at index %s is : %s" % (index, color [int(index)]))
print ("Deleting %s..." % color [int(index)] )
del color [int(index)]
print ("List is now: %s" % (color))

Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73f20be601, Feb 24 2015, 22:44:40) [MSC v.1400 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
['red', 'yellow', 'green', 'blue', 'pink', 'black', 'purple', 'indigo']
Enter index number for the item you want to delete: 5
The item at index 5 is : black
Deleting black...
List is now: ['red', 'yellow', 'green', 'blue', 'pink', 'purple', 'indigo']
>>> |
```

# Pop

Pop method can be used in two ways:

1. `<list_name>. pop ( )` : this will delete and return the last item of the list.
2. `<list_name>. pop ( index )`: this will delete the item at the given index and return the element removed.

## Exercise #41: Deleting item using the POP method

Open IDLE text editor and write the following code. Save the file

```
1. color=["red", "yellow", "green", "blue", "pink", "black", "purple", "indigo"]
2. print (color)

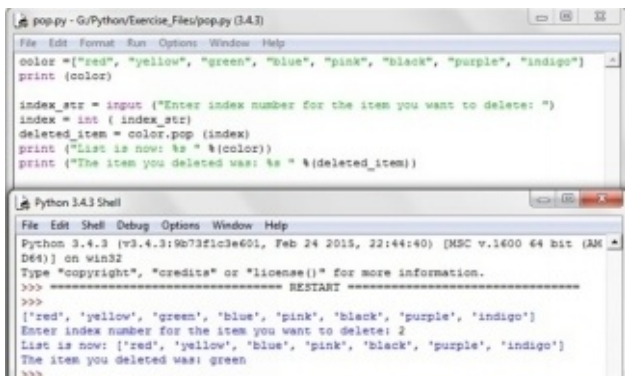
3.
4. index_str = input ("Enter index number for the item you want to delete: ")
   index = int ( index_str)

5. deleted_item = color.pop (index)
6. print ("List is now: %s " %(color))
   print ("The item you deleted was: %s " %(deleted_item))
```

**Run->Run module**

**Enter the index for the item you want to delete.**

Figure 51



In the above example you can see that the variable `deleted_item` stores the value *pop-ed* which we print in line 6.

## Mathematical operations on lists

We can perform mathematical operations on the lists such as addition and multiplication.

### *Adding lists*

You can combine lists using the addition (+) operator.

```
<list_name> + <list_name>
```

Note that there is a significant difference between adding two lists and using the extend method on lists. The extend( ) method does not return anything. It simply extends one list. In other words extend() method changes the lists in use. On the other hand the + operator combines the lists and resultant list can be stored as an independent list.

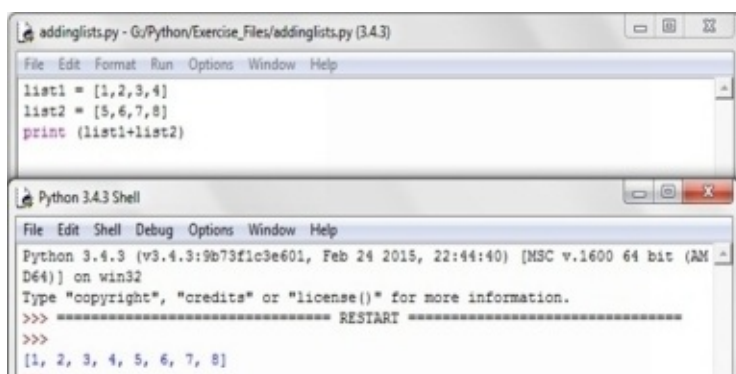
#### Exercise # 42: Adding Lists

Open IDLE Text editor and write the following code. Save as addinglists.py

```
list1 = [1,2,3,4]
list2 = [5,6,7,8]
print (list1+list2)
```

**Run-Run Module**

Figure 52



#### Exercise # 43: Adding Lists

Edit the addinglists.py as follows

```
list1 = [1,2,3,4]
```

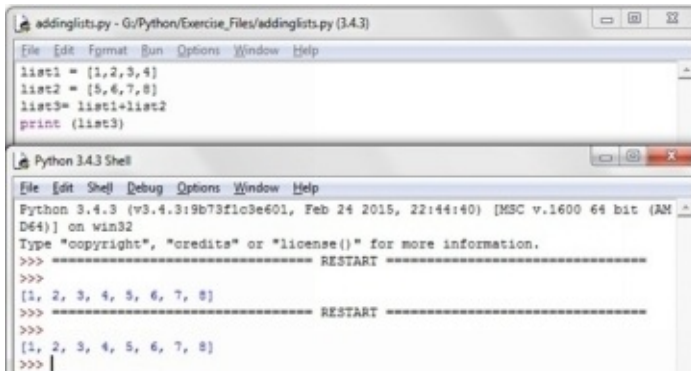
```
list2 = [5,6,7,8]
```

```
list3= list1+list2
```

```
print (list3)
```

**Run-Run Module**

Figure 53



## Multiplying list

You can multiply a list by an integer to repeat the content of the list a certain number of times. You can multiply content of list using the \* operator.

### Exercise # 44: Adding Lists

Open IDLE Text editor and write the following code. Save as addinglists.py

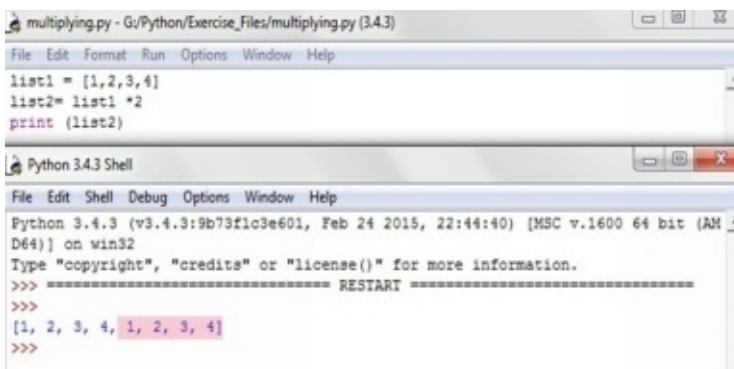
```
list1 = [1,2,3,4]
```

```
list2= list1 *2
```

```
print (list2)
```

**Run-Run Module**

Figure 54



## Data Structure Methods

### *Comparing lists*

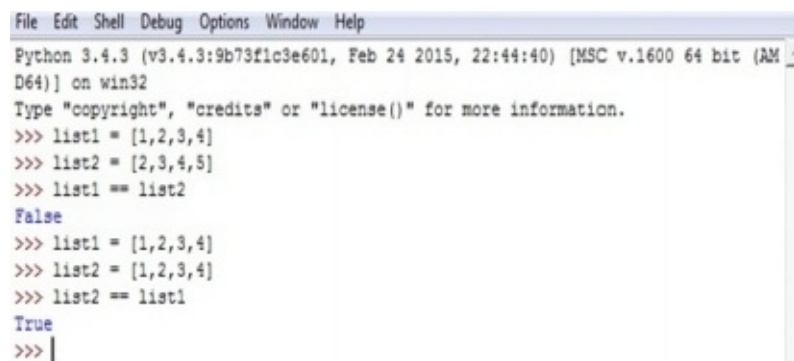
You can compare two lists to verify if they contain similar content using the comparison operator `==`. Note that the comparison operator is different than the assignment operator `=`. The comparison operator to compare lists returns true if the lists are equal and false when the lists are unequal.

#### Exercise #45: Comparing two lists

Open the Python Shell and type:

```
>>> list1= [1,2,3,4]
>>> list2= [2,3,4,5,]
>>>list1 == list2
>>>
>>> list1= [1,2,3,4]
>>> list2= [1,2,3,4]
>>>list2 == list1
>>>
```

Figure 55



The screenshot shows a Python Shell window with the following content:

```
File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> list1 = [1,2,3,4]
>>> list2 = [2,3,4,5]
>>> list1 == list2
False
>>> list1 = [1,2,3,4]
>>> list2 = [1,2,3,4]
>>> list2 == list1
True
>>> |
```

### *Sorting a list*

You can use the `sort( )` method on list containing string to arrange the contents of the list in alphabetic order or on a list containing numeric data items to sort the numbers in ascending order.

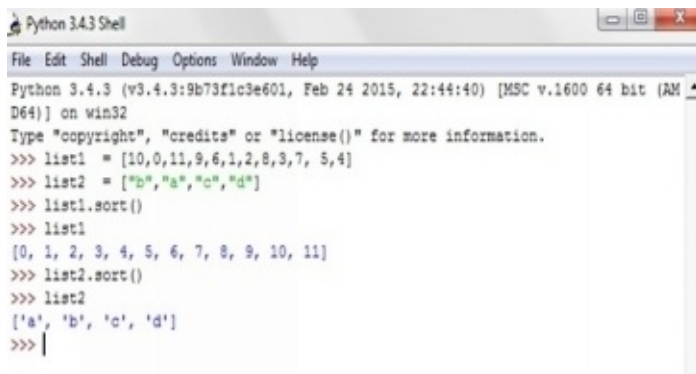
`<list_name>.sort( )`

#### Exercise #46: Comparing two lists

Open the Python Shell and type:

```
>>> list1= [10,0,11,9,6,1,2,8,3,7, 5,4]
>>> list2= ["b", "a", "c", "d"]
>>>
>>> list1.sort()
>>>list2.sort ()
>>> print (list1)
>>>print (list2)
>>>
```

Figure 56



```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> list1 = [10,0,11,9,6,1,2,8,3,7, 5,4]
>>> list2 = ["b","a","c","d"]
>>> list1.sort()
>>> list1
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
>>> list2.sort()
>>> list2
['a', 'b', 'c', 'd']
>>> |
```

## Chapter Seven



# Dictionaries

Lists organize data items in sequential position where each item can be accessed by its index number. Dictionaries also group items however they differ with Lists as they group items by associations. You may regard dictionaries as general container which allows you to access values using keys. Each item holds a unique key. Since each key for each item is unique search time faster.

## What are Dictionaries?

To understand the concept of dictionaries think of a real-life dictionaries. Each entry in the dictionary is a pair of a word and its meaning. When working with the lists we pair index and values; while for dictionaries we pair a key and a value. Like Lists, dictionaries are mutable. In contrast to lists, Index in lists is integer; whereas for dictionaries, keys can be of any data type. You can think of dictionaries as association lists or tables.

Creating dictionaries is similar to creating lists; but now instead of index-value we create key-value pairs. Note a few points about dictionaries:

- **Key data type:** key can be of any data type. Key can be numbers, strings or even complex data types such as objects.
- All the keys for a given dictionary must be of same data type.
- **Key is unique:** For every key-value pair, the key must be unique i.e. there can only be one key of its kind in the entire dictionary. This makes sense. Since we use keys to access values, having two same keys will cause conflict.
- Keys cannot change i.e. they are immutable.

Consider some typical examples of dictionaries:

Dictionary for Student and their ID card #

Sam	Qasim	Alexander	Steffen
1211	7860	1222	2144

Steffen	} Key value pair
2144	

Steffen	→ Key
2144	→ Value

Example #2: Dictionary which stores names of various states in the USA and their capital.

New Jersey	Massachusetts	New York	Illinois	Ohio
Trenton	Boston	Albany	Springfield	Columbus

Example #3: A Dictionary which saves a client's record

--	--	--	--	--

Name	ID-card-number	Location	Age	Date-of-Birth
Sam	1422	Albany	18	12-01-97

## Creating dictionaries & Accessing Values

Having discussed the structure of dictionaries let's create one. You create a dictionary using a set of curly brackets { }.

```
myDictionary = { }
```

#we just created an empty dictionary

```
myDictionary = { "NY": "New York", "OR": "Portland" }
```

In the above example we created a dictionary with two items. NY and New York are key-value pairs where NY is the key and New York is the value. Both key and the value are separated with colon. OR and Portland is the second key-value pair. Where OR is the key and Portland is the value. Both the pairs are separated with a comma.

### Exercise #47: Creating Dictionaries & Accessing Values with Keys

Open the IDLE Text editor and create a new file with the following code. Save as dictionary.py

```
>>> books = { 123: " Black Beauty", 345: "The Hobbit", 678: "The Alchemist", 91011:"the Little Prince" }
```

**Run->Run module**

**In the python shell type:**

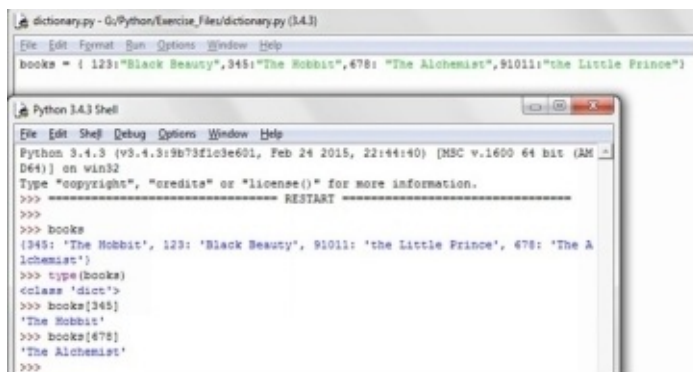
```
1. >>> books
2. >>>
3. >>>type (books)
4. >>>
5. >>>books[345]
6. >>>
7. >>>books[678]
```

Let's review the exercise line by line:

- For the first part of exercise we create a dictionary using the { } . Here the integer keys represent ISBN of books and the values represent the title of the books.
- With the *type(books)* function we query Python to return the type of *books*. Python shell responds with <class 'dict'> which confirms that we have created a dictionary called books.
- We then use the key to access the value corresponding to that key using the [ ] square brackets. When queried books[345], the shell responds with "The Hobbit". The Hobbit is the value corresponding to the key 345. Likewise we use the key 678 to access value

corresponding to 678 key i.e. Alchemist

Figure 57



The image shows a screenshot of a Python 3.4.3 IDE. The top window, titled 'dictionary.py - G:\Python\Exercise\_files\dictionary.py (3.4.3)', contains the following code:

```
books = { 123:"Black Beauty",345:"The Hobbit",678: "The Alchemist",91011:"the Little Prince"}
```

The bottom window, titled 'Python 3.4.3 Shell', shows the interactive prompt with the following commands and output:

```
>>> books
{345: 'The Hobbit', 123: 'Black Beauty', 91011: 'the Little Prince', 678: 'The A
lchemist'}
>>> type(books)
<class 'dict'>
>>> books[345]
'The Hobbit'
>>> books[678]
'The Alchemist'
>>>
```

## Adding and Updating Values

To add a pair of key and value in your dictionary use:

```
<dictionary_name> [key] = value
```

Do not forget to enclose the key/value in quotes if their data type is string.

### Exercise #48: Adding values in Dictionary

Open Dictionary.py. Run->Run Module. In python shell type:

```
>>> books [111] = "The Tale of two cities"  
>>> books[2222] = "The Ginger Man"  
>>> books[333] = "The Thorn Birds"  
  
>>>books  
>>>  
>>>
```

In the above exercise we have added three new key-value pairs in the dictionary. The dictionary has the following structure; the new items are depicted with blue color for distinction.

91011	'the Little Prince'
678	'The Alchemist'
345	'The Hobbit'
123	'Black Beauty'
111	'The tale of two cities'
2222	'The Ginger Man'
333	'The Thorn Bird'


Can you predict the behavior of the dictionary if we used the following code to add value to dictionary?

```
Books[123] = "Lolita"
```

Unlike when adding key-value pairs previously, now the key 123 is already present in the case under consideration. As pointed above we cannot have duplicate keys. Therefore when we try to add a value with a key which already exists in the dictionary; the old value

is replaced by the new value.

91011	'the Little Prince'
678	'The Alchemist'
345	'The Hobbit'
123	Lolita'
111	'The tale of two cities
2222	'The Ginger Man'
333	'The Thorn Bird'



#### Exercise #49: Updating Dictionaries

Open the IDLE Text editor and create a new file with the following code. Save as dictionary\_email.py

```
customer_email = { "Sam": "sam85@yahoo.com", "Jeff": "jeff_smith@hotmail.com",  
"Holly": "holly_Flin@outook.com", "Edmund": "thePrince@gmail.com" }
```

**Run->Run module**

**In the python shell type:**

```
1.      >>> customer_email  
2.      >>>  
3.      >>> customer_email["sam"] = "sam86@yahoo.com"  
4.      >>>  
5.      >>> customer_email
```

In the above example we update the email address for the person named "Sam".

## Removing a key Value Pair

We can have a key for which the corresponding value is 0 or an empty string. But there are situations when you would like to remove the key altogether. To remove key-value pair we use the `pop()` method, which we have already used for lists. The method removes the key and returns the value for the key.

`<dictionary_name>.pop(<key>)`

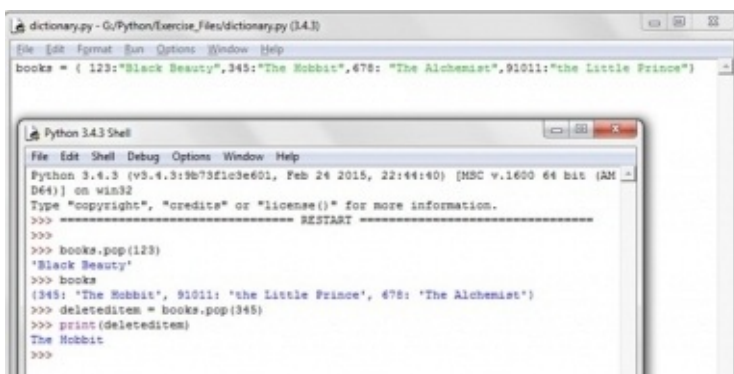
### Exercise #50: Deleting key and associated value using POP

Open Dictionary.py. Run->Run Module. In python shell type:

```
1. >>> books.pop(111)
2. >>>
3. >>> books
4. >>> deleteditem = books.pop(2222)
5.
6. >>> print(deleteditem)
>>> books
>>>
```

In the above example we have deleted the key-value pair for the key 111. You will see that the `pop` command prints the value for key 111; in other word the `pop` command returns the value for key 111 i.e. Black Beauty. In line 3 we print the contents of `books` dictionary to see the remaining items.

Figure 58





## Getting information about Dictionaries

### *Check if a key exists in dictionary*

To check whether a key is already present in dictionary or not use the **in** command like follows:

(key) **in** <dictionary\_name>

Note that: <dictionary\_name>.**has\_key**(key) method has been removed in python 3. Old books and tutorials use the above stated method.

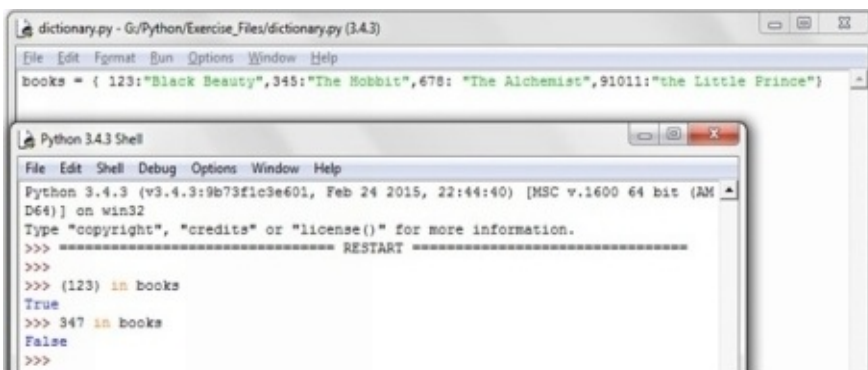
The method returns a boolean true or false.

Exercise #51: Check if a key is present in dictionary

Open Dictionary.py. Run->Run Module. In python shell type:

```
1.      >>> (123) in books
2.      >>>
3.      >>> 347 in books
4.      >>>
```

Figure 59



### *Listing Keys*

You may come across a situation where you want to list all the keys used in a dictionary. To list all the keys in a dictionary use the following method.

<dictionary\_name>.keys( )

### Exercise #52: Listing Keys in Dictionary

Open Dictionary.py. Run->Run Module. In python shell type:

```
1. >>> books.keys( )
2. >>>
```

## Listing Values

To list values in a dictionary, python provides the **value( )** method.

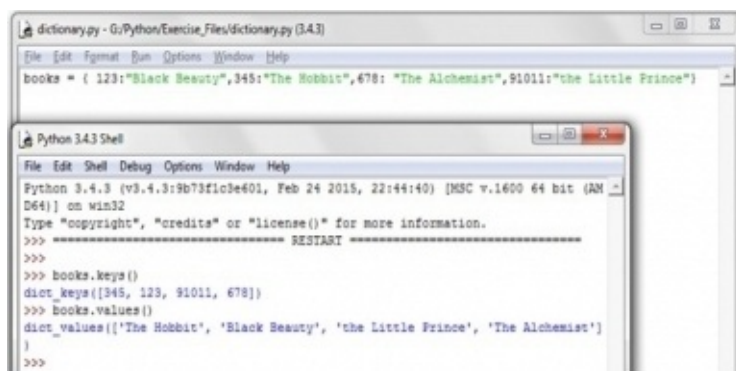
<dictionary\_name>.values( )

### Exercise #53: Listing Values in dictionary

Open Dictionary.py. Run->Run Module. In python shell type:

```
1. >>> books.values( )
2. >>>
```

Figure 60



Now as we have access to all the values in a dictionary with the value function, let's try to

store dictionary values.

#### Exercise #54: Dictionary Views

Open Dictionary.py. Run->Run Module. In python shell type:

```
1. >>> mybooklist = books.values()  
2. >>> mybooklist [0]
```

Unfortunately python will report an error. This would have worked in Python 2, however in the newer version of Python the method **dictionary.value ( )** returns a *view* of dictionary values instead of a list. Notice in the output, the list of values is wrapped in **dict\_values ( )**. To extract the list from the view we will use the **list ( )** function. Let's investigate the use of **list( )** method:

### Approach 1:

#### Exercise #55: Making list of dictionary values

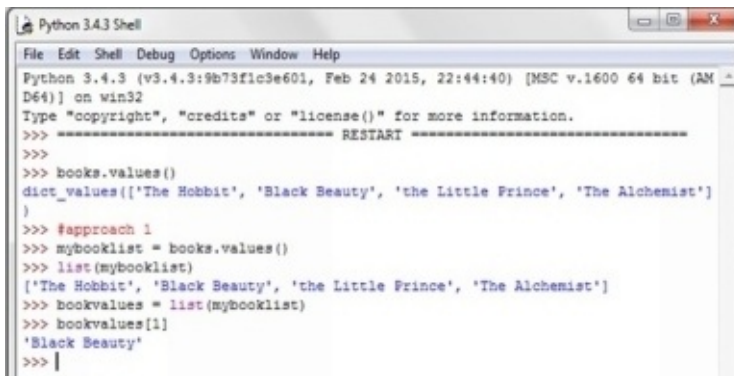
Open Dictionary.py. Run->Run Module. In python shell type:

```
1. >>> mybooklist = books.values()  
2. >>> list(mybooklist)  
3. >>>  
4. >>>  
5. >>>  
6. >>> bookvalues= list(mybooklist)  
   >>> bookvalues[1]  
   >>>
```

In this exercise we aim to extract the values of dictionary in a list.

1. In line 1, we use the **dictionary.values( )** function to extract dictionary values. However this is not a list, it's merely a view.
2. In line 2, we use the built in **list ( )** method to extract the list from the view.
3. At line 3, the python shell will print the list.
4. In line 4 we define a new variable to hold the list **bookvalues**. We assign the list from line 2 to **bookvalues** variable.
5. In line 5 we use index to access the value in the list.

Figure 61



```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
>>> books.values()
dict_values(['The Hobbit', 'Black Beauty', 'the Little Prince', 'The Alchemist'])
>>> #approach 1
>>> mybooklist = books.values()
>>> list(mybooklist)
['The Hobbit', 'Black Beauty', 'the Little Prince', 'The Alchemist']
>>> bookvalues = list(mybooklist)
>>> bookvalues[1]
'Black Beauty'
>>> |
```

## Approach 2:

In the previous example we created two variables;

- mybooklist: to store the dictionary values in a view
- bookvalues: to store the list from the view

Creating extraneous variables is inefficient. We can re-write the code in a succinct way:

`list(dictionary.values ( ))`

### Making list of dictionary values

Open Dictionary.py. Run->Run Module. In python shell type:

```
1. >>> newbookvalues = list(books.values ( ))
2.
3. >>> newbookvalues
4.
5. >>> # prints the list
>>> newbookvalues [1]
>>>
```

Figure 62



```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
>>> books.values()
dict_values(['The Hobbit', 'Black Beauty', 'the Little Prince', 'The Alchemist'])
>>> #approach 1
>>> mybooklist = books.values()
>>> list(mybooklist)
['The Hobbit', 'Black Beauty', 'the Little Prince', 'The Alchemist']
>>> bookvalues = list(mybooklist)
>>> bookvalues[1]
'Black Beauty'
>>> # approach 2
>>> newbookvalues = list(books.values())
>>> newbookvalues
['The Hobbit', 'Black Beauty', 'the Little Prince', 'The Alchemist']
>>> newbookvalues[1]
'Black Beauty'
>>> |
```

## *Finding the length of Dictionary*

To find the length of a dictionary i.e. the number of key-value pairs you can use python's built-in method `len()`

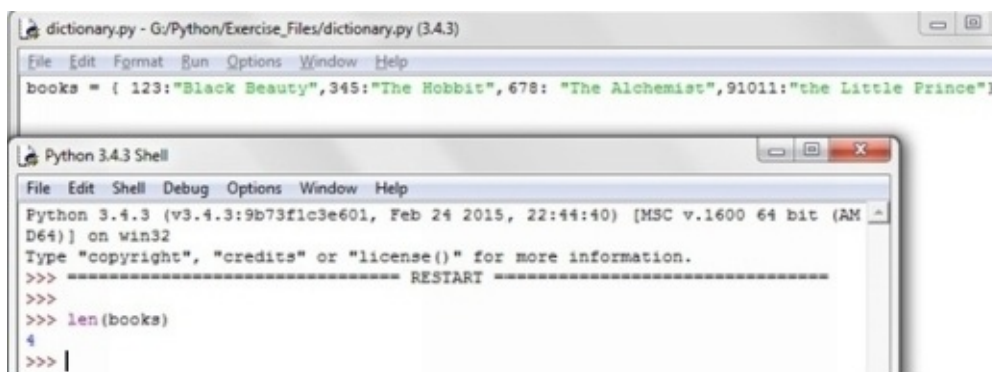
`len(dictionary_name)`

### Exercise #56: Finding Length of Dictionary

Open `Dictionary.py`. Run->Run Module. In python shell type:

1. `>>> len(books )`
2. `>>>`

Figure 63



## Chapter Eight

# Conditionals If-Else

When writing a computer program you often need to ask questions from users and proceed accordingly. Alternatively, you may want to skip some blocks of code and take a different route. You can accomplish this using the **if-else** statements. if-else statements are the most simplest type of selection statements and are used by the computers to decide which path to choose according to a certain criteria. Let's understand this by a simple example; examine the pseudo code below:

```
basicSalary = 10,000
bonus = 1000
employeeGetsBonus = True

If employeeGetsBonus is true
netSalary = basicSalary + bonus.

else
netSalary = basicSalary
```

The problem at hand is to calculate the netSalary of an employee. For every employee let's assume we have a variable employeeGetsBonus; based on his/her performance the variable is set to true or false. If the variable is true the computer adds the bonus amount to the basic salary. Here we need to make a decision, according to the variable "employeeGetBonus". Computer can take one of the two routes to calculate the netsalary ( i.e. with and without bonus) depending on the value of variable employeeGetBonus.

## If statement

### Exercise #57: Simple if statement

Open IDLE text editor, write the following code and save the program as simpleif.py

```
score = 100
```

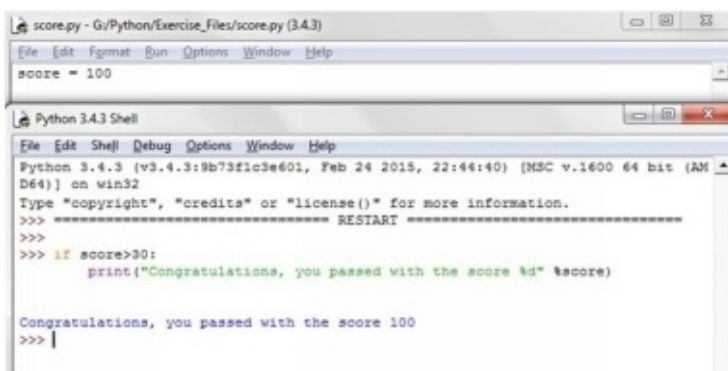
Run->Run Module. In the python Shell type:

1.	>>> if score>30:
2.	print ("Congratulations you passed with score %d" %score)
3.	>>>
4.	

For this exercise we first create a variable and assign it a value of 100 in the Python text editor. We then Run Module and proceed to Python shell where we write an **if** statement. The **if statement** is coupled with a comparison operators which compares the score variable with integer 30. The code on line 1 of python shell can be read aloud as “If the score is greater than 30 then...”. Note that the if statement is followed by a colon :.

After typing the if statement followed by colon hit the Enter key on your keyboard. You will notice that instead of getting a >>> prompt the cursor will be automatically indented. Here you can write the code which you want to execute after the if-check has been performed. For this example we will keep it simple and get the program to print a one liner message “Congratulations you passed with score..” with score.

Figure 64



Change the value of the score variable in simpleif.py and re-do the exercise to examine different outputs of the program. In the following exercise we will assign score variable a value of 30 and investigate the output.

### Exercise #58: Simple if statement (||)

Open simpleif.py and change the value of score variable.



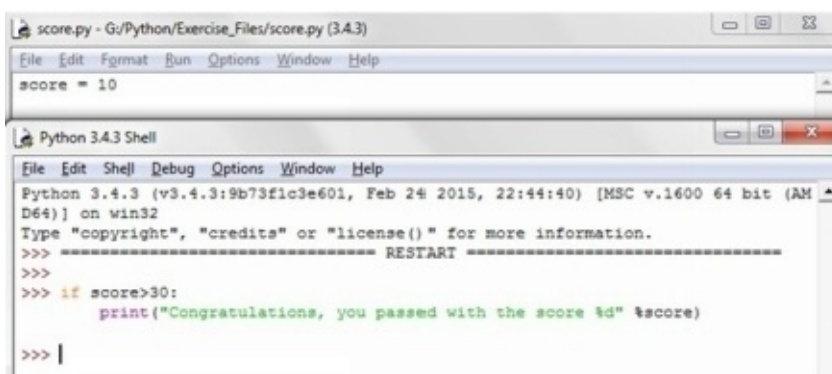
```
score = 30
```

**Run->Run Module. In the python Shell type:**

1. `>>> if score>30:`
2. `print ("Congratulations you passed with score %d" %score)`
3. `>>>`
- 4.

Now, since the condition i.e. score is greater than 30, is not satisfied the print statement will not be executed.

Figure 65



For the following exercise we will extend the program and instead of hard-coding a value of score we will prompt the user to input a value which we will store as score. We will also write the if-statement in text-file (not in python shell). Like before, we will use the if statement to compare user-input with the integer 30 and print a message if the condition is satisfied.

**Exercise #59: Simple if statement with user input**

**Open simpleif.py and change the code as follows:**

1. `score = int(input("Enter your score: "))`
2. `if score>30:`
3. `--print("Congratulations")`
4. `--print("You passed")`
5. `--print("Your score is %d" %score)`
6. `print("Good Bye!")`

**Run->Run Module. In the python Shell type:**

7. `>>> 50`

**Run the program again with different value:**

Before you run the program, take a minute and try to anticipate the output of the program with two user inputs (i.e. 30 and 50). Break-down of code is given below:

- In line1 we define a variable called score. We use the input ( ) function to prompt the user to enter a number. Recall the input ( ) function returns a string. Since we cannot compare a string to integer we first need to convert it to an int.
- In line 2 we compare score variable supplied by the user to 30. If the input value is greater than 30 the program prints 3 lines

```
print("Congratulations")
print("You passed")
print ("Your score is %d" %score)
```

I have used `--` to indicate that the three lines are at the same level of indentation. Refer to the screenshot below.

- Finally we print a good bye message. This time, however the print statement is not indented.

So what happens when the user enters 50?

Since 50 is greater than 30, the code block after the if statement executes printing the three lines,

*Congratulations*

*You passed*

*Your score is 50*

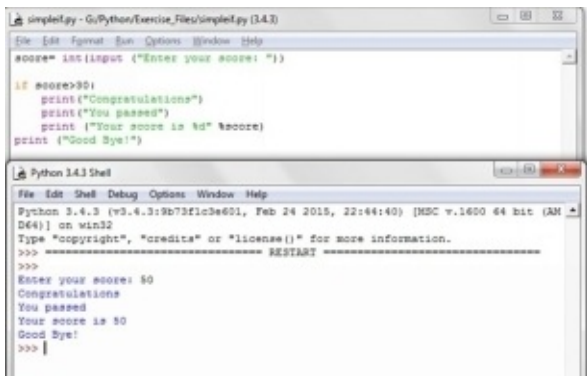
*Good Bye!*

So far so good! What happens when you enter 30?

Now as 30 is not greater than 30, the condition is not met and the code block does not execute. And the output is:

*Good Bye!*

Figure 66



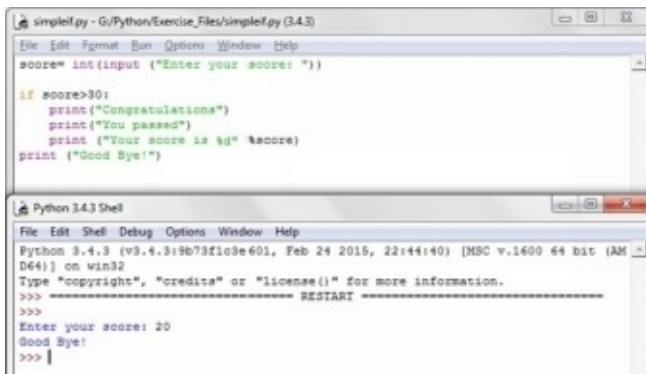
The image shows a screenshot of a Python IDE. The top window, titled 'simpleif.py - G:\Python\Exercise\_Files\simpleif.py (3.4.3)', contains the following code:

```
score= int(input("Enter your score: "))

if score>30:
    print("Congratulations")
    print("You passed")
    print("Your score is %d" %score)
print ("Good Bye!")
```

The bottom window, titled 'Python 3.4.3 Shell', shows the execution of the script. It displays the prompt 'Enter your score: 50', followed by the output 'Congratulations', 'You passed', 'Your score is 50', and 'Good Bye!'. The shell prompt '>>>' is visible at the end of the output.

Figure 67



The image shows a screenshot of a Python IDE. The top window, titled 'simpleif.py - G:\Python\Exercise\_Files\simpleif.py (3.4.3)', contains the following code:

```
score= int(input("Enter your score: "))

if score>30:
    print("Congratulations")
    print("You passed")
    print("Your score is %d" %score)
print ("Good Bye!")
```

The bottom window, titled 'Python 3.4.3 Shell', shows the execution of the script. It displays the prompt 'Enter your score: 20', followed by the output 'Good Bye!'. The shell prompt '>>>' is visible at the end of the output.

Were you expecting a blank console? Did the *Good Bye!* message surprise you? If so you are prepared for the next topic.

## Understanding Blocks

Blocks are programming statements grouped together. Generally speaking programming languages utilize set of curly brackets to define a block of code. However, in Python, you use indentation or spaces to identify a block of code. You can insert whitespaces by using tabs or by pressing space-bar.

In our last exercise we executed three print statements if the condition was met. All the three statements have similar level of indentation or spacing. Whereas the Goodbye! message was written without indentation i.e. it was not part of the **if** code and was executed in either case.

```
score = 100
if score>30:                                Block of code
    print("Congratulations")
    print("You passed")
    print ("Your score is%d" %score)
print ("Good Bye!")
```

## else with if

Having the understanding of code blocks we are now in the position to couple our *if* statement with an else statement. The *if* code block is executed if the condition is met, we add an *else* statement to cater situations when the condition is false.

### Exercise #60: Simple if-else

Open simpleif.py and change code as follows:

1. score = int (input ("Enter your score: "))
2. if score>30:
3.     --print("Congratulations")
4.     --print("You passed")
5.     --print ("Your score is %d" %score)
6. else :
7.     --print("Oops! You flung!")
8.     --print("Your score is %d" %score)
9.     --print ("Better Luck Next Time")
10. print ("Good Bye!")

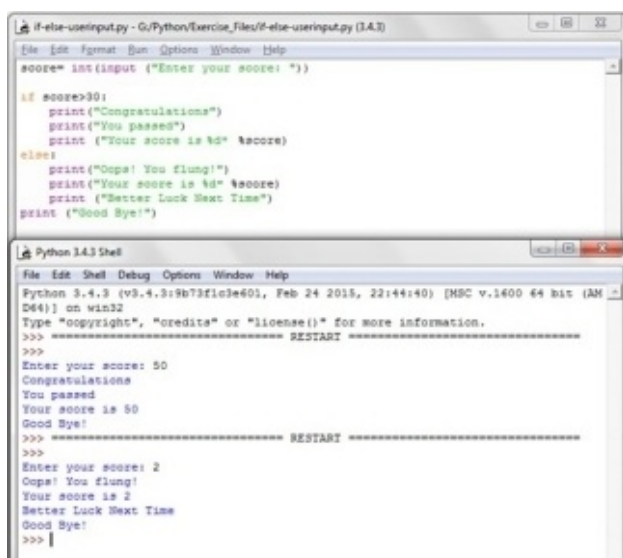
Run->Run Module. In the python Shell type:

```
>>> 50
```

Run the program again with different value:

```
>>> 2
```

Figure 68



In the above example when we first entered the test value 50. Since the condition (score>30) is true the if code block executes printing the first three statements at line 3, 4,

5 and exists the code block. The else code does not execute. Finally the good bye message, written out of the if-else code block is printed. For the second case, we enter a score of 2. The condition (score>30) is false and the if block is not executed; instead the else code block is executed printing the lines 7,8 and 9. After printing the else-block the program exists the block finally printing the good-bye message outside the code.

### ***Adding multiple else if with elif***

If-else statements help us pick one of two possible ways. However there are situations where we need more than two ways. In such a situation we use **elif** statements.

For example, in the exercise under-discussion, currently we are testing the score and giving the status of pass or fail. Let's incorporate another condition when the score entered is equal to the minimum passing value.

#### **Exercise #61: Multiple if-else with elif**

Open simpleif.py and change the code as follows:

```
1.  score= int(input ("Enter your score: "))
2.  if score>30:
3.      print("Congratulations")
4.      print("You passed")
5.      print ("Your score is %d" %score)
6.  elif: score == 30
7.      print ("Barely Passed")
8.      print ("Your score is %d" %score)
9.  else:
10.     print("Oops! You flung!")
11.     print("Your score is %d" %score)
    print ("Better Luck Next Time")
    print ("Good Bye!")
```

**Run->Run Module. In the python Shell type:**

```
7.  >>> 30
```

**Run the program again with different value:**

```
>>> 2
```

In the above example we see that now instead of two, the program has three routes:

1. When the score is greater than 30

2. When the score is equal to 30
3. When the score is less than 30

The first condition is implemented using the if statement. The second condition is implemented using the elif statement. Here it's important to recognize the difference between = and the == operator. We use the single equal to (=) operator to assign a value to a variable where as we use the double equal to (==) to compare two values. Naturally if the variable to test is not greater than 30 and not equal to 30 this implies that the variable is less than 30. Therefore we simply put an else condition and do not explicitly state it.

Figure 69



The image shows a screenshot of a Python IDE with two windows. The top window, titled "if-else-userinput.py - C:\Python\Exercise\_Files\if-else-userinput.py (3.4.3)", contains the following code:

```
score = int(input("Enter your score: "))
if score > 30:
    print("Congratulations")
    print("You passed")
    print("Your score is %d" % score)
elif score == 30:
    print("Barely Passed")
    print("Your score is %d" % score)
else:
    print("Ooops! You flung!")
    print("Your score is %d" % score)
    print("Better Luck Next Time")
print("Good Bye!")
```

The bottom window, titled "Python 3.4.3 Shell", shows the execution of the code. It displays the prompt "Enter your score: 50" followed by the output "Congratulations", "You passed", "Your score is 50", and "Good Bye!". It then shows the prompt "Enter your score: 30" followed by the output "Barely Passed", "Your score is 30", and "Good Bye!". Finally, it shows the prompt "Enter your score: 3" followed by the output "Ooops! You flung!", "Your score is 3", "Better Luck Next Time", and "Good Bye!".

## Combining conditions in if-else

You can combine conditions inside if-else blocks by using **and** and **or** operators. Speaking of the pass-fail example, assume we now need to integrate some sort grading in our program. To assign grades according to score we have to compare it against different ranges of numbers and assign grades accordingly. Let's extend our exercise #59 to incorporate a simple grading mechanism.

### Exercise #62: Grading System. Combining conditions in if-else

Open `simpleif.py` and change the value of `score` variable.

```
1. score= int(input ("Enter your score: "))
2. if score<=30:
3.     print("Your score is 30 or less!")
4.     print("FAIL")
5. elif score>30 and score<=50:
6.     print ("score is between 30-50 or 50")
7.     print ("Your grade is D")
8. elif score>50 and score<=70:
9.     print ("score is between 50-70 or 70")
10.    print ("Your grade is C")
11. elif score>70 and score<=80:
12.    print ("score is between 70-80 or 80")
13.    print ("Your grade is B")
14. elif score>80 and score<=90:
15.    print ("score is between 80-90 or 90")
16.    print ("Your grade is A")
17. else:
18.    print ("Your grade is A")
19.    print ("score is greater than 90")
20. print ("Good Bye!")
```

Run->Run Module. In the python Shell type:

In the above exercise we have combined different conditions inside the elif statements.

- In line1 we prompt the user to input a value for score. We define a variable and save the input after converting it to int.
- In line2 we compare the score with 30. This time we use the comparison operator `<=`, which reads as less than or equal to. We compare if the score variable is less than or equal to 30 the program prints "Fail message" and exits the loop. Finally it



prints a good bye message at line 20.

- If the condition given at line 2 is not met, computer checks for the second condition. The second condition given at line 5 is `elif score>30 and score<=50`: this condition combines two comparison statements using `and`. The statement reads aloud as: else if score is greater than 30 and less than or equal to 50. Here we have define a range of 30 – 50 including 50. If the score falls within this range the computer prints line 6 and 7.

Likewise, we check the score variable and compare it against different number ranges printing corresponding messages.

Note how we have defined different code blocks with spaces.

Figure 70

```
combining_conditions.py - G:\Python\Exercise_Files\combining.c
File Edit Format Run Options Window Help

score= int(input("Enter your score: "))
if score<=30:
    print("Your score is 30 or less!")
    print("FAIL")
elif score>30 and score<=50:
    print("score is between 30-50 or 50")
    print("Your grade is D")
elif score>50 and score<=70:
    print("score is between 50-70 or 70")
    print("Your grade is C")
elif score>70 and score<=80:
    print("score is between 70-80 or 80")
    print("Your grade is B")
elif score>80 and score<=90:
    print("score is between 80-90")
    print("Your grade is A")
else:
    print("Your grade is A")
    print("score is greater than 90")
print("Good Bye!")

Python 3.4.3 Shell
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2014) on win32
Type "copyright", "credits" or "license()" for more
>>> ===== RESTART =====
>>>
Enter your score: 30
Your score is 30 or less!
FAIL
Good Bye!
>>> ===== RESTART =====
>>>
Enter your score: 40
score is between 30-50 or 50
Your grade is D
Good Bye!
>>> ===== RESTART =====
>>>
Enter your score: 70
score is between 50-70 or 70
Your grade is C
Good Bye!
>>> ===== RESTART =====
>>>
Enter your score: 100
Your grade is A
score is greater than 90
Good Bye!
>>> |
```

If-else statements lie at the core of every programming language and are very useful when we need to take decisions or for error-handling when creating programs.

## Chapter Nine

# Loops

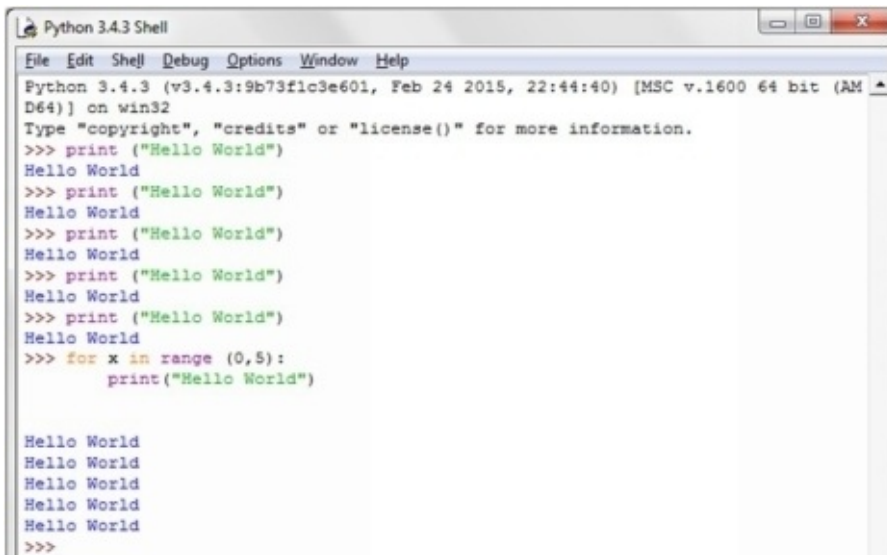
In the previous chapter we investigated the else-if statements or the conditionals. Conditionals evaluate a statement and choose and execute one of the different blocks of code. Loops like conditionals, also check and evaluate statements however unlike the conditionals they execute a piece of code repeatedly until the condition is satisfied. Loops may also repeat a piece of code for a fixed number of times. Loops are integral part of any program. Often game programs incorporate conditions such as “repeat game until user chooses to exit”. In this case the program will repeat until the condition is met; condition that is when user requests to quit. In this section we will explore different types of loops available in Python.

Loops can reduce a lot of work for you. Understand this with a small example.

```
print ("Hello World")
print ("Hello World")
print ("Hello World")
print ("Hello World")
print ("Hello World")
for x in (0,5):
    print("Hello World")
```

Both the pieces of code have the same output. For the first case we manually type 5 lines of Hello World, while in the second case we ask the computer to repeat the print command 5 times using the for loop. The computer repeats the loop (containing a single print statement) five times and exists. This simple example demonstrates the usefulness of loops.

Figure 71

A screenshot of a Python 3.4.3 Shell window. The window has a title bar that says "Python 3.4.3 Shell" and standard Windows window controls (minimize, maximize, close). Below the title bar is a menu bar with "File", "Edit", "Shell", "Debug", "Options", "Window", and "Help". The main area of the window contains the following text:

```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> print ("Hello World")
Hello World
>>> print ("Hello World")
Hello World
>>> print ("Hello World")
Hello World
>>> print ("Hello World")
Hello World
>>> print ("Hello World")
Hello World
>>> for x in range (0,5):
    print("Hello World")

Hello World
Hello World
Hello World
Hello World
Hello World
>>>
```

Before we proceed and investigate the for loop it's important to understand the Range functions as for loops and often coupled with range function.

## Range Function

A range function returns a range or series of number. Range function is used in different ways, the simplest way to use range function is as follows:

```
range(n)
```

returns numbers from 0 to n-1

Range function accepts an integer and returns numbers from 0 to one less than that number. For example, range (4) returns the numbers 0,1,2,3. You can get the list of numbers returned by the range function by coupling the range function with list function like follows:

```
list(range( <some_number> ) )
```

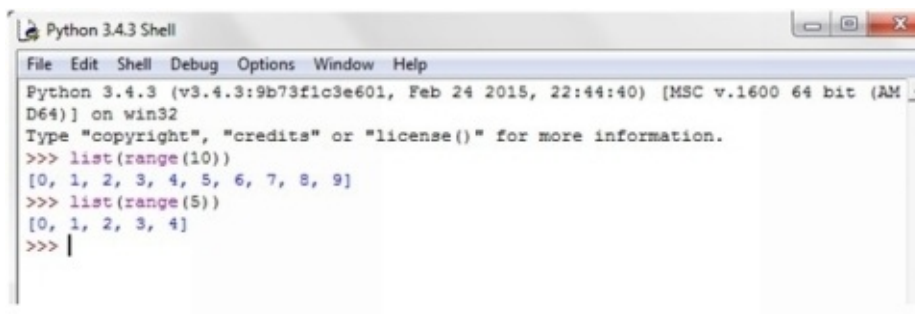
### Exercise #63: Understanding range

In python shell type:

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

>>>list(range(5))
>>>[0,1,2,3,4]
```

Figure 72



You can also use the range function to list numbers between any two numbers instead of starting the list from 0. To do so we need to supply two parameters to the range function.

```
range(n, m)
```

returns numbers from n to m-1

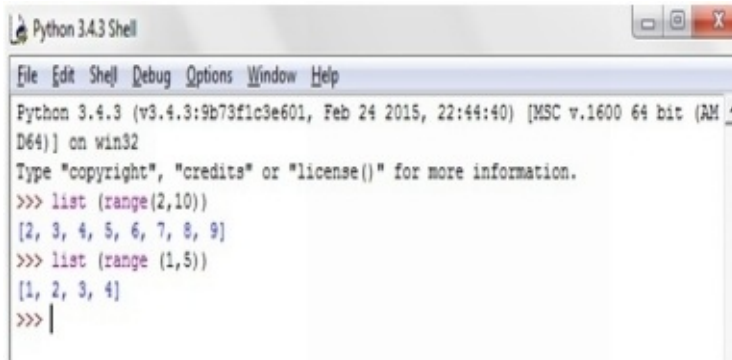
### Exercise #64: Understanding range (2)

In python shell type:

```
>>>list(range(2, 10))
>>>

>>> list(range(1,5))
>>>
```

Figure 73

A screenshot of a Python 3.4.3 Shell window. The window has a title bar that says "Python 3.4.3 Shell" and a menu bar with "File", "Edit", "Shell", "Debug", "Options", "Window", and "Help". The main text area shows the following output:

```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> list(range(2,10))
[2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1,5))
[1, 2, 3, 4]
>>> |
```

Having the understanding of the range function we are in position to discuss the for loops.

## For Loop

The general syntax of a for-loop is:

```
for <variable> in <collection/list>
```

### Exercise # 65: Understanding the For loop

Create a new IDLE text file and save it as for.py

```
for x in range (0,5):  
    print ("x is : %d" %x)
```

Let's investigate the code line by line.

- The loop starts with the keyword for.
- We create a variable x.
- We define a range (0,5). We know that this range returns 0,1,2,3,4

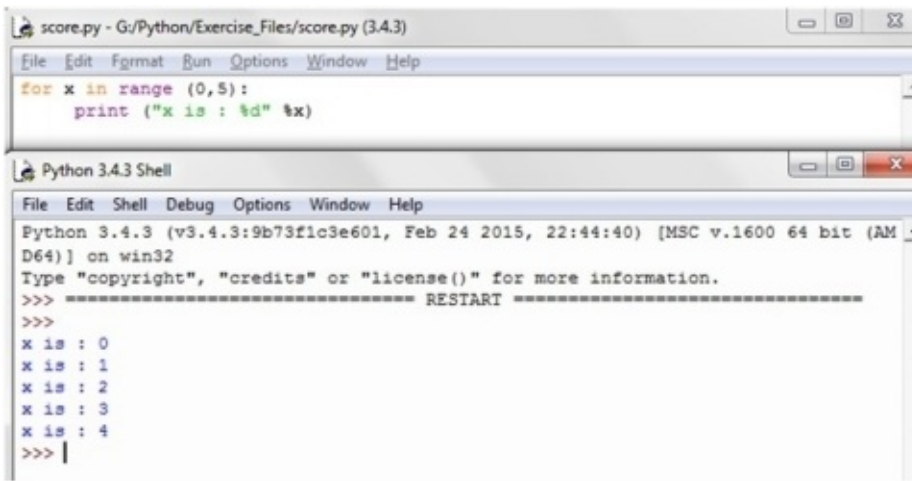
As the code starts execution, the variable x is assigned value 0 for the first iteration. The block of code after the for statement (i.e. print statement) executes. For this example we are only printing the value of x. After execution of the print statement the variable x assumes the value of the next number in list which is 1 and executes the code. Finally when x assumes the value of 4 and executes the code, there are no further numbers and hence the computer exits the loop. Retry the exercise using a different ranges.

### Exercise # 66: Understanding the For loop-2

Create a new IDLE text file and save it as for.py

```
for x in range (5,10):  
    print ("x is : %d" %x)
```

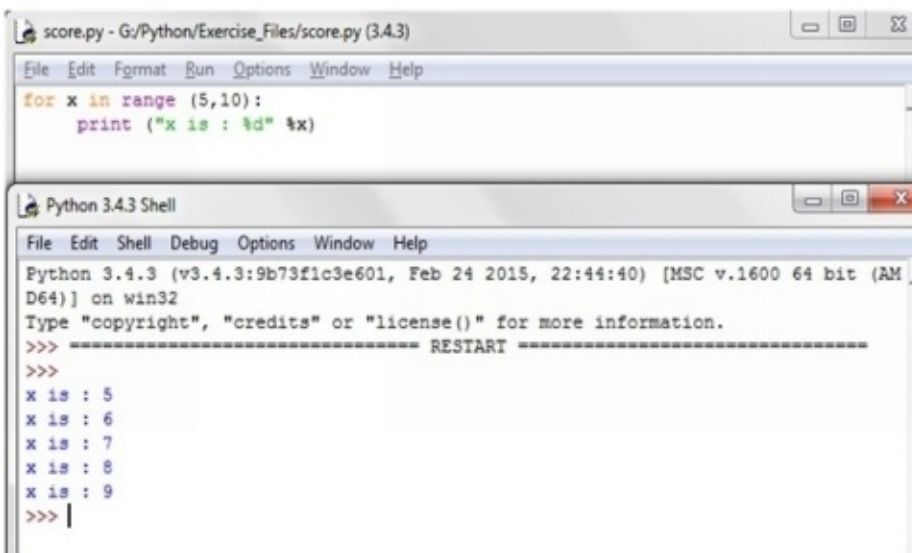
Figure 74



```
score.py - G:/Python/Exercise_Files/score.py (3.4.3)
File Edit Format Run Options Window Help
for x in range (0,5):
    print ("x is : %d" %x)

Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
x is : 0
x is : 1
x is : 2
x is : 3
x is : 4
>>> |
```

Figure 75



```
score.py - G:/Python/Exercise_Files/score.py (3.4.3)
File Edit Format Run Options Window Help
for x in range (5,10):
    print ("x is : %d" %x)

Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
x is : 5
x is : 6
x is : 7
x is : 8
x is : 9
>>> |
```

From the above exercises note that may the range be (0,5) or (5,10) the loop always executes for 5 times. This is because both the ranges contain 5 numbers each, even though the numbers are different. In a situation where we do not need to keep track of the counter variable and only want the loop to iterate for a particular number of times, the numbers contained within a range become irrelevant instead we are only concerned in how many numbers are contained in a range. For example:

#### Exercise # 67: Understanding the For loop

Create a new IDLE text file and save it. Run->Run module

```
for x in range (0,5):
    print ("Hello There")

print("-----")

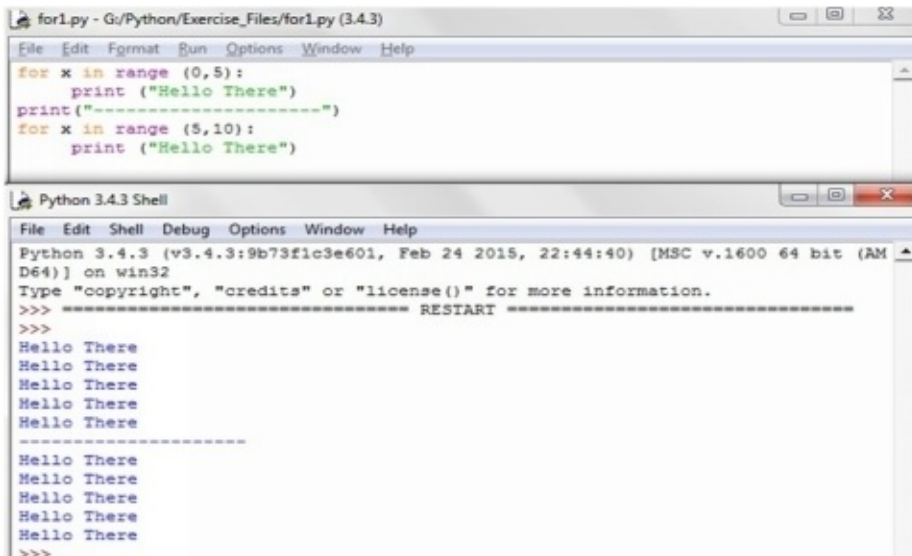
for x in range (5,10):
```



```
print ("Hello There")
```

The output of both the programs is the same though we have given different ranges. You might have already guessed the reason i.e. both the ranges contain same number of values.

Figure 76



The screenshot shows a Python IDE window titled 'for1.py - G:/Python/Exercise\_Files/for1.py (3.4.3)'. The code in the editor is:

```
for x in range (0,5):
    print ("Hello There")
print ("-----")
for x in range (5,10):
    print ("Hello There")
```

Below the editor is the 'Python 3.4.3 Shell' window. It shows the output of the program:

```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
Hello There
Hello There
Hello There
Hello There
Hello There
-----
Hello There
Hello There
Hello There
Hello There
Hello There
>>>
```

Recall from our discussion on Strings, Strings are sequence of characters. We can use for loops to traverse a string.

#### Exercise # 68: For loop and Strings

Create a new IDLE text file and save it. Run->Run module

```
myString = input ("Please Enter some String")
for eachChar in myString:
    print (eachChar)
```

In the above exercise the program prompts the user to input a string. We then use the for loop to print each character in the string in a line. In this example, we replace `x` with a variable called `eachChar`; both variables are essentially counter variable however `eachChar` has more descriptive name and eases readability. Every time the loop executes, the `eachChar` variable is assigned a character from the string. The for loop body executes printing the variable `eachChar`.

In the following exercise we take two numbers from the user and define a range i.e. list the numbers between the two numbers and print even numbers within the range.

#### Exercise #70: Printing even numbers in a given range with for loop

```
firstNumber = int(input ("Please enter First (smaller) Number: "))
secNumber = int(input ("Please enter Second (greater) Number: "))

for number in range(firstNumber, secNumber):
    if number%2 == 0:
        print (number)
```

```
for number in range(firstNumber, secNumber):
    if number%2 == 0:
        print (number)
```

The exercise begins by prompting the user for number. We use the input function to prompt the user for number and then use the int function to convert the number supplied by the user into integer. Later, we run a for loop in the list of numbers and for every number in the range we check if it is even or not. If the number is even it gets printed. Take two important notes from this exercise:

- We have defined two blocks of code here. The for loop code block and if code block. The if code block is contained inside the for loop block. Note how the indentation is different for different code blocks.
- We use the mathematical operator modulo % to check if the number is completely divisible by 2. If the number is completely divisible and the remainder is equal to 0, the number is even.

### ***Using the for loop to iterate a List***

In previous section we used the for loop to traverse a string. Since Lists are also sequential data set we may use loops for traversing a list item-by-item.

#### Exercise #71: Traversing a list with for loop

Create an IDLE text file, write the following code and save the file with .py extension.  
Once saved RUN->Run Module

```
colors=["red", "yellow", "green", "blue", "pink", "black", "purple", "indigo"]

print ("The items in the list are: ")
```

```
for color in colors:
    print (color)
```

In the above exercise we define a list and use a for loop to traverse it. We define a variable called “color” and assign items in list to color variable one by one. Note that color is a variable to be used within the for loop. We may have called it “col” or simply “c”; it would work the same! However for readability purpose it’s important to name the counter variable appropriately.

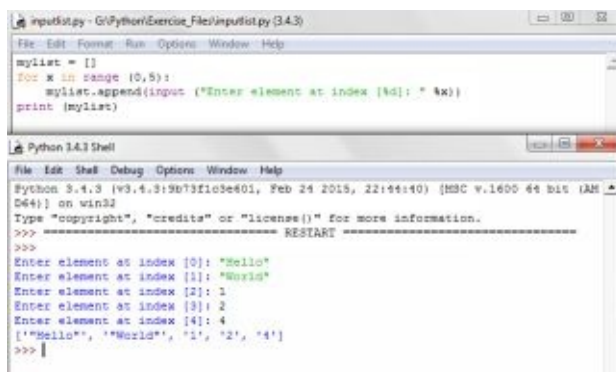
In the following exercise we will use for loop to insert values in a list.

#### Exercise #72: Input values in list using For loop

```
mylist = []
for x in range (0,5):
    mylist.append(input ("Enter element at index [%d]: " %x))
print (mylist)
```

In the above exercise we begin by defining an empty list. We then prompt the user to enter a value which the user wants to insert in a list. Note that user is permitted to add a numeric value as well as alpha-numeric value. The loop will be executed for 5 times, asking the user to enter 5 elements for the list. We have already studied the append function in detail in the chapter dedicated to lists.

Figure 77



Having seen a simple example of how we can use for loops to traverse a list let’s do something more useful. The following program converts a list of Celsius temperatures to equivalent Fahrenheit temperatures.

#### Exercise #73: Converting temperature: Using for loop with lists

Create an IDLE text file, write the following code and save the file with .py extension.  
Once saved RUN->Run Module

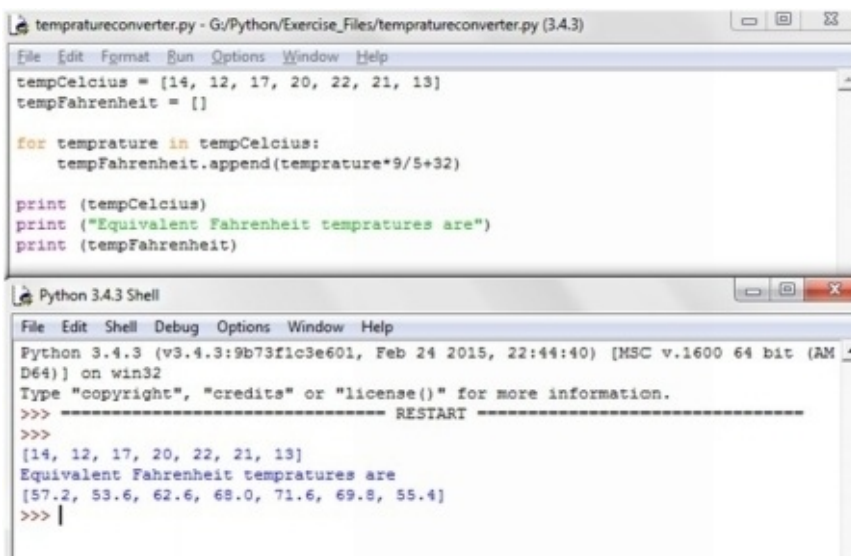
```
tempCelcius = [14, 12, 17, 20, 22, 21, 13]
tempFahrenheit = []

for temprature in tempCelcius:
    tempFahrenheit.append(temprature*9/5+32)

print (tempCelcius)
print ("Equivalent Fahrenheit tempratures are");
print (tempFahrenheit)
```

For this exercise we begin by defining a list of celcius temperatures and call it **tempCelcius**. We also define an empty list called tempFahrenheit. We will store our converted temperatures in this list. We then use the for loop to traverse the list. The counter variable temprature assumes value from the tempCelcius list one by one. For each value in the list we perform the mathematical calculation temprature\*9/5+32 which converts it to Celsius temperature. The converted value is appended in the tempFahrenheit list. Finally we print values of both the lists for comparison.

Figure 78



The screenshot displays two windows from a Python IDE. The top window, titled 'tempratureconverter.py - G:/Python/Exercise\_Files/tempratureconverter.py (3.4.3)', contains the following Python code:

```
tempCelcius = [14, 12, 17, 20, 22, 21, 13]
tempFahrenheit = []

for temprature in tempCelcius:
    tempFahrenheit.append(temprature*9/5+32)

print (tempCelcius)
print ("Equivalent Fahrenheit tempratures are")
print (tempFahrenheit)
```

The bottom window, titled 'Python 3.4.3 Shell', shows the output of the script after execution:

```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ----- RESTART -----
>>>
[14, 12, 17, 20, 22, 21, 13]
Equivalent Fahrenheit tempratures are
[57.2, 53.6, 62.6, 68.0, 71.6, 69.8, 55.4]
>>> |
```

## Using break

There are situations when you want to exit the execution of a loop before it finishes all of the iterations. For example consider a situation where you want to fetch a piece of data item from a list. It's likely for you to use a for loop to iterate over the list. As soon as you find the required item you quit. When you want to stop the loop and continue executing the code after the for-loop block you can use the **break** command.

Exercise #74: Simple game to demonstrate for and break.

For this exercise we develop a simple game. A list of 5 0's is displayed to the user and he/she is asked to enter values in the Python shell to replace the new value with 0. However he/she has to avoid prime numbers. Also the numbers entered must be smaller than 20. If the user enters a prime number or a number greater than 20 the game quits and a final score is displayed on the basis of number of zeros replaced in the list.

```
print ("Replace all zeros with numbers less than 20! Avoid Prime numbers")
mylist = [0,0,0,0,0]
print (mylist)
score = 0
print ("-----Game Starts!-----")

for x in range (0,5):
    numb = int (input ("Enter List [%d]: " %x))
    if numb == 2 or numb == 3 or numb == 5 or numb == 7 or numb == 11 or numb == 13 or numb == 17 or numb == 19 or
numb>20 :
        print ("You Entered a Prime number!")
        print ("The Game Quits")
        break
    else:
        mylist[x] = numb

print (mylist)

for element in mylist:
    if element != 0:
        score = score + 1

print ("Your score is %d" %score)
```

For this exercise we create a list of 5 zeros. The goal for the user is to enter a digit to replace the 0 for every zero in the list. For every zeros the user is able to replace by his/her proposed digit he earns points. To make the game challenging we impose a condition on the user input. The user may not enter a number greater than 20 or a prime number. The game starts! We use a for loop which will prompt the user to enter a number to replace every zero in the list of zeros. For every value the for loop checks if the number entered is

a prime number or if it's greater than 20 using the if command. Note that we have combined the conditions using the **or** logical operator. If either of the condition is met the game over message is printed and the execution is stopped. In case the user enters a valid entry the else statement will execute and the number entered by the user is assigned to the list.

Finally we calculate the score based on the number of zeros in the updated list. Here we use the **!=** (is not equal to) comparison operator. Every element in list is checked if it is not equal to zero, the score is incremented with 1. The maximum score is 5.

Figure 79

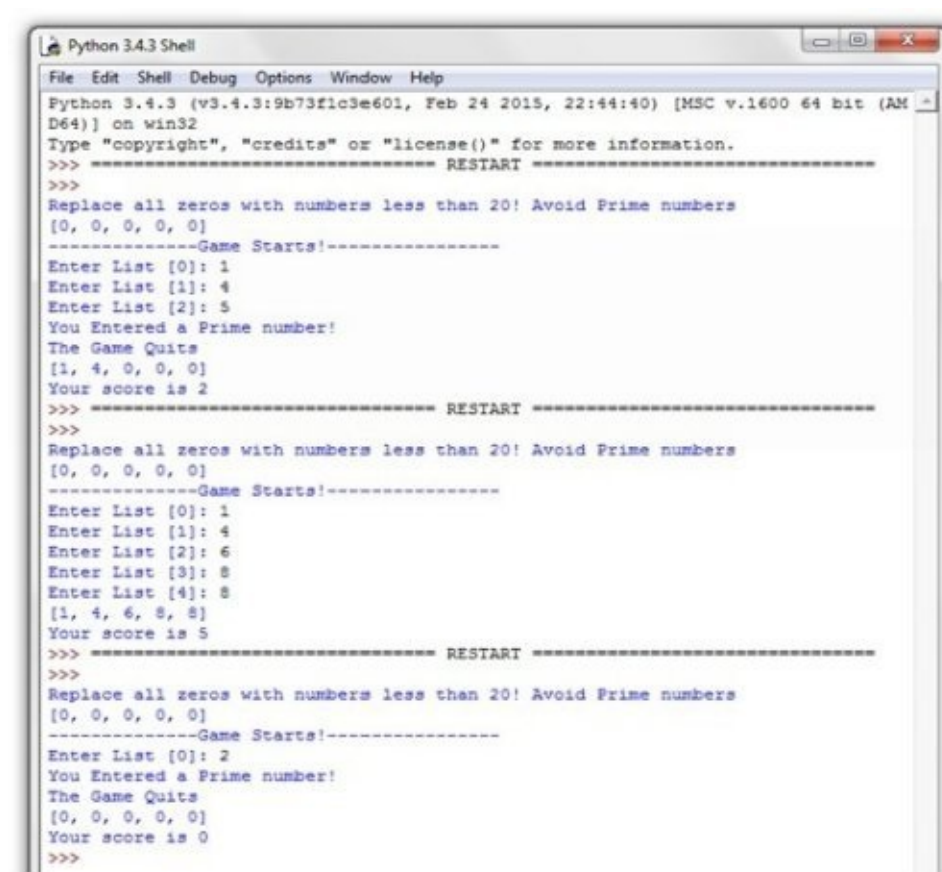


```
primesnumberfinder.py - G:\Python\Exercise_Files\primesnumberfinder.py (3.4.3)
File Edit Format Run Options Window Help
print ("Replace all zeros with numbers less than 20! Avoid Prime numbers")
mylist = [0,0,0,0,0]
print (mylist)
score = 0
print ("-----Game Starts!-----")

for x in range (0,5):
    numb = int (input ("Enter List [%d]: " %x))
    if numb == 2 or numb == 3 or numb == 5 or numb == 7 or numb == 11 or numb == 13 or numb == 17 or numb == 19:
        print ("You Entered a Prime number!")
        print ("The Game Quits")
        break
    else:
        mylist[x] = numb
print (mylist)

for element in mylist:
    if element != 0:
        score = score + 1
print ("Your score is %d" %score)
```

Figure 80



```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
Replace all zeros with numbers less than 20! Avoid Prime numbers
[0, 0, 0, 0, 0]
-----Game Starts!-----
Enter List [0]: 1
Enter List [1]: 4
Enter List [2]: 5
You Entered a Prime number!
The Game Quits
[1, 4, 0, 0, 0]
Your score is 2
>>> ===== RESTART =====
>>>
Replace all zeros with numbers less than 20! Avoid Prime numbers
[0, 0, 0, 0, 0]
-----Game Starts!-----
Enter List [0]: 1
Enter List [1]: 4
Enter List [2]: 6
Enter List [3]: 8
Enter List [4]: 8
[1, 4, 6, 8, 8]
Your score is 5
>>> ===== RESTART =====
>>>
Replace all zeros with numbers less than 20! Avoid Prime numbers
[0, 0, 0, 0, 0]
-----Game Starts!-----
Enter List [0]: 2
You Entered a Prime number!
The Game Quits
[0, 0, 0, 0, 0]
Your score is 0
>>>
```

## While Loop

In our last exercise we used for-loop with break statement. On encountering the break statement the execution of loop was stopped immediately and the code after the for code block was executed; While loops are more forgiving. While loop executes a code until a condition is met. The general syntax of while loop is :

```
while <condition>:  
    block of code telling to  
    do something until condition  
    is met
```

Understand the while loop with this simple example:

### Exercise # 75: Understanding while loop

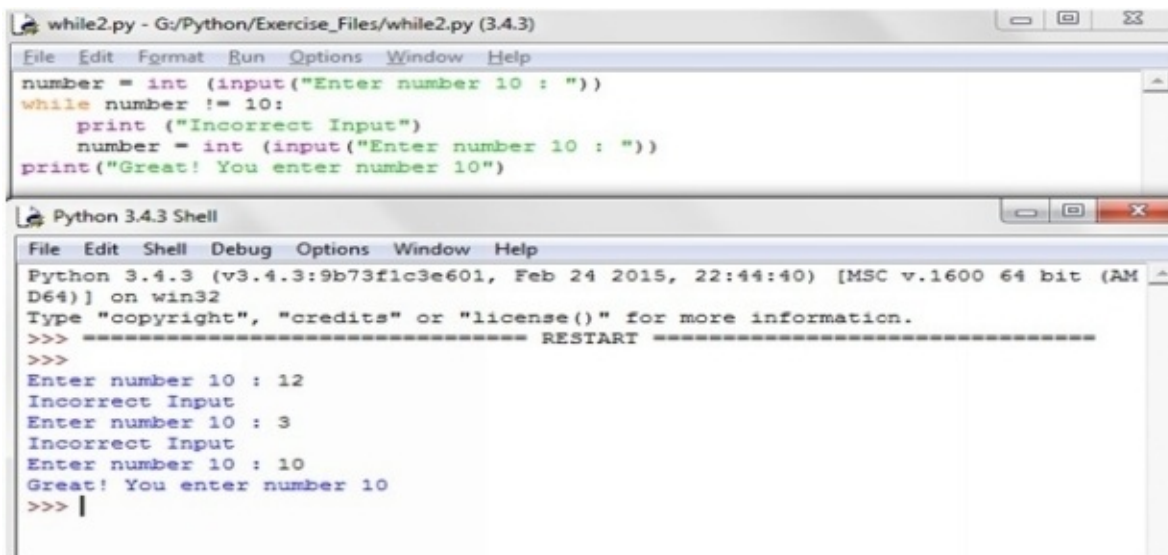
Open IDLE text editor. Write the following code and save it as .py file. Then Run->Run Module

```
number = int (input("Enter number 10 : "))  
while number != 10:  
    print ("Incorrect Input")  
    number = int (input("Enter number 10 : "))  
print("Great! You enter number 10")
```

In the above exercise the program prompts user to enter the number 10. If the user enters a number other than 10 the program prints an error message and prompts the user to re-enter number. If the user correctly enters a number, the program exists.

We use a while loop to implement this. The while loop compares the user input to 10, using the != (is not equal to) operator. You can read aloud the while statement as: while the number is not equal to 10, print incorrect input and prompt user to enter again. Once the user enters the correct digit (and the condition is met), the program exists the while loop and prints "Great you entered number 10".

Figure 81



The screenshot shows a Python 3.4.3 IDE with two windows. The top window, titled 'while2.py - G:/Python/Exercise\_Files/while2.py (3.4.3)', contains the following code:

```
number = int (input("Enter number 10 : "))
while number != 10:
    print ("Incorrect Input")
    number = int (input("Enter number 10 : "))
print("Great! You enter number 10")
```

The bottom window, titled 'Python 3.4.3 Shell', shows the execution of the program:

```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
Enter number 10 : 12
Incorrect Input
Enter number 10 : 3
Incorrect Input
Enter number 10 : 10
Great! You enter number 10
>>> |
```

This is a pretty straight forward example which demonstrates the use of While loop. We can use the while loop to validate any input by the user for example to check if the user has entered a numeric value when asked to enter age or date.

#### Exercise #76: Using While Loop to validate year

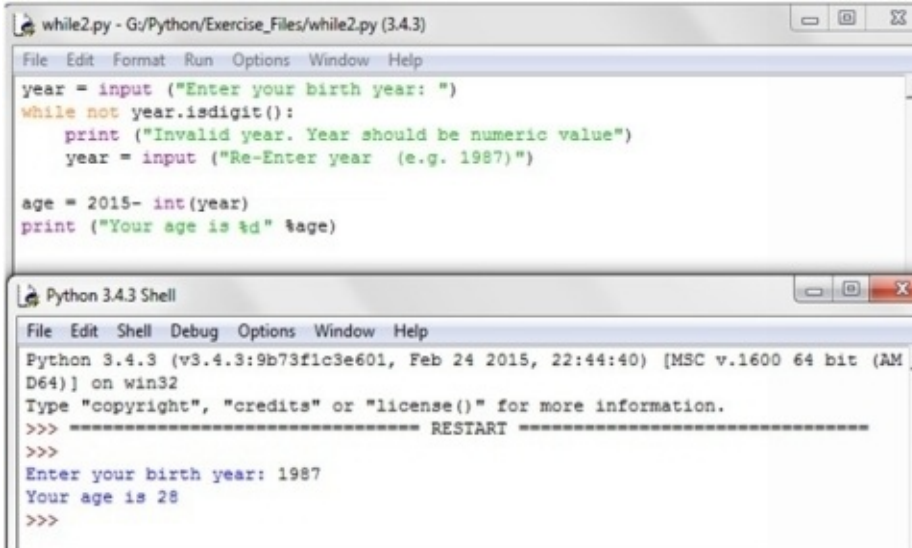
```
year = input ("Enter your birth year: ")
while not year.isdigit():
    print ("Invalid year. Year should be numeric value")
    year = input ("Re-Enter year (e.g. 1987)")

age = 2015- int(year)
print ("Your age is %d" %age)
```

In the above exercise we use the while loop to check if the entered value is numeric. While the year entered is not numeric the program will continue to prompt user to enter a valid value.

Figure 82





The image shows a screenshot of a Python IDE with two windows. The top window, titled 'while2.py - G:/Python/Exercise\_Files/while2.py (3.4.3)', contains a Python script. The script prompts the user to enter a birth year, uses a while loop to ensure the input is a digit, calculates the age by subtracting the birth year from 2015, and prints the result. The bottom window, titled 'Python 3.4.3 Shell', shows the execution of the script. It displays the Python version and architecture, followed by a restart prompt. The user enters '1987', and the program outputs 'Your age is 28'.

```
while2.py - G:/Python/Exercise_Files/while2.py (3.4.3)
File Edit Format Run Options Window Help
year = input ("Enter your birth year: ")
while not year.isdigit():
    print ("Invalid year. Year should be numeric value")
    year = input ("Re-Enter year (e.g. 1987)")

age = 2015- int(year)
print ("Your age is %d" %age)
```

```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
Enter your birth year: 1987
Your age is 28
>>>
```

## Chapter Ten

# Functions

Programs can be thought of as list of instructions. In almost all the example we have seen so far, the computer starts to execute the first instruction and continues line-by-line until the last instruction/code. This may be feasible for very small programs but practically programs comprise of hundreds and thousands of lines of code. If computer carries out execution of every line of code in a sequential way, it would take much longer to run a program and the computer will consume more system resources. Functions can be used to break code to reduce program size, improve performance and readability and to make programs easier to maintain.

Functions are self-contained code of blocks which perform a defined task. Any instructions in your code which appear more than once in your program can be encapsulated inside a function. These instructions are saved at one place the computer's memory and called inside the program whenever needed. A function can contain any type of code we have learnt so far.

To create a basic function the code syntax is:

```
def function_name (parameter_list):  
    some code
```

To call function the syntax is

```
function_name(parameters)
```

The following exercise demonstrates how we can break code, group similar instructions and save it in one place.

## Exercise # 78: Understanding Functions

### APPROACH 1:

```
#code without function  
participant1 = input ("please Enter your name: ")  
print ("Hello Dear %s" %participant1)  
print ("Find the Instructions in the Instruction Folder")  
print ("Before proceeding, please take time to fill the survey")  
print ("_____")  
  
participant2 = input ("please Enter your name: ")
```

```
print ("Hello Dear %s" %participant2)
print ("Find the Instructions in the Instruction Folder")
print ("Before proceeding, please take time to fill the survey")
print ("_____")
```

```
participant3 = input ("please Enter your name: ")
print ("Hello Dear %s" %participant3)
print ("Find the Instructions in the Instruction Folder")
print ("Before proceeding, please take time to fill the survey")
print ("_____")
```

```
participant4 = input ("please Enter your name: ")
print ("Hello Dear %s" %participant1)
print ("Find the Instructions in the Instruction Folder")
print ("Before proceeding, please take time to fill the survey")
print ("_____")
```

```
participant5 = input ("please Enter your name: ")
print ("Hello Dear %s" %participant5)
print ("Find the Instructions in the Instruction Folder")
print ("Before proceeding, please take time to fill the survey")
print ("_____")
```

**APPROACH 2: Now we will implement the same code but this time using a function.**

**Open another IDLE text file and place the following code:**

```
def print_Greeting (name_participant):
    print ("Hello Dear %s" %name_participant)
    print ("Find the Instructions in the Instruction Folder")
    print ("Before proceeding, please take time to fill the survey")
    print ("_____")

participant1 = input ("please Enter your name: ")
print_Greeting (participant1)

participant2 = input ("please Enter your name: ")
print_Greeting (participant2)

participant3 = input ("please Enter your name: ")
print_Greeting (participant3)

participant4 = input ("please Enter your name: ")
print_Greeting (participant4)
```

```
participant5 = input ("please Enter your name: ")  
print_Greeting (participant5)
```

Let's go through the exercise line by line.

- The purpose of the program is to prompt participant's name and print personalized greeting message, followed by a few other messages. Both the approaches 1 and 2 essentially perform the same task however approach 2 implements functions to reduce redundancy and make the code more readable and maintainable.
- **Approach 1:** For every participant we define a variable and prompt user for his/her name. We then print personalized greeting using the familiar string formatting operator.
- This approach uses a lot of redundant code. The same greeting (with different names) is printed for every participant. Notice that if we had to make even a minor change in the print statement in approach 1, we would have to manually alter the print statements for each of the 5 participants.
- **Approach 2:** Here, we group the similar print statements inside a function and name it as `print_Greeting`. Once we have defined the function, we write code to prompt the user for their names and assign them to variables. Now, instead of typing the entire print statement block, we simply call the function for every participant. The function `print_Greeting` accepts one parameter. You can regard parameter as an input to the function. The parameter here is the name of participant. The function prints a greeting inserting the name of participant passed to it as a parameter. Whenever we will call the function we will pass the name of the participant to the function for it to print personalized greeting.
- In approach2, if we need to introduce some changes in print lines, we can simply change the print statement inside the function `print_Greeting`. Also the new approach is more readable, easy to maintain and comprises of less number of code-lines.

Figure 83

The image shows two side-by-side Python IDE windows. The left window, titled 'simplefunction.py - G:\Python\Exercise\_Files\simplefunction.py (3.4.3)', contains code that prompts for five participant names and prints a greeting for each. The right window, titled 'simplefunction1.py - G:\Python\Exercise\_Files\simplefunction1.py (3.4.3)', contains a function definition 'def print\_Greeting(name\_participant):' and five calls to this function, each with an input prompt for a participant's name.

### Exercise # 79: Simple Function to calculate Average

Open IDLE text editor and place the following code.

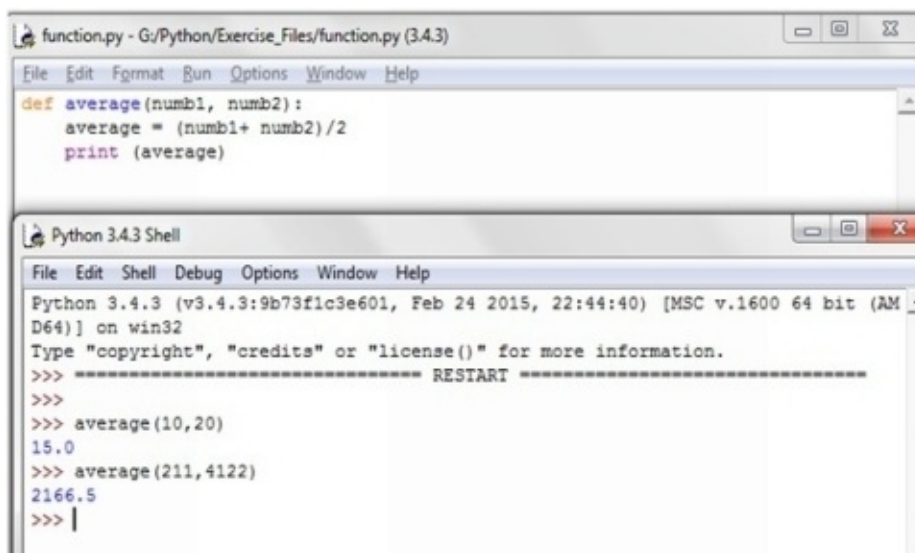
```
def average(num1, num2):  
    average = (num1+ num2)/2  
    print (average)
```

**Run->Run Module**

```
>>>average (10,20)  
>>>  
>>> average(211,4122)  
>>>
```

In the above example we defined a simple function which calculates average of two numbers. The function accepts two numbers as parameters, calculates the average and prints the result. Once we have defined the function in the text file, we will call the function in the Python shell with different parameters. To call the average we will write the name of the function as defined in code file and pass it two numbers in ( ). We can use the same function to calculate the average of different values, that is we are re-using the code.

Figure 84



The image shows a screenshot of a Python IDE with two windows. The top window, titled 'function.py - G:/Python/Exercise\_Files/function.py (3.4.3)', contains a function definition: 

```
def average(num1, num2):  
    average = (num1+ num2)/2  
    print (average)
```

. The bottom window, titled 'Python 3.4.3 Shell', shows the execution of this function. It displays the Python version and architecture, followed by a 'RESTART' message. The user has called the function twice: `average(10,20)` which returned `15.0`, and `average(211,4122)` which returned `2166.5`. The prompt `>>>` is shown at the end of the last line.

```
function.py - G:/Python/Exercise_Files/function.py (3.4.3)  
File Edit Format Run Options Window Help  
def average(num1, num2):  
    average = (num1+ num2)/2  
    print (average)  
  
Python 3.4.3 Shell  
File Edit Shell Debug Options Window Help  
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AMD64)] on win32  
Type "copyright", "credits" or "license()" for more information.  
>>> ===== RESTART =====  
>>>  
>>> average(10,20)  
15.0  
>>> average(211,4122)  
2166.5  
>>> |
```

## Passing Named Values

In our previous exercise we passed two numbers to a function which calculated and printed their average. Since both the parameters were numbers we did not bother about the order in which the parameters were being passed. However, order of the parameter becomes crucial when the list of parameters is long and comprises of different data types. Understand this by a simple example:

### Exercise #80: Understanding Function Parameters

Open IDLE file and type the following code and save the file as parameterfunc.py.

```
def greetingFunc (name, age):  
    print ("Hi %s, you are %s years old" %(name, age))
```

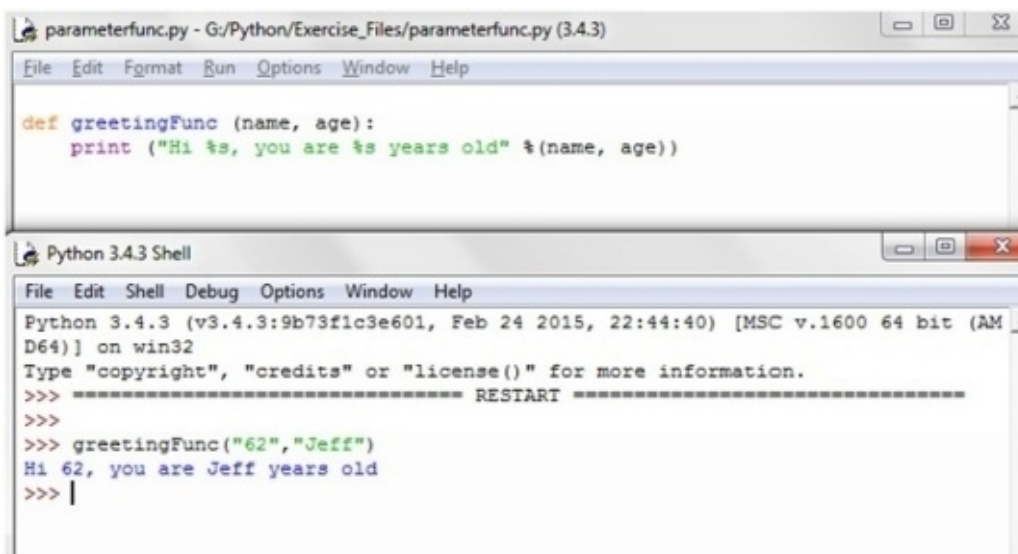
#### Run->Run Module

Now try to call the greetingFunc with the following parameters

```
>>> greetingFunc ("62", "Jeff")  
>>>
```

The function prints “Hi 62, you are Jeff years old”. This is indeed not the type of output we are looking for. Passing parameters in an erroneous order can cause the function to malfunction. Remembering a long list of parameters and their order is difficult for large programs.

Figure 85



You can ensure correct execution of the function by using parameter's name in the function call.



Open the parameterfunc.py file from our last exercise. Run->Run Module

This time we will call the function by changing the parameters. Observe the output for every case.

```
>>> greetingFunc (name="Jeff", age = "21")  
>>>  
>>> greetingFunc(age= "32", name ="Jeff")
```

Here we call both functions by passing named arguments. We specify that “Jeff” is the name while “21” is the age. In second call we change the order of arguments, but still specify the names. Both the function calls return output as desired.

## Returning values

In our discussions so far on functions we have only printed the values. However you can also return a value from a function and use it in your program. To return a value you use the **return** keyword.

Exercise #81: Summation of all the numbers in a range: Returning value from function

Open IDLE text editor. Create a new file and type the following code. Save the file as summation.py

```
def summation(number1, number2):  
    result = 0  
    for eachNum in list(range(number1, number2+1)):  
        result = result + eachNum  
    return result
```

Run->Run module

```
>>>myResult = summation (1,6)  
>>>myResult
```

The code should look familiar by now. In the above exercise we first create a function which takes two numbers as parameters and calculates the sum of all the numbers between the first and second number. To enumerate all the numbers between the two numbers we use the range function. We use the list function to create a list of all the numbers. A for loop iterates and sums all the numbers in the list. However this time instead of printing the result we use the **return** key to return the value of result. In the python shell we define a new variable myResult and assign it the value returned by the function summation. Finally we check the value of myResult value in the shell.

Note:

Any variable you create inside a function to store any information created by the function is only accessible within the block of the function. If you try to access it out of the block of function you will get an error. Such a variable is called local variable.

### *Calling functions within Functions*

We can call a function within a function. To demonstrate this, let's do an exercise. In the exercise we will define two functions, subtotal and member-discount. Sub-total function accepts two arguments

1. dictionary of books purchased by the. The Keys of the dictionary represent the names and their values represent the price of the books.

2. dictionary of customerData. Here the key-value pair of particular interest is key “IsMember”, for which the corresponding value is either true or false.

The sub-total function calculates the sum of all the values of the purchased books. The function also checks if the “isMember” key in customerData dictionary is true, if so, it calls the member-discount function. The member-discount function calculates member-discount. The discount is subtracted from the subtotal.

#### Exercise #82: Calculating the Bill for books

```
booksPurchased = {"Black Beauty": "12.05", "The Hobbit": "15.50", "The Alchemist": "29.99", "the Little Prince": "25.45"}
```

```
customerData = {"First Name": "Jannie", "Last Name": "Greg", "isMember": "False", "bonusPoint": "1000"}
```

```
def memberDiscount(total):
```

```
    discount = 0.10 * total
```

```
    total = total - discount
```

```
    print ("You Got a discount of %s" % discount)
```

```
    return total
```

```
def subtotal (booksPurchased, customerData):
```

```
    priceList = list((booksPurchased.values()))
```

```
    priceNum = []
```

```
    total = 0
```

```
    nettotal = 0
```

```
    for price in priceList:
```

```
        priceNum.append (float(price))
```

```
    for price in priceNum:
```

```
        total = total + price
```

```
    print ("Your total is %s" % total)
```

```
    if customerData ["isMember"] == 'True':
```

```
        subtotal = memberDiscount(total)
```

```
    else:
```

```
        subtotal = total
```

```
    return subtotal
```

**Run->RunModule**

```
>>> subtotal(booksPurchased,customerData)
```

Figure 86



```
greetingfunc.py - G:\Python\Exercise_Files\greetingfunc.py (3.4.3)
File Edit Format Run Options Window Help
booksPurchased = {"Black Beauty":12.05, "The Hobbit":15.50, "The Hitchhiker's Guide to the Galaxy":12.05}
customerData = {"First Name": "Jannie", "Last Name": "Greg", "isMember": "True"}

def memberDiscount(total):
    discount = 0.10 * total
    total = total - discount
    print ("You Got a discount of %d" %discount )
    return total

def subtotal (booksPurchased, customerData):
    priceList = list(booksPurchased.values())
    priceNum = []
    total = 0
    nettotal = 0

    for price in priceList:
        priceNum.append (float(price))

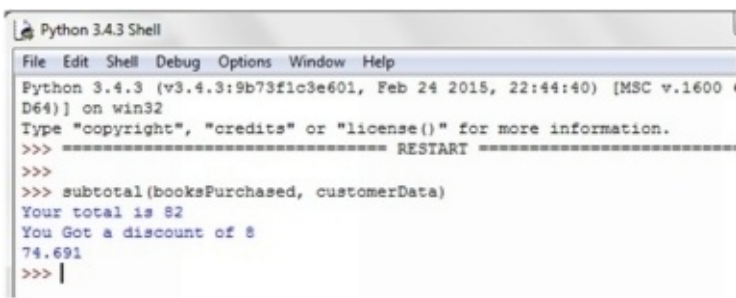
    for price in priceNum:
        total = total + price

    print ("Your total is %d" %total)

    if customerData ["isMember"] == 'True':
        subtotal = memberDiscount(total)
    else:
        subtotal = total

    return subtotal
```

Figure 87



```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 (
D64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
>>> subtotal(booksPurchased, customerData)
Your total is 82
You Got a discount of 8
74.691
>>> |
```

Output when isMember key is set to false

Figure 88

```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
>>> subtotal(booksPurchased, customerData)
Your total is 82
$2.99
>>> |
```

## Chapter Eleven

# Introducing Object Oriented Programming

Up till now in this book we have used data types which have been pre-defined by python such as strings, arrays, floats etc. Practically as our program becomes complex in nature we may require to define our own data types. Think of a situation when you want to create a book-store; having a dedicated type called “book” can be very useful in manipulating data. When working with graphics, you may want to define a type point (x, y).

## Introducing Oops

Object oriented programming, put simply, is a way to group some code together and express complex ideas in a more readable way. Object oriented programming deals with creating classes or user-defines types. Classes provide a template to create objects. Naturally, the first question to come in your mind would be: what are objects? Let's take a closer look.

### *What are Objects?*

Objects in programming are not much different from the objects in real life. Take a chair for example. A chair has certain characteristics, such as some sort of legs, a seat to sit, a shell/back to rest one's back and some sort of arm rests. Besides these characters, there are certain actions which can be performed on a chair, such as sitting on it, dragging and repositioning it, you may touch the seat's cushion to check for comfort ability etc. Likewise, objects in programming can be regarded as collection of data and associated behaviors. Let's try to translate the chair object in a pseudo code.

Attributes/characteristics

Legs

Arm-rest

Seat

Behaviors/functions

Sit ( )

Drag ( )

Comfort ability\_check ( )

A more relevant example of object in terms of programming can be of an object representing a customer; having first name, last name, address, some ID etc. The customer object can perform certain actions such as, buying a book in your book store, or may be marking a bunch of books as favorite, adding them to wishlist etc.

NOTE: Why create objects?

You might ask why is it important to create an object for a customer when we have dictionaries and lists to maintain user data- as we did in previous examples. While lists and dictionaries are great way to organize data, keeping track of user/customer attributes becomes difficult to manage using lists/dictionaries as the users and their attributes increase in number. Let's assume that you use a dictionary to maintain user data in an arrangement like:



KEITH	BEN
['Keith', 'Jackman', 'keithjack@hotmail.com', '0032145', '182345']	['Ben', 'Smith', 'mrSmith@hotmail.com', '001656', '182324']

Dictionary Keys

KEITH	BEN
['Keith', 'Jackman', 'keithjack@hotmail.com', '0032145', '182345']	['Ben', 'Smith', 'mrSmith@hotmail.com', '001656', '182324']

List corresponding to each key is a list. List saves first name, last name, email, phone # and ID

Let's investigate the issues we may face in this arrangement.

To access the individual data item, let's say the email of the customer you have to use:

```
Userdata = user["Ben"]
```

```
Userdata[3]
```

3 is simply index of a list. Since the index number does not say much about the type of data stored in the list we need to either document the fact that email address is stored on index 3,; or we have to remember it by heart which is difficult as the program becomes large.

Consider an alternative arrangement where:

USER
<u>firstName</u> = 'Keith', <u>SecondName</u> = 'Jackman', <u>Email</u> = 'keithjack@hotmail.com', <u>phoneNumber</u> = '0032145', <u>ID</u> = '182345']

```
User.firstName = "Keith"
```

```
User.Email = "keithjack@hotmail.com"
```

The figure represents a user object at a very basic level. In the above example, firstName,

SecondName, Email, phoneNumber are attributes of the object user. Besides attributes there are functions associated with objects called method; which we will explore shortly in this chapter.

In contrast to the previous approach, user data is saved as named attributed which makes properties easy to access, modify and maintain.

### ***What are classes?***

Continuing with our example of chair; there are different types of chairs; swivel chair possessing 5 legs each with a spinning wheel; a lawn bench, high chair, deck chair etc. All of the chairs share certain characteristics in common i.e. some sort of legs, seat, back support and arm rests. These attributes comprise the blue-print of the class called chairs i.e. every chair must have all these attributes. This blue print is called a class. Class provides a template and all the objects created from the class share the same pattern.

## Creating classes in python

To create a class in python you use the **class** keyword.

```
class <class_name> ( object ):
```

```
    # attributes of class
```

```
    # methods of class
```

### Exercise #83: Creating classes

Create a file in IDLE text editor and save it as customer\_class.py

```
class customer(object):
```

```
    firstName = ""
```

```
    lastName = ""
```

```
    customerID = ""
```

**Run->Run Module. In Python Shell type the following code. (Do not close the shell after execution )**

```
customer1 = customer()
```

```
>>> customer1.firstName = "Ben"
```

```
>>> customer1.lastName = "Jackman"
```

```
>>> customer1.customerID = "4822"
```

```
>>>
```

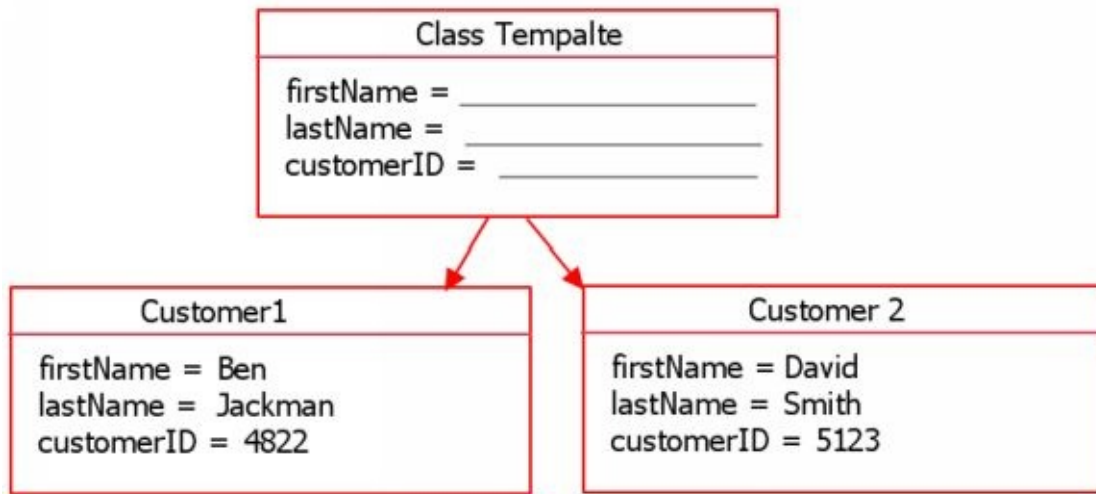
```
>>> customer2 = customer()
```

```
>>> customer2.firstName = "David"
```

```
>>> customer2.lastName = "Smith"
```

```
>>> customer2.customerID = "5123"
```

Let's investigate the code. We have created a custom data type called customer in customer\_class.py. The class has attributes or properties which will be distinct for every customer. The first name, last name and customer ID are set to empty strings in the class definition. When we will create objects of this class we will replace the values for individual objects (or customers) with empty strings. You may consider the class customer as a template and the name attributes as blank spaces; each object of the class will have its distinct information saved in those blank spaces.



Later in the exercise, in the python shell we create object of the customer class. Creating object of a class is much like defining variable of a specific type. We first create an object and name it customer1 and set its type to customer. Then we set all the attributes for that objects. Note:

1. To create an instance or an object of the class we use the following code.  
`<name of object> = <class_name> ( )`  
That is we are creating a variable of a custom type (here it is customer).
2. To access the attributes or the methods of a class we use . (dot)operator like follows.  
`<name of object> . <attribute_Name>`

Having created objects (customer1 and customer2) of the class customer, let's try to access the object's attributes using the dot operator.

#### Exercise #84: Creating classes

Create a file in IDLE text editor and save it as customer\_class.py

**Run->Run Module. In Python Shell type the following code. (Do not close the shell after execution )**

```
1.      >>> customer1.firstName
2.      >>> "Ben"
3.      >>> customer1.lastName
      >>> "Jackman"
4.      >>> customer1.lastName = "William"
5.      >>> customer1.lastName
      >>> "William"
6.
7.      >>> customer2.firstName
      >>>
8.
9.      >>> customer2.lastName
10.     >>>
```

In this exercise we used the dot operator to access and change attributes of the object. We proceed as:

1. In line 1 we access the first name of object **customer1** by typing *customer1.firstName*. You may read it aloud as “first name *of* customer1”.
2. In line 2 the shell prints the value stored in the attribute.
3. In line 3 and 4 we access and print the value of last name.
4. In line 5 we assign a new value to the lastName attribute of customer1 by providing a new value. In Line 8 through 10 we print values of same attributes (firstName and lastName) for the second object i.e. customer2.

### ***Adding Methods***

Methods are simply functions contained within the body of a class. We use functions to set the value of object and manipulate the objects. Methods are like regular functions; the only difference is that they take at least one parameter i.e. self. Let’s add methods to our class customer to understand their functionality.

#### **Exercise #85: Creating classes**

Open the `customer_class.py` file and add the following code under class definition. Take note of indents

```
#code to add
def printDetails( ):
    print (“Hi there!”)
#complete class definition
class customer(object):
    firstName = ""
    lastName = ""
    customerID = ""
    def printDetails( ):
        print (“Hi there!”)

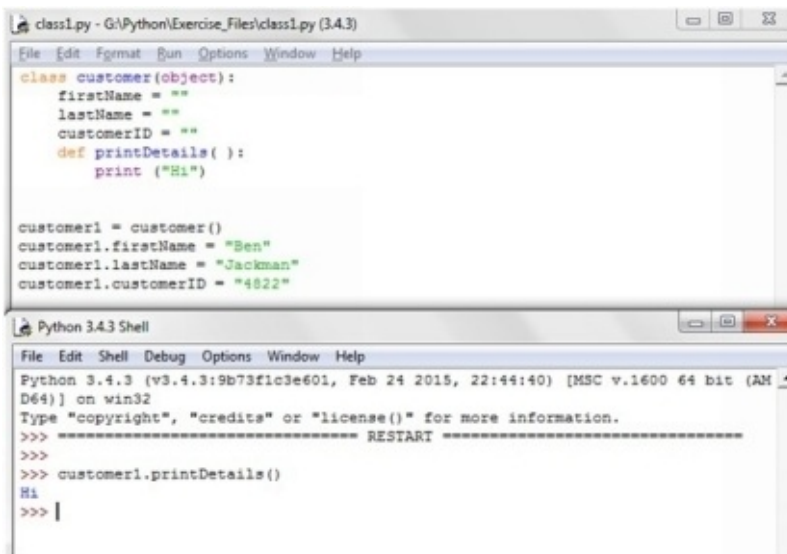
# create an object inside python file.

customer1 = customer()
customer1.firstName = “Ben”
customer1.lastName = “Jackman”
customer1.customerID = “4822”
```

**Run->Run Module. In Python Shell type the following code**

```
>>> customer1.printDetails( )
```

Figure 89



The screenshot shows two windows from a Python IDE. The top window, titled 'class1.py - G:\Python\Exercise\_Files\class1.py (3.4.3)', contains the following code:

```
class customer(object):
    firstName = ""
    lastName = ""
    customerID = ""
    def printDetails( ):
        print ("Hi")

customer1 = customer()
customer1.firstName = "Ben"
customer1.lastName = "Jackman"
customer1.customerID = "4822"
```

The bottom window, titled 'Python 3.4.3 Shell', shows the execution of the code:

```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
>>> customer1.printDetails()
Hi
>>> |
```

For this example we simply write a method which prints a string. Notice how we have indented the function to make it part of the class definition. Having defined our class in class1.py we continue to python shell where we create an object of the class. Like attributes, we use the dot operator to call a method on the object of the class followed by ( ). The printDetails( ) method does not take any arguments and simply prints a string. To call the method we used customer1.printDetails( ).

#### Exercise #86: Creating classes

Open the customer\_class.py file and add the edit the code under printDetails method definition. Take note of indents

#code to add

```
def printDetails(self):
    print ("Customer Name: " + self.firstName + self.lastName)
    print ("Customer ID: " + self.customerID)
```

#complete class definition

```
class customer(object):
    firstName = ""
    lastName = ""
    customerID = ""
    def printDetails(self):
        print ("Customer Name: " + self.firstName + self.lastName)
        print ("Customer ID: " + self.customerID)
```

# create an object inside python file.

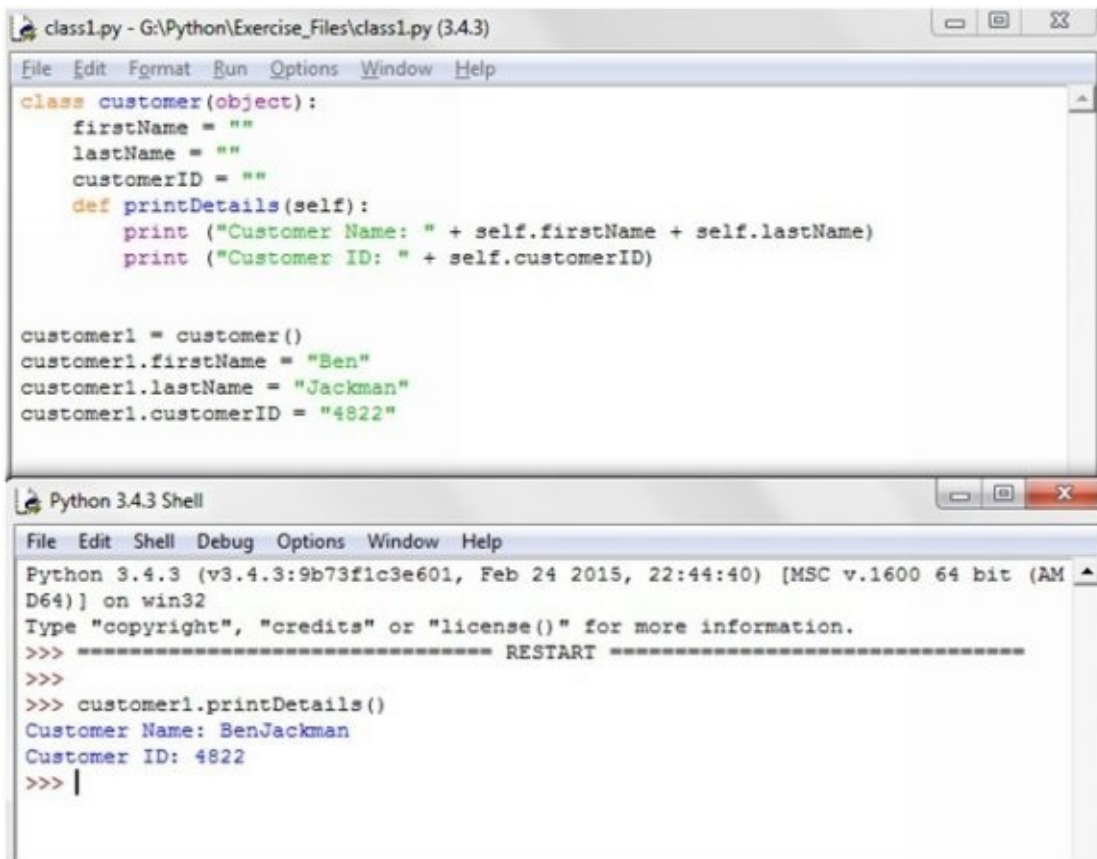
```
customer1 = customer()
```

```
customer1.firstName = "Ben"
customer1.lastName = "Jackman"
customer1.customerID = "4822"
```

Run->Run Module. In Python Shell type the following code

```
>>> customer1.printDetails()
```

Figure 90



The screenshot shows two windows from a Python IDE. The top window, titled 'class1.py - G:\Python\Exercise\_Files\class1.py (3.4.3)', contains the following code:

```
class customer(object):
    firstName = ""
    lastName = ""
    customerID = ""
    def printDetails(self):
        print ("Customer Name: " + self.firstName + self.lastName)
        print ("Customer ID: " + self.customerID)

customer1 = customer()
customer1.firstName = "Ben"
customer1.lastName = "Jackman"
customer1.customerID = "4822"
```

The bottom window, titled 'Python 3.4.3 Shell', shows the execution of the code:

```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
>>> customer1.printDetails()
Customer Name: BenJackman
Customer ID: 4822
>>> |
```

Let's investigate the syntax of the method we added to the class definition.

For easy referencing I have provided the code snippet here:

1. `def printDetails(self):`
2. `print ("Customer Name: " + self.firstName + self.lastName)`
3. `print ("Customer ID: " + self.customerID)`

In line 1 we have created a function within class definition using the `def` keyword. The

function takes an argument *self*. In our previous exercise we simply printed a line of text; however here, we need to print the details of the object. To print the attributes of the object we need to supply the object as a parameter to the print function. The *self* keyword refers to the object ***itself***; here we are passing all the information possessed by the object to the printDetails function for printing. The function itself uses string concatenation (the plus operator, as we discussed before) to print the attributes of the object. When we will call the method on the customer1 object, the interpreter will bind the parameter self with the object (here customer1) so that the method's code can refer to the object by name.

Note:

The example of earlier version of printDetails function (method which takes no argument) is presented for the explanation of self argument only. Practically, each method definition must include a first named parameter **self**.

This introductory book will not explore classes and functions any further. However the basic knowledge provided here will provide you with a strong base for advanced programming with objects and classes.



## Python modules

We just explored how to create classes and objects. There are number of commonly used aspects of any program such as date time, mathematical operations, working with graphical elements, etc. Since all of these functionalities hardly differ from program to program, python comes with pre-written classes and functions called modules to give you a jump start with commonly used features. Therefore to integrate functionality in your program you do not need to write function and classes from scratch you can simply import functionality or borrow code from python's library.

Python's additional functionality comes in packages. Each package includes modules. Packages can be regarded as folders while modules are files within a folder. Different packages provide different type of functionalities; some packages provide modules for solving mathematical problems, while some provide methods for developing user interfaces etc.

To import or borrow functionality from the pre-written code we use the **import** keyword.

`import module`

`from module import class`

`from module import function`

## Working with Mathematical function

Python provides math module for performing mathematical computations. We can simply import the math module and use the built-in functionality instead of writing the code from scratch.

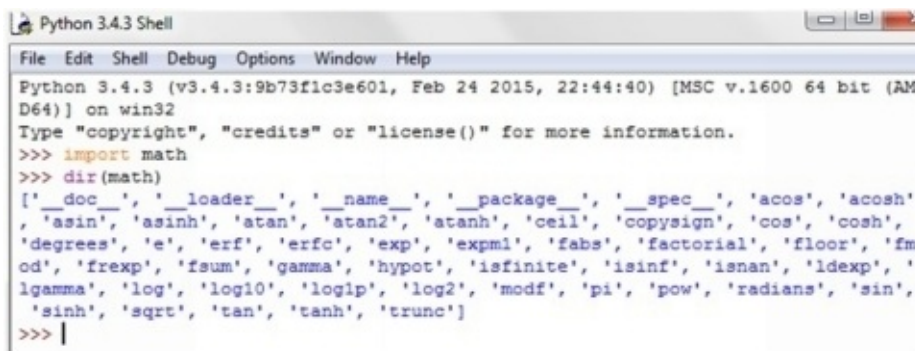
In python shell type:

```
>>>import math
>>>dir(math)
```

The dir command (above) enlists the directory of all the functionalities provided by the math module. A complete list and description for all the mathematical operations can be found here:

<https://docs.python.org/2/library/math.html>

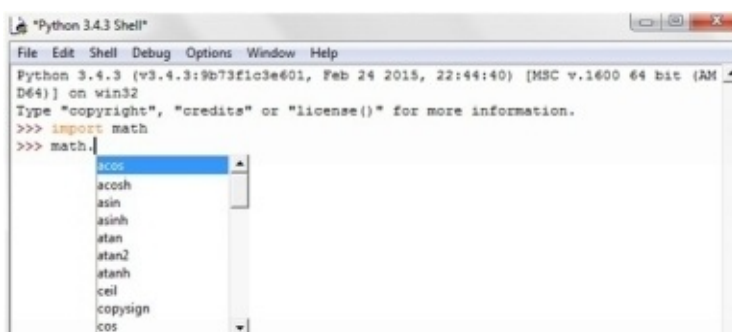
Figure 91



```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import math
>>> dir(math)
['_doc', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'hypot', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
>>> |
```

In the Python shell you may access any functionality by typing the math. operator.

Figure 92



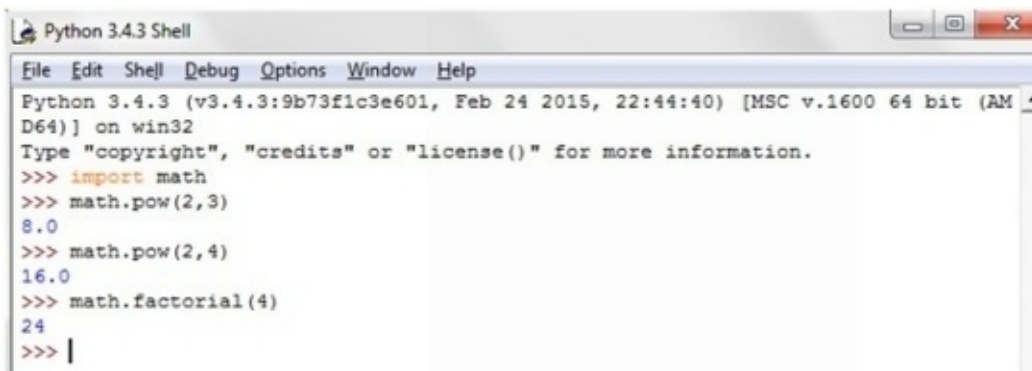
```
"Python 3.4.3 Shell"
File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import math
>>> math.
acos
acosh
asin
asinh
atan
atan2
atanh
ceil
copysign
cos
```

In python shell type:

```
>>>import math
>>>math.pow(2,3)
>>>math.pow(2,4)
>>>math.factorial (4)
```

The pow (x,y) function takes two arguments x and y and returns the x raised to the power of y. Likewise factorial function takes single argument and returns its factorial.

Figure 93

A screenshot of a Python 3.4.3 Shell window. The window has a title bar that says "Python 3.4.3 Shell" and a menu bar with "File", "Edit", "Shell", "Debug", "Options", "Window", and "Help". The main text area shows the following text:

```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import math
>>> math.pow(2,3)
8.0
>>> math.pow(2,4)
16.0
>>> math.factorial(4)
24
>>> |
```

## Random number module

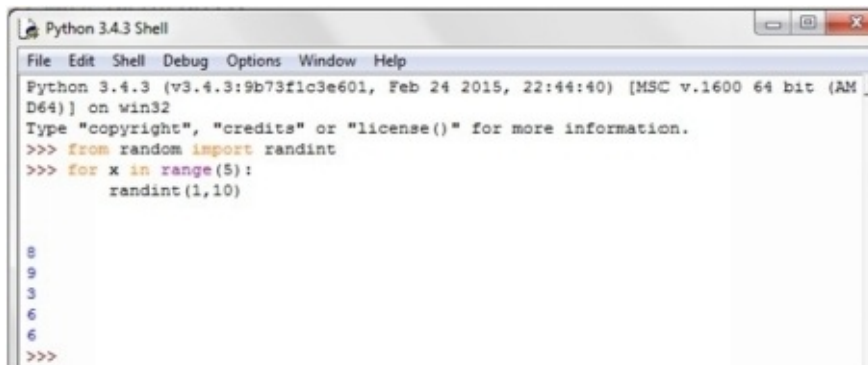
You often encounter situation when you need to generate a random number. Python provides a Random module to facilitate you with this task. To generate a random number within a range we use randint ( a, b ) function. randint (a,b) generates a random integer N such that  $a \leq N \leq b$ .

Exercise # 87: In the Python Shell type:

```
>>> from random import randint(a,b)
>>> for x in range(5):
    randint(1,10)
>>>
>>>
```

In the above exercise we use for loop to generate 5 random numbers using the randint( ) function. The randint function takes two arguments 1 and 10 and returns random numbers within the range 1-10.

Figure 94



```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> from random import randint
>>> for x in range(5):
    randint(1,10)
8
9
3
6
6
>>>
```

## Working with date and time

Python also provides a datetime module to work with date and time. The module datetime provides number of classes which allow you to manipulate the date and time. A complete documentation of available classes in module datetime can be found here: <https://docs.python.org/2/library/datetime.html>

In the following exercise we will use datetime class which is a combination of a date and a time with attributes: year, month, day, hour, minute, second, microsecond, and tzinfo.

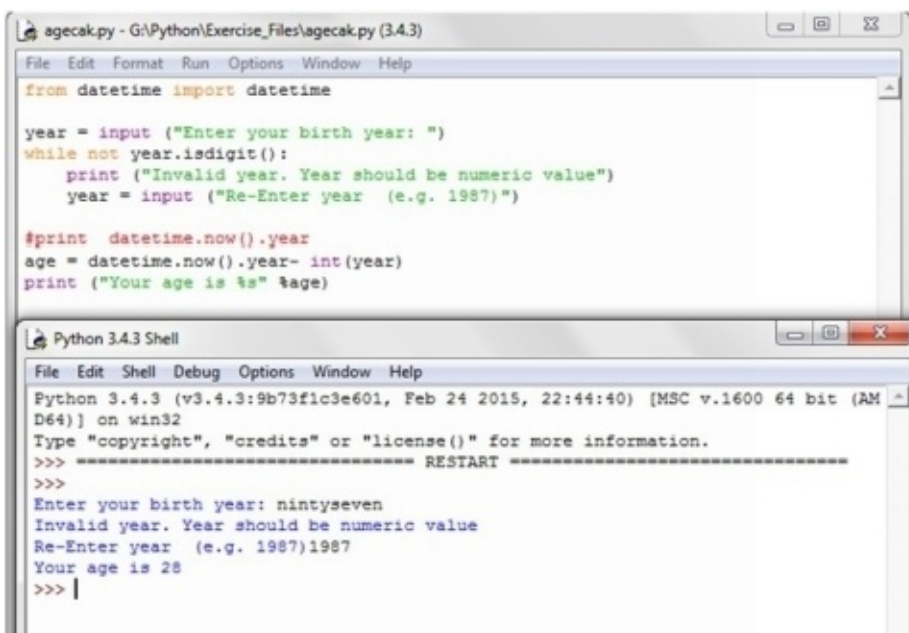
### Exercise #89: Age Calculator

```
from datetime import datetime

year = input("Enter your birth year: ")
while not year.isdigit():
    print("Invalid year. Year should be numeric value")
    year = input("Re-Enter year (e.g. 1987)")

#print datetime.now().year
age = datetime.now().year- int(year)
print("Your age is %s" %age)
```

Figure 95



The screenshot displays two windows from a Python 3.4.3 IDE. The top window, titled 'agecak.py - G:\Python\Exercise\_Files\agecak.py (3.4.3)', contains the Python code for the age calculator. The bottom window, titled 'Python 3.4.3 Shell', shows the output of the script. The user enters 'nintyseven', which is rejected as invalid. After entering '1987', the script calculates and prints the age as 28.

```
agecak.py - G:\Python\Exercise_Files\agecak.py (3.4.3)
File Edit Format Run Options Window Help

from datetime import datetime

year = input("Enter your birth year: ")
while not year.isdigit():
    print("Invalid year. Year should be numeric value")
    year = input("Re-Enter year (e.g. 1987)")

#print datetime.now().year
age = datetime.now().year- int(year)
print("Your age is %s" %age)

Python 3.4.3 Shell
File Edit Shell Debug Options Window Help

Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
Enter your birth year: nintyseven
Invalid year. Year should be numeric value
Re-Enter year (e.g. 1987)1987
Your age is 28
>>> |
```



# Conclusion

This book touches the very basics of programming concepts using Python. The foundation you have built using this book will prove to be handy as you progress to more advanced concepts and python applications. This section overviews various common applications of python. From here, you may proceed to your area of interest and pick up field-specific python book.

## Potentials of Python

Python now one of the popular scripting languages, is being widely adopted as programming language for teaching programming concepts in schools and colleges due to its readability and ease of learning. Python has multi-dimensional utility; From websites and web apps to complex computer vision applications, python is being extensively used. Below you can find a short list of popular applications written in Python.

1. Google apps. Most of Google applications are written in Python.
2. YouTube. Google's platform for sharing videos is written in python.
3. Paint-shop pro a full-fledged image editing software is written in Python.
4. Ubuntu Software center
5. Open source software 3D graphic and animation software Blender
6. Ebook creator software Calibre
7. BitTorrent

Python can be used for:

1. Web development with Django
2. Web apps with Flask and Neo4j
3. Python enabled web servers
4. Mobile apps which run on different platforms including Android
5. Build GUI based programs such as games, softwares
6. Advance computer vision and OpenCV applications which use complex image and pattern recognition, stereo imaging , augmented reality etc.
7. Advance machine learning applications.
8. You can also create full-fledged independent gadgets using Python with Raspberry Pi



# **One Last Thing...**

We would love to hear your feedback about this book!

If you enjoyed this book or found it useful, we would be very grateful if you would post a short review on Amazon. Your support does make a difference and we read every review personally, so we can get your feedback and make our books even better.

If you would like to leave a review, all you need to do is click the review link on this book's page on Amazon [here](#).

Thank you for your support!