PROGRAMMING IN PYTHON

S.RAFEEQ AHAMED.M.Sc.&
M.RIYAZ MOHAMMED M.C.A.,M.Phil.,

CONTENTS

Introduction to Computer Languages

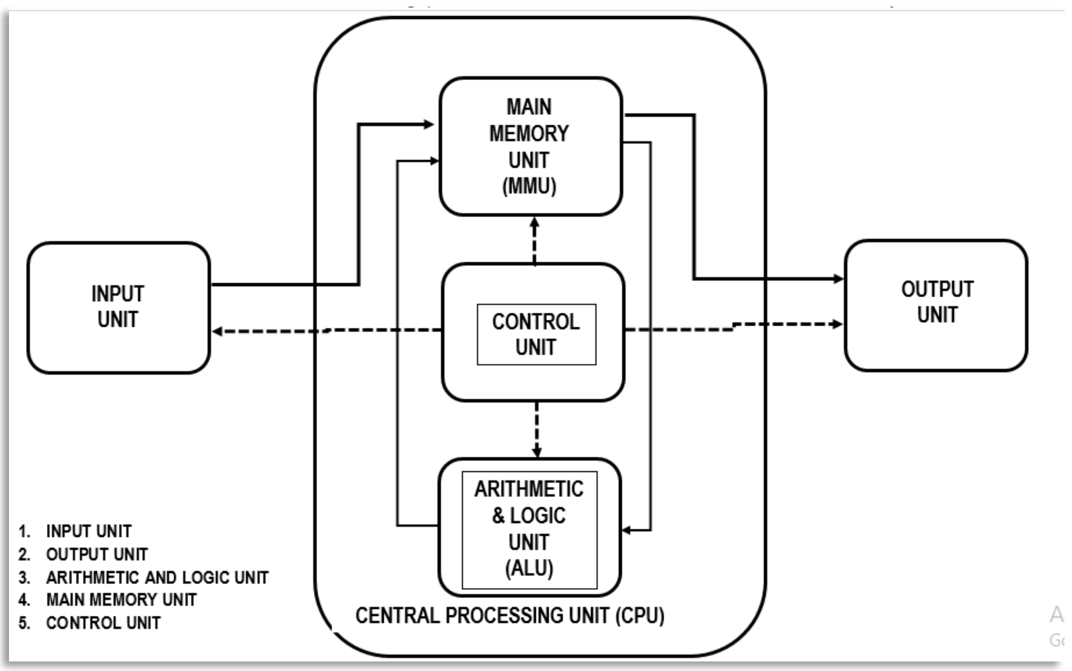
Computer Architecture

A computer can be divided into 3 main units as follows:

1. INPUT UNIT – Used to input data for processing. The input can be given to the computer using any one of the input devices such as Keyboard, Mouse, Scanner, Microphone, Tablets etc.

2. CENTRAL PROCESSING UNIT (CPU) – All the processes are done within this unit. The CPU consists of following sub-units inside to perform various tasks those are processed electronically inside the computer. The 3 sub-units of the CPU are described below:
    a. Main Memory Unit (MMU) – All the unprocessed, processed data (All questions and answers) and commands are stored in this Memory unit only.
    b. Arithmetic & Logic Unit (ALU) – This is the main processing area.
    c. Control Unit (CU) – This will control the operations of all units of a computer.

3. OUTPUT UNIT – All the information to be given to the user will be done through this unit. The output can be obtained from any of the devices such as Monitor screen, loudspeaker, printer, plotter etc.



Programming Fundamentals

A Program is defined as a collection of instructions those are commonly called as commands or statements.
All the instructions inside a program are processed by the computer line by line in a top-down order.Usualy a program may contain only collection of command and / or may have combination of commands and data those are unprocessed or partially processed.

Compiler and Interpreters

Commands of the program are in form of easily understandable by a programmer. They are made up of englshi phrases (Example: Input, Print, continue, break etc.) The program written by a programmer is called as High level language. This is not understood by the computer. Computer can understand only machine language that may contain binary numbers, hexadecimal numbers etc. To make a computer to understand the high level language commands of a program, a translating mechanism that can translate high level language to low level language and vice versa. This translating mechanism is a software called as compiler or interpreter. The differences between compiler and interpreter are:

- Compiler will read the entire program and translate it to machine language (or low level language) while interpreter compiles line by line of a program.
- Interpreters are always come with its own program editor. For compiler, we can use any editor to write the high level language and can translate it.

Compilation of a program

A program written in high level language is known as source code. A compiler reads the source code and does spelling check. Each phrase in a line of a source code is technically called as ***Lexus***and analyzing it is known as ***Lexical Analysis***. It is like checking a sentence of any languages like English, French, Tamil etc., for spelling check.

After ensuring that in a line there is no spelling mistake, the compiler goes for another analysis called ***Syntactic Analysis.*** This is to check that the entire line (or sentence) is grammatically correct or not.

If any of the above analysis is failed, the computer will ignore the process of translating the source code to machine code and throws an error message called ***Syntax Error.***

If both analysis are successfully passed, then the compiler will generate first level machine language code. This code is called ***object code.*** This can be executed by the compiler and it gives the result for which the source code is written. However, object codes are executable only by the compiler. So, we need the compiler to run your program that is translated to object code. In other words, object program is still compiler dependent and we need the compiler to run the program. To make the object program as compiler independent, that is to make the object program to be executed directly by the operating system, it has to undergo another process called ***linking.*** Linking is a process that links necessary libraries of a compiler to make it as a standalone ***executable*** file. This process is also called making executable program. After this process, there will be another file created named .EXE file. This file can be directly executed in a computer without a compiler.

The below are the processes carried out by a compiler:

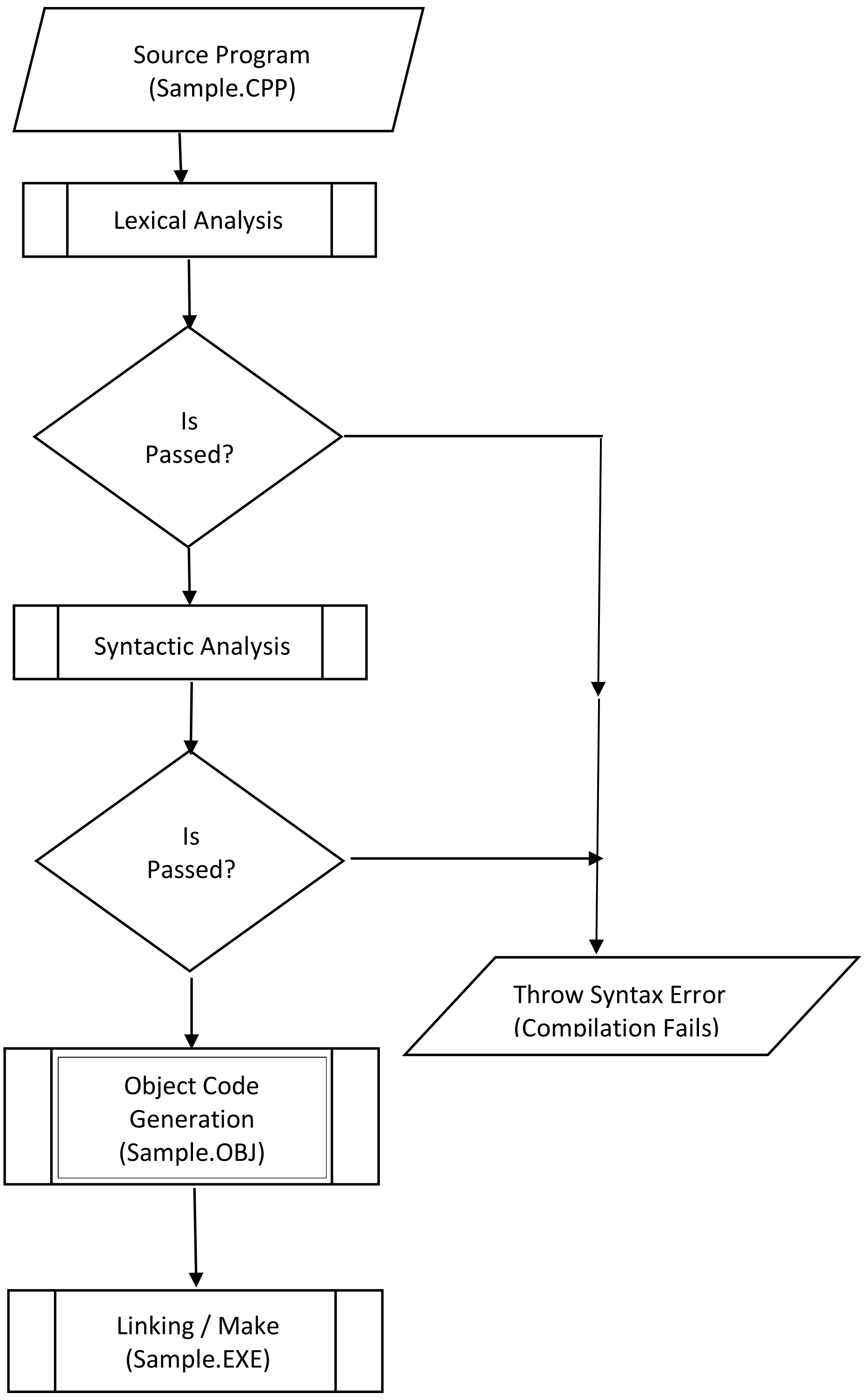
Lexical Analysis - Reads and Checks the source code (Sample.CPP, Sample.COB etc.,) for spelling check. On successful completion of lexical analysis, proceed to next step. Throw syntax error and abort compilation if failed.

Syntactic Analysis – Reads and checks the source program for grammar. If passed, proceed to object code generation. Throw syntax error and abort compilation if failed.

Object code generation – If both the above analysis are passed, the compiler will create object code equivalent to the source code and the code will be saved in a file named (Sample.OBJ). This file can be executed with the help of the compiler. The program written can be executed with the compiler.

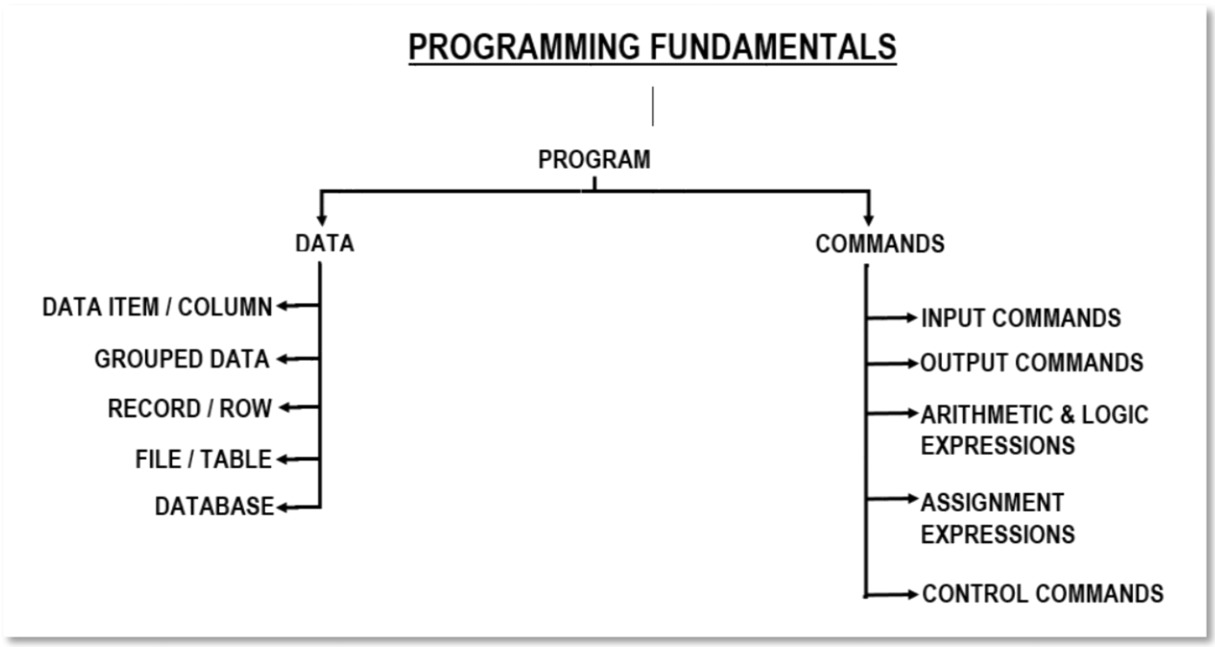
Make / Link – This process links some of the compiler's libraries with the object code to make the program as a standalone program that can be executed without the help of the compiler. Link process will create a file named Sample.EXE. It can be executed on any computer using same operating system even if the compiler is not available.

The below flowchart explains the process of a compiler and various formation of the source program:



The instructions or commands of a program can be logically categorized as follows:

1. Input Commands – Used to input data to the computer.
2. Output Commands – Used to get the information from the computer.
3. Control Commands – These commands are to control the flow (order) of the commands execution. The control commands can be classified as below:
    a. Unconditional Control Commands – To change the default order of line by line execution of the commands.
    b. Conditional Control Commands – To change the order of execution of the commands based on a condition.
    c. Repetitive Control Commands – To repeatedly execute a set of commands.
4. Sub-programs and functions – To avoid redundant set of commands written in a program, the redundant steps are separately written as a small program or a function and can be called as many times as needed in a program wherever it is required. The main difference between sub-program and function is that a sub-program will not return any values to the main program after the execution but a function will return a value to the main program.



Generations of Programming Languages

Languages for Text Mode Operating Systems

Earlier stage high level languages are used in operating systems like DOS, UNIX, CPMs those were used in the computers without graphics interface. They use monitors of text mode only. Few of the languages are BASIC, FORTRAN, COBOL, PASCAL, C etc. If you know how to write statements, including variable declarations and loops, in one of these three languages, then you have a head start in learning the others. However, many other details differ among the languages. In particular, the data types and libraries available are considerably different.

Languages with DBMS concept

As a development in data handling a new concept named Database Management system was introduced in few languages like dBASE, Clipper etc., to handle large volume of data in a better way.

Languages with GUI

Further, the operating systems like DOS were enhanced to handle Graphics also with a Graphics User Interface (GUI) program called Windows. Hence, few of the languages also included features to handle GUI. Visual Basic, Visual C++, Visual dBASE etc., are examples for such languages. They included Modules, Properties, Objects, Methods to write visual programs.

RDBMS

As a next stage of development in the DBMS concept, RDBMS (Relational Database Management System) was introduced and Microsoft introduced an RDBMS with GUI called Ms Access. Using MsAccess, we can write GUI Programs with RDBMS concepts. RDBMS itself uses a language to manipulate data. This language is called Structured Query Language (SQL). Databases with RDBMS concept like Ms SQL Server, Sybase, Oracle, MySQL etc. uses SQLs to manipulate data. They provide driver software with that, programming languages can connect to the database and access the databases for input and output operations. The concept of connecting a languages to work with databases by providing drivers are called ODBC (Open Database Connectivity), JDBC and Python Database Connectivity.

Introduction to Python

Python is very powerful and easy to learn language.

Features of Python

- It has a useful combination of features those, have made it very popular in recent years.
- It is a platform independent language that runs a onetime developed code on any OS.
- It can be used to develop Web applications.
- It can be used for files handling.
- It can connect to any database and can perform CRUD (Create, Read, Update and Delete) operations on any database.
- It can be used to handle big data and complex mathematics.
- It has simple syntax avoiding complicated rules and data declarations.
- Python is compatible for procedural programming, Object oriented programming (OOPS) or functional programming.
- Python code can be executed as soon as it is written as it runs on an interpreter facilitating prototyping is easy.

Python Compiler Installation

Many PCs and Macs will have python already installed. To check if you have python installed on a Windows PC, search in the start bar for Python or run the following on the Command Line (cmd.exe):

```
C:\Users\Your Name>python --version
```

To check if you have python installed on a Linux or Mac, then on linux open the command line or on Mac open the Terminal and type:

```
python --version
```

Python compiler can be downloaded for free from the following website: https://www.python.org/

Core Python Language

Getting Started

Python is an interpreter programming language.It provides its own program editor to type in the code and has built in option to run the code. Also,a developer can write Python (.py) files in a text editor and then put those files into the python interpreter to be executed.The way to run a python file is like this on the command line:

*C:\Users\*Your Name>*python MyPython1.py*

Where "MyPython1.py" is the name of the python file.

As mentioned earlier, Python program can also be written using any text editor. It is also possible to write Python program in an Integrated Development Environment (IDLE) of Python or using any IDE such as Thonny, Pycharm, Netbeans or Eclipse.

The program MyPython1.py, the below code is typed:

```
print("My first python program")
```

Now, type the below command in command prompt:

```
C:\>python Mypython1.py
```

The output is:

```
My first python program
```

Python syntax can be executed by writing directly in the Python Shell Command Line:

```
>>>print("My first python program ")
My first python program
```

Python Comment Lines

Comment lines are line that can be used to explain that for what purpose the program is written. Comment lines are not executed by the program. Comments can be used to make the code more readable. It can be used to prevent execution of some portion of codes as an act of testing code.

Example

```
#print("Hello, World!")
print("Hi, C3PO!")
```

Creating a Comment

Comment line starts with a **#** character. When Python reads this line it will ignore the line unexecuted.

Example

```
#This is a comment
print("I am a python code!")
```

Comments can be placed at the end of an executable code line as below:

Example

```
print("I am a python code!")  #This is a comment
```

Multi Line Comments

Python has no special syntax for multiline comments. To add a multiline comment, **#** can be inserted at the beginning of each line

```
#This is a comment
#written in
#more than just one line
print("Hello, World!")
```

or a multiline string can be used. Python will ignore string literals that are not assigned to a variable. So, a multiline string (triple quotes) can be used as multiple line comment as below:

```
"""
This is a comment line
written in
more than just one line
"""
print("Hello, World!")
```

Python Variables

A variable is a container to hold a data.

Example:
A=5
B=”Kaveriyin Pudhalvan”

In the above examples **A** is the variable to store an integer number while B stores a stream of characters (String) of non-numeric data. Unlike other languages like C++, java etc., Python has no declaration formalities for a variable.

A variable is created immediately when a value is assigned to it and the type of variable is understood by Python from the value stored to the variable. In other words, variables do not need type declaration and can even change type after they have been set.

Example

```
x = 5
b = "Raja RajaChozhan"
print(x)
print(b)
x = "Sally"
print(x)
```

The output will be:

```
5
Raja RajaChozhan
Sally
```

Variable Names

The rules to follow when naming a variable:

- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number

- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Variable names are case-sensitive (age, Age and AGE are three different variables) A variable can be a single letter name or can be in a descriptive way with more letters.

Example

Valid variable names:

```
myname = "Quigan Jinn"
my_name = "Leia"
my_name = "Anakin"
myName= "Lando"
MYNAME = "Carlission"
myname2 = "John Williams"
```

Example

Invalid variable names:

```
2myname = "Palpatein "
my-name= "Darth Sidius "
my name= "The evil emperor"
```

Note that variable names are case-sensitive.

Naming Variable with multiple words

Variables can be named with multiple words. There are 3methods followed when naming a varable such as ***Camel Case, Pascal Case*** and ***Snake Case***.

Camel Case

When naming using multiple words, use the capital letter for the first letter of each word except the very first letter of the variable name is ***Camel Case*** method.:

```
myFullName = "Ian Flemming"
```

Pascal Case

Naming Each word with a capital letter as the first letter for all the words is ***Pascal*** method

```
MyFullName = "Monty Norman"
```

Snake Case

Naming each word separated by an underscore is ***Snake*** method

```
my_full_name = "Desmond Lewylline"
```

Python allows to assign multiple values to multiple variables in one line:

Example

```
f1, f2, f3 = "Tie Fighter", "Imperial Walker", "Death Star"
print(f1)
print(f2)
print(f3)
```

Note: Make sure the number of variables matches the number of values, or else you will get an error.

Python allows to assign a single value to multiple variables in one line:

Example

```
a = b = c = "Alderon"
print(a)
print(b)
print(c)
```

Python Data Types

Python data types are listed in the below table:

PYTHON DATA TYPES		
Data Type	**Bytes used**	**Description**
byte	1	-128 to 127
str		
int	4	-2^{31} to 2^{31}-1
long	8	-2^{63} to 2^{63}-1
float	4	Up to 7 decimal digits
double	8	Up to 15 decimal digits
bool	1	True or False
complex		
tuple		
bytearray		
range		
set		
frozenset		
dict		
list		

Built-in Data Types

In programming, data type is an important concept. Variables can store data of different type of data. Python has the following data types as built-in by default, in the below mentioned categories:

Data Type	Keyword
Text	Str
Numeric	int, float, complex
Sequence	list, tuple, range
Mapping	Dict
Set	set, frozenset
Boolean	Bool
Binary	bytes, bytearray, memoryview
3None	NoneType

Python Operators

Bitwise Operators

Number System

The number system we use in daily life is called decimal number system, In decimal number system we use 10 numerical figures (0,1,2,3,4,5,6,7.8.9) to count numbers as follows: 0,1,2,3,4,5,6,7,8,9, to count next digit, we reset the Least significant digit to zero and we increase the value of most significant (second higher digit) by 1 and thus we get the next value to 9 as 10 and we keep counting further as 11,12,13,14,15,16,17,18,19,20…99,100,…199,200 and so on.

To understand the Bitwise Operator and the usage of Octal and Hexadecimal numbers, it is important to know how to convert a given decimal numbers to binary numbers, to Octal numbers, to Hexadecimal Numbers and Vice-Versa.

Binary Number System

We also have other number systems called Binary Number System that has only 2 numerical figures (0 and 1) to count numbers, , In binary number system we count numbers as follows: 0,1, to count next digit, we reset the Least significant digit to zero and we increase the value of most significant (second higher digit) by 1 and thus we get the next value to 1 as 10 and we keep counting further as 11, 100, 101, 110, 111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111 and so on. The binary data measurement unit is given in the below table:

BINARY TABLE	
1 Bit	Either 0 or 1
1 Byte	8 Bits
1024 Bytes	1 Kilo Byte (KB)
1024 KBs	1 Mega Byte (MB)
1024 MBs	1 Giga Byte (GB)
1024 GBs	1 Tera Byte (TB)

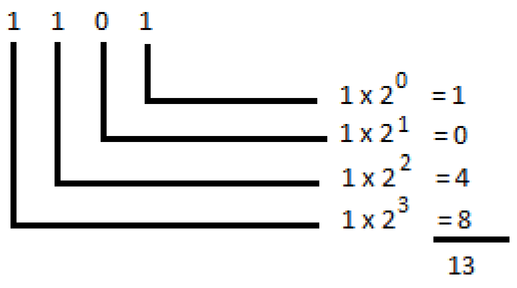
Converting a decimal number to a binary number

Let us take an example of a decimal number 4 and let us convert it to binary number. Keep dividing the given number 4 by 2 and note down the reminder. Repeat this procedure until the given number cannot be divided by 2 further and note down the reminder, Assemble the reminders from bottom to top direction as shown below:

```
2 | 4
2 | 2 –0        = 1 0 0.
    1 – 0
```

So, 100 is the binary equivalent for the decimal number 4. The answer is usually written as $(4)_{10}$= $(100)_2$. It can be shown in 8 bit number, it can be written as 00000100. (Note: Adding leading zeros to a whole number will not change its value)

Now, let us see how the binary number can be converted to its decimal equivalent,
Let us take a binary number $(1101)_2$for conversion.



Multiply each BIT (BInarydigiT) by 2 power the digit position number and add all together to get the decimal equivalent to the binary number.

The Bitwise operation can be performed by referring the below truth tables of Binary Operations:
Binary Truth Tables for AND,OR and XOR operations

AND		
A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

OR		
A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

XOR		
A	B	X
0	0	0
0	1	1
1	0	1
1	1	0

NOT	
B	X
0	1
1	0

If two binary numbers are given, each digits of both numbers will be considered as A and B and the result will be X

Octal Number System

There is another number systems called Octal Number System that has only 8 numerical figures (0, 1, 2, 3, 4, 5, 6 and 7) to count numbers. In Octal number system we use count numbers as follows: 0,1,2,3,4,5,6,7, to count next digit, we reset the Least significant digit to zero and we increase the value of most significant (second higher digit) by 1 and thus we get the next value to 1 as 10 and we keep counting further as 11, 12, 13, 14, 15, 16, 17, 20, 21, 22, 23, 24, 25, 26, 27, 30, 31, 32, 33, 34, 35. 36. 47, 40,…77,1 00, and so on.

Hexadecimal Number System

There is another number systems called Octal Number System that has only 16 numerical figures (0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,and F) to count numbers. In Hexadecimal number system we use count numbers as follows: 0,1,2,3,…8,9,A,B,C,D,E,F, to count next digit, we reset the Least significant digit to zero and we increase the value of most significant (second higher digit) by 1 and thus we get the next value to 1 as 10 and we keep counting further as 11,12,13,1A,1B,1C,1D,1E,1F,20,21,22,…FF,100, and so on.

PYTHON OPERATORS					
ARITHMETIC OPERATORS			**RELATIONAL OPERATORS**		
Purpose	**Real life**	**Python**	**Purpose**	**Real life**	**Python**
Assignment	=	=	Equal	=	==
Addition	+	+	Less than	<	<
Subtraction	-	-	Lesser Or Equal	≤	<=
Multiplication	x	*	Greater Than	>	>
Division	÷	/	Greater Or Equal	≥	>=
Floor Division		//	Not Equal To	<>	!=
Power	2^3	**			
Modular	mod	%			
LOGICAL OPERATORS			**BITWISE OPERATORS**		
Purpose	**Real life**	**Python**	**Purpose**	**Real life**	**Python**
And	&	and	AND	&	&
Or		or	OR		|
Not	<>	!	Exclusive OR		^
CONDITIONAL OPERATORS			Bitwise Compliment		~
Purpose	**Real life**	**Python**	Left Shift		<<
Evaluation based on a condition			Right Shift		>>

Python Comment Lines

Comment lines are line that can be used to explain that for what purpose the program is written. Comment lines are not executed by the program. Comments can be used to make the code more readable. It can be used to prevent execution of some portion of codes as an act of testing code.

Example

```
#print("Hello, World!")
print("Cheers, Mate!")
```

Creating a Comment

Comment line starts with a # character. When Python reads this line it will ignore the line unexecuted.

Example

```
#This is a comment
print("I am a python code!")
```

Comments can be placed at the end of an executable code line as below:

Example

```
print("I am a python code!")  #This is a comment
```

Multi Line Comments

Python has no specialsyntax for multi-line comments. To add a multiline comment, # can be inserted at the beginning of each line

```
#This is a comment
#written in
#more than just one line
print("Hello, World!")
```

Or a multiline string can be used.Python will ignore string literals that are not assigned to a variable. So, a multiline string (triple quotes) can be used as multiple line comment as below:

```
"""
This is a comment line
written in
more than just one line
"""
print("Hello, World!")
```

Commands

Input Commands

Input commands are used to get data from the user from console during runtime. Python provides an input command to get data from the user.

Input()

General format of the input() command usage:

    Variable Name = input(<Prompt message>)

The *input()* command reads a value from the user and stores in a variable name. The data keyed in by the user will always be stored as string type data.

Example-1:

a=input() will store the user input data in the variable a as string data. If a number is typed in and to involve that number in a mathematical calculation, it has to be converted to corresponding numeric type of data before involving it in mathematical calculation. So, a number can be inputted as shown below:

```
a=int(input())    or    a=input()
                        a=int(a)
```

Example-2:

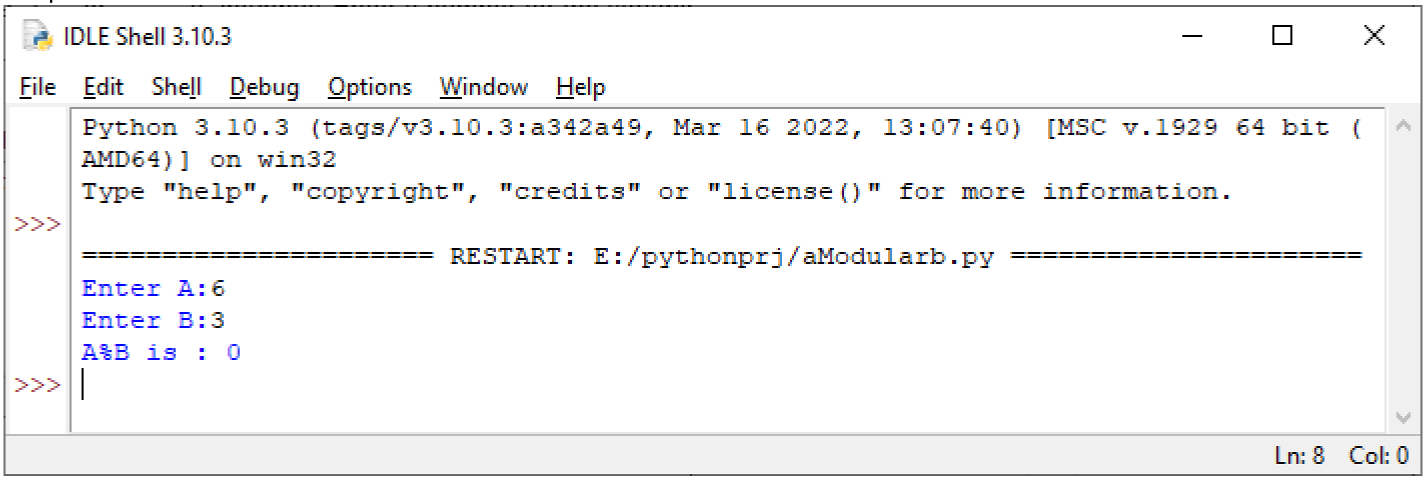
Python allows to provide a prompting message in the input() command as follows:

```
a=input("Enter a number for the variable a:")      or
a=int(input("Enter a number for the variable a:"))
```

Sample Program:

```
# Sample program with input() command.
a=int(input("Enter A:"))
b=int(input("Enter B:"))
print("A%B is :",a%b)
```

Output:



```
Python 3.10.3 (tags/v3.10.3:a342a49, Mar 16 2022, 13:07:40) [MSC v.1929 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
====================== RESTART: E:/pythonprj/aModularb.py ======================
Enter A:6
Enter B:3
A%B is : 0
>>> |
```

Type Casting

If the data type of a variable needs to be specified, it can be done as below:

Example

```
x = str(5)    # x will be '5'
y = int(5)    # y will be 5
z = float(5)  # z will be 5.0
```

Changing a data type to another is also called type casting. This is shown as below:

```
A=5
print(A)
print(str(A))
```

The output will be:

```
5
5
```

Also, the data type of a variable can be printed with the `type()` function as below:

```
x = 5
d = "Millennium Falcon"
print(type(x))
print(type(d))
```

String variables can be defined either by using single or double quotes:

```
n = "Darth Vader"
n = 'Darth Vader'
```

Both the definition will give same result of execution and no change in between them in the result.Note: Variable names are case-sensitive.

Example

This creates two variables:

```
        a = 4
        A = "Sally"
        #A will not overwrite as it is considered as different variable.
```

Output Commands

Print()

Python allows to use print() as output statement. The General format of print() command is:

```
print("<Constant to print>")
print(<variable to print)
print("<Constant>",<variable>)
```

For example, refer programs 1 & 2.

Outputting Variables

Python `print()` function is often used to output variables.

Example

```
s = "Man Machine Relationship"
print(x)
```

`print()` function outputs multiple variables separated by a comma:

Example

```
a = "Man"
b = "Machine"
c = "Relationship"
print(a, b, c)
```

Also can be printed multiple variables using the + operator.

Example

```
a = "Man "
b = "Machine "
c = "Relationship"
print(a + b + c)
```

Notice the space after each values. The result will be "ManMachineRelationship" without the spaces.
For numbers, the + character works as a mathematical / arithmetic operator:

Example

```
x = 5
y = 10
print(x + y)
```

Using `print()` function, if tried to combine a string and a number with the + operator, Python gives an error:

Example

```
a = 5
b = "Muhammadh"
print(a + b)
```

The best way to output multiple variables using the `print()` function by separating each variables with commas. This method supports different data types:

Example

```
a = 5
b = "Muhammadh"
print(a, b)
```

or using type casting, we can print them as below:

```
a = 5
b = "Muhammadh"
print(str(a) + b)
```

Note that the type casting is used to maintain all the variables type to be similar.

Assignment Expressions

To assign a numeric value, expression or string value to a variable, "=" symbol is used as sn assignment operator.

Example:

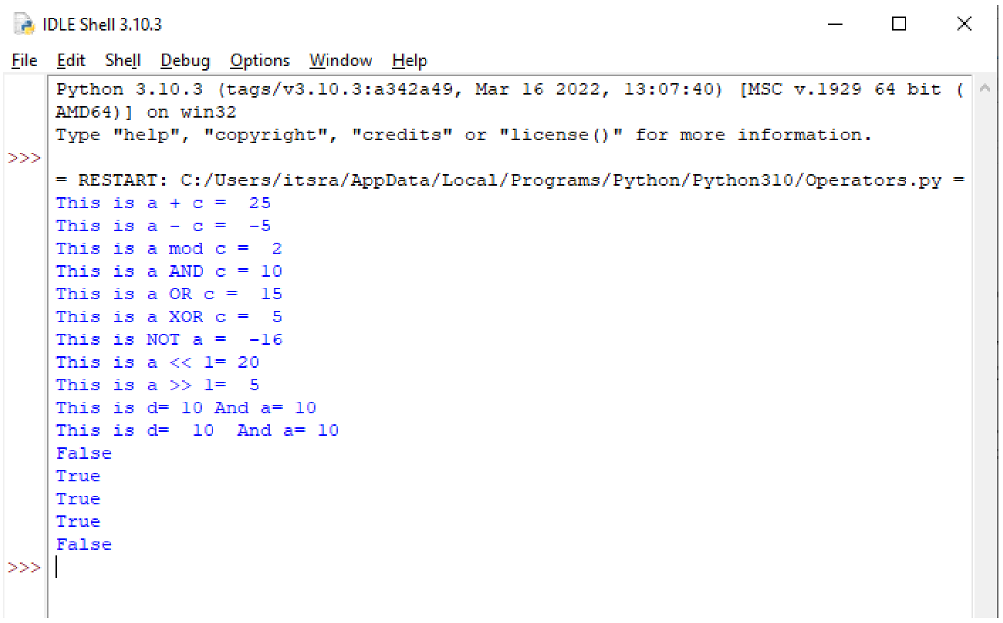
A=2+3
A=45
N="Darth Vader"

Program to test Python Operators

```

#This is sample program to test Python Operators
a=10
b=15
c=4
d = a + b                                      # Arithmetic
Operator - Addition
print("This is a + c = ",d)
d = a - b                                      # Arithmetic
Operation -Subtractor
print("This is a - c = ",d)
d = a % c                                      # Arithmetic
Operation - Modular
```

```
print("This is a mod c = ",d)
d=a&b                                         # Bitwise Operator
- AND
print("This is a AND c = "+str(d))
d=a | b                                         # Bitwise
Operator - OR
print("This is a OR c = ",d)
d=a ^ b                                         # Bitwise
Operator - XOR
print("This is a XOR c = ",d)
d= ~b
print("This is NOT a = ",d)
d= a<<1
print("This is a << 1= "+str(d))                # Bitwise
Operator - Left Shift
d= a>>1
print("This is a >> 1= ",d)                     # Bitwise
Operator - Right Shift
d= ++a
print("This is d= "+str(d)+" And a=",a)         # Increment
d= --a
print("This is d= ",d," And a=",a)              # Decrement

# Below segment is to test relational operators
a=4
b=5
c=6
d=4
print(a==b)                                  # Equality
print(a==d)                                  # Equality
print(a<b)                                   # Less than
print(c>=b)                                  # Greater Pr Equal
print(a!=d)                                  # Not Equal to
```

Run the program and observe the result.



```
Python 3.10.3 (tags/v3.10.3:a342a49, Mar 16 2022, 13:07:40) [MSC v.1929 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: C:/Users/itsra/AppData/Local/Programs/Python/Python310/Operators.py =
This is a + c =  25
This is a - c =  -5
This is a mod c =  2
This is a AND c = 10
This is a OR c =  15
This is a XOR c =  5
This is NOT a =  -16
This is a << 1= 20
This is a >> 1=  5
This is d= 10 And a= 10
This is d=  10  And a= 10
False
True
True
True
False
>>> |
```

Arithmetic Expressions

An expression derived using numbers and numerical operators are arithmetic expressions. Consider the expression a=2+4*5. The result stored in a are differently calculated as 30 by first adding 2+4 and multiplying it by 5 or 22 by multiplying 4*5 and then adding 2 to it. Here, 22 is the correct answer. To avoid this confusion, a methodology is followed to construct an expression to obtain correct result. It is called *Hierarchy of operation* .

Hierarchy of operation

Hierarchy of operation is nothing but following certain order to construct a numerical expression to obtain correct result when involving it in a computer program. The order is as shown as below:

1. Simplifying Unary Operators
2. Simplifying Exponents
3. Multiplication and Division
4. Addition and Subtraction

The above steps to be carried out from Left To Right direction in an expression. If any sub-expression is placed in an expression enclosed with parenthesis (brackets) it has to be simplified first. If more than one sub expression appears, the left most one must be simplified and the remaining will be in left-to-right direction.

Example:

Consider the arithmetic expression 2^3-(4+5) x 4/2+(7 x 2^2). This expression will be solved by following the Hierarchy of operation procedure as follows:

$2^3-(4+5) \times 4/2+(7 \times 2^2)$

Step-1: (Simplifying sub expressions before all. Here there are two sub expressions available. The left most sub expression will be simplified first)

$2^3-(\underline{4+5}) \times 4/2+(7 \times 2^2) = 2^3-9\times4/2+\underline{(7\times2^2})$

Step-2: (Simplifying the second sub expression. Within the second sub expression, there is 2^2. It has to be solved as per the Hierarchy)

$2^3-9\times4/2+(7\times2^2) = 2^3-9\times4/2+\underline{(7\times4)}$

Step-3: (Simplify the remaining portion of the second sub expression)

2^3-9x4/2+(7x2^2) = 2^3-9x4/2+28

Step-4: (No more sub expressions. Now find out if there are exponents in this expressions. There is 2^3.Simplify that).

8-9x4/2+28.

Step5: (No more exponents. Simplify multiplication and divisions if any, and solve them simultaneously from left to right direction)

8-36/2+28

=8-18+28

Step-6: (No more multiplications and divisions. Simplify addition and subtraction if any, and solve them simultaneously from left to right direction).

=36-18

=**18.**

Below python program simplifies and gives the result.

Program to simplify a complex arithmetic expression

```
#Sample program to simplify a complex arithmetic expression
print(2**3-(4+5)*4/2+(7*2**2))
```

Output:

```
Python 3.10.3 (tags/v3.10.3:a342a49, Mar 16 2022, 13:07:40) [MSC v.1929 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

=== RESTART: C:/Users/itsra/AppData/Local/Programs/Python/Python310/Cmplx.py ===
18.0
```

Control Statements

Unconditional Control Commands

Unconditional control statement is used to control the regular flow of program execution to change and jump to another command and continue execution.

Continue

Continue command will be used to ignore the code segments in ***for*** loop or ***while*** loop and proceed with the next iteration of the loop.

Example

Below segment of program shows how the control is deviated without any condition.

```
foriinrange(1,10):
    ifi==8:
        continue
    print(i)
print("For Loop Ends")
```

In the above example of code segment, the continue commands forces the computer control to jump back to the “for” loop ignoring the below command even if it is placed inside the “for” loop. The output is shown in the sample program shows various usages of “*for*” loop under “for” loop topic.

Break

break command will force the program to skip and quit executing the code segments given below the break in a block of commands segment given inside an ***if else*** command or within a ***for*** and ***while*** loop.

Example:

```
foriinrange(1,10):
   ifi==7:
       break
   else:
    print(i)
print("For Loop Ends")
```

Conditional Control Commands

Conditional control commands are used to control the regular flow of the program execution (line by line in a top to down direction) based on a condition. If a condition is checked and if the condition results True, the control can be diverted to execute a specific set of one or more commands. If the condition results False, the control can be diverted to execute another set of one or more commands leaving the set of commands of True result unexecuted.

If-elif-else:

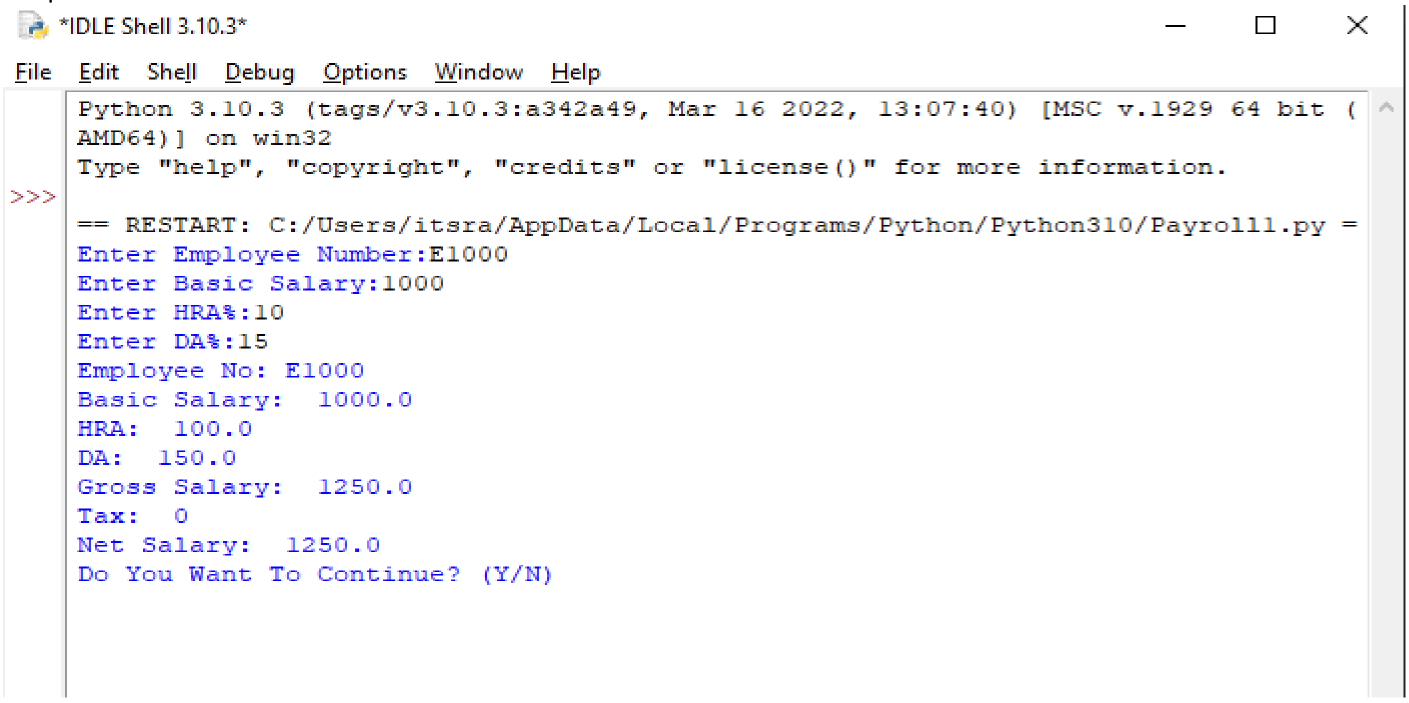
The general format of the usage of this command is:

```
if<condition1>:
        True part commands set
elif<condition2>:
        commands set if condition1 = false and condition2 = true
else:
        commands set if both condition1 and condition2 are false
```

Payroll Program:

```
#Sample Payroll program for a single employee
d="Y"
while d=="Y":
    emp_no=input("Enter Employee Number:")
    basic=float(input("Enter Basic Salary:"))
    hra=int(input("Enter HRA%:"))
    da=int(input("Enter DA%:"))
    hra=basic * hra/100
    da=basic * da/100
    gross=basic + hra + da
    if gross<50000:
        tax=0
    else:
        tax=gross * 0.02
    net=gross-tax
    print("Employee No:",emp_no)
    print("Basic Salary: ",basic)
    print("HRA: ",hra)
    print("DA: ",da)
    print("Gross Salary: ",gross)
    print("Tax: ",tax)
    print("Net Salary: ",net)
    d=input("Do You Want To Continue? (Y/N)")
print("Program Ends")
```

Output:



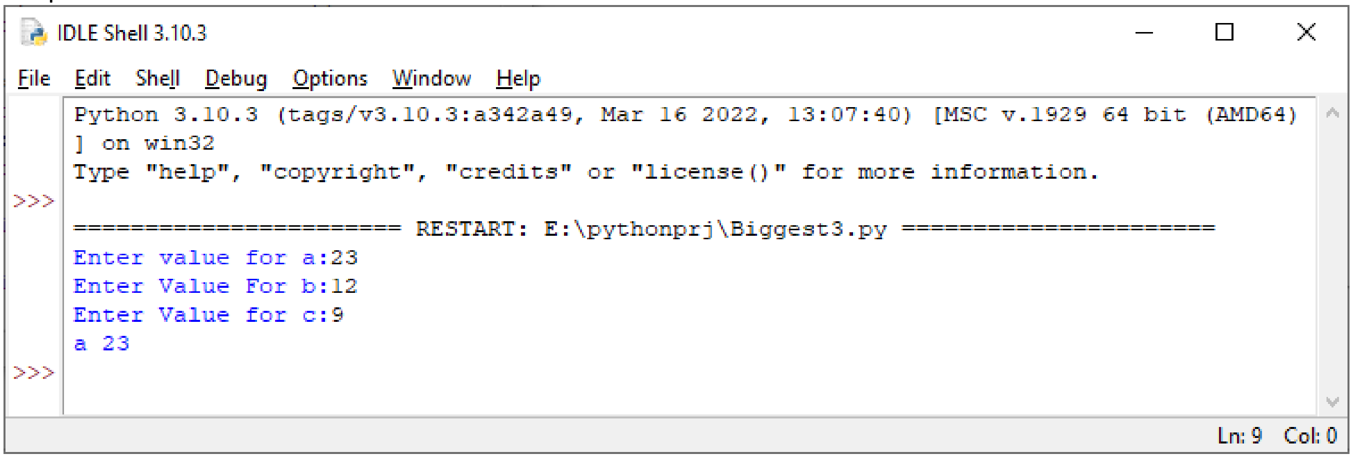
```
Python 3.10.3 (tags/v3.10.3:a342a49, Mar 16 2022, 13:07:40) [MSC v.1929 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
== RESTART: C:/Users/itsra/AppData/Local/Programs/Python/Python310/Payroll1.py =
Enter Employee Number:E1000
Enter Basic Salary:1000
Enter HRA%:10
Enter DA%:15
Employee No: E1000
Basic Salary:  1000.0
HRA:  100.0
DA:  150.0
Gross Salary:  1250.0
Tax:  0
Net Salary:  1250.0
Do You Want To Continue? (Y/N)
```

Program to find biggest among three numbers

```
#Finding biggest among three numbers
a=int(input("Enter value for a:"))
b=int(input("Enter Value For b:"))
c=int(input("Enter Value for c:"))
ifa>b:
      ifa>c:
          print("a",a)
```

```
        else:
            print("c",c)
elifb>c:
        print("b",b)
else:
        print("c",c)
```

Output:



```
Python 3.10.3 (tags/v3.10.3:a342a49, Mar 16 2022, 13:07:40) [MSC v.1929 64 bit (AMD64)
] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
======================= RESTART: E:\pythonprj\Biggest3.py ======================
Enter value for a:23
Enter Value For b:12
Enter Value for c:9
a 23
>>>
```

Repetitive Control Commands

Repetitive control commands are used to execute a set of one or more commands repeatedly specific number of times or until a condition becomes false.

For loop

for command is used to execute a set of commands repeatedly multiple times. There are various usages of *for* command.

The General format of *for* loop usage-1:

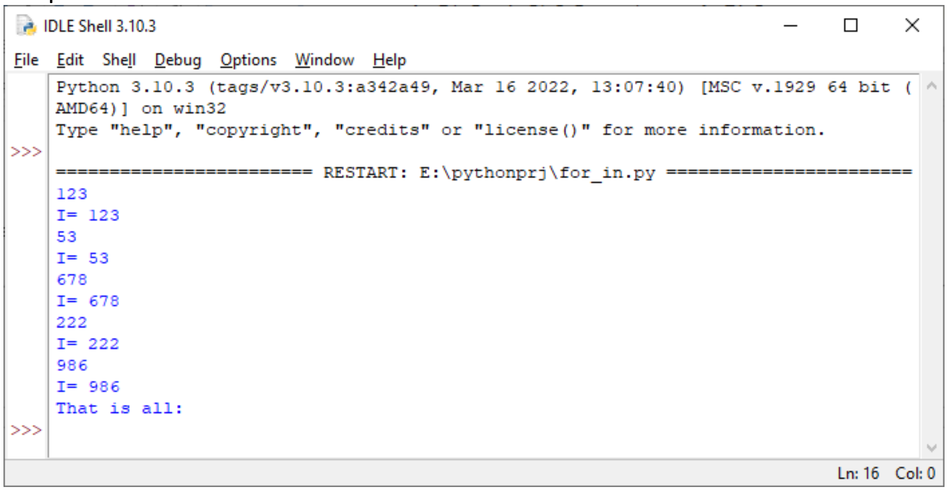
```
for<variable> in  <list/array>:
        <commands to be executed repeatedly, until the existence of expression elements>
<commands out of for loop>
```

Example:

```
#Usage-1 of For loop
exp=[123,53,678,222,986]
foriinexp:
    print(i)
    print("I=",i)
print("For Loop Ends")
#Usage-2 of For loop
foriinrange(1,11,2):
    print(i)
```

```
foriinrange(1,10):
   ifi==7:
       break
   else:
    print(i)
print("For Loop Ends")
#Usage-3 of For loop
foriinrange(1,10):
    ifi==8:
        continue
    print(i)
print("For Loop Ends")
```

Output:



```
Python 3.10.3 (tags/v3.10.3:a342a49, Mar 16 2022, 13:07:40) [MSC v.1929 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
======================= RESTART: E:\pythonprj\for_in.py =======================
123
I= 123
53
I= 53
678
I= 678
222
I= 222
986
I= 986
That is all:
>>>
```

In the above example, there is an array named “exp” that contains 5 numeric elements is used. A variable “i” is initialized in *for* loop. A print command to print the value of “I” is placed inside the “*for”* loop. Another print command is placed outside the “for” loop with a display message “That is all” to mention that the loop has ended.

Sample program using for in loop

```
exp=[123,53,678,222,986]
foriinexp:
    print(i)
    print("I=",i)
print("That is all:")
```

Program to print Prime numbers within a given N range

```
#Program to find Prime Numbers within a range of given N number
x=0.0
```

```
n=int(input("Enter A number to find Prime numbers within its
range: "))
print(1,"is a prime number")
foriinrange(1,n+1):
    forjinrange(2,i+1):
       x=i%j
       if (x==0andj<i):
           s="NP"
       else:
           print(i,"is a prime number")
       break
print("End of Program.")
```

The Output is

```
Python 3.10.3 (tags/v3.10.3:a342a49, Mar 16 2022, 13:07:40) [MSC v.1929
64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

======================== RESTART: E:\pythonprj\prime.py
========================
Enter A number to find Prime numbers within its range: 40
1 is a prime number
2 is a prime number
3 is a prime number
5 is a prime number
7 is a prime number
9 is a prime number
11 is a prime number
13 is a prime number
15 is a prime number
17 is a prime number
19 is a prime number
21 is a prime number
23 is a prime number
25 is a prime number
27 is a prime number
29 is a prime number
31 is a prime number
33 is a prime number
35 is a prime number
37 is a prime number
39 is a prime number
End
```

While loop

While loop is used to execute one or more commands repeatedly until as long as a condition is true.

General Format:

```
While <Condition>:
        S1
        S2
         |
         |
        Sn
Sc1
Sc2
|
|
```

Where S1, S2, Sn are the commands to be executed repeatedly until the <condition> given besides the ***While*** command becomes false. If it becomes false, the commands under the ***while*** loop (S1,S2,…Sn) will not be executed and the computer control starts executing commands Sc1,Sc2,…and continues.

Note: A program segment that makes the condition to become false, must be present inside the while loop. Otherwise, the program segment inside While loop will be executed repeatedly infinetly.

Example:

```
I=0
While i<-10
      Print(i)
      I=i+1
Print("Program Ends")
```

In the above example, a variable 'I' is initialized to 0. The condition i<=10 is checked. If it is True, the commands within while loop i.e., ***print(i)*** and ***i=i+1*** will be executed repeatedly as long as the condition is True. If i>10, the repeated execution will end and the computer will execute the command ***print("Program Ends").***

In the above example, ***i=i+1*** is the expression that changes the status of the condition to **false.**

Exercises:

1. Write a python program to read 2 names and print them line by line in alphabetically sorted in ascending order or in descending order as per the option given as an input, as follows:

```
Enter Name1:_MAN MACHINE
Enter Name2:_MORRIS TOWN
Ascending or Descending? (Type A/D):D
MORRIS TOWN
MAN MACHINE
Do you want to continue (Y/N)?Y

Enter Name1:_REX KING_
Enter Name2:_KIT WALKER
Ascending or Descending? (Type A/D): A
KIT WALKER
REX KING
Do you want to continue (Y/N)? N
Thank You!
```

2. Write a program to evaluate below arithmetic expression and verify the result that it is evaluated as per the Hierarchy of operation on an arithmetic expression.

   $(2^2-3^2)(35 – 7/4 \times (1+2)$

3. Write a program to evaluate and print the result. Also check that it is true or false and print TRUE or FALSE accordingly:

   $5^2 - 10^2 = (5 + 10)\ (5-10)$

4. Write a program to find N! (factorial value for N i.e., N!=1 x 2 x 3 x…x N)

Python (Data) Collections

More Data types

Setting the Data Type

In Python, the data type is set when you assign a value to a variable:

Data	Type
x = "Hello World"	str
x = 20	Int
x = 20.5	Float
x = 1j	Complex
x = ["apple", "banana", "cherry"]	List
x = ("apple", "banana", "cherry")	Tuple
x = range(6)	Range
x = {"name" : "John", "age" : 36}	Dict
x = {"apple", "banana", "cherry"}	Set
x = frozenset({"apple", "banana", "cherry"})	Frozenset
x = True	Bool
x = b"Hello"	Bytes
x = bytearray(5)	Bytearray
x = memoryview(bytes(5))	Memoryview
x = None	NoneType

Setting the Specific Data Type

If you want to specify the data type, you can use the following constructor functions:

Data	Type
x = str("Hello World")	Str
x = int(20)	Int
x = float(20.5)	float
x = complex(1j)	complex
x = list(("apple", "banana", "cherry"))	List
x = tuple(("apple", "banana", "cherry"))	tuple
x = range(6)	range
x = dict(name="John", age=36)	dict
x = set(("apple", "banana", "cherry"))	Set
x = frozenset(("apple", "banana", "cherry"))	frozenset
x = bool(5)	bool
x = bytes(5)	bytes
x = bytearray(5)	bytearray
x = memoryview(bytes(5))	memoryview

The following code example would print the data type of x, what data type would that be?

```
x = 5
print(type(x))
```

Arrays

An array is a special variable can hold more than one value at a time. Arrays are used to store multiple values in one single variable. Python uses List type of data as arrays.

Example

Create an array containing car names:

```
cars = ["Ford", "Volvo", "BMW"]
```

If you have a list of items (a list of car names, for example), storing the cars in single variables could look like this:

```
car1 = "Ford"
car2 = "Volvo"
car3 = "BMW"
```

Using of array gives you a solution to find one car among 300 cars. An array can hold many values under a single name, and you can access the values by referring to an index number.

Access the Elements of an Array

We can access an elementary value from an array by referring to the *index number*.

Example

If we want to get the first car name from an array named cars:

```
x = cars[0]
```

Modifying an array element

Modify the value of the first array item:

```
cars[0] = "Toyota"
```

Length of an Array

To find the length of an array i.e., how many elements are in an array, the ***len()*** method can be used. It returns the length of an array (the number of elements in an array).

Example

```
x = len(cars)
```

will return the length of the array ***cars.***

Looping Array Elements

By using for in loop we can loop through all the elements of an array.

Example

Print each item in the cars array:

```
for x in cars:
  print(x)
```

Adding Array Elements

To add an element to an array variable, ***append()*** method is used.

Example

To add one more element to the cars array:

```
cars.append("Honda")
```

Removing Array Elements

pop() method is to remove an element from the array by index.

Example
Delete the second element of the cars array:

```
cars.pop(1)
```

remove() method is used to remove an element from an array variable by its value,
Example
Delete the element that has the value "Volvo":

```
cars.remove("Volvo")
```

The above command willremove the first occurrence of the specified value.
Sample program using for loop and Array

```
#Sample Program To Sort 10 Car names inputted unordered and
sort them in ascending order
cars=[]
foriinrange(0,10):
    cars.append(input("Enter Car Name"+str(i)+": "))
foriinrange(0,9):
    forjinrange(i+1,10):
        ifcars[i] >= cars[j]:
            t=cars[j]
            cars[j]=cars[i]
            cars[i]=t
foriinrange(0,10):
    print(cars[i])
```

Lists

Lists are used to store multiple items in a single variable.
Lists are one of 4 built-in data types in Python used to store collections of data, the other 3 are Tuple, Set, and Dictionary, all with different qualities and usage.
Lists are created using square brackets:

Create a List

Assigning multiple values to a variable separated by commas within a "[]" bracket will create a list

```
List1 = ["apple", "jackfruit", "cherry"]
print(list1)
```

List Items

List items are ordered, changeable, and allow duplicate values.They are indexed, the first item has index **[0],** the second item has index **[1]** etc.

Ordered List

A list is an ordered list when its items have a defined unchanged order. When an item is added to a list it will be placed as the last item at the end of the list. There are exceptions when some list methods are used to change the order of list.

Changeable List

The list can be change by adding and removing items to or from a list after it was created.

Duplicates

lists can have duplicate items with different index.

Example

Lists can have duplicate items

```
List1 = ["Apple", "Orange", "Mango", “Orange”, "Jack Fruit"]
print(List1)
```

len() function can be used to return how many items are there in a list has.

Example

Print the number of items in the list:

```
list=["BoneyM", "ABBA", "Ventures"]
print(len(list))
```

List items can be of any data type:

Example

String, int and boolean data types:

```
list1=["BoneyM", "ABBA", "Commandos"]
list2=[1, 5, 7, 9, 3]
list3=[True, False, False]
```

A list can contain different data types:

Example

A list with strings, integers and boolean values are given below:

```
list1 = ["Bond", 7, True, 58, "MI6”,”Q”,”Valter PPK]
```

type()

type() function returns the type of a list

Lists in Python are defined as objects with the data type 'list':

```
<class 'list'>
```

Example

```
Bondlist=["Moore", “Connery”,"Lazenby",”Dalton”, "Brosnon"]
print(type(Bondlist))
```

list() Constructor

`list()` can be used as constructor when creating a new list.

Example

The below example shows hos a `List()` is constructedfrom an array:

```
a=[]
a.append('Auston Martin”)
a.append('DB5')
a.append('BMT216')
list=list(a)
print(list)
```

The output is:

```
Python 3.10.3 (tags/v3.10.3:a342a49, Mar 16 2022, 13:07:40) [MSC v.1929 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

=================== RESTART: E:/pythonprj/List Constructor.py
==================
['Auston Martin’, ‘DB5', 'BMT216']
```

Python provides four collection data types. They are:

List - A collection of ordered and changeable items. It also, allows duplicate items to be members.
Tuple - A collection of ordered and unchangeable items. It also allows duplicate items as members.
Set - A collection of unordered, unchangeable (but addition and removal is possible), and unindexed items No duplicate members are allowed.
Dictionary - A collection of ordered and changeable. In older versions of python, unordered items are allowed. No duplicate members are allowed.

Choosing the right type of collection a data set will be done by considering their properties.

Exercises:
1. Write a program to sort N numbers using While loop
2. Write a program to read a list with multiple data types and to create another list with the same elements of the first list and of same data type.

Program to test list

```
#Program to test Python Lists
#Creating a List
lst = ['Lists',6.3,2.2,12,32]
print(lst)

#List with size
size = [1]*10
print(size)
size = ['series']*5
print(size)

#variable as size and list value
n = 4
x=7
lst2 = [x]*n
print(lst2

#Splitting words from a sentence as list items
```

```
value = "Fear leads to anger; Anger leads to Hate; Hate leads
to sufferings"
print(value.split())

#Forming a sentence from list items
jedi = ['The','Force','is strong']
pverb = ' '.join(jedi)
print(pverb)

#List of lists
lst1=[1542,1547,1539,1548]
lst2=[1525,1516]
cmbLst= []
cmbLst.append(lst1)
cmbLst.append(lst2)
print("List1: ",lst1)
print("List2: ",lst2)
print("combined lists:",cmbLst)

#Creating List from dictionary
snames = {'Luke': 11, 'Leia': 12, 'Han':13, 'Lando': 14}
Lst = [(key,val) for key,val in snames.items()]
print(f"List from Dictionary : {Lst}")

#List of floating point numbers
fl_list = ["1.1", "3.5", "7.6"]
print(fl_list)
nfl_list = []
foriinfl_list:
    nfl_list.append(float(i))
print(nfl_list)

#Creating a list with elements with range
rno = [*range(1501,1549)]
print(rno)
```

```
#Adding elements to list after creating it, using for loop
token = ['667', '632', '567', '416', '839']
lst = []
foriintoken:
    lst.append(int(i))
print(lst)

#Creating list with a range
max_range = 10
lst = list(range(max_range))
print(lst)

# Correcting mistake values in a list
odd = [2, 4, 6, 8]
print(odd)
    # change the 1st item
odd[0] = 1
print(odd)
    # change 2nd to 4th items
odd[1:4] = [3, 5, 7]
print(odd)
```

List Methods

LIST METHODS	
Method	**Description**
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the first item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

Tuples

Tuples are used to store multiple items in a single variable.
Tuple is one of 4 built-in data collection of Python.
A tuple is a collection that is ***ordered*** and ***unchangeable***.
Tuples are written with round brackets.

Example
The below code will create a Tuple:

```
tuple = ("apple", "banana", "cherry")
print(tuple)
```

Tuple Items

Tuple items are ***ordered, unchangeable***, and ***allow duplicate values***.
Tuple items are indexed, the first item has index **`[0]`**, the second item has index `[1]` etc.

Ordered

If the items are in a defined order, then the tuple is ordered and that order ***will not change***.

Unchangeable

Tuples are ***unchangeable***. No items can be changed, added or removed directly after the tuple has been created.

Allow Duplicates

Tuples are indexed. They can have items with the same value in different indices.

Example

```
tuple = ("apple", "banana", "cherry", "apple", "cherry")
print(tuple)
```

Tuple Length

`Len()` function can be used to get the length of a Tuple.

Example
Print the number of items in the tuple:

```
tuple = ("apple", "banana", "cherry")
print(len(tuple))
```

Create Tuple With a single Item

A single item tuple can be created by adding a comma after the item. Otherwise Python will not recognize it as a tuple.
Example
One item tuple, remember the comma:

```
tuple1 = ("apple",)
print(type(tuple1))
```

Below is not a tuple:

```
tuple2 = ("apple")
print(type(tuple2))
```

Tuple Items - Data Types

Tuple items can be of any data type:
Example
String, int and boolean data types:

```
      tuple1 = ("apple", "banana", "cherry")
      tuple2 = (1, 5, 7, 9, 3)
      tuple3 = (True, False, False)
```

A tuple can contain different data types:
Example
A tuple with strings, integers and boolean values:

```
      tuple1 = ("abc", 34, True, 40, "male")
```

type()
From Python's perspective, tuples are defined as objects with the data type 'tuple':

```
      <class 'tuple'>
```

Example
To check what is the data type of a tuple, type() function is used in tuple as follows:

```
      tuple1 = ("apple", "banana", "cherry")
      print(type(tuple1))
```

Array Methods

Below table contains all methods can be used on Arrays and Lists:

```
      mytuple = ("apple", "banana", "cherry")
```

The tuple() Constructor
It is also possible to use the `tuple()` constructor to make a tuple.

Example
The below example shows how to use the tuple() method to make a tuple:

```
      tuple = tuple(("apple", "banana", "cherry")) # note the double round-brackets
      print(tuple)
```

Access Tuple Items
You can access tuple items by referring to the index number, inside square brackets:
Example
Print the second item in the tuple:

```
      Tuple1 = ("apple", "banana", "cherry")
      print(tuple[1])
```

Note: The first item is placed in index 0.

Update Tuples

Once the tuple is created, they are unchangeable, i.e., you cannot change, add, or remove items. But it is possible indirectly.

Change Tuple Values
Example
Convert the tuple into a list to be able to change it:

```
      x = ("apple", "banana", "cherry")
      y = list(x)
      y[1] = "kiwi"
      x = tuple(y)
      print(x)
```

Add Items
Since tuples are immutable, they do not have a build-in `append()` method, but there are other ways to add items to a tuple.

1. Convert into a list: Just like the workaround for *changing* a tuple, you can convert it into a list, add your item(s), and convert it back into a tuple.

Example

Convert the tuple into a list, add "orange", and convert it back into a tuple:

```
tuple1 = ("apple", "banana", "cherry")
y = list(tuple1)
y.append("orange")
tuple1 = tuple(y)
```

2. Add tuple to a tuple. You are allowed to add tuples to tuples, so if you want to add one item, (or many), create a new tuple with the item(s), and add it to the existing tuple:

Example

Create a new tuple with the value "orange", and add that tuple:

```
thistuple = ("apple", "banana", "cherry")
y = ("orange",)
thistuple += y
print(thistuple)
```

Note: When creating a tuple with only one item, remember to include a comma after the item, otherwise it will not be identified as a tuple.

Remove Items

Note: You cannot remove items in a tuple.

Tuples are ***unchangeable***, so you cannot remove items from it, but you can use the same workaround as we used for changing and adding tuple items:

Example

Convert the tuple into a list, remove "apple", and convert it back into a tuple:

```
Tuple1 = ("apple", "banana", "cherry")
y = list(tuple1)
y.remove("apple")
tuple1 = tuple(y)
```

or you can delete the tuple completely:

Example

The `del` keyword can delete the tuple completely:

```
tuple1 = ("apple", "banana", "cherry")
del tuple1
print(tuple1) #this will raise an error because the tuple no longer exists
```

Unpack Tuples

When we create a tuple, we normally assign values to it. This is called "packing" a tuple:

Example

Packing a tuple:

```
fruits = ("apple", "banana", "cherry")
```

But, in Python, we are also allowed to extract the values back into variables. This is called "unpacking":

Example

Unpacking a tuple:

```
fruits = ("apple", "banana", "cherry")
(green, yellow, red) = fruits
print(green)
print(yellow)
print(red)
```

Note: The number of variables must match the number of values in the tuple, if not, you must use an asterisk to collect the remaining values as a list.

Using Asterisk(*)

If the number of variables is less than the number of values, you can add an * to the variable name and the values will be assigned to the variable as a list:

Example

Assign the rest of the values as a list called "red":

```
fruits = ("apple", "banana", "cherry", "strawberry", "raspberry")
(green, yellow, *red) = fruits
print(green)
print(yellow)
print(red)
```

If the asterisk is added to another variable name than the last, Python will assign values to the variable until the number of values left matches the number of variables left.

Example

Add a list of values the "tropic" variable:

```
fruits = ("apple", "mango", "papaya", "pineapple", "cherry")
(green, *tropic, red) = fruits
print(green)
print(tropic)
print(red)
```

Loop Tuples

You can loop through the tuple items by using a `for` loop.

Example

Iterate through the items and print the values:

```
thistuple = ("apple", "banana", "cherry")
for x in thistuple:
  print(x)
```

Loop Through the Index Numbers

You can also loop through the tuple items by referring to their index number.
Use the `range()` and `len()` functions to create a suitable iterable.

Example

Print all items by referring to their index number:

```
thistuple = ("apple", "banana", "cherry")
for i in range(len(thistuple)):
  print(thistuple[i])
```

Using a While Loop

You can loop through the list items by using a `while` loop.
Use the **`len()`** function to determine the length of the tuple, then start at 0 and loop your way through the tuple items by refering to their indexes.
The index has to be increased by 1 after each iteration.

Example

Print all items, using a `while` loop to go through all the index numbers:

```
tuple = ("apple", "banana", "cherry")
i = 0
```

```
while i< len(tuple):
  print(thistuple[i])
  i = i + 1
```

Join Tuples

Using (+) operator, two tuples can be joined:

Example

Join two tuples:

```
tuple1 = ("a", "b", "c")
tuple2 = (1, 2, 3)
tuple3 = tuple1 + tuple2
print(tuple3)
```

Multiply Tuples

If you want to multiply the content of a tuple a given number of times, you can use the * operator:

Example

Multiply the fruits tuple by 2:

```
fruits = ("apple", "banana", "cherry")
mytuple = fruits * 2
print(mytuple)
```

Tuple Methods

Python has two built-in methods that you can use on tuples.

TUPLE METHODS	
Method	**Description**
count()	Returns the number of times a specified value occurs in a tuple
index()	Searches the tuple for a specified value and returns the position of where it was found

Sample program with Tuples exercises

```
#Sample Program with Tuples Excercise
tuple = ("apple", "banana", "cherry")
print(tuple)
print(type(tuple))

#Duplicates
tuple = ("apple", "banana", "cherry", "apple", "cherry")
print(tuple)

#Length
tuple = ("apple", "banana", "cherry")
print(len(tuple))

#Create Tuple
tuple1 = ("apple",)
print(type(tuple1))

#Not a Tuple
tuple1 = ("apple")
print(type(tuple1))
```

```
#Tuples with different data types
tuple1 = ("abc", 34, True, 40.55, "male")
print(tuple1)
print(type(tuple1[0]))
print(type(tuple1[1]))
print(type(tuple1[2]))
print(type(tuple1[3]))
print(type(tuple1[4]))

#Add item Indirectly - Convert Tuple to list
x = ("apple", "banana", "cherry")
y = list(x)
y.append("Jackfruit")
print(y)

# Add an item while creating tuple
tuple1 = ("apple", "banana", "cherry")
y = ("orange",)
tuple1 += y
print(tuple1)

#Update item Indirectly - Convert Tuple to list
x = ("apple", "banana", "cherry")
y = list(x)
y[1] = "kiwi"
print(y)
#v=tuple(y)
print(type(y))
print(y)

# Remove item indirectly
tuple2 = ("apple", "banana", "cherry")
y = list(tuple2)
y.remove("apple")
#tuple2 = tuple(y)
print(tuple2)

#Unpacking tuple
fruits = ("apple", "banana", "cherry")
(green, yellow, red) = fruits
print(green)
print(yellow)
print(red)

# Using '*' to unpack
fruits = ("apple", "banana", "cherry", "strawberry", "raspberry")
(green, yellow, *red) = fruits
print(green)
print(yellow)
print(red)
```

```
#type2
fruits = ("apple", "mango", "papaya", "pineapple", "cherry")
(green, *tropic, red) = fruits
print(green)
print(tropic)
print(red)
```

Dictionaries

Dictionaries are used to store data values in ***key: value*** pairs.
A dictionary is an ordered*, changeable collection of data and does not allow duplicates.

```
dict1 = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
```

Dictionaries are written with curly brackets, and have keys and values:
Example
Create and print a dictionary:

```
dict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(dict)
```

Dictionary Items

Dictionary items are ordered, changeable, and does not allow duplicates.
Dictionary items are presented in ***key: value*** pairs, and can be referred to by using the key name.
Example
Print the "brand" value of the dictionary:

```
dict1 = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(dict["brand"])
```

Ordered / Unordered

From Python version 3.7, dictionaries are *ordered*. Dictionaries are *unordered in* Python 3.6 and earlier versions.
If items are in a defined order and that order will not change, then the Dictionary is ordered.
If items are not in a defined order, you cannot refer to an item by using an index. This is unordered dictionary

Changeable

Dictionaries are changeable. Add or remove items can be done is a dictionary after it is created.

Duplicates Not Allowed

Dictionaries cannot have two items with the same key:

Example

Duplicate values will overwrite existing values:

```
dict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964,
  "year": 2020
}
print(dict)
```

Dictionary Length

To determine how many items a dictionary has, use the `len()` function:

Example

Print the number of items in the dictionary:

```
print(len(dict))
```

Items Data Types

The values in dictionary items can be of any data type:

Example

String, int, boolean, and list data types:

```
dict = {
  "brand": "Ford",
  "electric": False,
  "year": 1964,
  "colors": ["red", "white", "blue"]
}
```

type()

Dictionaries are defined as objects with the data type 'dict':

```
<class 'dict'>
```

Example

Print the data type of a dictionary:

```
dict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(type(dict))
```

Accessing Items

The items of a dictionary can be accessed by referring to its key name, inside square brackets:

Example

Get the value of the "model" key:

```
dict = {
"brand": "BMW",
"model": "750D",
"year": 2000
}
x = dict["model"]
```

There is also a method called **`get()`** that will give you the same result:

Example

Get the value of the "model" key:

```
x = dict.get("model")
```

Get Keys

The `keys()` method will return a list of all the keys in the dictionary.

Example

Get a list of the keys:

```
x = thisdict.keys()
```

The list of the keys is a *view* of the dictionary, meaning that any changes done to the dictionary will be reflected in the keys list.

Example

Add a new item to the original dictionary, and see that the keys list gets updated as well:

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}

x = car.keys()
print(x) #before the change
car["color"] = "white"
print(x) #after the change
```

Get Values

The `values()` method will return a list of all the values in the dictionary.

Example

Get a list of the values:

```
x = dict.values()
```

The list of the values is a *view* of the dictionary, meaning that any changes done to the dictionary will be reflected in the values list.

Example

Make a change in the original dictionary, and see that the values list gets updated as well:

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}
x = car.values()
print(x) #before the change
car["year"] = 2020
print(x) #after the change
```

Example

Add a new item to the original dictionary, and see that the values list gets updated as well:

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}
x = car.values()
print(x) #before the change
car["color"] = "red"
print(x) #after the change
```

Get Items

The **`items()`** method will return each item in a dictionary, as tuples in a list.

Example

Get a list of the key: value pairs

```
x = dict.items()
```

The returned list is a *view* of the items of the dictionary, meaning that any changes done to the dictionary will be reflected in the items list.

Example

Make a change in the original dictionary, and see that the items list gets updated as well:

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}
x = car.items()
print(x) #before the change
car["year"] = 2020
print(x) #after the change
```

Example

Add a new item to the original dictionary, and see that the items list gets updated as well:

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}
x = car.items()
print(x) #before the change
car["color"] = "red"
print(x) #after the change
```

Check if Key Exists

To determine if a specified key is present in a dictionary use the `in` keyword:

Example

Check if "model" is present in the dictionary:

```
dict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
if "model" in dict:
  print("Yes, 'model' is one of the keys in the dict dictionary")
```

Change Values

You can change the value of a specific item by referring to its key name:

Example

Change the "year" to 2018:

```
dict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
dict["year"] = 2018
```

Update Dictionary

The **`update()`** method will update the dictionary with the items from the given argument.
The argument must be a dictionary, or an iterable object with key:value pairs.

Example

Update the "year" of the car by using the **`update()`** method:

```
dict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
dict.update({"year": 2020})
```

Adding Items

Adding an item to the dictionary is done by using a new index key and assigning a value to it:

Example

```
dict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
dict["color"] = "red"
print(dict)
```

Removing Items

There are several methods to remove items from a dictionary:

Example

The **`pop()`** method removes the item with the specified key name:

```
dict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
dict.pop("model")
print(dict)
```

Example

The **`popitem()`** method removes the last inserted item (in versions before 3.7, a random item is removed instead):

```
dict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
dict.popitem()
print(thisdict)
```

Example

The **`del`** keyword removes the item with the specified key name:

```
dict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
del dict["model"]
print(dict)
```

Example

The **`del`** keyword can also delete the dictionary completely:

```
dict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
del dict
print(dict) #this will cause an error because "dict" no longer exists.
```

Example

The **`clear()`** method empties the dictionary:

```
dict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
dict.clear()
print(dict)
```

Loop through a Dictionary

You can loop through a dictionary by using a `for` loop.

When looping through a dictionary, the return value are the *keys* of the dictionary, but there are methods to return the *values* as well.

Example

Print all key names in the dictionary, one by one:

```
for x in thisdict:
  print(x)
```

Example

Print all *values* in the dictionary, one by one:

```
for x in thisdict:
  print(thisdict[x])
```

Example

You can also use the `values()` method to return values of a dictionary:

```
for x in thisdict.values():
  print(x)
```

Example
You can use the `keys()` method to return the keys of a dictionary:

```
for x in thisdict.keys():
  print(x)
```

Example
Loop through both *keys* and *values*, by using the `items()` method:

```
for x, y in thisdict.items():
  print(x, y)
```

Copy a Dictionary
There are many ways to make a copy of a dictionary. one of the way is to use the built-in dictionary method `copy()`.
Example
Make a copy of a dictionary with the **`copy()`** method:

```
dict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
dict2 = dict.copy()
print(dict)
```

Another way to make a copy is to use the built-in function **`dict()`**.
Example
Make a copy of a dictionary with the **`dict()`** function:

```
dict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
dict2 = dict(dict)
print(dict2)
```

Nested Dictionaries
A dictionary can contain dictionaries, this is called nested dictionaries.
Example
Create a dictionary that contain three dictionaries:

```
myfamily = {
  "child1" : {
    "name" : "Emil",
    "year" : 2004
  },
  "child2" : {
    "name" : "Tobias",
    "year" : 2007
  },
  "child3" : {
    "name" : "Linus",
    "year" : 2011
  }
}
```

Or, if you want to add three dictionaries into a new dictionary:

Example

Create three dictionaries, then create one dictionary that will contain the other three dictionaries:

```
      child1 = {
        "name" : "Emil",
        "year" : 2004
      }
      child2 = {
        "name" : "Tobias",
        "year" : 2007
      }
      child3 = {
        "name" : "Linus",
        "year" : 2011
      }

      myfamily = {
        "child1" : child1,
        "child2" : child2,
        "child3" : child3
      }
```

Dictionary Methods

Python has a set of built-in methods that you can use on dictionaries.

DICTIONARY METHODS	
Method	**Description**
clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and value
get()	Returns the value of the specified key
items()	Returns a list containing a tuple for each key value pair
keys()	Returns a list containing the dictionary's keys
pop()	Removes the element with the specified key
popitem()	Removes the last inserted key-value pair
setdefault()	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
update()	Updates the dictionary with the specified key-value pairs
values()	Returns a list of all the values in the dictionary

Sample Program using Dictionary Methods

```
# DIctionary keys
food = {'Tom': 'Burger', 'Jim': 'Pizza', 'Tim': 'Donut'}
f = food.keys()
print(f)

# DIctionary values
```

```
food = {'Tom': 'Burger', 'Jim': 'Pizza', 'Tim': 'Donut'}
f = food.values()
print(f)

# Dictionary Get()
employee = {1020: 'Kim', 1021: 'Ani', 1022: 'Mishka'}
print(employee.get(1021))
employee = {1020: 'Kim', 1021: 'Ani', 1022: 'Mishka'}
print(employee.get(1023))

# Dictionary Add
employee = {1020: 'Kim', 1021: 'Ani', 1022: 'Mishka'}
employee[1023] = 'Tom'
print(employee)

#Dictionary Len()
employee = {1020: 'Kim', 1021: 'Ani', 1022: 'Mishka'}
print(len(employee))

#Dictionary Update()
employee = {1020: 'Kim', 1021: 'Ani', 1022: 'Mishka'}
employee.update({1023: 'Ritika'})
print(employee)

#Dictionary List
dict = {}
dict["Name"] = ["Jack"]
dict["Marks"] = [45]
print(dict)

#Dictionary Comprehension
dict1 = {n:n*3for n inrange(6)}
print(dict1)

#Key Existence check
my_dict = {"name": "Harry", "roll": "23", "marks": "64"}
if"marks"inmy_dict:
    print("Yes, 'marks' is one of the keys in dictionary")

#Removing a key
    my_dict = {"name": "Harry", "roll": "23", "marks": "64"}
delmy_dict["roll"]
print(my_dict)

#Pop()
my_dict = {"name": "Harry", "roll": "23", "marks": "64"}
my_dict.pop("roll")
print(my_dict)

#Finding max() value
```

```
my_dictionary = {"avinav": 11, "John": 22, "nick": 23}
maximum_key = max(my_dictionary, key=my_dictionary.get)
print(maximum_key)

#Finding min() value
my_dictionary = {"avinav": 111, "John": 222, "nick": 223}
minimum_key = min(my_dictionary, key=my_dictionary.get)
print(minimum_key)

#Clear() method
Student = {
  "Nick": "America",
  "Roll": 154,
  "year": 2019
}
print(Student)
Student.clear()
print(Student)
```

Sets

Sets store multiple items in a single variable like Arrays, List, Tuples and Dictionary.
A set is a collection of data that is ***unordered, unchangeable, and unindexed.***

```
set1 = {"apple", "banana", "cherry"}
```

Note: Set ***items*** are unchangeable.But you can remove items and add new items.
Sets are written with **curly brackets**.

Create a Set

Example

```
set1 = {"apple", "banana", "cherry"}
print(set)
```

Note: Sets are unordered, so you cannot be sure in which order the items will appear.

Set Items

Set items do not allow duplicate values.

Unordered

Unordered means that the items in a set do not have a defined order.
Set items appear in a different order every time when used, and cannot be referred to by index or key.

Unchangeable

The items cannot be changed in a set after creating it.
Once a set is created, its items cannot be changed.But items can be removed and can be addedwith the new items.

Duplicates Not Allowed

Two items with the same value will not be allowed to be placed in a set.

Example

Duplicate values will be ignored:

```
set1 = {"apple", "banana", "cherry", "apple"}
print(set1)
```

The output will be

```
==================== RESTART: D:/pythonprj/set_duplicte.py ====================
{'banana', 'cherry', 'apple'}
```

Length of a Set

To determine how many items a set has, use the `len()` function.

Example

Get the number of items in a set:

```
      set1 = {"apple", "banana", "cherry"}
      print(len(set))
```

Data Types of Set Items

Set items can have data of any type

Example

```
      set1 = {"apple", "banana", "cherry"}
      set2 = {1, 5, 7, 9, 3}
      set3 = {True, False, False}
```

A set can contain items with mixed data types:

Example

A set with strings, integers and boolean values:

```
      set1 = {"abc", 34, True, 40, "male"}
```

type()

From Python's perspective, sets are defined as objects with the data type 'set':

```
      <class 'set'>
```

Example

What is the data type of a set?

```
      myset = {"apple", "banana", "cherry"}
      print(type(myset))
```

set() Constructor

It is also possible to use the `set()` constructor to make a set.

Example

Using the set() constructor to make a set:

```
      set1 = set(("apple", "banana", "cherry")) # note the double round-brackets
      print(set1)
```

Access Items

You cannot access items in a set by referring to an index or a key.
But you can loop through the set items using a '`for`' loop, or ask if a specified value is present in a set, by using the `in` keyword.

Example

Loop through the set, and print the values:

```
set1 = {"apple", "banana", "cherry"}
for x in set1:
  print(x)
```

Example

Check if "banana" is present in the set:

```
set1 = {"apple", "banana", "cherry"}
print("banana" in set1)
```

Change Items

Once a set is created, you cannot change its items, but you can add new items.

Add Items

Once a set is created, you cannot change its items, but you can add new items.
To add one item to a set use the `add()` method.

Example

Add an item to a set, using the `add()` method:

```
set1 = {"apple", "banana", "cherry"}
set1.add("orange")
print(set1)
```

Add Sets

To add items from another set into the current set, use the `update()` method.

Example

Add elements from `tropical` into `set1`:

```
set1 = {"apple", "banana", "cherry"}
set2 = {"grape", "watermelon", "pomegranade"}
set1.update(set2)
print(set1)
```

Add Any Iterable

The object in the `update()` method does not have to be a set, it can be any iterable object (tuples, lists, dictionaries etc.).

Example

Add elements of a list to at set:

```
set1 = {"apple", "banana", "cherry"}
list1 = ["Lemon", "Grapefruit"]
set1.update(list1)
print(set1)
```

Remove Item

To remove an item in a set, use the `remove()`, or the `discard()` method.

Example

Remove "banana" by using the `remove()` method:

```
bikes = {"YAMAHA", "HONDA", "KAWASAKI","SUZUKI"}
print(bikes)
bikes.remove("SUZUKI")
```

```
    print(bikes)
```
Note: *If the item to remove does not exist, remove() will raise an error.*

Example

Remove "banana" by using the `discard()` method:
```
    bikes = {"FIAT", "OPEL", "BMW","LEXUS","MERCEDES"}
    print(bikes)
    bikes.discard("LEXUS")
    print(bikes)
```
Note: If the item to remove does not exist, `discard()` will **NOT** raise an error.

`pop()` method can also be used to remove an item.But this method will remove the *last* item, not an item at any position. Since sets are unordered, it will not be known that which item was removed when used pop() method. The return value of the `pop()` method is the removed item.

Example

Remove the last item by using the `pop()` method:
```
    set1 = {"BOEING", "AIRBUS", "DOUGLAS"}
    print(set1)
    ri = set1.pop()
    print(ri)
    print(set1)
```
Note: Sets are *unordered*, so when using the `pop()` method, you do not know which item that gets removed.

Example

The `clear()` method empties the set:
```
    Set1 = {"apple", "banana", "cherry"}
    set1.clear()
    print(set1)
```

Example
```
    set1 = {"apple", "banana", "cherry"}
    del set1
    print(set1)
```

The `del` keyword will delete the set completely and the print(set1) statement will return error due to the inexistence of set1 after deletion.

Loop Items

You can loop through the set items by using `for` loop:

Example

Loop through the set, and print the values:
```
    Set1 = {"apple", "banana", "cherry"}
    for x in set1:
        print(x)
```

Join Two Sets

There are several ways to join two or more sets in Python.
You can use the `union()` method that returns a new set containing all items from both sets, or the `update()` method that inserts all the items from one set into another:

Example

The `union()` method returns a new set with all items from both sets:

```
set1 = {"a", "b" , "c"}
set2 = {1, 2, 3}
set3 = set1.union(set2)
print(set3)
```

Example

The `update()` method inserts the items in set2 into set1:

```
set1 = {"a", "b" , "c"}
set2 = {1, 2, 3}
set1.update(set2)
print(set1)
```

Note: Both `union()` and `update()` will exclude any duplicate items.

Keeping Only the Duplicate Items

The `intersection_update()` method will keep only the items that are present in both sets.

Example

Keep the items that exist in both set `x`, and set `y`:

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
x.intersection_update(y)
print(x)
```

The `intersection()` method will return a *new* set, that only contains the items that are present in both sets.

Example

Return a set that contains the items that exist in both set `x`, and set `y`:

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
z = x.intersection(y)
print(z)
```

Keep All, except the Duplicates

The `symmetric_difference_update()` method will keep only the elements that are NOT present in both sets.

Example

Keep the items that are not present in both sets:

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
x.symmetric_difference_update(y)
print(x)
```

The `symmetric_difference()` method will return a new set, that contains only the elements that are NOT present in both sets.

Example

Return a set that contains all items from both sets, except items that are present in both:

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
```

```
z = x.symmetric_difference(y)
print(z)
```

Set Methods

Python has a set of built-in methods that you can use on sets.

SET METHODS	
Method	**Description**
add()	Adds an element to the set
clear()	Removes all the elements from the set
copy()	Returns a copy of the set
difference()	Returns a set containing the difference between two or more sets
difference_update()	Removes the items in this set and also included in another, specified set
discard()	Remove the specified item
intersection()	Returns a set, that is the intersection of two other sets
intersection_update()	Removes the items in this set that are not present in other, specified set(s)
isdisjoint()	Returns whether two sets have a intersection or not
issubset()	Returns whether another set contains this set or not
issuperset()	Returns whether this set contains another set or not
pop()	Removes an element from the set
remove()	Removes the specified element
symmetric_difference()	Returns a set with the symmetric differences of two sets
symmetric_difference_update()	inserts the symmetric differences from this set and another
union()	Return a set containing the union of sets
update()	Update the set with the union of this set and others

Python Strings

Strings

Strings in python are surrounded by either single quotation marks, or double quotation marks.

`'hello'` is the same as `"hello"`.

You can display a string literal with the `print()` function:

Example

```
print("Hello")
print('Hello')
```

Assign String to a Variable

Assigning a string to a variable is done with the variable name followed by an equal sign and the string:

Example

```
a = "Hello"
print(a)
```

Multiline Strings

You can assign a multiline string to a variable by using three quotes:

Example

You can use three double quotes:

```
a = """Dark side of the force has many,
abilities some,
considered yo be unnatural”””
print(a)
```

Or three single quotes:

Example

```
a = '''The force is strong in you,
your inside serves you well,
but it could be made to serve
the dark side.'''
print(a)
```

Note: in the result, the line breaks are inserted at the same position as in the code.

Strings are Arrays

Like many other popular programming languages, strings in Python are arrays of bytes representing unicode characters.
However, Python does not have a character data type, a single character is simply a string with a length of 1.
Square brackets can be used to access elements of the string.

Example

Get the character at position 1 (remember that the first character has the position 0):

```
a = "Hello, World!"
print(a[1])
```

Looping Through a String

Since strings are arrays, we can loop through the characters in a string, with a `for` loop.

Example

Loop through the letters in the word "banana":

```
for x in "banana":
      print(x)
```

String Length

To get the length of a string, use the `len()` function.

Example

The `len()` function returns the length of a string:

```
a = "Hello, World!"
print(len(a))
```

Check String

To check if a certain phrase or character is present in a string, we can use the keyword `in`.

Example

Check if "free" is present in the following text:

```
txt = "The best things in life are free!"
print("free" in txt)
```

Example
Print only if "free" is present:

```
txt = "The best things in life are free!"
if "free" in txt:
  print("Yes, 'free' is present.")
```

Check if NOT

To check if a certain phrase or character is NOT present in a string, we can use the keyword `not in`.

Example

Check if "expensive" is NOT present in the following text:

```
txt = "The best things in life are free!"
print("expensive" not in txt)
```

Example
print only if "expensive" is NOT present:

```
txt = "Anakin is good pilot!"
if "ship" not in txt:
  print("No, 'ship' is NOT present.")
```

Slicing

You can return a range of characters by using the slice syntax.
Specify the start index and the end index, separated by a colon, to return a part of the string.

Example
Get the characters from position 2 to position 5 (not included):

```
b = "Hello, World!"
print(b[2:5])
```

Note: The first character has index 0.

Slice From the Start

By leaving out the start index, the range will start at the first character:

Example
Get the characters from the start to position 5 (not included):

```
b = "Hello, World!"
print(b[:5])
```

Slice To the End

By leaving out the *end* index, the range will go to the end:

Example
Get the characters from position 2, and all the way to the end:

```
b = "Hello, World!"
print(b[2:])
```

Negative Indexing

Use negative indexes to start the slice from the end of the string:

Example

Get the characters:
From: "o" in "World!" (position -5)
To, but not included: "d" in "World!" (position -2):

```
b = "Hello, World!"
print(b[-5:-2])
```

Modify Strings

Python has a set of built-in methods that you can use on strings.

Upper Case

Example

The `upper()` method returns the string in upper case:

```
a = "Hello, World!"
print(a.upper())
```

Lower Case

Example

The `lower()` method returns the string in lower case:

```
a = "Hello, World!"
print(a.lower())
```

Remove Whitespace

Whitespace is the space before and/or after the actual text, and very often you want to remove this space.

Example

The `strip()` method removes any whitespace from the beginning or the end:

```
a = " Hello, World! "
print(a.strip()) # returns "Hello, World!"
```

Replace String

Example

The `replace()` method replaces a string with another string:

```
a = "Hello, World!"
print(a.replace("H", "J"))
```

Split String

The `split()` method returns a list where the text between the specified separator becomes the list items.

Example

The `split()` method splits the string into substrings if it finds instances of the separator:

```
a = "R2D2,C3PO"
print(a.split(",")) # returns ['R2D2', ' C3PO']
```

String Methods

String Concatenation

To concatenate, or combine, two strings you can use the + operator.

Example

Merge variable a with variable b into variable c:

```
a = "Queen"
b = "Amidala"
c = a + b
print(c)
```

Example

To add a space between them, add a " ":

```
a = "Queen"
b = Amidala"
c = a + " " + b
print(c)
```

Format - Strings

String Format

As we learned in the Python Variables chapter, we cannot combine strings and numbers like this:

Example

```
age = 36
txt = "My name is Bond, I am " + age
print(txt)
```

But we can combine strings and numbers by using the `format()` method!
The `format()` method takes the passed arguments, formats them, and places them in the string where the placeholders `{}` are:

Example

Use the `format()` method to insert numbers into strings:

```
age = 36
txt = "My name is Bond, and I am {}"
print(txt.format(age))
```

The format() method takes unlimited number of arguments, and are placed into the respective placeholders:

Example

```
quantity = 3
itemno = 567
price = 49.95
myorder = "I bought {} pieces of item {} for Rs.{}’-"
print(myorder.format(quantity, itemno, price))
```

You can use index numbers `{0}` to be sure the arguments are placed in the correct placeholders:

Example

```
quantity = 3
itemno = 567
```

```
price = 49.95
myorder = "I want to pay {2} dollars for {0} pieces of item {1}."
print(myorder.format(quantity, itemno, price))
```

Python String Formatting

To make sure a string will display as expected, we can format the result with the `format()` method.

String format()

The `format()` method allows you to format selected parts of a string.
Sometimes there are parts of a text that you do not control, maybe they come from a database, or user input?
To control such values, add placeholders (curly brackets `{}`) in the text, and run the values through the `format()` method:

Example

Add a placeholder where you want to display the price:

```
price = 49
txt = "The price is {} dollars"
print(txt.format(price))
```

You can add parameters inside the curly brackets to specify how to convert the value:

Example

Format the price to be displayed as a number with two decimals:

```
txt = "The price is {:.2f} dollars"
```

Multiple Values

If you want to use more values, just add more values to the format() method:

```
print(txt.format(price, itemno, count))
```

And add more placeholders:

Example

```
quantity = 3
itemno = 567
price = 49
myorder = "I want {} pieces of item number {} for {:.2f} dollars."
print(myorder.format(quantity, itemno, price))
```

Index Numbers

You can use index numbers (a number inside the curly brackets `{0}`) to be sure the values are placed in the correct placeholders:

Example

```
quantity = 3
itemno = 567
price = 49
myorder = "I want {0} pieces of item number {1} for {2:.2f} dollars."
print(myorder.format(quantity, itemno, price))
```

Also, if you want to refer to the same value more than once, use the index number:

Example
```
age = 36
name = "John"
txt = "His name is {1}. {1} is {0} years old."
print(txt.format(age, name))
```

Named Indexes
You can also use named indexes by entering a name inside the curly brackets `{carname}`, but then you must use names when you pass the parameter values `txt.format(carname = "Ford")`:

Example
```
myorder = "I have a {carname}, it is a {model}."
print(myorder.format(carname = "Ford", model = "Mustang"))
```

Escape Character
To insert characters that are illegal in a string, use an escape character.
An escape character is a backslash \ followed by the character you want to insert.
An example of an illegal character is a double quote inside a string that is surrounded by double quotes:

Example
You will get an error if you use double quotes inside a string that is surrounded by double quotes:
```
txt = "We are the so-called "Vikings" from the north."
```
To fix this problem, use the escape character `\"`:

Example
The escape character allows you to use double quotes when you normally would not be allowed:
```
txt = "We are the so-called \"Vikings\" from the north."
```

Escape Characters

Other escape characters used in Python:

ESCAPE CHARACTERS	
Code	**Result**
\'	Single Quote
\\	Backslash
\n	New Line
\r	Carriage Return
\t	Tab
\b	Backspace
\f	Form Feed
\ooo	Octal value
\xhh	Hex value

String Methods
Python has a set of built-in methods that you can use on strings.

Note: All string methods return new values. They do not change the original string.

STRING METHODS	
Method	**Description**
capitalize()	Converts the first character to upper case
casefold()	Converts string into lower case
center()	Returns a centered string
count()	Returns the number of times a specified value occurs in a string
encode()	Returns an encoded version of the string
endswith()	Returns true if the string ends with the specified value
expandtabs()	Sets the tab size of the string
find()	Searches the string for a specified value and returns its position
format()	Formats specified values in a string
format_map()	Formats specified values in a string
index()	Searches the string for a specified value and returns its position
isalnum()	Returns True if all characters in the string are alphanumeric
isalpha()	Returns True if all characters in the string are in the alphabet
isdecimal()	Returns True if all characters in the string are decimals
isdigit()	Returns True if all characters in the string are digits
isidentifier()	Returns True if the string is an identifier
islower()	Returns True if all characters in the string are lower case
isnumeric()	Returns True if all characters in the string are numeric
isprintable()	Returns True if all characters in the string are printable
isspace()	Returns True if all characters in the string are whitespaces
istitle()	Returns True if the string follows the rules of a title
isupper()	Returns True if all characters in the string are upper case
join()	Joins the elements of an iterable to the end of the string
ljust()	Returns a left justified version of the string
lower()	Converts a string into lower case
lstrip()	Returns a left trim version of the string
maketrans()	Returns a translation table to be used in translations
partition()	Returns a tuple where the string is parted into three parts
replace()	Returns a string where a specified value is replaced with a specified value
rfind()	Searches the string for a specified value and returns its last position
rindex()	Searches the string for a specified value and returns its last position
rjust()	Returns a right justified version of the string
rpartition()	Returns a tuple where the string is parted into three parts
rsplit()	Splits the string at the specified separator, and returns a list
rstrip()	Returns a right trim version of the string
split()	Splits the string at the specified separator, and returns a list
splitlines()	Splits the string at line breaks and returns a list
startswith()	Returns true if the string starts with the specified value
strip()	Returns a trimmed version of the string
swapcase()	Swaps cases, lower case becomes upper case and vice versa
title()	Converts the first character of each word to upper case
translate()	Returns a translated string
upper()	Converts a string into upper case
zfill()	Fills the string with a specified number of 0 values at the beginning

Functions

Functions are a small program segment that can be separately written and can be called in a program as many as times wherever required. The function will run small segment of a program and returns a value to the calling program as a result.

The mathematical expression $^{n}C_{r}$=n! r! / (n-r)! uses multiple factorial values n!, r! and (n-r)!. To evaluate this formula programmatically, we need to write factorial finding program segment in 3 places if written in a single program. First we need to find factorial value for ***n, r*** and for ***(n-r).***and then we need to calculate value for ncr. To simplify this process, if a function is written to find factorial value and to return the result to the calling program, it can be easily evaluated by avoiding writing code repeatedly to find factorial value.
A function can be written as below:

```
G.F:    def <function name>(<optional input parameter1,optional input parameter2,…>:
          C1
                  C2
                  |
                  |
                  Cn
                  return<result>
```

This function can be called in the calling program as

```
          <variable>=<function name>(<Value for parameter1,Value for parameter2,…>)
```

The function will execute C1,C2,…Cn commands and returns the result to the calling program and the calling program will store the result in <variable>.

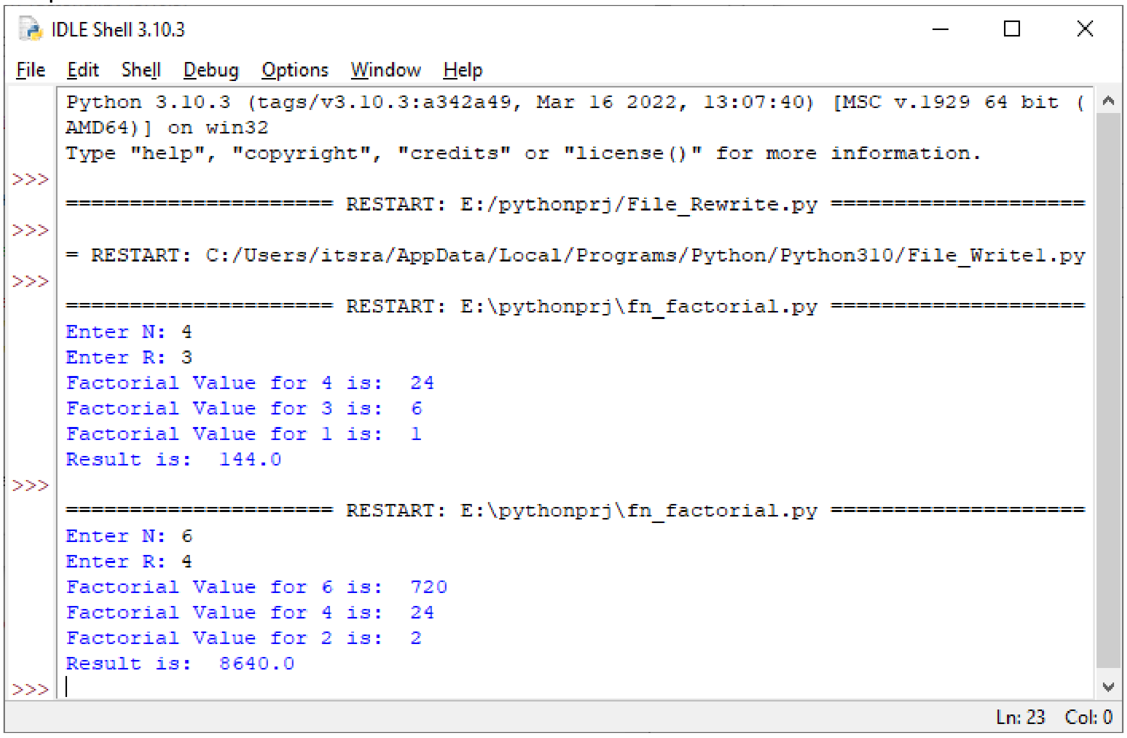
The below program explains how a function can be written and called in a main program.

Program to find $^{n}C_{r}$=n! r! / (n-r)!

```
#Program for n!r!/(n-r)!
#----Function fact(x) is the function
deffact(x):
    prod=1
    foriinrange(1,x+1):
        prod=prod*i
    print("Factorial Value for "+str(x)+" is: ",prod)
    return prod
#----Main program segment calling function fact()
n=int(input("Enter N: "))
r=int(input("Enter R: "))
f1=fact(n)
f2=fact(r)
```

```
f3=fact(n-r)
result=f1*f2/f3
print("Result is: ",result)
```

Output:


```
Python 3.10.3 (tags/v3.10.3:a342a49, Mar 16 2022, 13:07:40) [MSC v.1929 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> 
==================== RESTART: E:/pythonprj/File_Rewrite.py ====================
>>> 
= RESTART: C:/Users/itsra/AppData/Local/Programs/Python/Python310/File_Write1.py
>>> 
==================== RESTART: E:\pythonprj\fn_factorial.py ====================
Enter N: 4
Enter R: 3
Factorial Value for 4 is:  24
Factorial Value for 3 is:  6
Factorial Value for 1 is:  1
Result is:  144.0
>>> 
==================== RESTART: E:\pythonprj\fn_factorial.py ====================
Enter N: 6
Enter R: 4
Factorial Value for 6 is:  720
Factorial Value for 4 is:  24
Factorial Value for 2 is:  2
Result is:  8640.0
>>> |
```

Lambda Function

A lambda function is a small anonymous function. A lambda function can take any number of arguments, but can only have one expression.

General Format:

```
lambda arguments : expression
```

The expression is executed and the result is returned:

Example

Add 10 to argument a, and return the result:

```
x = lambda a : a + 10
print(x(5))
```

Lambda functions can take any number of arguments:

Example

Multiply argument **a** with argument b and return the result:

```
x = lambda a, b : a * b
print(x(5, 6))
```

Example

Summarize argument a, b, and c and return the result:

```
x = lambda a, b, c : a + b + c
print(x(5, 6, 2))
```

Purpose of Lambda Functions

Lambda is powerful when used as an anonymous function inside another function. If a function definition if found one argument, that argument will be multiplied with an unknown number:

```
def myfunc(n):
  return lambda a : a * n
```

Use that function definition to make a function always doubles the number sent in:

Example

```
def myfunc(n):
  return lambda a : a * n
mydoubler = myfunc(2)
print(mydoubler(11))
```

Or, use the same function definition to make a function that always *triples* the number you send in:

Example

```
def myfunc(n):
  return lambda a : a * n
mytripler = myfunc(3)
print(mytripler(11))
```

Or, use the same function definition to make both functions, in the same program:

Example

```
def myfunc(n):
  return lambda a : a * n
mydoubler = myfunc(2)
mytripler = myfunc(3)
print(mydoubler(11))
print(mytripler(11))
```

Use lambda functions when an anonymous function is required for a short period of time.

Global Variables

Variables created outside of a function are known as global variables.Global variables can be used both inside of functions and outside.

Example

Create a variable outside of a function, and use it inside the function

```
x = "May the force be with you"
def myfunc():
 print(x)
myfunc()
```

If a variable with the same name or with different name inside a function will be known as local and it can only be used inside the function inside where the variable is created. The global variable with the same name will remain as global and with the original value.

Example
Create a variable inside a function, with the same name as the global variable

```
x = "May the force be with you "
def myfunc():
  x = "Always "
  print(x)
myfunc()
print(x)
```

The global Keyword

Normally, when a variable is created inside a function that variable is local, and can only be used inside that function. To create a global variable inside a function, it can use the global keyword.

Example
If the **global** keyword is used, the variable belongs to the global scope:

```
def myfunc():
  global x
 x = "fantastic"
myfunc()
print("Python is " + x)
```

Also if a global variable needs to be changed, it must use the global keyword inside a function.

Example
To change the value of a global variable inside a function, it mut be referred by using the global keyword:

```
x = "Use the force"
def myfunc():
  global x
  x = "Power of Force"
myfunc()
print(x)
```

Introduction to Object Oriented Programming:

Classes

The important feature of OOP is to handle classes. Class is a collection of classified methods and attributes. An attribute is nothing but a variable defined and used inside a class. A method is a function defined and used inside a class.

Objects

An object is an instance created to use class, its attributes, methods and/or everything in python. A program may have multiple classes and each classes can be used in the program by creating object for each class.

Attributes

An attribute is a variable defined and used inside a class. A class may have one or more attributes.

Methods

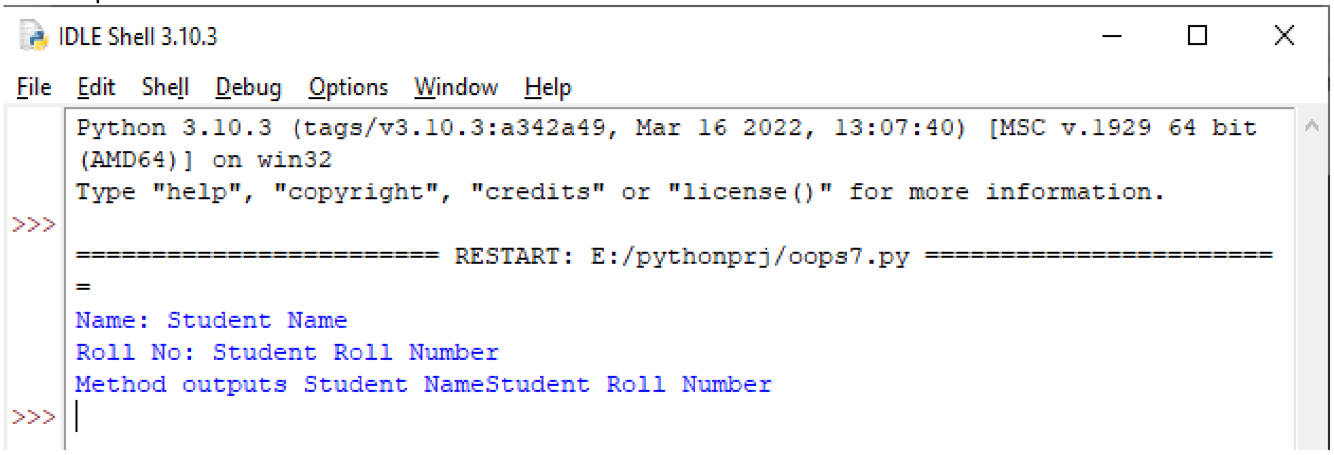
A method is a function defined and used inside a class. A class may have one or more methods.
The below example shows how a class and its attributes are defined in a program and are used.

Create a Class

The keyword **`class:`** is used to create a class

```
class student:
    Name="Student Name"
    R_No="Student Roll Number"
    defstudent_details(self):
        result=self.Name+self.R_No
        return result
        #return "You have entered Name as “+self.Name+” and “+self.R_No
obj=student()
print("Name: "+obj.Name)
print("Roll No: "+obj.R_No)
print("Method outputs "+obj.student_details())
```

The output is



```
Python 3.10.3 (tags/v3.10.3:a342a49, Mar 16 2022, 13:07:40) [MSC v.1929 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
======================= RESTART: E:/pythonprj/oops7.py ======================
=
Name: Student Name
Roll No: Student Roll Number
Method outputs Student NameStudent Roll Number
>>>
```

Here, **student** is the name of class, ***Name*** and***R_No*** are the attributes and ***student_details()*** is the method defined inside the class ***student.***

An object ***obj*** is created as instance to the class ***student*** in the program and the attributes and method inside the class is used in the program as below:

```
print("Name: "+obj.Name)
print("Roll No: "+obj.R_No)
print("Method outputs "+obj.student_details())
```

__init__() Function

All classes have a function called ***__init__(),*** which is always executed when the class is being initiated. ***__init__()*** function is used to assign values to object properties, and other operations that are necessary to do when the object is being created.

The ***__init__()*** function is called automatically every time the class is being used to create a new object.

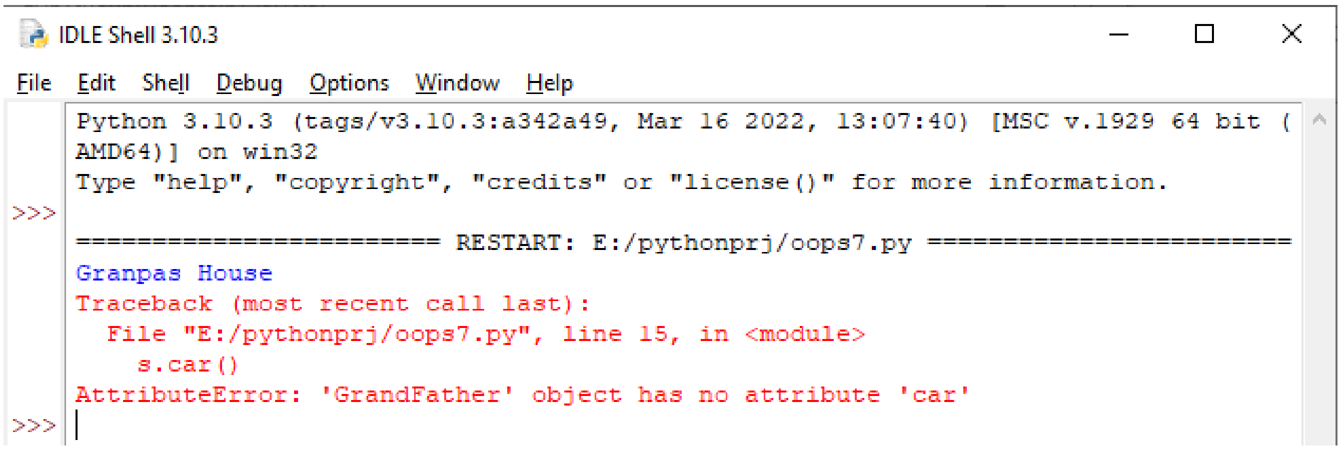
Example
Create a class named Person, use the __init__() function to assign values for name and age:

```
      class Person:
        def __init__(self, name, age):
          self.name = name
          self.age = age
      p1 = Person("John", 36)
      print(p1.name)
      print(p1.age)
```

Normally in a program if more than one classes are defined each of their attributes and methods can be used in the program by separately creating object instances to each classes and used. This is because, an attribute or a method inside a class belongs to that class and it cannot be used or called inside another class. Consider the below program that uses three classes:

```
classGrandFather:
    def House(self):
        print("Granpas House")
class Father:
    def car(self):
        print("Fathers Car")
class me:
    def bike():
        print("My Bike")
s=GrandFather()
s.House()
s.car()
s.bike()
```

The output prints only the string used in the print statement under the class***GrandFather*** by calling the method ***House()*** that belongs to this class. Others show error as below:

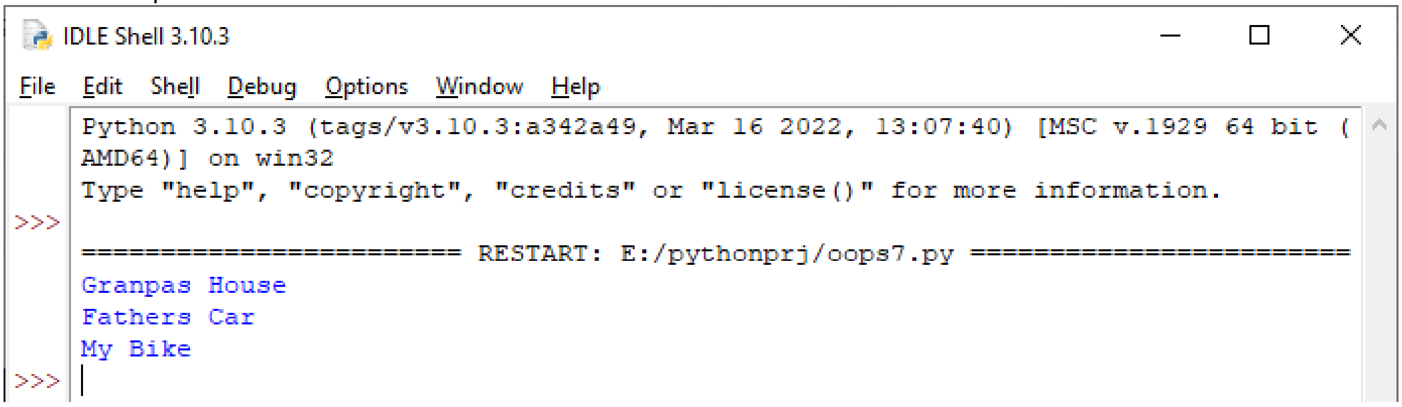


```
Python 3.10.3 (tags/v3.10.3:a342a49, Mar 16 2022, 13:07:40) [MSC v.1929 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
======================== RESTART: E:/pythonprj/oops7.py ========================
Granpas House
Traceback (most recent call last):
  File "E:/pythonprj/oops7.py", line 15, in <module>
    s.car()
AttributeError: 'GrandFather' object has no attribute 'car'
>>> |
```

To print others it is mandate to create separate object for each class to avoid the error as in the below program:

```
classGrandFather:
    def House(self):
        print("Granpas House")
class Father:
    def car(self):
        print("Fathers Car")
class me:
    def bike(self):
        print("My Bike")
s=GrandFather()
c=Father()
v=me()
s.House()
c.car()
v.bike()
```

Now the output is



```
Python 3.10.3 (tags/v3.10.3:a342a49, Mar 16 2022, 13:07:40) [MSC v.1929 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
======================== RESTART: E:/pythonprj/oops7.py ========================
Granpas House
Fathers Car
My Bike
>>> |
```

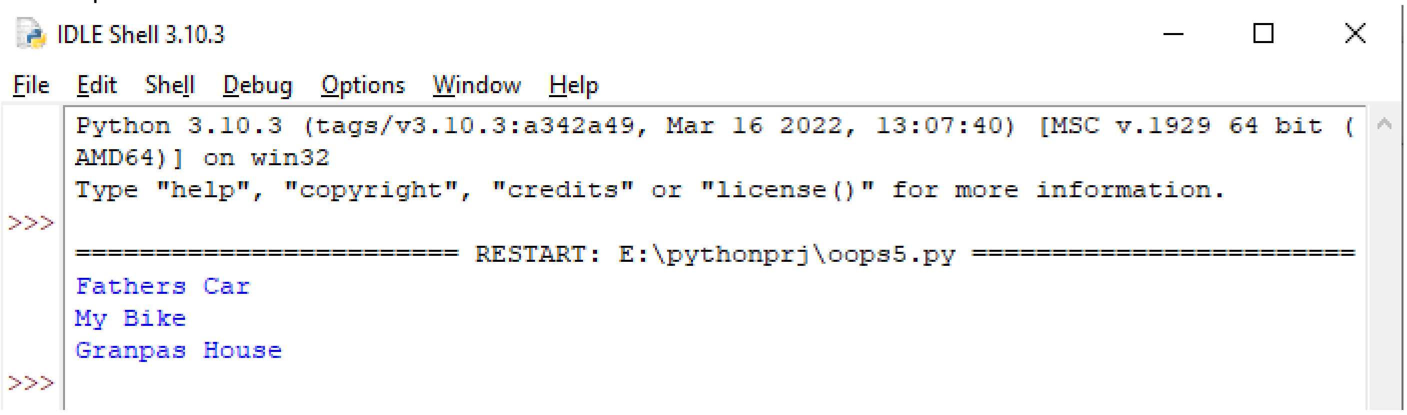
Inheritance

It is also possible to access all the classes, their attributes and methods by creating single object instance. This can be done to make each classes to inherit each other. Let us see the below example program of Inheritance.

```
# Inheritance
classGrandFather:
    def House(self):
        print("Granpas House")
```

```
class Father(GrandFather):
    def car(self):
        print("Fathers Car")
class me(Father):
    def bike(self):
        print("My Bike")
s=GrandFather()
s.House()
s.car()
s.bike()
```

The output is



```
Python 3.10.3 (tags/v3.10.3:a342a49, Mar 16 2022, 13:07:40) [MSC v.1929 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
======================== RESTART: E:\pythonprj\oops5.py ========================
Fathers Car
My Bike
Granpas House
>>>
```

Inheritance allows us to define a class that inherits all the methods and properties from another class.
Parent class is the class being inherited from, also called base class.
Child class is the class that inherits from another class, also called derived class

The self Parameter

The `self` parameter is a reference to the current instance of the class. It is used to access variables that belongs to the class. It is not compulsary to be named as `self`, you can call it whatever you like, but it must be placed as the first parameter of any function in the class.

Example
Use the words *mysillyobject* and *abc* instead of *self*:

```
class Person:
  def __init__(mysillyobject, name, age):
    mysillyobject.name = name
    mysillyobject.age = age

  def myfunc(abc):
    print("Hello my name is " + abc.name)
p1 = Person("John", 36)
p1.myfunc()
```

Modify Object Properties

You can modify properties on objects like this:

Example
Set the age of p1 to 40:

```
p1.age = 40
```

Delete Object Properties

You can delete properties on objects by using the `del` keyword:

Example

Delete the age property from the p1 object:

```
del p1.age
```

The pass Statement

`class` definitions cannot be empty, but if you for some reason have a `class` definition with no content, put in the **`pass`** statement to avoid getting an error.

Example

```
class Person:
  pass
```

Polymorphism

When calling a function having input parameters, all parameters must be assigned with a value without leaving any parameter. Otherwise, python shows error.

There are special situation where any one of the input parameters will be either filled with a value or can be ignored passing value. Such function call is called polymorphism.

Example 1:

```
def fnPoly1(a,b,c=0):
     return a+b+c
```

The above function can be called in anyone of the below ways:

Usage-1:

```
print(fnPoly1(3,4,5))
```

Will output the result as:

```
>>>12
```

Usage-2:

```
print(fnPoly(3,4))
```

Will output the result as:

```
>>7
```

Example 2:

```
def fnFullName(firstName,lastName,surName=””):
      return firstName+” “+lastName+” “+surName
```

The above functions can be called in anyone of the below method:

Usage-1:

```
print(fnFullName(“Mark Hamill”,”Anakin”)
```

Will output the result as:

```
>>>Mark Hamill Anakin
```

Usage-2:

```
Print(fnFullName(“Mark Hamill”,”Anakin”,”SkyWalker”)
```

Will output the result as:

```
>>>Mark Hamill Anakin SkyWalker
```

Python Iterators

An iterator is an object that contains a countable number of values.
An iterator is an object that can be iterated upon, meaning that you can traverse through all the values.
Technically, in Python, an iterator is an object which implements the iterator protocol, which consist of the methods ***__iter__()*** and ***__next__().***

Iterator vs Iterable

Lists, tuples, dictionaries, and sets are all iterable objects. They are iterable containers which you can get an iterator from.

All these objects have a ***iter()*** method which is used to get an iterator:

Example
Return an iterator from a tuple, and print each value:

```
mytuple = ("apple", "banana", "cherry")
myit = iter(mytuple)

print(next(myit))
print(next(myit))
print(next(myit))
```

Even strings are iterable objects, and can return an iterator:
Example
Strings are also iterable objects, containing a sequence of characters:

```
mystr = "banana"
myit = iter(mystr)

print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
```

Looping Through an Iterator

We can also use a **for** loop to iterate through an iterable object:

Example
Iterate the values of a tuple:

```
mytuple = ("apple", "banana", "cherry")
for x in mytuple:
  print(x)
```

Example

Iterate the characters of a string:
```
mystr = "banana"
for x in mystr:
  print(x)
```

The for loop actually creates an iterator object and executes the next() method for each loop.

Create an Iterator

To create an object/class as an iterator you have to implement the methods ***__iter__()*** and ***__next__()*** to your object.

As you have learned in the Python Classes/Objects chapter, all classes have a function called ***__init__()***, which allows you to do some initializing when the object is being created.

The ***__iter__()*** method acts similar, you can do operations (initializing etc.), but must always return the iterator object itself.

The ***__next__()*** method also allows you to do operations, and must return the next item in the sequence.

Example

Create an iterator that returns numbers, starting with 1, and each sequence will increase by one (returning 1,2,3,4,5 etc.):

```
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self

  def __next__(self):
    x = self.a
    self.a += 1
    return x
myclass = MyNumbers()
myiter = iter(myclass)

print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
stopIteration
```

The example above would continue forever if you had enough ***next()*** statements, or if it was used in a for loop.
To prevent the iteration to go on forever, we can use the ***stopIteration*** statement.

In the ***__next__()*** method, we can add a terminating condition to raise an error if the iteration is done a specified number of times:

Example

Stop after 20 iterations:

```
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self

  def __next__(self):
    if self.a <= 20:
      x = self.a
      self.a += 1
```

```
            return x
          else:
            raise stopIteration

      myclass = MyNumbers()
      myiter = iter(myclass)

      for x in myiter:
        print(x)
```

Python Scope

A variable is only available from inside the region it is created. This is called scope.

Local Scope

A variable created inside a function belongs to the local scope of that function, and can only be used inside that function.

Example

A variable created inside a function is available inside that function:

```
def myfunc():
  x = 300
  print(x)
myfunc()
```

Function Inside Function

As explained in the example above, the variable x is not available outside the function, but it is available for any function inside the function:

Example

The local variable can be accessed from a function within the function:

```
def myfunc():
  x = 300
  def myinnerfunc():
    print(x)
  myinnerfunc()
myfunc()
```

Global Scope

A variable created in the main body of the Python code is a global variable and belongs to the global scope.

Global variables are available from within any scope, global and local.

Example

A variable created outside of a function is global and can be used by anyone:

```
x = 300
def myfunc():
  print(x)
myfunc()
print(x)
```

Naming Variables

If you operate with the same variable name inside and outside of a function, Python will treat them as two separate variables, one available in the global scope (outside the function) and one available in the local scope (inside the function):

Example

The function will print the local x, and then the code will print the global x:

```
x = 300
def myfunc():
  x = 200
  print(x)
myfunc()
print(x)
```

Global Keyword

If you need to create a global variable, but are stuck in the local scope, you can use the global keyword.
The global keyword makes the variable global.

Example

If you use the global keyword, the variable belongs to the global scope:

```
def myfunc():
  global x
  x = 300
myfunc()
print(x)
```

Also, use the global keyword if you want to make a change to a global variable inside a function.

Example

To change the value of a global variable inside a function, refer to the variable by using the global keyword:

```
x = 300
def myfunc():
  global x
  x = 200
myfunc()
print(x)
```

Python Modules

A module is an object like a code library.
A file containing a set of functions you want to include in your application.

Create a Module

Save the code in a file with the file extension .py:

Example

Save this code in a file named mymodule.py

```
def greeting(name):
  print("Hello, " + name)
```

Use a Module

Now we can use the module we just created, by using the import statement:

Example
Import the module named ***mymodule***, and call the greeting function:
```
import mymodule
mymodule.greeting("Ben Kenobi")
```

Note: When using a function from a module, use the syntax:
module_name.function_name.

Variables in Module
The module can contain functions, and variables of all types (arrays, dictionaries, objects etc.):

Example
Save this code in the file mymodule.py
```
person1 = {
  "name": "John",
  "age": 36,
  "country": "Norway"
}
```

Example
Import the module named mymodule, and access the person1 dictionary:
```
import mymodule
a = mymodule.person1["age"]
print(a)
```

Naming a Module
You can name the module file whatever you like, but it must have the file extension .py

Re-naming a Module
You can create an alias when you import a module, by using the as keyword:

Example
Create an alias for ***mymodule*** called mx:
```
import mymodule as mx
a = mx.person1["age"]
print(a)
```

Built-in Modules
There are several built-in modules in Python, which you can import whenever you like.

Example
Import and use the platform module:
```
import platform
x = platform.system()
print(x)
```

Using the dir() Function
There is a built-in function to list all the function names (or variable names) in a module. The dir() function:

Example
List all the defined names belonging to the platform module:
```
import platform
x = dir(platform)
```

```
print(x)
```

Note: The dir() function can be used on all modules, also the ones you create yourself.

Import From Module
You can choose to import only parts from a module, by using the from keyword.

Example
The module named mymodule has one function and one dictionary:

```
def greeting(name):
  print("Hello, " + name)
person1 =
{
  "name": "John",
  "age": 36,
  "country": "Norway"
}
```

Example
Import only the person1 dictionary from the module:

```
from mymodule import person1
print (person1["age"])
```

Note: When importing using the from keyword, do not use the module name when referring to elements in the module.

Example:

```
person1["age"], not mymodule.person1["age"]
```

Python Datetime

Python Dates
A date in Python is not a data type of its own, but we can import a module named datetime to work with dates as date objects.

Example
Import the datetime module and display the current date:

```
import datetime
x = datetime.datetime.now()
print(x)
```

Date Output
When we execute the code from the example above the result will be:

```
2022-11-21 22:43:37.111041
```

The date contains year, month, day, hour, minute, second, and microsecond.
The datetime module has many methods to return information about the date object.
Here are a few examples, you will learn more about them later in this chapter:

Example
Return the year and name of weekday:

```
import datetime
x = datetime.datetime.now()
print(x.year)
print(x.strftime("%A"))
```

Creating Date Objects
To create a date, we can use the datetime() class (constructor) of the datetime module.
The datetime() class requires three parameters to create a date: year, month, day.

Example
Create a date object:

```
import datetime
x = datetime.datetime(2020, 5, 17)
print(x)
```

The datetime() class also takes parameters for time and timezone (hour, minute, second, microsecond, tzone), but they are optional, and has a default value of 0, (None for timezone).

The strftime() Method
The datetime object has a method for formatting date objects into readable strings.
The method is called strftime(), and takes one parameter, format, to specify the format of the returned string:

Example
Display the name of the month:

```
import datetime
x = datetime.datetime(2018, 6, 1)
print(x.strftime("%B"))
```

A reference of all the legal format codes:

Handling files

Creating a file

In python, if a file is opened in output mode (write, append), the file will be created if does not exist.

Opening a file

To perform any input or output operation, the file must be opened first. To open a file python provides ***Open()*** command. The usage of ***Open()*** command is described below:

General Format:

Open(<filename>,<mode>)

Filename: Name of the file. If the filename is given along with the pathname, the file will be opened from the specified path. If path is not specified the file will be opened from the default folder from where the python program runs and if it does not exist, python will throw error.

Mode: Tells the program to open the specific file either in read, write or append etc. modes. Python provides a list of modes as below:

w- Write mode. If the specific file exists in the specific path, it will be opened in output mode and it will clear its previous contents and writes the content newly.

r- Read mode. It is used when a file is opened for reading its contents.

a- Append mode. It used to add contents to the file. The difference between ***Append*** and ***Write*** mode is that when a file is opened in a ***Write*** mode, it will clear previously stored contents but Append will add the newly written contents at the end of previously stored contents.

Example

```
f1=open(“E:\Pythonprj\file1.txt”,”w”)
```

This line will open a file named file1.txt on the specified path ***E:\Pythonprj\file1.txt*** and it is ready to write contents in it as it is opened in “**w**” mode. This mode will open the specified file if exists in the path. If it does not exist, it will be created. Thus ***open*** command will open/create a file.

Writing contents to the file

Writing contents to the file can be done in many methods.The second line of the program will be written in the file aster opening/creating the file.

```
f1=open(“E:\Pythonprj\file1.txt”,”w”)
f1.write("This is sample line1")
```

A write statement can write only one line at a time. To repeatedly write multiple lines to the file, write command must be executed repeatedly with different contents. This can be done using either ***For*** or ***While*** loop.

The below program creates a file in write mode:

Example:

Program1:

```
# Opening a file in Write mode
f1=open("E:\pythonprj\File1.txt","w")
f1.write("This is sample line1")
f1.close()
```

Program2:

```
# Opening a file in Write mode
f1=open("E:\pythonprj\File1.txt","w")
f1.write("This is sample lineA")
f1.write("This is sample lineB")
f1.close()
```

The first program in the above example creates a file named as ***File1.txt*** in the specified path ***E:\pythonprj*** and writes the line "**This is sample line 1**". If we observe the second program, the same file is opened in *Write* mode same as in program1 and it writes 2 lines "**This is sample line A**" and "**This is sample line B**" in it by deleting previously written line.

If observed the below program, the two new lines are written as a single line.

```
# Opening a file in Write mode
f1=open("E:\pythonprj\File1.txt","w")
f1.write("This is sample line1")
f1.write("This is sample line2")
f1.close()
```

To write these lines as separate line, add a "\n" at the beginning/end as shown below:

```
# Opening a file in Write mode
f1=open("E:\pythonprj\File1.txt","w")
f1.write("This is sample line1\n")
f1.write("This is sample line2\n")
f1.close()
```

Appending contents to the file

Writing contents to an existing file by opening it in "**a**" (append) mode will add the new lines in write statement, to the bottom of the file.

```
# Opening a file in Append Mode
f1=open("E:\pythonprj\File1.txt","a")
f1.write("\nThis is sample line3")
f1.write("\nThis is sample line4")
f1.close()
```

The above program will add the lines to the file that is opened in "a" mode. Unlike "w" (write) mode, opening this file will not delete previous contents of the file but will add the new contents along with the old contents of the file. After executing the above program, the file contains below lines:

```
This is sample line1
This is sample line2
This is sample line3
This is sample line4
```

Reading contents from file

Read operation can be done only in a file that exists physically in the specified path. Reading a file can be done using either ***read()*** or ***readln()*** commands.

Example

```
f3=open("E:\pythonprj\File1.txt","r")
data=f3.readline()
print(data)
f3.close
```

The above example program reads the content of the file line by line. Here only one ***readline()*** statement is found and it will read only one line (usually first line). If the ***readline()*** command is put within a loop, it reads successive lines when executed repeatedly.

Example

```
try:
    f3=open("E:\pythonprj\File1.txt","r")
    print(f3.writable())
    print(f3.readable())
    print(f3.mode)
    data=f3.readline()
    data1=f3.readlines()
    print(type(data1))
    print(data1[0])
    f3.close()
except FileNotFoundError:
    print("File Not Found")
```

Rewriting contents to the file

Sample Program for Rewriting Contents (r+ mode)

```
#Rewriting/Updating contents of a file
#=====================================
f = open('D:\pythonprj\score.txt','w')
score=45
print(score)
f.write(str(score))
f.close()

f = open('D:\pythonprj\score.txt','r+')
PlayersScore = 1
oldscore = (f.read())
oldscore = int(oldscore)
score = oldscore + PlayersScore
print(score)
f.write(str(score))
f.close()
```

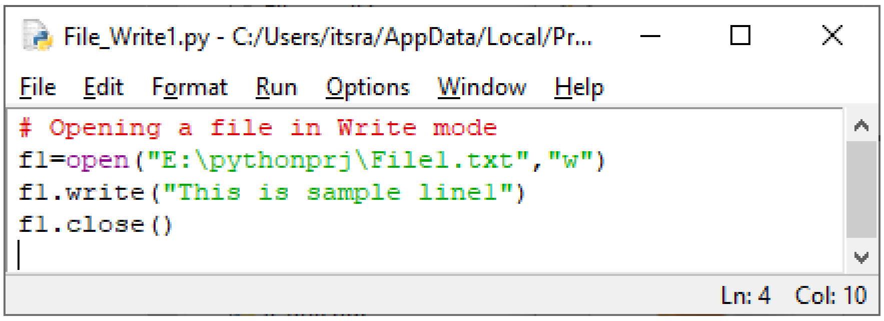
Closing a file

Example:
The below programs show how to open a file in different modes and how to read, write and to append contents from/to a file.

The first program creates a file named as ***File1.txt*** in the specified path ***E:\pythonprj*** and writes the line "**This is sample line 1**". If we observe the second program, the same file is opened in *Write* mode same as in program1 and it writes 2 lines "**This is sample line A**" and "**This is sample line B**" in it by deleting previously written line. Also if observed, the two new lines are written as a single line. To write these lines as separate line, add a "\n" at the end of the first line. This is explained in Program-3.

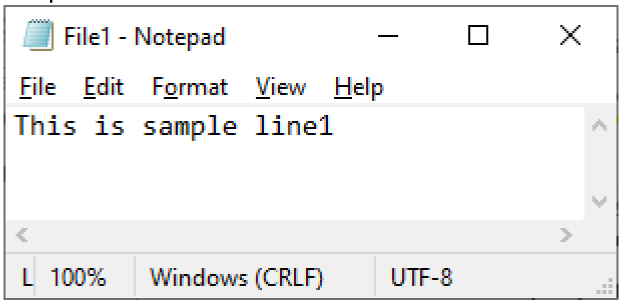
A read / write statement can read / write only one line at a time. To repeatedly read / write multiple lines from / to the file, these commands must be executed repeatedly. For this purpose, we can either use ***For*** or ***While*** loop. Program-4 explains how to read contents from a file in different ways using ***read()***, ***readline(), readlines()*** and to write contents to a file in different ways using ***write(), writeline(),writelines()*** commands.
Program-5 explains how a file content can be read and rewritten as an update.

Program1:


File_Write1.py - C:/Users/itsra/AppData/Local/Pr...
File Edit Format Run Options Window Help
Ln: 4 Col: 10

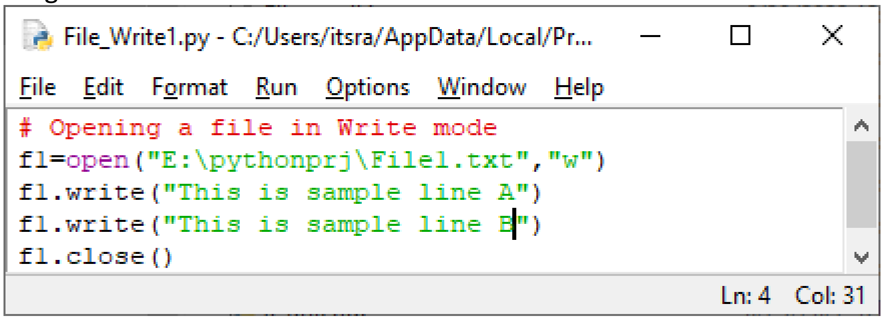
```
# Opening a file in Write mode
f1=open("E:\pythonprj\File1.txt","w")
f1.write("This is sample line1")
f1.close()
```

Output1:



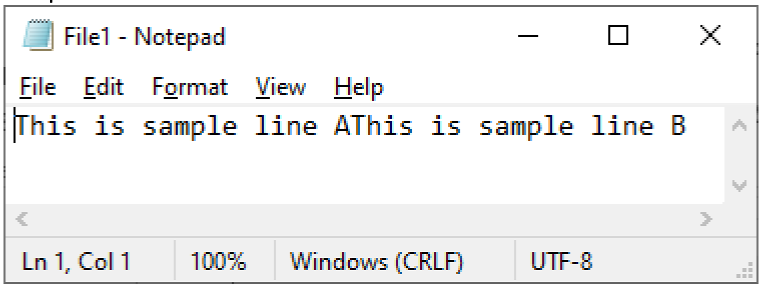
```
This is sample line1
```

Program-2:


File_Write1.py - C:/Users/itsra/AppData/Local/Pr...
File Edit Format Run Options Window Help
Ln: 4 Col: 31

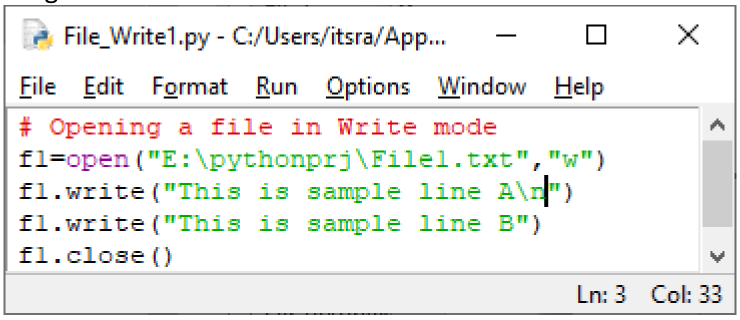
```
# Opening a file in Write mode
f1=open("E:\pythonprj\File1.txt","w")
f1.write("This is sample line A")
f1.write("This is sample line B")
f1.close()
```

Output-2:


File1 - Notepad
File Edit Format View Help
Ln 1, Col 1 100% Windows (CRLF) UTF-8

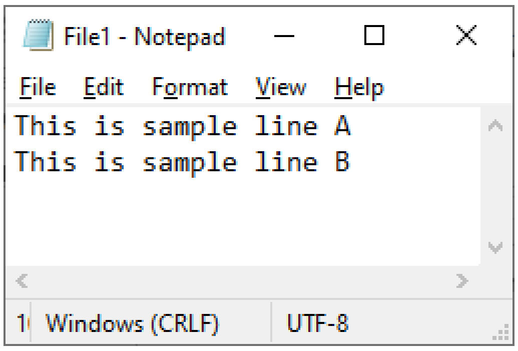
```
This is sample line AThis is sample line B
```

Program-3:


File_Write1.py - C:/Users/itsra/App...
File Edit Format Run Options Window Help
Ln: 3 Col: 33

```
# Opening a file in Write mode
f1=open("E:\pythonprj\File1.txt","w")
f1.write("This is sample line A\n")
f1.write("This is sample line B")
f1.close()
```

Output-3:

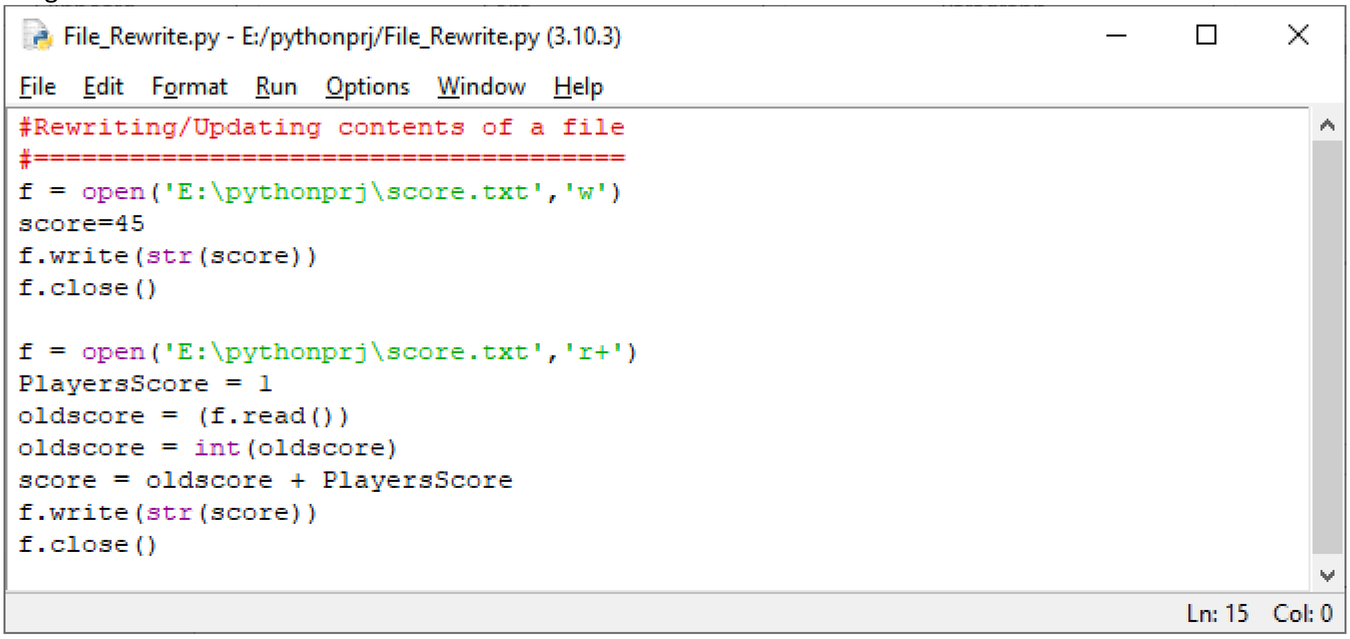


```
This is sample line A
This is sample line B
```

Program-4:

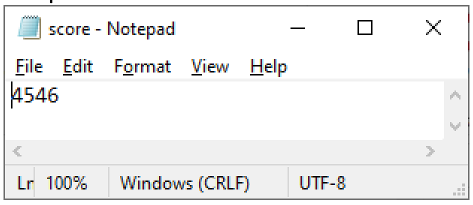
```
# Opening a file in Write mode
f1=open("E:\pythonprj\File1.txt","w")
f1.write("This is sample line1")
f1.close()

# Opening a file in Write mode
f1=open("E:\pythonprj\File1.txt","w")
f1.write("This is sample line1")
f1.write("This is sample line2")
f1.close()

# Opening a file in Append Mode
f1=open("E:\pythonprj\File1.txt","a")
f1.write("\nThis is sample line5")
f1.write("\nThis is sample line6")
f1.close()

#====
f2=open("E:\pythonprj\File2.txt","w")
lst=["This is first line","This is Second Line","This is 3rd Line"]
for each_line in lst:
    f2.write(each_line)
    print(each_line)
f2.close

#===========
f3=open("E:\pythonprj\File1.txt","r")
data=f3.readline()
print(data)
f3.close

#========
try:
    f3=open("E:\pythonprj\File1.txt","r")
    print(f3.writable())
    print(f3.readable())
    print(f3.mode)
    data=f3.readline()
    data1=f3.readlines()
    print(type(data1))
    print(data1[0])
    f3.close()
except FileNotFoundError:
    print("File Not Found")
```

Program-5:


File_Rewrite.py - E:/pythonprj/File_Rewrite.py (3.10.3)
File Edit Format Run Options Window Help
Ln: 15 Col: 0

```
#Rewriting/Updating contents of a file
#=====================================
f = open('E:\pythonprj\score.txt','w')
score=45
f.write(str(score))
f.close()

f = open('E:\pythonprj\score.txt','r+')
PlayersScore = 1
oldscore = (f.read())
oldscore = int(oldscore)
score = oldscore + PlayersScore
f.write(str(score))
f.close()
```

Output-5:



```
4546
```

Sample program for opening file in all modes

```
# Opening a file in Write mode
f1=open("D:\pythonprj\File1.txt","w")
f1.write("\nThis is sample line1")
f1.write("\nThis is sample line2")
f1.close()

# Opening a file in Append Mode
f1=open("D:\pythonprj\File1.txt","a")
f1.write("\nThis is sample line3")
f1.write("\nThis is sample line4")
f1.close()

# Opening a file in Read Mode
f3=open("D:\pythonprj\File1.txt","r")
for i in range(0,5):
    data=f3.readline()
    print(data)
f3.close

# Opening a file in Write mode
f1=open("D:\pythonprj\File1.txt","w")
f1.write("This is sample line1")
f1.write("This is sample line2")
```

```
f1.close()

# Opening a file in Append Mode
f1=open("D:\pythonprj\File1.txt","a")
f1.write("This is sample line3")
f1.write("This is sample line4")
f1.close()

# Opening a file in Read Mode
f3=open("D:\pythonprj\File1.txt","r")
for i in range(0,5):
    data=f3.readline()
    print(data)
f3.close

# Opening a file in Read Mode
f3=open("D:\pythonprj\File1.txt","r")
data=f3.readline()
print(data)
f3.close

# Opening a file in Write mode
f1=open("D:\pythonprj\File1.txt","w")
f1.writelines("This is sample line1")
f1.writelines("This is sample line2")
f1.writelines("This is sample line3")
f1.writelines("This is sample line4")
f1.close()

# Opening a file in Read Mode
f3=open("D:\pythonprj\File1.txt","r")
data=f3.readlines()
print(data)
f3.close

#ReadLine File
f2=open("D:\pythonprj\File1.txt","w")
lst=["This is first line","This is Second Line","This is 3rd Line"]
for each_line in lst:
    f2.write(each_line)
    print(each_line)
print(type(lst))
print(f2.mode)
f2.close
```

Reading Contents Of CSV (Comma Seperated) Excel File Contents:

```
import csv
with open("book1.csv", 'r') as csvfile:
    rows = csv.reader(csvfile)
    for row in rows:
        print(row)
```

Reading EXCEL File

```
import xlrd
location = "book1.xlsx"
wb = xlrd.open_workbook(location)
sheet = wb.sheet_by_index(0)
print(sheet.cell_value(0, 0)
```

Python with databases.

Using python we can develop database application. To develop a database application through any language, the below steps are to be followed:

- Create Connectivity between database and the language.
- Open connection
  - Create database if does not exist.
  - Open database if exists.
    - Create table if does not exist in the opened database.
    - Add new rows to the table (adding records),
    - Update existing row(s) (modifying existing row content)
    - Delete selected row(s)
  - Close database
- Close Connection.

Most of the language provides connectivity solution in such a way that, a database canbe opened while establishing a connection.

Python Database Connectivity

Python also allows us to connect with any database and to work with that. Python has many modules to write programs for various functions using its packages and classes, To work with databases the the class ***connect*** inside the ***connector*** is used. For example, to work with MySQL database, python provides ***mysql.connector*** package. The class ***mysql.connector.connect*** allows python to establish a connection to mySql database. This class has the below parameters:

host=<Database Server Name>
user=<User name>
password=<Password>
database=<Database name>

Python with MySQL

Create MySql database Connection

Using ***connect()*** method of mysql.connector class, we can create a connection to a MySql database by specifying the attributes for connect() method.
The general format is:

```
<Object Name>=mysql.connector.connect
        (
                host=”<Server Name>”,
                user=”<User Name>”,
                password=”<Password>”
```

Test MySQL Connectio

The below program connects to a database named ***trade*** that is available in mysql database server in the ***localhost:***

```
import mysql.connector
con=mysql.connector.connect(host='localhost',user='root',password='',database='trade')
if con:
   print('connected')
else:
   print('not connected')
```

Username and password are of the MySQL database:

Afer establishing / creating the connection to the database, we can start querying the database using SQL statements.

Database Creation

To create a database, table, adding rows, updating rows, deleting and reading rows from a table can be done with SQL Statements, There are separate SQL statements are available for each action. Here, as part of the python tutorial, we use the appropriate SQL statements for each action. Though it will be easy to understand, ***it is important to undergo detail study about the SQL that is available as a separate course.***

Example

The below code segment is used to create a database named "mydb"

```
import mysql.connector
con = mysql.connector.connect
(
    host="localhost",
    user="<username>",
    password="<password>"
)
cur = con.cursor()
cur.execute("CREATE DATABASE <database Name>")
```

Executing the above program segment with no errors will create a database in the name mentioned as <database Name> successfully.

Check if a Database Exists

You can check if a database exist by listing all databases in your system by using the "SHOW DATABASES" statement:

Example

Return a list of your system's databases:

```
import mysql.connector
con = mysql.connector.connect
(
  host="localhost",
  user="<user name>",
  password="<password>"
)

cur = con.cursor()
```

```
cur.execute("SHOW DATABASES")
for dbName in cur:
    print(dbName)
```

Or the database can be accessed when making the connection:

Example

Try connecting to the database "dB":

```
import mysql.connector
con = mysql.connector.connect
(
  host="localhost",
  user="<username>",
  password="<password>",
  database="dB"
)
```

If the database does not exist, you will get an error.

Create a Table

To create a table in MySQL, use the "CREATE TABLE" statement.
Make sure you define the name of the database when you create the connection

Example

Create a table named "Employees":

```
import mysql.connector
con = mysql.connector.connect
(
  host="localhost",
  user="<username>",
  password="<password>",
  database="dB"
)
cur = con.cursor()
cur.execute("CREATE TABLE Employees (Emp_No VARCHAR(10), Basic Float)")
```

If the above code is executed with no errors, a new table named “Employees” is successfully created.

Check if Table Exists

To check if a table exists, list all the tables in a database with the "SHOW TABLES" statement:

Example

```
import mysql.connector
con = mysql.connector.connect
(
  host="localhost",
  user="<username>",
  password="<password>",
  database="dB"
)
```

```
cur = con.cursor()
cur.execute("SHOW TABLES")
for tblName in curr:
    print(tblName)
```

Return a list of tables in a database:

Insert a row in a table

To fill a table in MySQL, use the "INSERT INTO" statement.

```
import mysql.connector
con=mysql.connector.connect(host='localhost',user='root',password='',database='test')
cur = con.cursor()
sql = "INSERT INTO Employees (Emp_No, Basic) VALUES (%s, %s)con"
val = ("Mr. John Williams", "23000")
cur.execute(sql, val)
con.commit()
print(cur.rowcount, " record(s) inserted.")
```

Important!: Notice the statement: **db.commit()**. It is required to make the changes, otherwise no changes are made to the table.

Insert Multiple Rows

To insert multiple rows into a table, use the ***executemany()*** method.
The second parameter of the *executemany()* method is a list of tuples, containing the data you want to insert:

Example

```
import mysql.connector
dbCon = mysql.connector.connect
(
  host="localhost",
  user="root",
  password="",
  database="test"
)
cur = dbCon.cursor()
sql = "INSERT INTO Employees (Emp_No, Basic) VALUES (%s, %s)"
val = [
  ('Diana Palmer', '5500'),
  ('Lamanda Luaga', '10400'),
  ('Coranda', '9500'),
  ('Col.Worobu', '7200'),
  ('Rex King', '4500'),
  ('Mozz', '5000'),
  ('Guran', '3400'),
  ('Tagama', '5800'),
  ('Dave Palmer', '10000')
]
cur.executemany(sql, val)
dbCon.commit()
print(cur.rowcount, " rows (s) where inserted.")
```

Get Inserted ID

You can get the id of the row you just inserted by asking the cursor object.
Note: If you insert more than one row, the id of the last inserted row is returned.

Example

Insert one row, and return the ID:

```
import mysql.connector
dbCon = mysql.connector.connect
(
  host="localhost",
  user="root",
  password="",
  database="test"
)
cur = dbCon.cursor()
sql = "INSERT INTO Employees (Emp_No, Basic) VALUES (%s, %s)"
val = ("Tom Tom", "3300")
cur.execute(sql, val)
dbCon.commit()
print("1 record inserted, ID:", cur.lastrowid)
```

Select From a Table

Using the "SELECT" statement, table rows can be Selected / Read:

Example

Select all records from the "Employees" table, and display the result:

```
import mysql.connector
dbCon = mysql.connector.connect
(
  host="localhost",
  user="root",
  password="",
  database="test"
)
cur = dbCon.cursor()
cur.execute("SELECT * FROM Employees")
rs = cur.fetchall()
for rows in rs:
  print(rows)
```

Note: We use the ***fetchall()*** method, which fetches all rows from the last executed statement.

Selecting Columns

To select only some of the columns in a table, use the "SELECT" statement followed by the column name(s):

Example

Select only the name and address columns:

```
import mysql.connector
dbCon = mysql.connector.connect
(
  host="localhost",
  user="root",
  password="",
  database="test"
)
cur = dbCon.cursor()
cur.execute("SELECT emp_No FROM Employees")
rs = cur.fetchall()
for rpw in rs:
    print(rs)
```

fetchone() Method

If you are only interested in one row, you can use the ***fetchone()*** method.

Example

Fetch single row will return the first row of the result:

```
import mysql.connector
dbCon = mysql.connector.connect
(
  host="localhost",
  user="root",
  password="",
  database="test"
)
cur = dbCon.cursor()
cur.execute("SELECT * FROM Employees")
rs = cur.fetchone()
print(rs)
```

Select With a Filter

When selecting records from a table, you can filter the selection by using the "WHERE" statement:

Example

Select record(s) where Basic is "3400":

```
import mysql.connector
dbCon = mysql.connector.connect
(
  host="localhost",
  user="yourusername",
  password="yourpassword",
  database="test"
)
cur = dbCon.cursor()
sql = "SELECT * FROM Employees WHERE Basic ='3400'"
cur.execute(sql)
rs = cur.fetchall()
for row in rs:
    print(row)
```

Wildcard Characters

To select the rows of columns that starts, includes, or ends with a given letter or phrase, the wildcard characters % * are used:

Example

Select records where the contains the word "ABC":

```
import mysql.connector
dbCon = mysql.connector.connect
(
  host="localhost",
  user="root",
  password="",
  database="test1"
)
cur = dbCon.cursor()
sql = "SELECT * FROM Employees WHERE Emp_No LIKE '%ABC%'"
cur.execute(sql)
rs = cur.fetchall()
for row in rs:
    print(row)
```

Prevent SQL Injection

When query values are provided by the user, you should escape the values. This is to prevent SQL injections, which is a common web hacking technique to destroy or misuse your database. The ***mysql.connector*** module has methods to escape query values:

Example

Escape query values by using the placholder %s method:

```
import mysql.connector
mydb = mysql.connector.connect
(
  host="localhost",
  user="yourusername",
  password="yourpassword",
  database="mydatabase"
)
mycursor = mydb.cursor()
sql = "SELECT * FROM customers WHERE address = %s"
adr = ("Yellow Garden 2", )
mycursor.execute(sql, adr)
myresult = mycursor.fetchall()
for x in myresult:
    print(x)
```

Delete row

You can delete rows from an existing table by using the "DELETE FROM" statement:

Example

Delete any record where the address is "Mountain 21":

```
import mysql.connector
con=mysql.connector.connect(host='localhost',user='root',password='',database='test')
if con:
    id=input("Enter Sales ID to delete: ")
    res=con.cursor()
    sql="delete from sales where sale_id=%s"
    print(sql)
    user=(id,)
    res.execute(sql,user)
    con.commit()
    print(mycursor.rowcount, "record(s) deleted")
else:
    print("Cannot connect to database")
```

Note WHERE clause in the **DELETE -** The WHERE clause specifies which record(s) that should be deleted. If you omit the WHERE clause, all records will be deleted!

CRUD Program

CRUD is the abbreviated term for **C**reate, **R**ead, **U**pdate and **D**elete. A program written to create a database content, read, update and delete it is called CRUD program.

There are special database language commands are available in all the databases for data manipulations such as add/insert, update/modify, delete / remove a data row to a data table in the database in connection. The special data manipulation language is called SQL (Structured Query Language).

To execute any SQL command to manipulate a data table of a database, the steps to be carried out in an order listed below:

1. Establish database connection successfully.
2. Create an instance for a cursor to the connection.
3. Execute the SQL command by calling execute method of the connection cursor.
4. If any read and write operation (Insert, Update or Delete SQL) was performed, make the changes permenant to the data table using the method ***commit()***

Creating a new data row

An INSERT SQL will be executed to insert a data row to a table. Below python code segment inserts a data row:

```
import mysql.connector
con=mysql.connector.connect(host='localhost',user='root',password='',database='trade')
if con:
   print('connected')
else:
   print('not connected')

res=con.cursor()
sql="insert into smpl (id,name,age) values (%s, %s, %s)"
```

```
print(sql)
user=(id,name,age)
res.execute(sql,user)
con.commit()
print("Inserted")
```

Updating existing data row

An UPDATE SQL will be executed to modify an existing data row in a table. Below python code segment inserts a data row:

```
import mysql.connector
con=mysql.connector.connect(host='localhost',user='root',password='',database='trade')
if con:
   print('connected')
else:
   print('not connected')
res=con.cursor()
sql="update smpl set id=%s,name=%s,age=%s where id=%s"
print(sql)
user=(id,name,age,id1)
res.execute(sql,user)
con.commit()
print("Updated")
```

Deleting an existing row

A DELETE SQL will be executed to remove an existing data row in a table. Below python code segment inserts a data row:

```
import mysql.connector
con=mysql.connector.connect(host='localhost',user='root',password='',database='trade')
if con:
   print('connected')
else:
   print('not connected')
res=con.cursor()
sql="delete from smpl where id=%s"
print(sql)
user=(id,)
res.execute(sql,user)
con.commit()
print("Record id is deleted")
```

Reading a data row

To read a data row from a data table of a database, SELECT SQL will be executed as in the below code segment:

```
from tabulate import tabulate
import mysql.connector
con=mysql.connector.connect(host='localhost',user='root',password='',database='trade')
if con:
   print('connected')
else:
   print('not connected')
res=con.cursor()
sql="SELECT * FROM SMPL"
res.execute(sql)
result=res.fetchall()
print(tabulate(result,headers=["ID","NAME","AGE"]))
```

The module ***tabulate*** is imported to display the data rows in tabular form.

A sample python program to perform CRUD operation on a database can explain more in detail about working with a database. The below sample program that provides a menu and allows user to choose an option to carry out one of the CRUD operation on a mysql database named ***trade***

In below program, code for each action (INSERT, READ (SELECT), UPDATE and DELETE are written as separate functions. And a Menu is provided to select an option to perform any one of the action.

```
from tabulate import tabulate
import mysql.connector
con=mysql.connector.connect(host='localhost',user='root',password='',database='trade')
if con:
   print('connected')
else:
   print('not connected')
def insert(id,name,age):   # Creating database rows by adding information to a specific
                             table of the database connected.
    res=con.cursor()
    sql="insert into smpl (id,name,age) values (%s, %s, %s)"
    print(sql)
    user=(id,name,age)
    res.execute(sql,user)
    con.commit()
    print("Inserted")
def update(id,name,age,id1):  # Updating database rows by adding information to a specific
                                table of the database connected.
    res=con.cursor()
    sql="update smpl set id=%s,name=%s,age=%s where id=%s"
    print(sql)
    user=(id,name,age,id1)
    res.execute(sql,user)
    con.commit()
    print("Updated")

def select():            # Reading database rows by adding information to a
                         specific table of the database connected.
    res=con.cursor()
    sql="SELECT * FROM SMPL"
    res.execute(sql)
    result=res.fetchall()
    print(tabulate(result,headers=["ID","NAME","AGE"]))
```

```
def delete(id):          # Deleting database rows by adding information to a
                           specific table of the database connected.
    res=con.cursor()
    sql="delete from smpl where id=%s"
    print(sql)
    user=(id,)
    res.execute(sql,user)
    con.commit()
    print("Record id is deleted")
while True:
      print("1.INSERT")
      print("2.UPDATE")
      print("3.SELECT")
      print("4.DELETE")
      print("5.QUIT")
      choice=int(input("Enter Choice: "))
      if choice==1:
         id=input("Enter ID: ")
         name=input("Enter Name: ")
         age=input("Enter Age: ")
         insert(id,name,age)
      elif choice==2:
         id=input("Enter ID: ")
         name=input("Enter Name: ")
         age=input("Enter Age: ")
         id1=input("Enter Which ID: ")
         update(id,name,age,id1)
      elif choice==3:
         print("----------------------------------------")
         select()
         print("----------------------------------------")
      elif choice==4:
         id=input("Enter ID: ")
         delete(id)
      elif choice==5:
         print("Process Ends")
         quit()
      else:
         print("Invalid Selection. Try Again")
```