MODULE-4
PYTHON CONCEPTS AND DATANWRANGLING

Data Structure

Data structures are fundamental concepts of computer science which helps is writing efficient programs in any language. Python is a high-level, interpreted, interactive and object-oriented scripting language using which we can study the fundamentals of data structure in a simpler way as compared to other programming languages.

In this chapter we are going to study a short overview of some frequently used data structures in general and how they are related to some specific python data types. There are also some data structures specific to python which is listed as another category.

Liner Data Structures

These are the data structures which store the data elements in a sequential manner.

- **Array** − It is a sequential arrangement of data elements paired with the index of the data element.
- **Linked List** − Each data element contains a link to another element along with the data present in it.
- **Stack** − It is a data structure which follows only to specific order of operation. LIFO(last in First Out) or FILO(First in Last Out).
- **Queue** − It is similar to Stack but the order of operation is only FIFO(First In First Out).
- **Matrix** − It is two dimensional data structure in which the data element is referred by a pair of indices.

Non-Liner Data Structures

These are the data structures in which there is no sequential linking of data elements. Any pair or group of data elements can be linked to each other and can be accessed without a strict sequence.

- **Binary Tree** − It is a data structure where each data element can be connected to maximum two other data elements and it starts with a root node.
- **Heap** − It is a special case of Tree data structure where the data in the parent node is either strictly greater than/ equal to the child nodes or strictly less than it's child nodes.
- **Hash Table** − It is a data structure which is made of arrays associated with each other using a hash function. It retrieves values using keys rather than index from a data element.
- **Graph** − It is an arrangement of vertices and nodes where some of the nodes are connected to each other through links.

Python Specific Data Structures

These data structures are specific to python language and they give greater flexibility in storing different types of data and faster processing in python environment.

- **List** − It is similar to array with the exception that the data elements can be of different data types. You can have both numeric and string data in a python list.
- **Tuple** − Tuples are similar to lists but they are immutable which means the values in a tuple cannot be modified they can only be read.
- **Dictionary** − The dictionary contains Key-value pairs as its data elements.

THE PYTHON INTERPRETER

The Python programming language was conceived in the late 1980s, and itsimplementation was started in December 1989 by Guido van Rossum at CWI in theNetherlands as a successor to the ABC programming language. When he began implementingPython, Guido van Rossum was also reading the published scripts from "**Monty Python's**Flying Circus‖, a BBC comedy series from the 1970s. Van Rossum thought he needed aname that was short, unique, and slightly mysterious, so he decided to call the languagePython.

Python is a cross-platform programming language, meaning, it runs onmultiple platforms like Windows, Mac OS X, Linux, Unix and has even been ported to theJava and .NET virtual machines. It is free and open source.

Starting the Interpreter

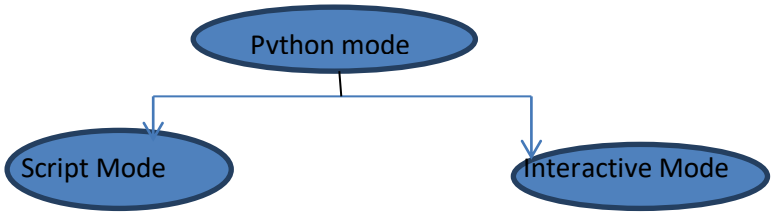
The Python interpreter is a program that reads and executes Python code.
Even though most of today's Linux and Mac have Python preinstalled in it, the version might be out-of-date. So, it is always a good idea to install the most current version.

After installation, the python interpreter lives in the installed directory.

By default it is /usr/local/bin/pythonX.X in Linux/Unix and C:\PythonXX in Windows, where the 'X' denotes the version number. To invoke it from the shell or the command prompt we need to add this location in the search path.

Search path is a list of directories (locations) where the operating systemsearches for executables. For example, in Windows command prompt, we can type setpath=%path%;c:\python27 (python27 means version 2.7, it might be different in your case) toadd the location to path for that particular session. In Mac OS we need not worry about thisas the installer takes care about the search path. Now there are two ways to start Python.

Figure shows Modes of Python Interpreter



Interactive Mode

Typing python in the command line will invoke the interpreter in interactive mode.

When it starts, you should see output like this:

```
PYTHON 2.7.13 (V2.7.13:A06454B1AFA1, DEC 17 2016, 20:42:59) [MSC V.1500 32 BIT (INTEL)] ON WIN32
TYPE "COPYRIGHT", "CREDITS" OR "LICENSE()" FOR MORE INFORMATION.
>>>
```

The first three lines contain information about the interpreter and the operating system it's running on, so it might be different for you. But you should check that the version number, which is 2.7.13 in this example, begins with 2, which indicates that you are running

Python 2. If it begins with 3, you are running Python 3.
The last line is a prompt that indicates that the interpreter is ready for you to enter code.
If you type a line of code and hit Enter, the interpreter displays the result:

```
>>> 5 + 4
9
```

This prompt can be used as a calculator. To exit this mode type exit() or quit() and press enter.

Script Mode

This mode is used to execute Python program written in a file. Such a file is calleda script. Scripts can be saved to disk for future use. Python scripts have the extension .py,meaning that the filename ends with .py.

For example: helloWorld.py

To execute this file in script mode we simply write python helloWorld.py at thecommand prompt.

Integrated Development Environment (IDE)

We can use any text editing software to write a Python script file.
We just need to save it with the .py extension. But using an IDE can make our life a lot easier. IDE is a piece of software that provides useful features like code hinting, syntax highlighting and checking, file explorers etc. to the programmer for application development.

Using an IDE can get rid of redundant tasks and significantly decrease the time required for application development.

IDEL is a graphical user interface (GUI) that can be installed along with the Python programming language and is available from the official website.

We can also use other commercial or free IDE according to our preference. PyScripter IDE is one of the Open source IDE.

Hello World Example

Now that we have Python up and running, we can continue on to write our first Python program.

Type the following code in any text editor or an IDE and save it as ***helloWorld.py***

```
print("Hello world!")
```

Now at the command window, go to the loaction of this file. You can use the **cd** command to change directory.
To run the script, type, ***python helloWorld.py*** in the command window. We should be able to see the output as follows:

```
Hello world!
```

If you are using *PyScripter,* there is a green arrow button on top. Press that button orpress Ctrl+F9 on your keyboard to run the program.
In this program we have used the built-in function ***print(),*** to print out a string to thescreen. String is the value inside the quotation marks, i.e. Hello world!.

VALUES AND TYPES

A value is one of the basic things a program works with, like a letter or a number. Some example values are 5, 83.0, and 'Hello, World!'. These values belong to different types: 5 is an integer, 83.0 is a floating-point number, and 'Hello, World!' is a string, so-called because the letters it contains are strung together. If you are not sure what type a value has, the interpreter can tell you:

```
>>>type(5)
<class 'int'>
>>>type(83.0)
<class 'float'>
>>>type('Hello, World!')
<class 'str'>
```

In these results, the word ―class‖ is used in the sense of a category; a type is a category of values. Not surprisingly, integers belong to the type int, strings belong to str and floating-point numbers belong to float. What about values like '5' and '83.0'? They look like numbers, but they are in quotation marks like strings.

```
>>>type('5')
<class 'str'>
>>>type('83.0')
<class 'str'>
```

They’re strings. When you type a large integer, you might be tempted to use commas between groups of digits, as in 1,000,000. This is not a legal integer in Python, but it is legal:

```
>>> 1,000,000
 (1, 0, 0)
```

That’s not what we expected at all! Python interprets 1,000,000 as a comma-separated sequence of integers. We’ll learn more about this kind of sequence late.

Standard Data Types

Python has various standard data types that are used to define the operations possible on them and the storage method for each of them.

Python has five standard data types –

- Numbers
- String
- List
- Tuple
- Dictionary

Python Number

Number data types store numeric value Number objects are created when you assign a value to then for example-

```
Var1=0
Var2=10
```

You can also delete the reference to a number object by using the del statement.

The syntax of the del statement is-

```
Del var1[var2[var[var3[….,varN]]]]
```

You can delete a single object or multiple objects by using the del statement .for example-

delvar
delvar_a,var_b
python support four different numerical types-

- int(signed integers)
- long(long integers they can also be represented in octal and hexadecimal)
- float(floating point real values)
- complex(complex numbers)

Examples
Here are some examples of numbers-

Int	Long	Float	complex
10	51924361L	0.0	3.14J
100	-0x19323L	15.20	45J
-786	0122L	-21.9	9.322e-36J
080	OxDEFABCECBDAECBFBAEI	32.+e18	.876J
-0490	535633629843l	-90	-6545+0J
-0x260	-052318172735L	-32,54e100	3e+26J
Ox69	-4721885298529l	70.2.E12	4,53e-7J

Python allows you to use a lowercase l with long, but it is recommended that you use only an uppercase L to avoid confusion with the number 1. Python displays long integers with an uppercase L. A complex number consists of an ordered pair of real floating-point numbers denoted by x + yj, where x and y are the real numbers and j is the imaginary unit.

Python Strings Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes. Subsets of strings can be taken using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end.

The plus (+) sign is the string concatenation operator and the asterisk (*) is the repetition operator. For example –

```
str = 'Python Programming'
printstr              # Prints complete string
printstr[0]       # Prints first character of the string
printstr[-1]      # Prints last character of the string
printstr[2:5]     # Prints characters starting from 3rd to 5th
printstr[2:]      # Prints string starting from 3rd character
printstr * 2      # Prints string two times
printstr + " Course"     # Prints concatenated string
```

This will produce the following result –

```
Python Programming
P
G
tho
thon Programming
Python ProgrammingPython Programming
Python Programming Course
```

Python Lists Lists are the most versatile of Python's compound data types. A list contains items separated by commas and enclosed within square brackets ([]). To some extent, lists are similar to arrays in C. One difference between them is that all the items belonging to a list can be of different data type.

The values stored in a list can be accessed using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1.

The plus (+) sign is the list concatenation operator, and the asterisk (*) is the repetition operator.

For example −

```
list = [ 'Hai', 123 , 1.75, 'vinu', 100.25 ]
smalllist = [251, 'vinu']
print list              # Prints complete list
print list[0]          # Prints first element of the list
print list[-1]         # Prints last element of the list
print list[1:3]        # Prints elements starting from 2nd till 3rd
print list[2:]         # Prints elements starting from 3rd element
printsmalllist * 2     # Prints list two times
```

This produces the following result –

```
['Hai', 123, 1.75, 'vinu',100.25]
Hai
100.25
 [123, 1.75]
 [1.75, 'vinu', 100.25]
 [251, 'vinu', 251, 'vinu']
 ['Hai', 123, 1.75, 'vinu', 100.25, 251, 'vinu']
```

Python Bolean

A Boolean type was added to Python 2.3. Two new constants were added to the __builtin__ module, **True and False.** True and False are simply set to integer values of 1 and 0 and aren't a different type. The type object for this new type is named bool; the constructor for it takes any Python value and converts it to True or False.

```
>>>bool(1)
 True
>>>bool(0) False
```

Python's Booleans were added with the primary goal of making code clearer. For example, if you're reading a function and encounter the statement return 1, you might wonder whether the 1 represents a Boolean truth value, an index, or a coefficient that multiplies some other quantity. If the statement is return True, however, the meaning of the return value is quite clear.

Python's Booleans were not added for the sake of strict type-checking. A very strict language such as Pascal would also prevent you performing arithmetic with Booleans, and would require that the expression in an if statement always evaluate to a Boolean result. Python is not this strict and never will be. This means you can still use any expression in an if statement, even ones that evaluate to a list or tuple or some random object. The Boolean type is a subclass of the int class so that arithmetic using a Boolean still works.

```
>>>True+1
2
>>>False+1
1
>>>False*85
87
>>>True+True
2
>>>False+False
```

Data type conversion

Sometime, you may need to perform conversions between the built –in types.

To convert between types, you simply use the type name as a function.

There are several built-in functions to perform conversion from one data type to another .These functions return a new object representing the converted value.

Table: Data type conversion

Function	description
int(x,[,base])	Converts x to an integer base specifies the base if x is a string
long(x[,base])	Converts x to a long integer base specifies the if x is a string
float(x)	Converts x to a floating point number
complex(real[,image])	Creates a complex number
str(x)	Converts object x to a string representation
repr(x)	Converts object x to an expression string
eval(x)	Evaluates a string and returns an object
list(s)	Converts to a list
char(x)	Converts an integer to a character
unichar(x)	Converts an integer to a Unicode character
ord(x)	Converts a single character to its integer value
hex(x)	Converts an integer to a hexadecimal string
oct(x)	Converts an integer to an octal string

Variables

A variable is a name that refers to a value. Variable reserved memory location to store value. This means that when you create a variable you reserve some space in memory. Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory.

Assignment statements

An assignment statement creates a new variable and gives it a value:

>>>message= ‘introducing python variable’

>>>num=15

>>>radius=5.4

This example makes three assignments.The first assigns a string to a new variable named message the second gives the integer 15 to num the third assigns floating point value 5.4 to variable radius.

Variable names

Programmers generally choose names for their variables that are meaningful

The rules

Variable names must start with a letter or an underscore such as

- _mark
- mark_

The remainder of your variable name may consist of letters,numbers and underscores

- subject1
- my2ndsubject
- un_der_scores

Names are case sensitive.

- case_sensitive, CASE_SENSITIVE. and Case_Sensitive are each a different variable

can be any(reasonable)length

There are some reserved(key words)words which you cannot use as a variable name because python uses for other things
The interpreter uses keywords to recognize the structure of the program and they cannot be used as variable names.
Python 3 has these keywords:
False, class, finally, is, return, one, pass, break, except, with, as ,del, non, local, if,
Continue, for, lambda, try, true, def, from, not, while etc.,
You don't have to memorize this list.In most development enivironments,keywords are displayed in a different colour;if you try to use one as a variable name,you 'll know.
If you give a variable an illegal name,you get a syntax error:
>>Ibook='python'
syntaxError:invalid syntax
>>>more@=1000000
syntaxError:invalidsyntax

ibook is illegal because it begins with a number.more@is illegal because it contains an illegal character, @class is illegal because it is keyword

Good Variable Name

- Choose meaningful name instead of short name. roll_no is better than rn.
- Maintain the length of a variable name. Roll_no_of_a_student is too long?
- Be consistent; roll_no or orRollNo
- Begin a variable name with an underscore(_) character for a special case.

EXPRESSIONS AND STATEMENTS

An **expression** is a combination of values, variables, and operators. A value all by itself is considered an expression, and so is a variable, so the following are all legal expressions:

```
>>> 50
50
>>>10<5
False
>>>50+20
70
```

When you type an expression at the prompt, the interpreter evaluates it , which means that it finds the value of the expression
A statement is a unit of code that has an effect, like creating a variable or displaying a value.

```
>>>n=25
>>>print(n)
```

The first line is an assignment statement that gives a value to n .The second line is a print statement that displays the value of n. when you type a statement, the interpreter executes it, which means that it does whatever the statement says. In general, statements don't have values.

EXPRESSIONS AND STATEMENTS

An expression is a combination of values, variables, and operators. A value all by itself is considered an expression, and so is a variable, so the following are all legal expressions: >>> 50 50

```
>>> 10<5
False
>>> 50+20
70
```

When you type an expression at the prompt, the interpreter evaluates it, which means that it finds the value of the expression.

A statement is a unit of code that has an effect, like creating a variable or displaying a value.

```
>>> n = 25
>>>print(n)
```

The first line is an assignment statement that gives a value to n. The second line is a print statement that displays the value of n. When you type a statement, the interpreter executes it, which means that it does whatever the statement says. In general, statements don't have values.

Difference Between a Statement and an Expression

A statement is a complete line of code that performs some action, while an expression is any section of the code that evaluates to a value. Expressions can be combined ―horizontally‖ into larger expressions using operators, while statements can only be combined ―vertically‖ by writing one after another, or with block constructs. Every expression can be used as a statement, but most statements cannot be used as expressions.

OPERATORS

In this section, you'll learn everything about different types of operators in Python, their syntax and how to use them with examples.

Operators are special symbols in Python that carry out computation. The value that the operator operates on is called the operand.

For example:

```
>>>10+5
```

15 Here, + is the operator that performs addition. 10 and 5 are the operands and 15 is the output of the operation.

Python has a number of operators which are classified below.

- Arithmetic operators
- Comparison (Relational) operators
- Logical (Boolean) operators
- Bitwise operators
- Assignment operators
- Special operators

Arithmetic Operators

Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication etc.

Table : Arithmetic operators in Python

operator	Meaning	example
+	Add two operands or unary plus	X + y+2
-	Sub right operand from the left or unary minus	x-y-2
*	Multiply two operands	X*y
/	Divide left operand by the right one (always results into float)	x/y
%	Modulus – remainder of the division of left operand by the right	X% y(remainder of x/y)
/	Floor division-	X//y

/	division that results into whole number adjusted to the left in the number line	
**	Exponent-left operand raised to the power of right	X**y(x to the power y)

Example:

- Print('x –y=',x – y)
- X=7
- Y=3
- Print('x +y= ',x + y)
- Print('x *y=',x*y)
- Print('x/y=',x/y)
- Print('x//y=',x//Y)
- Print('x% y=',x%y)
- Print('x**y=',x**y)

When you run the program the out put will be

X+y=10

X-y=4

X*y=21

x/y=2.33333333335

X//y=2

X%y=1

X**y=343

Operator	Meaning	example
>	Greater that-true if left operand is greater than the right	X>y
<	Less that –true if left operand is less than the right	X<y
==	Equal to –true if both operands are equal	X==y
!=	Not equal to –true if operands are not equal	X!=y
>=	Greater than or equal to- true if left operand is greater than or equal	X>=y
<=	Less than or equal to –true if left operand is less than or equal to the right	X<=y

Comparison (Relational) operators

Comparison operators are used to compare values its returns as true or false

Example

- X=5
- Y=7
- Print(‘x>y is ‘,x>y)
- Print(‘x<y is ‘,x<y)
- Print(‘x==y is ‘,x==)
- Print(‘x ! =y is ’, x!=y)
- Print(‘x>=y is ‘, x>=y)
- Print(‘x<=y is ‘,x<=y)

When you run the program the output will be

- X>y is false
- X<y is true
- X ==y is false
- X ! =y is false
- X >= y is false
- X<=y is true

Logical operators

Logical operators are the and, or not operators

Operator	meaning	example
And	True if both operands are true	x and y
Or	True if either of the operands is true	x or y
Not	True if operand is false	Not y

Example

- X=True
- y = False
- print('x and y is',x and y)
- print('x or y is',x or y)
- print('not x is',not x)

When you run the program, the output will be:

- x and y is False
- x or y is True
- not x is False

BITWISE OPERATORS

- Bitwise operators act on operands as if they were string of binary digits. it operates bit by bit, hence the name
- For example 2is 10 in binary and 7 is 111
- Let x=10 (0000 1010 in binary)and y= 4 (0000 0100 in binary)

Operator	Meaning	example
&	Bitwise AND	X&y =0(0000 0000)
|	Bitwise OR	X} Y=14(0000 1110)
~	Bitwise NOT	~x=-11(1111 0101)
^	Bitwise XOR	X^ Y=14(0000 1110)
>>	Bitwise right shift	X>> 2=2(0000 0010)
<<	Bitwise left shift	X<< 2=40 (0010 1000)

Example

- x=10
- y=4
- print('x& y=',x& y)
- print('x | y=',x | y)
- print('~x=',~x)
- print('x ^ y=',x ^ y)
- print('x>> 2=',x>> 2)
- print('x<< 2=',x<< 2)

When you run the program, the output will be:

- x& y= 0
- x | y= 14
- ~x= -11
- x ^ y= 14

PRECEDENCE OF PYTHON OPERATORS

The combination of values, variables, operators and function calls is termed as an expression. Python interpreter can evaluate a valid expression. When an expression contains more than one operator, the order of evaluation dependson the **Precedence** of operations.

For example, multiplication has higher precedence than subtraction.

```
>>> 20 – 5*3
5
```

But we can change this order using parentheses () as it has higher precedence.

```
>>> (20 - 5) *3
45
```

The operator precedence in Python are listed in the following table. It is in descending order, upper group has higher precedence than the lower ones.

Operator precedence rule in Python

Operators	meaning
()	parentheses
**	exponent
+x,-x,¬x	Unary plus, unary minus ,bitwise not
*,/,//,%	Multiplication ,division ,floor division modulus
+,-	Add,sub
<<,>>	Bitwise shift operators

&	Bitwise AND
^	Bitwise XOR
Pipe	Bitwise OR
==,!=,>,>=,<,<=,is ,isnot,in,not in	Comparisons, identity, membership operators
Not	Logical NOT
And	Logical AND
Or	Logical OR

ASSOCIATIVITY OF PYTHON OPERATORS

We can see in the above table that more than one operator exists in the same group. These operators have the same precedence.

When two operators have the same precedence, associativity helps to determine which the order of operations.

Associativity is the order in which an expression is evaluated that has multiple operator of the same precedence. Almost all the operators have left-to-right associativity.

18

For example, multiplication and floor division have the same precedence.

Hence, if both of them are present in an expression, left one is evaluates first.

```
>>> 10 * 7 // 3
23
>>> 10 * (7//3)
20
>>> (10 * 7)//3
23
```

We can see that 10 * 7 // 3is equivalent to (10 * 7)//3.

Exponent operator ** has right-to-left associativity in Python.

```
>>> 5 ** 2 ** 3
390625
>>> (5** 2) **3
15625
>>> 5 **(2 **3)
390625
```

We can see that 2 ** 3 ** 2 is equivalent to 2 ** (3 ** 2).

Non Associative Operators

Some operators like assignment operators and comparison operators do not have associativity in Python. There are separate rules for sequences of this kind of operator and cannot be expressed as associativity.

For example, $x < y < z$ neither means $(x < y) < z$ nor $x < (y < z)$. $x < y < z$ is equivalent to $x < y$ and $y < z$, and is evaluates from left-to-right.

Furthermore, while chaining of assignments like $x = y = z$ is perfectly valid, $x = y += z$ will result into error.

TUPLE ASSIGNMENT

It is often useful to swap the values of two variables. With conventional assignments, you have to use a temporary variable. For example, to swap a and b:

```
>>>temp = a
>>> a = b
>>> b = temp
```

This solution is cumbersome; **tuple assignment** is more elegant:

```
>>>a, b = b, a
```

The left side is a tuple of variables; the right side is a tuple of expressions.

Each value is assigned to its respective variable. All the expressions on the right side are evaluated before any of the assignments. The number of variables on the left and the number of values on the right have to be the same:

```
>>>a, b = 1, 2, 3
ValueError: too many values to unpack
```

More generally, the right side can be any kind of sequence (string, list or tuple). For example, to split an email address into a user name and a domain, you could write:

```
>>>addr = 'monty@python.org'
>>>uname, domain = addr.split('@')
```

The return value from split is a list with two elements; the first element is assigned to uname, the second to domain.

```
>>>uname 'monty'
>>>domain
'python.org'
```

COMMENTS

As programs get bigger and more complicated, they get more difficult to read. Formal languages are dense, and it is often difficult to look at a piece of code and figure out what it is doing, or why. For this reason, it is a good idea to add notes to your programs to explain in natural language what the program is doing. These notes are called comments, and they start with the # symbol:

```
# compute Area of a triangle using Base and Height area= (base*height)/2
```

In this case, the comment appears on a line by itself. You can also put comments at the end of a line: area= (base*height)/2 # Area of a triangle using Base and Height

Everything from the # to the end of the line is ignored—it has no effect on the execution of the program. Comments are most useful when they document non-obvious features of the code. It is reasonable to assume that the reader can figure out what the code does; it is more useful to explain why.

This comment is redundant with the code and useless: base= 5 # assign 5 to base

This comment contains useful information that is not in the code: base = 5 # base is in centimetre.

Good variable names can reduce the need for comments, but long names can make complex expressions hard to read, so there is a trade off.

If we have comments that extend multiple lines, one way of doing it is to use hash (#) in the beginning of each line. For example:

```
#This is a long comment
#and it extends
#to multiple lines
```

Another way of doing this is to use triple quotes, either ''' or """.

These triple quotes are generally used for multi-line strings. But they can be used as multi- line comment as well. Unless they are not docstrings, they do not generate any extra code.

```
"""This is also a
perfect example of
multi-line comments"""
```

Docstring in Python

Docstring is short for documentation string. It is a string that occurs as the first statement in a module, function, class, or method definition. We must write what a function/class does in the docstring. Triple quotes are used while writing docstrings. For example:

```
def area(r):
"""Compute the area of Circle"""
return 3.14159*r**2
```

Docstring is available to us as the attribute __doc__ of the function. Issue the following code in shell once you run the above program.

```
>>>print(area.__doc__)
Compute the area of Circle
```

MODULES AND FUNCTIONS

Functions

In the context of programming, a function is a named sequence of statements that performs a computation. When you define a function, you specify the name and the sequence of statements. Later, you can ―call‖ the function by name. Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable. Furthermore, it avoids repetition and makes code reusable.

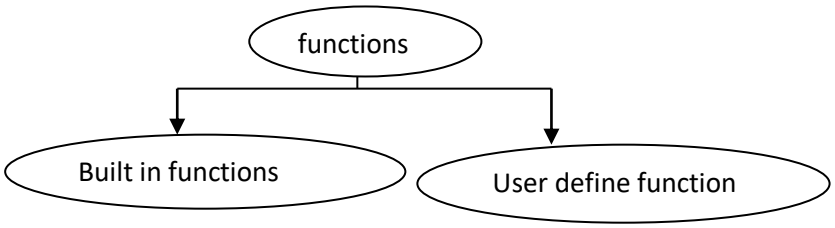


Figure shows Types of Functions

Functions can be classified into

- Built-in Functions
- User Defined Functions

Built-in Functions

The Python interpreter has a number of functions that are always available for use. These functions are called built-in functions. We have already seen one example of a function call:type()

```
>>>type(25)
<class ‘int’>
```

The name of the function is type. The expression in parentheses is called the argument of the function. The result, for this function, is the type of the argument. It is common to say that a function ―takes‖ an argument and ―returns‖ a result. The resultis also called the return value.

Python provides functions that convert values from one type to another. The int function takes any value and converts it to an integer, if it can, or complains otherwise: >>>int('25')

```
25
>>>int('Python')
ValueError: invalid literal for int(): Python
```

int can convert floating-point values to integers, but it doesn’t round off; it chops off the fraction part

```
>>>int(9.999999)
9
>>>int(-2.3)
-2
```

float converts integers and strings to floating-point numbers:

```
>>>float(25)
25.0
>>>float('3.14159')
3.14159
```

Finally, str converts its argument to a string:

```
>>>str(25)
'25'
>>>str(3.14159)
```

'3.14159'

range() – function

The range() constructor returns an immutable sequence object of integers between the given start integer to the stop integer.

Python's range() Parameters

The range() function has two sets of parameters, as follows: range(stop)

- stop: Number of integers (whole numbers) to generate, starting from zero.
  eg. range(3) == [0, 1, 2].

range([start], stop[, step])

- start: Starting number of the sequence.
- stop: Generate numbers up to, but not including this number.
- step: Difference between each number in the sequence
- All parameters must be integers.
- All parameters can be positive or negative.

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

Printing to the Screen

In python 3, print function will prints as strings everything in a comma- separated sequence of expressions, and it will separate the results with single blanks by default. Note that you can mix types: anything that is not already a string is automatically converted to its string representation.

Eg.

```
>>> x=10
>>> y=7
>>>print('The sum of', x, 'plus', y, 'is', x+y)
```

The sum of 10 plus 7 is 17 You can also use it with no parameters:

print()

to just advance to the next line.

In **python 2,** simplest way to produce output is using the print statement where you can pass zero or more expressions separated by commas. This function converts the expressions you pass into a string and writes the result to standard output as follows

```
>>> x=10
>>> y=7
>>>print'The sum of', x, 'plus', y, 'is', x+y
```

The sum of 10 plus 7 is 17

Reading Keyboard Input

Python provides two built-in functions to read a line of text from standard input, which by default comes from the keyboard. These functions are −

- raw_input
- input

The raw_input Function The raw_input([prompt]) function reads one line from standard input and returns it as a string.

```
>>>str = raw_input("Enter your input: ");
Enter your input: range(0,10)
```

```
>>> print "Received input is : ", str
Received input is :  range(0,10)
```

This prompts you to enter any string and it would display same string on the screen.

The input Function

The input([prompt]) function is equivalent to raw_input, except that it assumes the input is a valid Python expression and returns the evaluated result to you.

```
>>>str = input("Enter your input: ");
Enter your input: range(0,10)
>>> print "Received input is : ", str
Received input is :  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Input data is evaluated and the list is generated

User-defined functions

As you already know, Python gives you many built-in functions like print(), input(), type() etc. but you can also create your own functions. These functions are called user-defined functions.

Function Definition and Use

Syntax of Function definition

```
def function_name(parameters):
"""docstring"""
statement(s)
```

Above shown is a function definition which consists of following components.

1. Keyword def marks the start of function header.
2. A function name to uniquely identify it. Function naming follows the same rules of writing identifiers in Python.
3. Parameters (arguments) through which we pass values to a function. They are optional.
4. A colon (:) to mark the end of function header.
5. Optional documentation string (docstring) to describe what the function does.
6. One or more valid python statements that make up the function body. Statements must have same indentation level (usually 4 spaces).
7. An optional return statement to return a value from the function.

Example of a function

```
def welcome(person_name):
"""This function welcome
the person passed in as
parameter"""
print(" Welcome " ,  person_name , " to Python Function Section")
```

Using Function or Function Call

Once we have defined a function, we can call it from another function, program or even the Python prompt. To call a function we simply type the function name with appropriate parameters.

```
>>> welcome('CSBS')
```

Welcome CSBS to Python Function Section

The return statement

The return statement is used to exit a function and go back to the place from where it was called.

Syntax of return

```
return [expression_list]
```

This statement can contain expression which gets evaluated and the value is returned. If there is no expression in the statement or the return statement itself is not present inside a function, then the function will return the None object.

```
def absolute_value(num):
"""This function returns the absolute  value of the entered number"""
if num >= 0:
return num
else:
return -num
print(absolute_value(5))
print(absolute_value(-7))
```

When you run the program, the output will be:

5

7

Flow of Execution

To ensure that a function is defined before its first use, you have to know the order statements run in, which is called the flow of execution. Execution always begins at the first statement of the program. Statements are run one at a time, in order from top to bottom. Function definitions do not alter the flow of execution of the program, but remember that

statements inside the function don't run until the function is called. A function call is like a detour in the flow of execution. Instead of going to the next statement, the flow jumps to the body of the function, runs the statements there, and then comes back to pick up where it left off. Figure shows the flow of execution

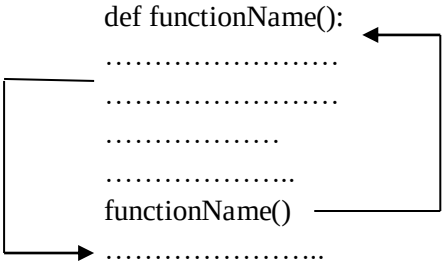


Figure shows Flow of execution when function Call

That sounds simple enough, until you remember that one function can call another. While in the middle of one function, the program might have to run the statements in another function. Then, while running that new function, the program might have to run yet another function! Fortunately, Python is good at keeping track of where it is, so each time a function completes, the program picks up where it left off in the function that called it. When it gets to the end of the program, it terminates.

In summary, when you read a program, you don't always want to read from top to bottom. Sometimes it makes more sense if you follow the flow of execution.

Parameters and Arguments

Some of the functions we have seen require arguments. For example, when you call type() you pass a variable or value as an argument. Some functions take more than one argument: eg, range() function take one or two or three arguments.

Inside the function, the arguments are assigned to variables called parameters. Here is a definition for a function that takes an argument:

def welcome(person_name): print(" Welcome " , person_name , " to Python Function Section") This function assigns the argument to a parameter named person_name. When the function is called, it prints the value of the parameter (whatever it is). This function works with any value that can be printed.

```
>>> welcome("csbs")
Welcome  csbs  to Python Function Section
>>> welcome(100)
Welcome  100  to Python Function Section
>>> welcome(50.23)
Welcome  50.23  to Python Function Section
```

The argument is evaluated before the function is called, so in the below examples the expressions 'csbs'*3 is evaluated.

```
>>> welcome('csbs'*3) Welcome  csbscsbscsbs  to Python Function Section
```

You can also use a variable as an argument:

```
>>> student_name='Ranjith'
>>> welcome(student_name)
Welcome  Ranjith  to Python Function Section
```

Function Arguments

You can call a function by using the following types of formal arguments:

- Required arguments
- Default arguments
- Keyword arguments
- Variable-length arguments

Required Arguments

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition

```
def add(a,b):
return a+b
a=10
b=20
print "Sum of ", a ,"and ", b, "is" , add(a,b)
```

To call the function add(), you definitely need to pass two argument, otherwise it gives a syntax error as follows

When the above code is executed, it produces the following result: Sum of 10 and 20 is 30

If we miss to give an argument it will show syntax error.

Example

```
def add(a,b):
return a+b
a=10
b=20
print "Sum of ", a ,"and ", b, "is" , add(a)
```

It will produce Error message as follows

```
Sum of  10 and  20 is
Traceback (most recent call last):
File "G:/class/python/code/required_arguments.py", line 5, in <module>
```

```
print "Sum of ", a ,"and ", b, "is" , add(a)
TypeError: add() takes exactly 2 arguments (1 given)
```

Default Arguments

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The following example gives an idea on default arguments, it add default value of b if it is not passed while calling.

```
def add(a,b=0):
print "Sum of ", a ,"and ", b, "is" ,a+b
a=10
b=20
add(a,b)
add(a)
```

When the above code is executed, it produces the following result:

```
Sum of  10 and  20 is 30
Sum of  10 and  0 is 10
```

When default argument is used in program, Non default arguments should come before default arguments.

Keyword Arguments

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

This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters. You can also make keyword calls to the add() function in the following ways

```
def add(a,b):
print "Sum of ", a ,"and ", b, "is" ,a+b
a=10
b=20
add(b=a,a=b)
```

When the above code is executed, it produces the following result − Sum of 20 and 10 is 30

Variable-Length Arguments

You may need to process a function for more arguments than you specified while defining the function. These arguments are called variable-length arguments and are not named in the function definition, unlike required and default arguments.

Syntax for a function with non-keyword variable arguments is this

```
def functionname([formal_args,] *var_args_tuple ):
"function_docstring"
function_suite
return [expression]
```

An asterisk (*) is placed before the variable name that holds the values of all nonkeyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call.

Following is a simple example

```
def printvalues( arg1, *vartuple ):
print "Output is: "
print arg1
for var in vartuple:
print var
```

printvalues(20)
printvalues(50, 60, 55)
When the above code is executed, it produces the following result:-
Output is:
20
Output is:
50
60
55

The Anonymous Functions or Lambda Functions

These functions are called anonymous because they are not declared in the standard manner by using the def keyword. You can use the lambda keyword to create small anonymous functions.

- Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
- An anonymous function cannot be a direct call to print because lambda requires an expression
- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.
- Although it appears that lambda's are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is by passing function stack allocation during invocation for performance reasons.

Syntax

The syntax of lambda functions contains only a single statement, which is as follows

lambda [arg1 [,arg2,.....argn]]:expression

Following is the example to show how lambda form of function works

```
sum = lambda a, b: a + b;
 print "Sum is : ", sum( 5, 10 )
 print "Sum is : ", sum( 30, 50 )
```

When the above code is executed, it produces the following result

Sum is: 15
Sum is: 80

Modules

A module allows you to logically organize your Python code. Grouping related code into a module makes the code easier to understand and use. A module is a file that contains a collection of related functions. Python has lot of built-in modules; math module is one of them. math module provides most of the familiar mathematical functions.

Before we can use the functions in a module, we have to import it with an import statement:

>>> import math

This statement creates a module object named math. If you display the module object, you get some information about it:

>>> math <module 'math' (built-in)>

The module object contains the functions and variables defined in the module. To access one of the functions, you have to specify the name of the module and the name of the function, separated by a dot (also known as a period). This format is called dot notation.

```
>>> math.log10(200)
2.3010299956639813
>>> math.sqrt(10)
```

3.1622776601683795

Math module have functions like log(), sqrt(), etc… In order to know what are the functions available in particular module, we can use dir() function after importing particular module. Similarly if we want to know detail description about a particular module or function or variable means we can use help() function.

Eg. >>> import math

>>> dir(math)

['__doc__', '__name__', '__package__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'hypot', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc'] >>> help(pow)

Help on built-in function pow in module __builtin__:

pow(...)

pow(x, y[, z]) -> number

With two arguments, equivalent to x**y. With three arguments, equivalent to (x**y) % z, but may be more efficient (e.g. for longs).

Writing modules

Any file that contains Python code can be imported as a module. For example, suppose you have a file named addModule.py with the following code:

```
def add(a, b):
result = a + b
print(result)
add(10,20)
```

If you run this program, it will add 10 and 20 and print 30. We can import it like this:

```
>>> import addModule
30
```

Now you have a module object addModule

```
>>> addModule
<module 'addModule' from 'G:/class/python/code\addModule.py'>
```

The module object provides add():

```
>>> addModule.add(120,150)
270
```

So that's how you write modules in Python.

The only problem with this example is that when you import the module it runs the test code at the bottom. Normally when you import a module, it defines new functions but it doesn't run them.

Programs that will be imported as modules often use the following idiom: if __name__ == '__main__':

```
add(10,20)
```

__name__ is a built-in variable that is set when the program starts. If the program is running as a script, __name__ has the value '__main__'; in that case, the test code runs. Otherwise, if the module is being imported, the test code is skipped. Modify addModule.py file as given below.

```
def add(a, b):
  result = a + b
print(result)
if __name__ == '__main__':
add(10,20)
```

Now while importing addModule test case is not running >>> import addModule

__name__ has module name as its value when it is imported. Warning: If you import a module that has already been imported, Python does nothing. It does not re-read the file, even if it has changed. If you want to reload a module, you can use the built-in function reload, but it can be tricky, so the safest thing to do is restart the interpreter and then import the module again

CONDITIONALS

Flow of execution of instruction can be controlled using conditional statements. Conditional statements have some Boolean expression. Boolean expression can have relational operators or logical operators or both.

Boolean Expressions

A boolean expression is an expression it's result is either true or false. The following examples use the operator ==, which compares two operands and produces True if they are equal and False otherwise:

```
>>> 9 == 9
True
>>> 9 == 6
False
```

True and False are special values that belong to the type bool; they are not strings:

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

Boolean expression can have relational operators or logical operators. The == operator is one of the relational operators; the others are:

```
x==y  #x is equal to y
x != y   # x is not equal to y
x > y   # x is greater than y
x < y   # x is less than y
x >= y   # x is greater than or equal to y
x <= y   # x is less than or equal to y
```

More explanation can found in already. Although these operations are probably familiar to you, the Python symbols are different from the mathematical symbols. A common error is to use a single equal sign (=) instead of a double equal sign (==). Remember that = is an assignment operator and == is a relational operator. There is no such thing as =< or =>.

Logical operators

There are three logical operators: and, or, and not. The semantics (meaning) of these operators is similar to their meaning in English.

For example

x> 0 and x < 10 is true only if x is greater than 0 and less than 10.

n%2 == 0 or n%3 == 0 is true if either or both of the conditions is true, that is, if the number is divisible by 2 or 3.

not operator negates a boolean expression, so not (x > y) is true if x > y is false, that is, if x is less than or equal to y.

Note:- Any nonzero number is interpreted as True

>>>95 and True True >>>mark=95 >>>mark>0 and mark<=100 True >>> mark=102 >>> mark>0 and mark<=100 False

SELECTION

You were introduced to the concept of flow of control: the sequence of statements that the computer executes. In procedurally written code, the computer usually executes instructions in the order that they appear. However, this is not always the case. One of the ways in which programmers can change the flow of control is the use of selection control statements.

Now we will learn about selection statements, which allow a program to choose when to execute certain instructions. For example, a program might choose how to proceed on the basis of the user's input. As you will be able to see, such statements make a program more versatile.

In python selection can be achieved through

- if statement
- The elif Statement
- if...elif...else
- Nested if statements

Conditional Execution

In order to write useful programs, we almost always need the ability to check conditions and change the behaviour of the program accordingly. Selection or Conditional statements give us this ability. The simplest form is the if statement:

General Syntax of if statement is

```
if TEST EXPRESSION:
    STATEMENT(S)                    # executed if condition evaluates to True
```

Here, the program evaluates the TEST EXPRESSION and will execute statement(s) only if the text expression is True. If the text expression is False, the statement(s) is not executed.

A few important things to note about if statements:

1The colon (:) is significant and required. It separates the header of the compound statement from the body.

2. The line after the colon must be indented. It is standard in Python to use four spaces for indenting.

3. All lines indented the same amount after the colon will be executed whenever the BOOLEAN_EXPRESSION is true.

4. Python interprets non-zero values as True. None and 0 are interpreted as False.

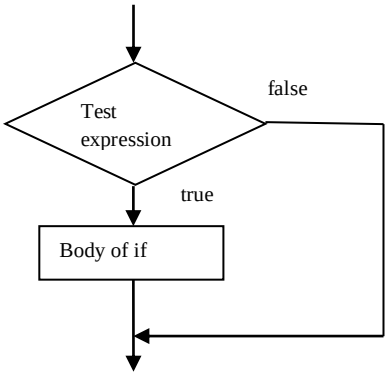


Figure shows if Statement Flowchart

Here is an example:

```
mark = 102
if mark >= 100:
 print(mark, " is a Not a valid mark.")
 print("This is always printed.")
```

Output will be:

102 is a Not a valid mark.

This is always printed.

The boolean expression after the if statement (here mark>=100)is called the condition. If it is true, then all the indented statements get executed.

There is no limit on the number of statements that can appear in the body, but there has to be at least one. Occasionally, it is useful to have a body with no statements (usually as a place keeper for code you haven't written yet). In that case, you can use the pass statement, which does nothing.

```
if x < 0:
        pass    # TODO: need to handle negative values!
```

Alternative execution

A second form of the if statement is "alternative execution", in which there are two possibilities and the condition determines which one runs. In other words , It is frequently the case that you want one thing to happen when a condition it true, and something else to happen when it is false. For that we have if else statement. The syntax looks like this:

General Syntax of if .. else statement is

```
if TEST EXPRESSION:
    STATEMENTS_1        # executed if condition evaluates to True
else:
    STATEMENTS_2        # executed if condition evaluates to False
```

Each statement inside the if block of an if else statement is executed in order if the test expression evaluates to True. The entire block of statements is skipped if the test expression evaluates to False, and instead all the statements under the else clause are executed.

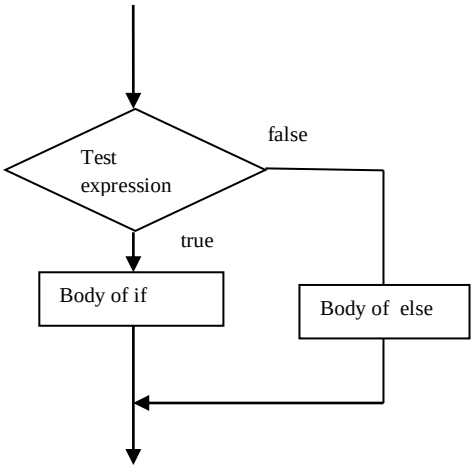


Figure show if..else Flowchart

There is no limit on the number of statements that can appear under the two clauses of an if else statement, but there has to be at least one statement in each block. Occasionally, it is useful to have a section with no statements, for code you haven't written yet. In that case, you can use the pass statement, which does nothing except act as a placeholder.

```
if True:                    # This is always true
    pass                    # so this is always executed, but it does nothing
else:
    pass
```

Here is an example:

```
age=21                      #age=17
```

```
if age >= 18:
        print("Eligible to vote")
else:
        print("Not Eligible to vote")
```

Output will be:

Eligible to vote

In the above example, when age is greater than 18, the test expression is true and body of if is executed and body of else is skipped. If age is less than 18, the test expression is false and body of else is executed and body of if is skipped. If age is equal to 18, the test expression is true and body of if is executed and body of else is skipped.

Chained Conditionals

Sometimes there are more than two possibilities and we need more than two branches. One way to express a computation like that is a chained conditional. The syntax looks like this:

General Syntax of if..elif...else statement is

```
if TEST EXPRESSION1:
    STATEMENTS_A
elif TEST EXPRESSION2:
    STATEMENTS_B
else:
    STATEMENTS_C
```

The elif is short for else if. It allows us to check for multiple expressions. If the condition for if is False, it checks the condition of the next elif block and so on. If all the conditions are False, body of else is executed. Only one block among the several if...elif...else blocks is executed according to the condition. The if block can have only one else block. But it can have multiple elif blocks

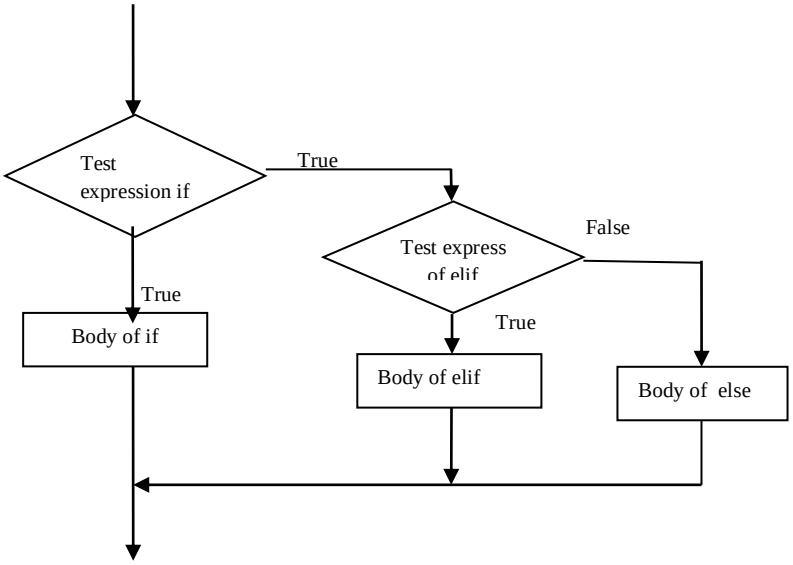


Figure show Flowchart of if...elif...else

Each condition is checked in order. If the first is false, the next is checked, and so on. If one of them is true, the corresponding branch executes, and the statement ends. Even if more than one condition is true, only the first true branch executes.

Here is an example:

```
time=17          #time=10, time=13, time=17, time=22
if time<12:
```

```
        print("Good Morning")
 elif time<15:
        print("Good Afternoon")
elif time<20:
        print("Good Evening")
else:
        print("Good Night")
```

Output will be:

Good Evening

When variable time is less than 12, Good Morning is printed. If time is less than 15, Good Afternoon is printed. If time is less than 20, Good Evening is printed. If all above conditions fails Good Night is printed.

Nested Conditionals

One conditional can also be nested within another. ie, We can have a if...elif...else statement inside another if...elif...else statement. This is called nesting in computer programming.

Any number of these statements can be nested inside one another. Indentation is the only way to figure out the level of nesting.

General Syntax of Nested if..else statement is

```
if TEST EXPRESSION1:
    if TEST EXPRESSION2:
        STATEMENTS_B
    else:
        STATEMENTS_C
    else:
  if TEST EXPRESSION3:
        STATEMENTS_D
    else:
       STATEMENTS_E
```

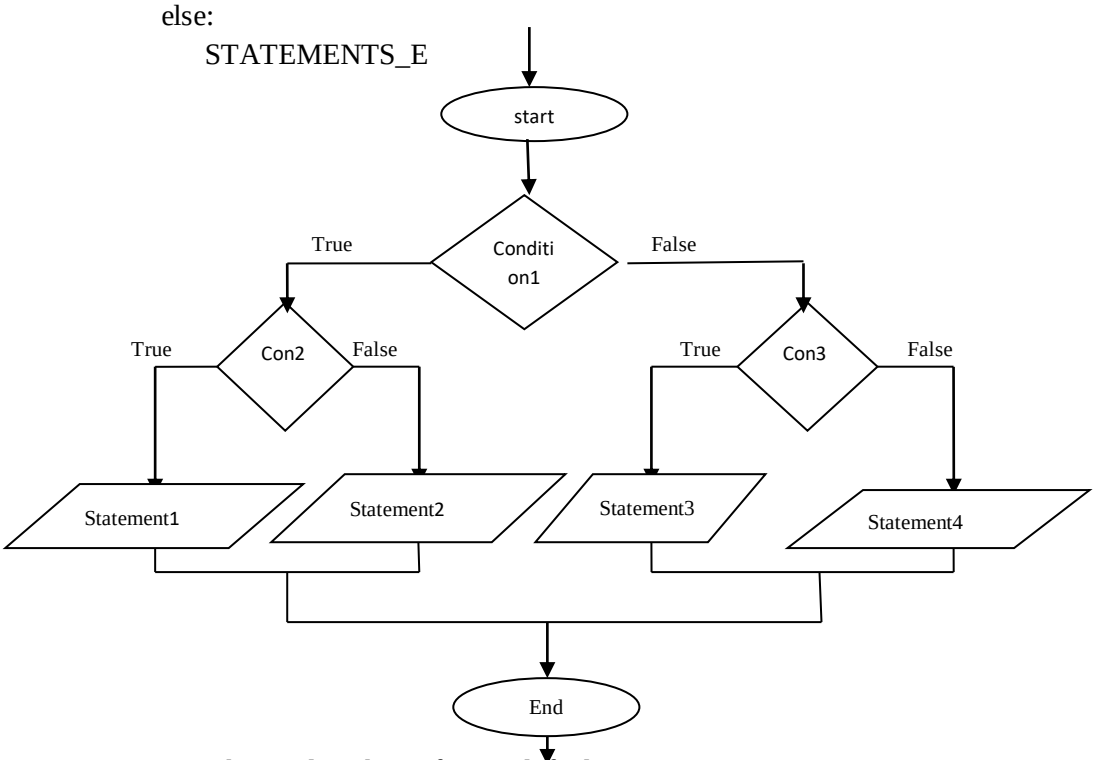


Figure show Flowchart of Nested if..else

The outer conditional contains two branches. Those two branches contain another if… else statement, which has two branches of its own. Those two branches could contain conditional statements as well.

Here is an example:

```
a=10
b=20
c=5
if a>b:
    if a>c:
      print("Greatest number is ",a)
    else:

     print("Greatest number is ",c)
else:
     if b>c:
     print("Greatest number is ",b)
else:
     print("Greatest number is ",c)
```

Output will be :

Greatest number is 20

Although the indentation of the statements makes the structure apparent, nested conditionals very quickly become difficult to read. In general, it is a good idea to avoid them when you can.

Logical operators often provide a way to simplify nested conditional statements. For example, we can rewrite the above code using single if…elif…. else statement:

```
a=10
b=20
c=5
if a>b and a>c:
   print("Greatest number is ",a)
elif b>a and b>c:
  print("Greatest number is ",b)
else:
print("Greatest number is ",c)
```

Another example, we can rewrite the following code using a single conditional:

```
if 0 < x:
   if x < 10:
      print('x is a positive single-digit number. ')
```

The print statement runs only if we make it pass both conditionals, so we can get the same effect with the and operator:

```
if 0 < x and x < 10:
print('x is a positive single-digit number. ')
```

For this kind of condition, Python provides a more concise option:

```
if 0 < x < 10:
print('x is a positive single-digit number. ')
```

ITERATION

Computers are often used to automate repetitive tasks. Repeating identical or similar tasks without making errors is something that computers do well and people do poorly.

Repeated execution of a set of statements is called iteration. Python has two statements for iteration

- for statement
- while statement

Before we look at those, we need to review a few ideas.

Reassignmnent

As we saw back in the variable section, it is legal to make more than one assignment to the same variable. A new assignment makes an existing variable refer to a new value (and stop referring to the old value).

```
age = 26
print(age)
age = 17
print(age)
```

The output of this program is

```
26
17
```

because the first time age is printed, its value is 26, and the second time, its value is 17. Here is what reassignment looks like in a state snapshot:

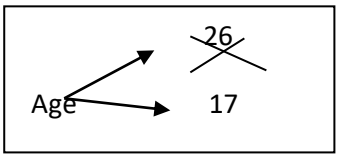


Figure show State Diagram for Reassignment

With reassignment it is especially important to distinguish between an assignment statement and a boolean expression that tests for equality. Because Python uses the equal token (=) for assignment, it is tempting to interpret a statement like a = b as a boolean test. Unlike mathematics, Remember that the Python token for the equality operator is ==.

Note too that an equality test is symmetric, but assignment is not. For example, if a == 7 then 7 == a. But in Python, the statement a = 7 is legal and 7 = a is not.

Furthermore, in mathematics, a statement of equality is always true. If a == b now, then a will always equal b. In Python, an assignment statement can make two variables equal, but because of the possibility of reassignment, they don't have to stay that way:

```
a = 5
b = a      # after executing this line, a and b are now equal
a = 3      # after executing this line, a and b are no longer equal
```

The third line changes the value of a but does not change the value of b, so they are no longer equal.

Updating variables

When an assignment statement is executed, the right-hand-side expression (i.e. the expression that comes after the assignment token) is evaluated first. Then the result of that evaluation is written into the variable on the left hand side, thereby changing it.

One of the most common forms of reassignment is an update, where the new value of the variable depends on its old value

```
n = 5
n = 3 * n + 1
```

The second line means “get the current value of n, multiply it by three and add one, and put the answer back into n as its new value”. So after executing the two lines above, n will have the value 16.

If you try to get the value of a variable that doesn’t exist yet, you’ll get an error:

```
>>> x = x + 1
Traceback (most recent call last):
 File "<interactive input>", line 1, in
 NameError: name 'x' is not defined
```

Before you can update a variable, you have to initialize it, usually with a simple assignment:

```
>>> x = 0
>>> x = x + 1
```

This second statement — updating a variable by adding 1 to it. It is very common. It is called an increment of the variable; subtracting 1 is called a decrement.

The while statement

The while loop in Python is used to iterate over a block of code as long as the test expression (condition) is true. We generally use this loop when we don't know beforehand, the number of times to iterate.

The general syntax for the while statement :

```
while TEST_EXPRESSION:
        STATEMENTS
```

In while loop, test expression is checked first. The body of the loop is entered only if the TEST_EXPRESSION evaluates to True. After one iteration, the test expression is checked again. This process continues until the TEST_EXPRESSION evaluates to False.

In Python, the body of the while loop is determined through indentation. Body starts with indentation and the first unindented line marks the end.

Python interprets any non-zero value as True. None and 0 are interpreted as false

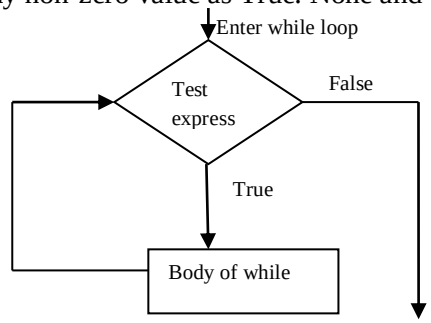


Figure show Flowchart of while Loop

Example:

Python program to find sum of first n numbers using while loop

```
n = 20
sum = 0     # initialize sum and counter
i = 1
while i <= n:
    sum = sum + i
    i = i+1        # update counter
print("The sum is", sum)   # print the sum
```

When you run the program, the output will be:

The sum is 210

In the above program, the test expression will be True as long as our counter variable i is less than or equal to n (20 in our program).

We need to increase the value of counter variable in the body of the loop. This is very important (and mostly forgotten). Failing to do so will result in an infinite loop (never ending loop). Finally the result is displayed.

The for Statement

The for loop in Python is used to iterate over a sequence (list, tuple, string) or other iterable objects. Iterating over a sequence is called traversal.

The general syntax for the while statement:

```
for LOOP_VARIABLE in SEQUENCE:
    STATEMENTS
```

Here, LOOP_VARIABLE is the variable that takes the value of the item inside the sequence on each iteration.

Loop continues until we reach the last item in the sequence. The body of for loop is separated from the rest of the code using indentation.

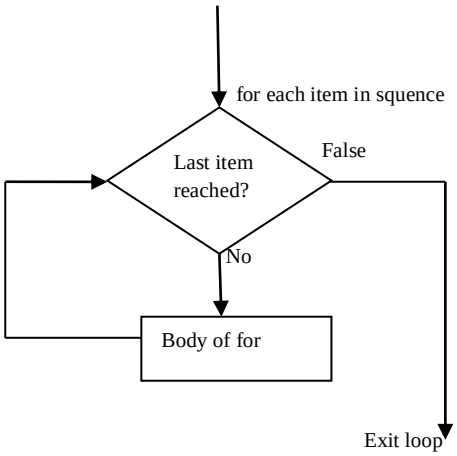


Figure show Flowchart of for Loop

Example

```
marks = [95,98,89,93,86]
total = 0
for subject_mark in marks:
        total = total+subject_mark
print("Total Mark is ", total)
```

when you run the program, the output will be:

Total Mark is 461

We can use the range() function in for loops to iterate through a sequenc Numbers.

Example:

```
sum=0
for i in range(20):
      sum=sum+i
```

```
Print("Sum is ", sum)
```

Output will be:

```
Sum is  190
```

Break, Continue, Pass

You might face a situation in which you need to exit a loop completely when an external condition is triggered or there may also be a situation when you want to skip a part of the loop and start next execution.

Python provides break and continue statements to handle such situations and to have good control on your loop. This section will discuss the break, continue and pass statements available in Python.

The break Statement

The break statement in Python terminates the current loop and resumes execution at the next statement. The most common use for break is when some external condition is triggered requiring a hasty exit from a loop. The break statement can be used in both while and for loops.

Example:

```
for letter in 'Welcome':        # First Example
    if letter == 'c':
        break
print('Current Letter :', letter)
var = 10                    # Second Example
while var > 0:
        print('Current variable value :', var)
        var = var -1
        if var == 5:
            break
 print "End!"
```

This will produce the following output:

```
Current Letter : W
Current Letter : e
Current Letter : l
Current variable value : 10
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
End!
```

The continue Statement: The continue statement in Python returns the control to the beginning of the while loop. The continue statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop. The continue statement can be used in both while and for loops.

Example:

```
for letter in 'Welcome':        # First Example
            if letter == 'c':
                continue
            print('Current Letter :', letter)
 var = 10                    # Second Example
 while var > 0:
```

```
        print('Current variable value :', var)
          var = var -1
          if var == 5:
         continue
print "End!"
```

This will produce the following output:

```
Current Letter : W
 Current Letter : e
 Current Letter : l
Current Letter : o
 Current Letter : m
 Current Letter : e
 Current variable value : 10
 Current variable value : 9
 Current variable value : 8
 Current variable value : 7
 Current variable value : 6
Current variable value : 5
Current variable value : 4
 Current variable value : 3
 Current variable value : 2
Current variable value : 1
End!
```

The else Statement Used with Loops

Python supports to have an else statement associated with a loop statements.

- If the else statement is used with a for loop, the else statement is executed when the loop has exhausted iterating the list.
- If the else statement is used with a while loop, the else statement is executed when the condition becomes false.

while loop with else

We can have an optional else block with while loop as well. The else part is executed if the condition in the while loop evaluates to False. The while loop can be terminated with a break statement. In such case, the else part is ignored. Hence, a while loop's else part runs if no break occurs and the condition is false.

Here is an example to illustrate this.

```
counter = 0
while counter < 3:
        print("Inside loop")
        counter = counter + 1
       else:
       print("Inside else")
```

Output

```
Inside loop
Inside loop
Inside loop
Inside else
```

Here, we use a counter variable to print the string Inside loop three times. On the forth iteration, the condition in while becomes False. Hence, the else part is executed.

for loop with else

A for loop can have an optional else block as well. The else part is executed if the items in the sequence used in for loop exhausts. break statement can be used to stop a for loop. In such case, the else part is ignored. Hence, a for loop's else part runs if no break occurs.

Here is an example to illustrate this.

```
digits = [0, 1, 5]
for i in digits:
    print(i)
else:
print("No items left.")
```

When you run the program, the output will be:

```
0
1
5
No items left.
```

Here, the for loop prints items of the list until the loop exhausts. When the for loop exhausts, it executes the block of code in the else and prints

The pass Statement

In Python programming, pass is a null statement. The difference between a comment and pass statement in Python is that, while the interpreter ignores a comment entirely, pass is not ignored. However, nothing happens when pass is executed. It results into no operation (NOP).

Syntax of pass

```
pass
```

We generally use it as a placeholder.

Suppose we have a loop or a function that is not implemented yet, but we want to implement it in the future. They cannot have an empty body. The interpreter would complain. So, we use the pass statement to construct a body that does nothing.

Example

```
sequence = {'p', 'a', 's', 's'}
for val in sequence:
    pass
We can do the same thing in an empty function
def function(args):
   pass
```

Text & Binary Files

FILES

Most of the programs we have seen so far are transient in the sense that they run for a short time and produce some output, but when they end, their data disappears. If you run the program again, it starts with a clean slate.

Other programs are persistent: they run for a long time (or all the time); they keep at least some of their data in permanent storage (a hard drive, for example); and if they shut down and restart, they pick up where they left off.

One of the simplest ways for programs to maintain their data is by reading and writing text files. An alternative is to store the state of the program in a database.

Text Files

A text file is a sequence of characters stored on a permanent medium like a hard drive, flash memory, or CD-ROM. Text file contain only text, and has no special formatting such as bold text, italic text, images, etc. Text files are identified with the .txt file extension.

Reading and Writing to Text Files

Python provides inbuilt functions for creating, writing and reading files. There are two types of files that can be handled in python, normal text files and binary files (written in binary language,0s and 1s).

Text files: In this type of file, each line of text is terminated with a special character called EOL (End of Line), which is the new line character („\n") in python by default.

Binary files: In this type of file, there is no terminator for a line and the data is stored after converting it into machine understandable binary language.

In order to perform some operations on files we have to follow below steps

- Opening
- Reading or writing
- Closing Here we are going to discusses about opening, closing, reading and writing data in a text file.

File Access Modes

Access modes govern the type of operations possible in the opened file. It refers to how the file will be used once its opened. These modes also define the location of the File Handle in the file. File handle is like a cursor, which defines from where the data has to be read or written in the file. There are 6 access modes in python.

- Read Only ('r') : Open text file for reading. The handle is positioned at the beginning of the file. If the file does not exists, raises I/O error. This is also the default mode in which file is opened.
- Read and Write ('r+') : Open the file for reading and writing. The handle is positioned at the beginning of the file. Raises I/O error if the file does not exists.
- Write Only ('w') : Open the file for writing. For existing file, the data is truncated and over-written. The handle is positioned at the beginning of the file. Creates the file if the file does not exists.
- Write and Read ('w+') : Open the file for reading and writing. For existing file, data is truncated and over-written. The handle is positioned at the beginning of the file.
- Append Only ('a') : Open the file for writing. The file is created if it does not exist. The handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data.
- Append and Read ('a+') : Open the file for reading and writing. The file is created if it does not exist. The handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data.

Opening a File It is done using the open() function. No module is required to be imported for this function.

Syntax: File_object = open(r"File_Name","Access_Mode")

The file should exist in the same directory as the python program file else, full address (path will be discussed in later section of this unit) of the file should be written on place of filename. Note: The r is placed before filename to prevent the characters in filename string to be treated as special character. For example, if there is \temp in the file address, then \t is treated as the tab character and error is

raised of invalid address. The r makes the string raw, that is, it tells that the string is without any special characters. The r can be ignored if the file is in same directory and address is not being placed.

Example:

```
>>> f1 = open("sample.txt","a")
>>> f2 = open(r"G:\class\python\sample3.txt","w+")
```

Here, f1 is created as object for sample.txt and f3 as object for sample3.txt (available in G:\class\python directory)

Closing a File close() function closes the file and frees the memory space acquired by that file. It is used at the time when the file is no longer needed or if it is to be opened in a different file mode.

Syntax: File_object.close()

Example:

```
>>> f1 = open("smapl.txt","a")
>>> f1.close()
```

After closing a file we can‟t perform any operation on that file. If want to do so we have to open the file again

Reading from a File To read the content of a file, we must open the file in reading mode. There are three ways to read data from a text file.

1.read() : Returns the read bytes in form of a string. Reads n bytes, if no n specified, reads the entire file.

File_object.read([n])

2. readline() : Reads a line of the file and returns in form of a string.For specified n, reads at most n bytes. However, does not reads more than one line, even if n exceeds the length of the line.

File_object.readline([n])

3. readlines() : Reads all the lines and return them as each line a string element in a list.

File_object.readlines()

Example: Consider the content of file sample.txt that is present in location G:\class\python\code\ as

```
Read Only
Read and Write
Write OnlyWrite and Read
Append Only Append and Read
```

Now execute the following file reading script.

```
>>> f1=pen("G:\class\python\code\sample.txt","r")
>>> f1.read() '
Read Only\nRead and Write\nWrite Only\nWrite and Read\nAppend Only\nAppend and Read'
```

Here \n denotes next line character. If you again run the same script, you will get empty string. Because during the first read statement itself file handler reach the end of the file. If you read again it will return empty string

```
>>> f1.read() ''
```

So in order to take back the file handler to the beginning of the file you have to open the file again or use seek() function. We will discuss about seek function in upcoming section.

```
>>> f1=open("G:\class\python\code\sample.txt","r")
>>> f1.read(10)
'Read Only\n'
>>> f1=open("G:\class\python\code\sample.txt","r")
>>> f1.readline() 'Read Only\n' >>> f1=open("G:\class\python\code\sample.txt","r") >>>
f1.readlines()
```

['Read Only\n', 'Read and Write\n', 'Write Only\n', 'Write and Read\n', 'Append Only\n','Append and Read']

File Positions

tell(): The tell() method tells you the current position within the file; in other words, the next read or write will occur at that many bytes from the beginning of the file.

seek(): The seek(offset[, from]) method changes the current file position. The offset argument indicates the number of bytes to be moved. The from argument specifies the reference position from where the bytes are to be moved.

If from is set to 0, it means use the beginning of the file as the reference position and 1 means use the current position as the reference position and if it is set to 2 then the end of the file would be taken as the reference position. If the second argument is omitted, it also means use the beginning of the file as the reference position.

Example:

```
>>> f1=open("G:\class\python\code\sample.txt","r")
 >>> f1.tell()
 0L
 >>> f1.readline()
 'Read Only\n'
 >>> f1.tell()
11L
 >>> f1.seek(0)
 >>> f1.tell()
 0L
 >>> f1.seek(5)
 >>> f1.tell()
 5L
 >>> f1.readline()
 'Only\n'
```

Writing to a File

In order to write into a file we need to open it in write 'w' or append 'a'. We need to be careful with the 'w' mode as it will overwrite into the file if it already exists. All previous data are erased.

There are two ways to write in a file.

1. write() : Inserts the string str1 in a single line in the text file. File_object.write(str1)
2. writelines() : For a list of string elements, each string is inserted in the text file.Used to insert multiple strings at a single time.

File_object.writelines(L) for L = [str1, str2, str3]

Example:

```
>>> f4=open("fruit.txt","w")
>>> fruit_list=['Apple\n','Orange\n','Pineapple\n']
 >>> f4.writelines(fruit_list)
 >>> f4.write('Strawberry\n')
 >>> f4.close()
 >>> f4=open("fruit.txt","r")
 >>> f4.read()
'Apple\nOrange\nPineapple\nStrawberry\n
```

Appending to a File Adding content at the end of a file is known as append. In order to do appending operation, we have to open the file with append mode.

Example:

```
>>> f4=open('fruit.txt','a')
 >>> f4.write('Banana')
 >>> f4.close()
>>> f4=open('fruit.txt','r')
 >>> f4.read()
'Apple\nOrange\nPineapple\nStrawberry\nBanana\n'
```

The File Object Attributes Once a file is opened and you have one file object, you can get various information related to that file. Here is a list of all attributes related to file object:

Attribute	Description
File_object.closed	Returns true if file is closed, false otherwise.
File_object.mode	Returns access mode with which file was opened.
File_object.name	Returns name of the file.
File_object.softspace	Returns false if space explicitly required with print, true otherwise.

Format Operator The argument of write has to be a string, so if we want to put other values in a file, we have to convert them to strings. The easiest way to do that is with str:

```
 >>> f5=open('stringsample.txt','w')
 >>> f5.write(5)
 TypeError: expected a string or other character buffer object
 >>> f5.write(str(5))
```

An alternative is to use the format operator, %. When applied to integers, % is the modulus operator. But when the first operand is a string, % is the format operator. The first operand is the format string, which contains one or more format sequences, which specify how the second operand is formatted. The result is a string. For example, the format sequence '%d' means that decimal value is converted to string.

```
 >>> run=8
 >>> '%d'%run
          '8'
```

The result is the string '8', which is not to be confused with the integer value 8. Some other format strings are.

Conversion	Meaning
d	Signed integer decimal.
I	Signed integer decimal.
O	Unsigned octal.
u	Unsigned decimal.
x	Unsigned hexadecimal (lowercase).
X	Unsigned hexadecimal (uppercase).
e	Floating point exponential format (lowercase).
E	Floating point exponential format (uppercase)
. f	Floating point decimal format.
F	Floating point decimal format.
g	Same as "e" if exponent is greater than -4 or less than precision, "f" otherwise.
G	Same as "E " if exponent is greater than -4 or less than precision, "F" otherwise.

C Single character (accepts integer or single character string).
r String (converts any python object using repr()).
s String (converts any python object using str()).
% No argument is converted, results in a "%" character in the result.

A format sequence can appear anywhere in the string, so you can embed a value in a sentence:

```
>>> 'India need %d runs'%3
'India need 3 runs'
```

If there is more than one format sequence in the string, the second argument has to be a tuple. Each format sequence is matched with an element of the tuple, in order.

```
>>> 'India need %d runs in %d balls'%(3,5)
'India need 3 runs in 5 balls'
```

The following example uses '%d' to format an integer, '%g' to format a floating-point number, and '%s' to format a string:

```
>>> '%d %s price is %g rupees'%(5,'apple',180.50)
'5 apple price is 180.500000 rupees'
```

The number of elements in the tuple has to match the number of format sequences in the string. Also, the types of the elements have to match the format sequences:

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string >>>
'%d' % 'apple'
TypeError: %d format: a number is required, not str
```

In the first example, there aren‟t enough elements; in the second, the element is the wrong type.

Filenames and Paths Files are organized into directories (also called “folders”). Every running program has a “current directory”, which is the default directory for most operations. For example, when you open a file for reading, Python looks for it in the current directory. The os module provides functions for working with files and directories (“os” stands for “operating system”). os.getcwd returns the name of the current directory:

```
>>> import os
>>> os.getcwd()
'C:\\Python27'
```

cwd stands for “current working directory”. A string like 'C:\\Python27' that identifies a file or directory is called a path.

A simple filename, like 'stringsample.txt' is also considered a path, but it is a relative path because it relates to the current directory.

If the current directory 'C:\\Python27', the filename 'stringsample.txt' would refer to 'C:\\Python27\\stringsample.txt '.

A path that begins with drive letter does not depend on the current directory; it is called an absolute path. To find the absolute path to a file, you can use os.path.abspath: >>> os.path.abspath('stringsample.txt') 'C:\\Python27\\stringsample.txt'

os.path provides other functions for working with filenames and paths. For example, os.path.exists checks whether a file or directory exists:

```
>>> os.path.exists('memo.txt')
True
```

If it exists, os.path.isdir checks whether it‟s a directory:

```
>>> os.path.isdir('memo.txt')
```

False
>>> os.path.isdir ('C:\\Python27')
True
Similarly, os.path.isfile checks whether it‟s a file. os.listdir returns a list of the files (and other directories) in the given directory:
>>> cwd=os.getcwd()
>>> os.listdir(cwd)
['DLLs', 'Doc', 'include', 'infinitLoop.py', 'Lib', 'libs', 'LICENSE.txt', 'NEWS.txt', 'parameter.py', 'python.exe', 'pythonw.exe', 'README.txt', 'sample.txt', 'sample2.txt', 'Scripts', 'stringsample.txt', 'swapwith third.py', 'tcl', 'Tools', 'w9xpopen.exe', 'wc.py', 'wc.pyc']
To demonstrate these functions, the following example “walks” through a directory, prints the names of all the files, and calls itself recursively on all the directories.

```
>>> def walk(dirname):
for name in os.listdir(dirname):
path = os.path.join(dirname, name)
if os.path.isfile(path):
  print(path)
  else:
   walk(path)
>>> cwd=os.getcwd()
>>> walk(cwd)
```

Output:

```
C:\Python27\DLLs\bz2.pyd
C:\Python27\DLLs\py.ico
C:\Python27\DLLs\pyc.ico
C:\Python27\DLLs\pyexpat.pyd
C:\Python27\DLLs\select.pyd
C:\Python27\DLLs\sqlite3.dll
C:\Python27\DLLs\tcl85.dll
C:\Python27\include\abstract.h
C:\Python27\include\asdl.h
C:\Python27\include\ast.h
```

os.path.join takes a directory and a file name and joins them into a complete path.

```
>>> os.path.join(cwd,'stringsample.txt')
'C:\\Python27\\stringsample.txt'
```

Data Wrangling in Python

Data Wrangling is the process of gathering, collecting, and transforming Raw data into another format for better understanding, decision-making, accessing, and analysis in less time. Data Wrangling is also known as Data Munging.

Importance Of Data Wrangling

Data Wrangling is a very important step. The below example will explain its importance as :

Books selling Website want to show top-selling books of different domains, according to user preference. For example, a new user search for motivational books, then they want to show those motivational books which sell the most or having a high rating, etc.

But on their website, there are plenty of raw data from different users. Here the concept of Data Munging or Data Wrangling is used. As we know Data is not Wrangled by System. This process is done by Data Scientists. So, the data Scientist will wrangle data in such a way that they will sort that motivational books that are sold more or have high ratings or user buy this book with these package of Books, etc. On the basis of that, the new user will make choice. This will explain the importance of Data wrangling.

Data Wrangling in Python

Data Wrangling is a crucial topic for Data Science and Data Analysis. Pandas Framework of Python is used for Data Wrangling. Pandas is an open-source library specifically developed for Data Analysis and Data Science. The process like data sorting or filtration, Data grouping, etc.

Data wrangling in python deals with the below functionalities:

1. **Data exploration:** In this process, the data is studied, analyzed and understood by visualizing representations of data.
2. **Dealing with missing values:** Most of the datasets having a vast amount of data contain missing values of *NaN, they are needed to be taken* careof by replacing them with mean, mode, the most frequent value of the column or simply by dropping the row having a *NaN*value.
3. **Reshaping data:** In this process, data is manipulated according to the requirements, where new data can be added or pre-existing data can be modified.
4. **Filtering data:** Some times datasets are comprised of unwanted rows or columns which are required to be removed or filtered
5. **Other:** After dealing with the raw dataset with the above functionalities we get an efficient dataset as per our requirements and then it can be used for a required purpose like data analyzing, machine learning, data visualization, model training etc.

Below is an example which implements the above functionalities on a raw dataset:

- **Data exploration**, here we assign the data, and then we visualize the data in a tabular format.

Python3

```
# Import pandas package
importpandas as pd
# Assign data
data ={'Name': ['Jai', 'Princi', 'Gaurav',
               'Anuj', 'Ravi', 'Natasha', 'Riya'],
      'Age': [17, 17, 18, 17, 18, 17, 17],
      'Gender': ['M', 'F', 'M', 'M', 'M', 'F', 'F'],
      'Marks': [90, 76, 'NaN', 74, 65, 'NaN', 71]}
  # Convert into DataFrame
df =pd.DataFrame(data)
# Display data
df
```

output:

	Name	Age	Gender	Marks
0	Jai	17	M	90
1	Princi	17	F	76
2	Gaurav	18	M	NaN
3	Anuj	17	M	74
4	Ravi	18	M	65
5	Natasha	17	F	NaN
6	Riya	17	F	71

- **Dealing with missing values**, as we can see from the previous output, there are NaNvalues present in the MARKS column which are going to be taken care of by replacing them with the column mean.

Python3

```
# Compute average
c =avg =0
forele indf['Marks']:
    ifstr(ele).isnumeric():
        c +=1
        avg +=ele
avg /=c
# Replace missing values
df =df.replace(to_replace="NaN",
               value=avg)
 # Display data
Df
```

Output:

	Name	Age	Gender	Marks
0	Jai	17	M	90.0
1	Princi	17	F	76.0
2	Gaurav	18	M	75.2
3	Anuj	17	M	74.0
4	Ravi	18	M	65.0
5	Natasha	17	F	75.2
6	Riya	17	F	71.0

- **Reshaping data**, in the *GENDER* column, we can reshape the data by categorizing them into different numbers.

Python3

```
# Categorize gender
df['Gender'] =df['Gender'].map({'M': 0,
                                'F': 1, }).astype(float)
```

```
 # Display data
df
```

	Name	Age	Gender	Marks
0	Jai	17	0.0	90.0
1	Princi	17	1.0	76.0
2	Gaurav	18	0.0	75.2
3	Anuj	17	0.0	74.0
4	Ravi	18	0.0	65.0
5	Natasha	17	1.0	75.2
6	Riya	17	1.0	71.0

- **Filtering data**, suppose there is a requirement for the details regarding name, gender, marks of the top-scoring students. Here we need to remove some unwanted data

Python3

```
# Filter top scoring students
df =df[df['Marks'] >=75]
# Remove age row
df =df.drop(['Age'], axis=1)
 # Display data
Df
```

Output:

	Name	Gender	Marks
0	Jai	0.0	90.0
1	Princi	1.0	76.0
2	Gaurav	0.0	75.2
5	Natasha	1.0	75.2

- Hence, we have finally obtained an efficient dataset which can be further used for various purposes.
- Now that we know the basics of data wrangling. Below we will discuss various operations using which we can perform data wrangling:

Wrangling Data Using Merge Operation

Merge operation is used to merge raw data and into the desired format.

Syntax

pd.merge(data_frame1,data_frame2, on="field ")

Here the field is the name of the column which is similar on both data-frame.

For example: Suppose that a Teacher has two types of Data, first type of Data consist of Details of Students and Second type of Data Consist of Pending Fees Status which is taken from Account Office. So The Teacher will use merge operation here in order to merge the data and provide it meaning. So that teacher will analyze it easily and it also reduces time and effort of Teacher from Manual Merging.

FIRST TYPE OF DATA:

Python3

```
# import module
importpandas as pd
 # creating DataFrame for Student Details
details =pd.DataFrame({
    'ID': [101, 102, 103, 104, 105, 106,
          107, 108, 109, 110],
    'NAME': ['Jagroop', 'Praveen', 'Harjot',
           'Pooja', 'Rahul', 'Nikita',
           'Saurabh', 'Ayush', 'Dolly', "Mohit"],
    'BRANCH': ['CSE', 'CSE', 'CSE', 'CSE', 'CSE',
            'CSE', 'CSE', 'CSE', 'CSE', 'CSE']})
  # printing details
print(details)
```

Output:

	ID	NAME	BRANCH
0	101	Jagroop	CSE
1	102	Praveen	CSE
2	103	Harjot	CSE
3	104	Pooja	CSE
4	105	Rahul	CSE
5	106	Nikita	CSE
6	107	Saurabh	CSE
7	108	Ayush	CSE
8	109	Dolly	CSE
9	110	Mohit	CSE

SECOND TYPE OF DATA:

Python3

```
# Import module
importpandas as pd
 # Creating Dataframe for Fees_Status
fees_status =pd.DataFrame(
    {'ID': [101, 102, 103, 104, 105,
          106, 107, 108, 109, 110],
     'PENDING': ['5000', '250', 'NIL',
               '9000', '15000', 'NIL',
               '4500', '1800', '250', 'NIL']})
  # Printing fees_status
print(fees_status)
```

Output:

```
    ID PENDING
0  101    5000
1  102     250
2  103     NIL
3  104    9000
4  105   15000
5  106     NIL
6  107    4500
7  108    1800
8  109     250
9  110     NIL
```

WRANGLING DATA USING MERGE OPERATION:

Python3

```
# Import module
importpandas as pd
# Creating Dataframe
details =pd.DataFrame({
   'ID': [101, 102, 103, 104, 105,
         106, 107, 108, 109, 110],
   'NAME': ['Jagroop', 'Praveen', 'Harjot',
          'Pooja', 'Rahul', 'Nikita',
          'Saurabh', 'Ayush', 'Dolly', "Mohit"],
   'BRANCH': ['CSE', 'CSE', 'CSE', 'CSE', 'CSE',
           'CSE', 'CSE', 'CSE', 'CSE', 'CSE']})

# Creating Dataframe
fees_status =pd.DataFrame(
   {'ID': [101, 102, 103, 104, 105,
         106, 107, 108, 109, 110],
    'PENDING': ['5000', '250', 'NIL',
             '9000', '15000', 'NIL',
             '4500', '1800', '250', 'NIL']})
  # Merging Dataframe
print(pd.merge(details, fees_status, on='ID'))
```

Output:

```
    ID     NAME BRANCH PENDING
0  101  Jagroop    CSE    5000
1  102  Praveen    CSE     250
2  103   Harjot    CSE     NIL
3  104    Pooja    CSE    9000
4  105    Rahul    CSE   15000
5  106   Nikita    CSE     NIL
6  107  Saurabh    CSE    4500
7  108    Ayush    CSE    1800
8  109    Dolly    CSE     250
9  110    Mohit    CSE     NIL
```

Wrangling Data using Grouping Method

The grouping method in Data analysis is used to provide results in terms of various groups taken out from Large Data. This method of pandas is used to group the outset of data from the large data set.

Example: There is a Car Selling company and this company have different Brands of various Car Manufacturing Company like Maruti, Toyota, Mahindra, Ford, etc. and have data where different cars are sold in different years. So the Company wants to wrangle only that data where cars are sold during the year 2010. For this problem, we use another Wrangling technique that is*groupby()* method.

CARS SELLING DATA:

Python3

```
# Import module
importpandas as pd
# Creating Data
car_selling_data ={'Brand': ['Maruti', 'Maruti',
'Maruti',
                           'Maruti', 'Hyundai', 'Hyundai',
                           'Toyota', 'Mahindra', 'Mahindra',
                           'Ford', 'Toyota', 'Ford'],
                   'Year':  [2010, 2011, 2009, 2013,
                           2010, 2011, 2011, 2010,
                           2013, 2010, 2010, 2011],
                   'Sold': [6, 7, 9, 8, 3, 5,
                          2, 8, 7, 2, 4, 2]}

# Creating Dataframe of car_selling_data
df =pd.DataFrame(car_selling_data)
 # printing Dataframe
print(df)
```

Output:

	Brand	Year	Sold
0	Maruti	2010	6
1	Maruti	2011	7
2	Maruti	2009	9
3	Maruti	2013	8
4	Hyundai	2010	3
5	Hyundai	2011	5
6	Toyota	2011	2
7	Mahindra	2010	8
8	Mahindra	2013	7
9	Ford	2010	2
10	Toyota	2010	4
11	Ford	2011	2

DATA OF THEYEAR 2010:

Python3

```
# Import mdoule
importpandas as pd
 # Creating Data
car_selling_data ={'Brand': ['Maruti', 'Maruti', 'Maruti',
                           'Maruti', 'Hyundai', 'Hyundai',
                           'Toyota', 'Mahindra', 'Mahindra',
                           'Ford', 'Toyota', 'Ford'],
                   'Year':  [2010, 2011, 2009, 2013,
                           2010, 2011, 2011, 2010,
                           2013, 2010, 2010, 2011],
```

```
                'Sold': [6, 7, 9, 8, 3, 5,
                         2, 8, 7, 2, 4, 2]}

# Creating Dataframe for Provided Data
df =pd.DataFrame(car_selling_data)
 # Group the data when year = 2010
grouped =df.groupby('Year')
print(grouped.get_group(2010))
```

Output:

	Brand	Year	Sold
0	Maruti	2010	6
4	Hyundai	2010	3
7	Mahindra	2010	8
9	Ford	2010	2
10	Toyota	2010	4

Wrangling data by removing Duplication

Pandas *duplicates()* method helps us to remove duplicate values from Large Data. An important part of Data Wrangling is removing Duplicate values from the large data set.

Syntax:

```
DataFrame.duplicated(subset=None, keep='first')
```

Here subset is the column value where we want to remove Duplicate value.

In *keep*, we have 3 options :

- if *keep ='first'* then the first value is marked as original rest all values if occur will be removed as it is considered as duplicate.
- if *keep='last'* then the last value is marked as original rest all above same values will be removed as it is considered as duplicate values.
- if *keep ='false'* the all the values which occur more than once will be removed as all considered as a duplicate value.

For example, A University will organize the event. In order to participate Students have to fill their details in the online form so that they will contact them. It may be possible that a student will fill the form multiple time. It may cause difficulty for the event organizer if a single student will fill multiple entries. The Data that the organizers will get can be Easily Wrangles by removing duplicate values.

DETAILS STUDENTS DATA WHO WANT TO PARTICIPATE IN THE EVENT:

Python3

```
# Import module
importpandas as pd
# Initializing Data
student_data ={'Name': ['Amit', 'Praveen', 'Jagroop',
                        'Rahul', 'Vishal', 'Suraj',
                        'Rishab', 'Satyapal', 'Amit',
                        'Rahul', 'Praveen', 'Amit'],

               'Roll_no': [23, 54, 29, 36, 59, 38,
                           12, 45, 34, 36, 54, 23],

               'Email': ['xxxx@gmail.com', 'xxxxxx@gmail.com',
                         'xxxxxx@gmail.com', 'xx@gmail.com',
                         'xxxx@gmail.com', 'xxxxx@gmail.com',
```

```
                'xxxxx@gmail.com', 'xxxxx@gmail.com',
                'xxxxx@gmail.com', 'xxxxxx@gmail.com',
                'xxxxxxxxxx@gmail.com', 'xxxxxxxxxx@gmail.com']}
  # Creating Dataframe of Data
df =pd.DataFrame(student_data)
# Printing Dataframe
print(df)
```

Output:

	Name	Roll_no	Email
0	Amit	23	xxxx@gmail.com
1	Praveen	54	xxxxxx@gmail.com
2	Jagroop	29	xxxxxx@gmail.com
3	Rahul	36	xx@gmail.com
4	Vishal	59	xxxx@gmail.com
5	Suraj	38	xxxxx@gmail.com
6	Rishab	12	xxxxx@gmail.com
7	Satyapal	45	xxxxx@gmail.com
8	Amit	34	xxxxx@gmail.com
9	Rahul	36	xxxxxx@gmail.com
10	Praveen	54	xxxxxxxxxx@gmail.com
11	Amit	23	xxxxxxxxxx@gmail.com

Python Classes/Objects

Python is an object oriented programming language.

Almost everything in Python is an object, with its properties and methods.

A Class is like an object constructor, or a "blueprint" for creating objects.

Create a Class

To create a class, use the keyword class:

Example

Create a class named MyClass, with a property named x:

```
class MyClass:
  x = 5
```

Create Object

Now we can use the class named MyClass to create objects:

Example

Create an object named p1, and print the value of x:

```
p1 = MyClass()
print(p1.x)
```

The __init__() Function

The examples above are classes and objects in their simplest form, and are not really useful in real life applications.

To understand the meaning of classes we have to understand the built-in __init__() function.

All classes have a function called __init__(), which is always executed when the class is being initiated.

Use the __init__() function to assign values to object properties, or other operations that are necessary to do when the object is being created:

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)
```

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

Object Methods

Objects can also contain methods. Methods in objects are functions that belong to the object.

Let us create a method in the Person class:

Example

Insert a function that prints a greeting, and execute it on the p1 object:

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

  def myfunc(self):
    print("Hello my name is " + self.name)

p1 = Person("John", 36)
p1.myfunc()
```

Note: The self parameter is a reference to the current instance of the class, and is used to access variables that belong to the class.

Inheritance

Inheritance allow 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

Create a parent class

Any class can be a parent class so the syntax is the same as creating any class

Example

create a class named person, with first name and last name properties and a print name method.

```
Class person:
def__init__(self,fname,lname):
Self.firstname=fname
Self.lastname=lname
def printname(self):
Print(self.firstname, self.lastname)
```

Use the person class to create an object and execute the print name method:

```
X=person("john","doc")
x.printname()
```

create a child class

To create a class that inherits the functionality from another class, send the parent class as a parents when creating the child class

Example

Create a class named student, which will inherit the properties and method from the person class

Class student (person):
pass
now the student class has the some properties methods as the person class
Example
use the student class to create an object and the execute the print name method:
x=student("milk","water")
x.printname()
Add the__init__() function
So for have created a child class that inherits the properties &methods from its parent.we want to add the__init __()function to the child class (instead of the pass keyword)

Note:The __init__()function is called automatically every time is being used to create new object.

Example:Add the __init__()function to the student class
Classstudemt(person):
def__init__(self,fname,lname):
when you add the __init__()function, the child class will no longer inherit the parents
__init__()function

Note: The child __init__()function overrides the inheritance of the parent's__init__()function.
To keep the inheritance of the parent's

__init__()function,add a cell to the parents
__init__()function:
Example:
class student(person):
def__init__(self,fname,lname):
person__init__(self,fname,lname)
added the __init__()function &kept the inheritance of the parent class use the super()function
python also has a super()function that will make the child class inherit all the methods &properties from its parent
Example:
class student(person):
def__init__(self,fname,lname):
super()__init__(fname,lname):
super()function you do not have the name of the name of the parent element, it will automatically inherit the methods &properties from its parent
Add properties
Add properties called graduation year to the student class
Class student(person)
def__init__(self,fname,lname):
super()__init__(fname,lname)
self.graduation year=2019
2019should be variable &passed into the students class creating student object to add another parameter in__init__()function
Example
Add year parameter &pass the correct year when creating object
Create student(person):
def__init__(self,fname,lname,year)
super()__init__(fname,lname)
self.graduation year=year
x=student("milk","water",2019)
Add method
Addmethod called welcome to the student class

Class student(person)
def__init__(self,fname,lname,year):
super()__init__(fname,lname)
self.graduation year=year
def welcome(self):
print("welcome",self.firstname,self.lastnames," to the class of" self.graduationyear)
child class with the same name as function in the parent class the inheritance of the parent method will be overriden

constructor
constructor is a special method that is used to initialize a newely created object and is called just after the memory is allocated for the object.
Default constructor: it is simple constructor which doesn't accept any arguments .It's definition has only one argument which is a reference to the instance being constructed.
Parameterized constructor: constructor with parameters is known as parameterized constructor take its first argument as a reference to the instance being constructed known as self and the rast of the arguments are provided by the programmer
Syntax:def_init_(self.[argument values(3)]):
Self.memberName=argument value.

Features of Python Constructors:

- In Python, a Constructor begins with double underscore (_) and is always named as __init__().
- In python Constructors, arguments can also be passed.
- In Python, every class must necessarily have a Constructor.
- If there is a Python class without a Constructor, a default Constructor is automatically created without any arguments and parameters.
- **Example:**

```
class Employees():
    def __init__(self, Name, Salary):
        self.Name = Name
        self.Salary = Salary

    def details(self):
        print "Employee Name : ", self.Name
        print "Employee Salary: ", self.Salary
        print "\n"

first = Employees("Khush", 10000)
second = Employees("Raam", 20000)
third = Employees("Lav", 10000)
fourth = Employees("Sita", 30000)
fifth = Employees("Lucky", 50000)

first.details()
second.details()
third.details()
fourth.details()
fifth.details()
```

- 
vibhuti@kmradi-Inspiron-3541:~/Desktop$ python pythonexample.py
Employee Name : Khush
Employee Salary: 10000
Employee Name : Raam
Employee Salary: 20000
Employee Name : Lav
Employee Salary: 10000
Employee Name : Sita
Employee Salary: 30000
Employee Name : Lucky
Employee Salary: 50000
vibhuti@kmradi-Inspiron-3541:~/Desktop$