PYTHON PROGRAMMING

- **What is Python?**
- Python is an example of a high-level language; other high-level languages you might have heard of are C++, PHP, and Java.

- Multi-purpose (Web, GUI, Scripting, etc.)

- • Object Oriented

- • Interpreted

- • Strongly typed and Dynamically typed

- • Focus on readability and productivity

- ***History of Python:***
- Python was developed by Guido van Rossum in the late eighties and early nineties at the National Research Institute for Mathematics and Computer Science in the Netherland

- Python is derived from many other languages, including ABC, Modula-3, C, C++, Algol-68, SmallTalk, and Unix shell and other scripting languages.

- • Python is copyrighted. Like Perl, Python source code is now available under the GNU General Public License (GPL).

- • Python is now maintained by a core development team at the institute, although Guido van Rossum still holds a vital role in directing its progress.

- • Python 1.0 was released in November 1994. In 2000, Python 2.0 was released. Python
- 2.7.11 is the latest edition of Python 2.
- • Meanwhile, Python 3.0 was released in 2008. Python 3 is not backward compatible with Python 2. The emphasis in Python 3 had been on the removal of duplicate programming constructs and modules so that "There should be one -- and preferably only one -- obvious way to do it." Python 3.5.1 is the latest version of Python 3

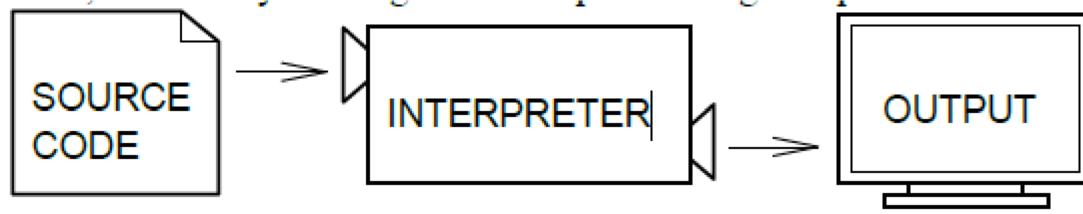
- ***Features of python:***
- Python's features include-
- • **Easy-to-learn:** Python has few keywords, simple structure, and a clearly defined syntax. This allows a student to pick up the language quickly.
- • **Easy-to-read:** Python code is more clearly defined and visible to the eyes.
- • **Easy-to-maintain:** Python's source code is fairly easy-to-maintain.
- • **A broad standard library:** Python's bulk of the library is very portable and cross platform compatible on UNIX, Windows, and Macintosh.
- • **Interactive Mode:** Python has support for an interactive mode, which allows interactive testing and debugging of snippets of code.

- • **Portable:** Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
- • **Extendable:** You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.
- • **Databases:** Python provides interfaces to all major commercial databases.
- • **GUI Programming:** Python supports GUI applications that can be created and ported
- 4 to many system calls, libraries and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.
- • **Scalable:** Python provides a better structure and support for large programs than shell scripting.

- Apart from the above-mentioned features, Python has a big list of good features. A few are listed below-
- • It supports functional and structured programming methods as well as OOP.
- • It can be used as a scripting language or can be compiled to byte-code for building large applications.
- • It provides very high-level dynamic data types and supports dynamic type checking.
- • It supports automatic garbage collection.
- • It can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java.

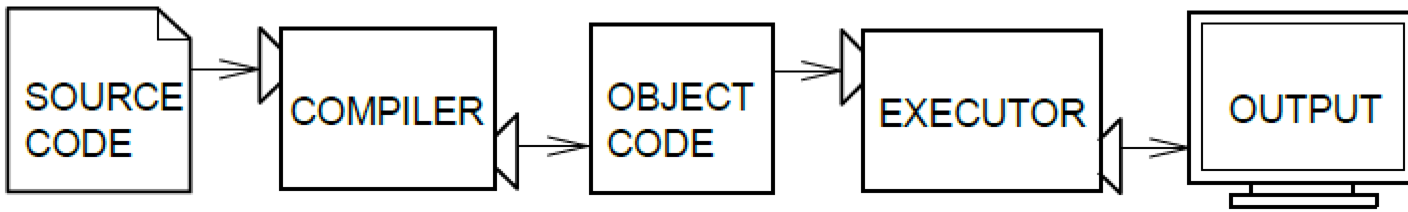
- **Interpreter VS Compiler**
- Two kinds of applications process high-level languages into low-level languages: **interpreters** and **compilers**.

- An interpreter reads a high-level program and executes it, meaning that it does what the program says. It processes the program a little at a time, alternately reading lines and performing computations.



- --

- A compiler reads the program and translates it into a low-level program, which can then be run.
- In this case, the high-level program is called the **source code**, and the translated program is called the **object code** or the **executable**. Once a program is compiled, you can execute it repeatedly without further translation.



- Many modern languages use both processes. They are first compiled into a lower
- level language, called **byte code**, and then interpreted by a program called **a virtual machine.** Python uses both processes, but because of the way programmers interact with it, it is usually considered an interpreted language
- ---

- There are two ways to use the Python interpreter: **shell mode and script mode**.
- In shell mode, you type Python statements into the **Python shell** and the interpreter immediately prints the result.

- In this course, we will be using an IDE (Integrated Development Environment) called IDLE. When you first start IDLE it will open an interpreter window.1
-
- The first few lines identify the version of Python being used as well as a few other messages; you can safely ignore the lines about the firewall. Next there is a line identifying the version of IDLE. The last line starts with >>>, which is the **Python prompt**. The interpreter uses the prompt to indicate that it is ready for instructions.
- If we type print 1 + 1 the interpreter will reply 2 and give us another prompt.2
-

```
>>> print 1 + 1
2
>>>
```

- Running Python program:
- There are three different ways to start Python-
- **(1) Interactive Interpreter**
- You can start Python from Unix, DOS, or any other system that provides you a command line interpreter or shell window.
- Enter python the command line.
- **(2) Script from the Command-line**
- **(3) Integrated Development Environment**

- ***What is Debugging ?***
- Programming is a complex process, and because it is done by human beings, programs often contain errors. For whimsical reasons, programming errors are called **bugs** and the process of tracking them down and correcting them is called **debugging**.
- Three kinds of errors can occur in a program: syntax errors, runtime errors, and semantic errors. It is useful to distinguish between them in order to track them down more quickly

---

- ***Syntax errors***
- Python can only execute a program if the program is syntactically correct; otherwise, the process fails and returns an error message. **Syntax** refers to the structure of a program and the rules about that structure. For example, in English, a sentence must begin with a capital letter and end with a period. this sentence contains a **syntax error**.
- So does this one For most readers, a few syntax errors are not a significant problem, which is why we can read the poetry of e. e. cummings without spewing error messages. Python is not so forgiving. If there is a single syntax error anywhere in your program, Python will print an error message and quit, and you will not be able to run your program.
- During the first few weeks of your programming career, you will probably spend a lot of time tracking down syntax errors. As you gain experience, though, you will make fewer syntax errors and find them faster.

- ***Runtime errors***
- The second type of error is a runtime error, so called because the error does not appear until you run the program. These errors are also called **exceptions** because they usually indicate that something exceptional (and bad) has happened. Runtime errors are rare in the simple programs you will see in the first few chapters, so it might be a while before you encounter one.

- ***Semantic errors***
- The third type of error is the **semantic error**. If there is a semantic error in your program, it will run successfully, in the sense that the computer will not generate any error messages, but it will not do the right thing. It will do something else.
- Specifically, it will do what you told it to do.
- The problem is that the program you wrote is not the program you wanted to write. The meaning of the program (its semantics) is wrong. Identifying semantic errors can be tricky because it requires you to work backward by looking at the output of the program and trying to figure out what it is doing.

- ***Formal and Natural Languages:***
- **Natural languages** are the languages that people speak, such as English, Spanish, and French. They were not designed by people (although people try to impose some order on them); they evolved naturally.
- **Formal languages** are languages that are designed by people for specific applications. For example, the notation that mathematicians use is a formal language that is particularly good at denoting relationships among numbers and symbols.

- ***Programming languages are formal languages that have been designed to express computations***.

- Although formal and natural languages have many features in common—tokens, structure, syntax, and semantics—there are many differences:
- **ambiguity:** Natural languages are full of ambiguity, which people deal with by using contextual clues and other information. Formal languages are designed to be nearly or completely unambiguous, which means that any statement has exactly one meaning, regardless of context.
- **redundancy:** In order to make up for ambiguity and reduce misunderstandings, natural languages employ lots of redundancy. As a result, they are often verbose. Formal languages are less redundant and more concise.
- **literalness:** Natural languages are full of idiom and metaphor. If someone says, “The penny dropped”, there is probably no penny and nothing dropped. Formal languages mean exactly what they say.

- People who grow up speaking a natural language—everyone—often have a hard time adjusting to formal languages. In some ways, the difference between formal and natural language is like the difference between poetry and prose, but more so:
- **Poetry:** Words are used for their sounds as well as for their meaning, and the whole poem together creates an effect or emotional response. Ambiguity is not only common but often deliberate.
- **Prose:** The literal meaning of words is more important, and the structure contributes
- 12 more meaning. Prose is more amenable to analysis than poetry but still often ambiguous.
- **Programs:** The meaning of a computer program is unambiguous and literal, and can be understood entirely by analysis of the tokens and structure.

- ***The Difference Between Brackets, Braces, and Parentheses:***

- Braces are used for different purposes. If you just want a list to contain some elements and organize them by index numbers (starting from 0), just use the [] and add elements as necessary. {} are special in that you can give custom id's to values like a = {"John": 14}. Now, instead of making a list with ages and remembering whose age is where, you can just access John's age by a["John"].
- ***The [] is called a list and {} is called a dictionary (in Python).***
- Dictionaries are basically a convenient form of list which allow you to access data in a much easier way.
- However, there is a catch to dictionaries. Many times, the data that you put in the dictionary doesn't stay in the same order as before. Hence, when you go through each value one by one, it won't be in the order you expect. There is a special dictionary to get around this, but you have to add this line from collections import OrderedDict and replace {} with OrderedDict(). But, I don't think you will need to worry about that for now.

- **Variables and Expressions:**
- ***Values and Types:***
- A **value** is one of the fundamental things—like a letter or a number—that a program manipulates. The values we have seen so far are 2 (the result when we added 1 + 1), and “Hello, World!”. These values belong to different **types**: 2 is an **integer**, and “Hello, World!” is a **string**. You (and the interpreter) can identify strings because they are enclosed in quotation marks.
- The print statement also works for integers.

```
>>> print(4)
4
```

If you are not sure what type a value has, the interpreter can tell you.

```
>>> type("Hello, World!")
```

- ***Variables:***
- One of the most powerful features of a programming language is the ability to manipulate **variables**. A variable is a name that refers to a value. The **assignment statement** creates
- new variables and assigns them values:

```
>>> message = "What's your name?"
>>> n = 17
>>> pi = 3.14159
```

Programmers generally choose names for their variables that are meaningful—they document what the variable is used for. **Variable names** can be arbitrarily long. They can contain both letters and numbers, but they have to begin with a letter. Although it is legal to use uppercase letters, by convention we don't. If you do, remember that case matters. Bruce and bruce are different variables. The underscore character ( ) can appear in a name. It is often used in names with multiple words, such as myname or priceofteainchina. If you give a variable an illegal name, you get a syntax error:

```
>>> 76trombones = "big parade"
SyntaxError: invalid syntax
>>> more$ = 1000000
SyntaxError: invalid syntax
>>> class = "COMP150"
SyntaxError: invalid syntax
```

Keywords define the language’s rules and structure, and they cannot be used as variable names. Python has thirty-one keywords:

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

- ***Type conversion:***
- Sometimes, 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.

Function	Description
int(x [,base])	Converts x to an integer. The base specifies the base if x is a string.
float(x)	Converts x to a floating-point number.
complex(real [,imag])	Creates a complex number.
str(x)	Converts object x to a string representation.
repr(x)	Converts object x to an expression string.
eval(str)	Evaluates a string and returns an object.
tuple(s)	Converts s to a tuple.

list(s)	Converts s to a list.
set(s)	Converts s to a set.
dict(d)	Creates a dictionary. d must be a sequence of (key,value) tuples.
frozenset(s)	Converts s to a frozen set.
chr(x)	Converts an integer to a character.
unichr(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.

- ***Operators and Operands:***
- **Operators** are special symbols that represent computations like addition and multiplication. The values the operator uses are called **operands**. The following are all legal Python expressions whose meaning is more or less clear:
- A + B   here A and B are operands and + is operator
- --
- **Expressions:**
- An expression is a combination of values, variables, and operators. If you type an expression on the command line, the interpreter evaluates it and displays the result:

```
>>>1+1
2
```

The *evaluation of an expression* produces a value, which is why expressions can appear on the right hand side of assignment statements. A value all by itself is a simple expression, and so is a variable.

- ***Interactive Mode and ScriptMode:***
- **Interactive Mode Programming**
- Invoking the interpreter without passing a script file as a parameter brings up the following prompt-

```
$ python
Python 3.3.2 (default, Dec 10 2013, 11:35:01)
[GCC 4.6.3] on Linux
Type "help", "copyright", "credits", or "license" for more information. >>>
On Windows:
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
```

Type the following text at the Python prompt and press Enter-

```
>>> print ("Hello, Python!")
```

If you are running the older version of Python (Python 2.x), use of parenthesis as **inprint** function is optional. This produces the following result-

```
Hello, Python!
```

- **Script Mode Programming**
- Invoking the interpreter with a script parameter begins execution of the script and continues until the script is finished. When the script is finished, the interpreter is no longer active.
- Let us write a simple Python program in a script. Python files have the extension.py. Type the following source code in a test.py file-

```
print ("Hello, Python!")
```

We assume that you have the Python interpreter set in **PATH** variable. Now, try to run this program as follows-

On Linux

```
$ python test.py
```

This produces the following result-

```
Hello, Python!
```

On Windows

```
C:\Python34>Python test.py
```

This produces the following result-

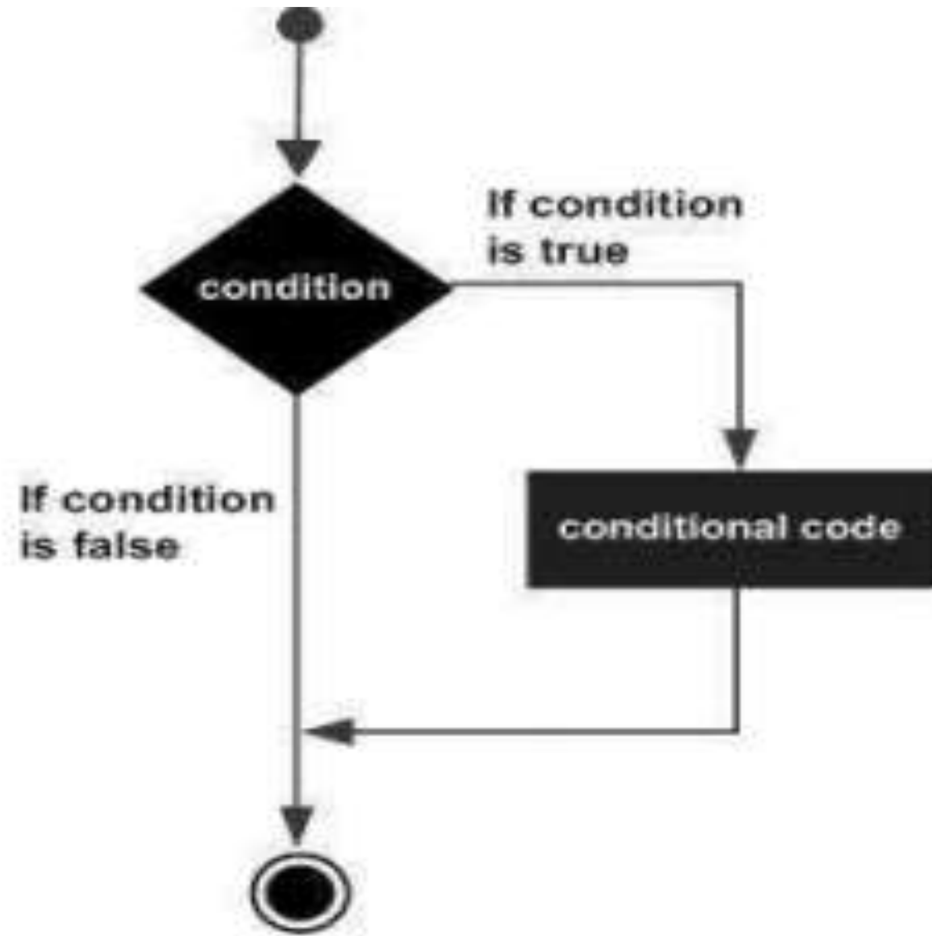
Hello, Python!

- **Order of Operations:**
- When more than one operator appears in an expression, the order of evaluation depends on the **rules of precedence**. Python follows the same precedence rules for its mathematical operators that mathematics does. The acronym **PEMDAS** is a useful way to remember the order of operations:
- 1. **P**arentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, 2*(3-1) is 4, and (1+1)**(5-2) is 8. You can also use parentheses to make an expression easier to read, as in (minute*100)/60, even though it doesn't change the result.
- 2. **E**xponentiation has the next highest precedence, so 2**1+1 is 3 and not 4, and 3*1**3 is 3 and not 27.
- 3. **M**ultiplication and **D**ivision have the same precedence, which is higher than **A**ddition and **S**ubtraction, which also have the same precedence. So 2*3-1 yields 5 rather than 4, and 2/3-1 is -1, not 1 (remember that in integer division, 2/3=0).
- Operators with the same precedence are evaluated from left to right. So in the expression minute*100/60, the multiplication happens first, yielding 5900/60, which in turn yields 98. If the operations had been evaluated from right to left, the result would have been 59*1, which is 59, which is wrong. Similarly, in evaluating 17-4-3,
- 22
- 17-4 is evaluated first.
- If in doubt, use parentheses.

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

- **Syntax:**
- **if expression: statement(s)**

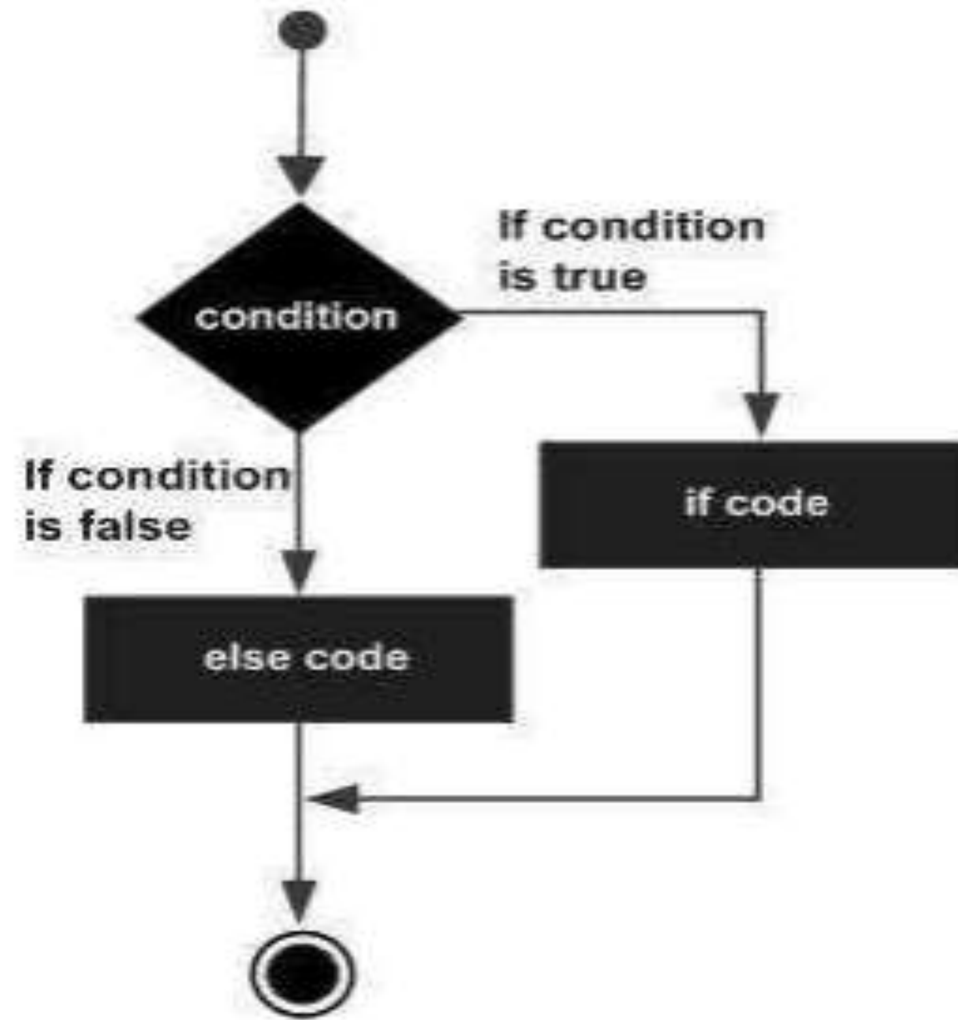
- If the boolean expression evaluates to TRUE, then the block of statement(s) inside the if statement is executed. In Python, statements in a block are uniformly indented after the : symbol. If boolean expression evaluates to FALSE, then the first set of code after the end of block is executed.
- ***nested if statements You can use one if or else if statement inside another if or else if statement(s).***


condition
If condition
is true
conditional code
If condition
is false

Example

```
num = 3

if num > 0:

print(num, "is a positive number.")
print("This is always printed.")
        num = -1 if

        num > 0:


        print(num, "is

a positive number.")


        print("This is

also always

printed.")
```

- **Guess the output?**
- 3 is a positive number. This is always printed.
- **--**
- **IF ELSE Statements:**
- An **else** statement can be combined with an **if** statement.
- An **else** statement contains a block of code that executes **if** the conditional expression in the **if** statement resolves to 0 or a FALSE value.
- The **else** statement is an optional statement and there could be at the most only one else statement following **if**.
- Syntax
- *The syntax of the if...else statement is*
- **if expression:**
- **statement(s)**
- **else: statement(s)**


condition
If condition
is true
If condition
is false
if code
else code

Example

```
# Program checks if the number is positive or negative
# And displays an appropriate message

num = 3

# Try these two variations as well.
# num = -5
# num = 0

if num >= 0:
   print("Positive or Zero")
else:
   print("Negative number")
```

```
Positive or Zero
```

In the above example, when num is equal to 3, the test expression is true and body of if is executed and body of else is skipped.

If num is equal to -5, the test expression is false and body of else is executed and body

- **Nested IF -ELSE Statements:**
- There may be a situation when you want to check for another condition after a condition resolves to true. In such a situation, you can use the nested if construct. In a nested if construct, you can have an if...elif...else construct inside another if...elif...else construct.
- **Syntax** The syntax of the nested if...elif...else construct may be
- **if expression1:**
- **statement(s)**
- **if expression2:**
- **statement(s)**
- **elif expression3:**
- **statement(s)**
- **else:**
- **statement(s)**
- **elif expression4:**
- **statement(s)**
- **else: statement(s)**

Example

```
# !/usr/bin/python3
# In this program, we input a number
# check if the number is positive or
# negative or zero and display
# an appropriate message
# This time we use nested if

num = float(input("Enter a number: "))
if num >= 0:
    if num == 0:
        print("Zero")
    else:
        print("Positive number")
else:
    print("Negative number")
num=int(input("enter number"))
```

- Output 1
- Enter a number: 5
- Positive number
- Output 2
- Enter a number: -1
- Negative number
- Output 3
- Enter a number: 0
- Zero Divisible by 3 and 2
- enter number5
- not Divisible by 2 not divisible by 3

- **Looping:**
- **For:**
- ***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.***

- Syntax of for Loop:

- **for val in sequence:**
- **Body of for**

- ***Note :***Here, val 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.

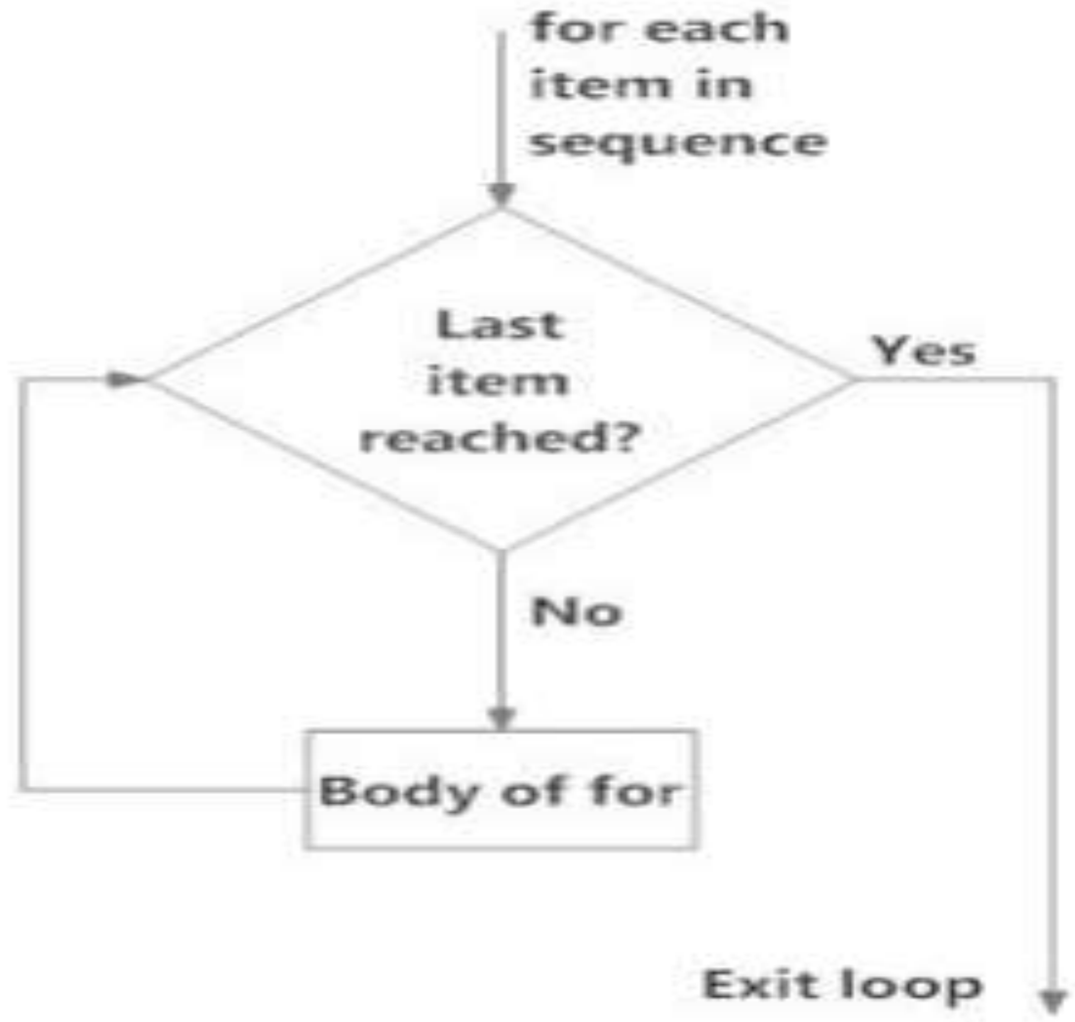


Fig: operation of for loop

- **Example: Python for Loop**

- **# Program to find the sum of all numbers stored in a list**
- **# List of numbers**
- **numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]**
- **# variable to store the sum**
- **sum = 0**
- **# iterate over the list**
- **sum = sum+val**
- **# Output: The sum is 48**
- **print("The sum is", sum)**
- **for val in numbers:**

- *when you run the program, the output will be: The sum is 48*

- **2. While Loop**
- 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.
- ---
- **Syntax of while Loop in Python**
- **while test_expression:**
- **Body of while**
- **---**
- **Note:** 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.

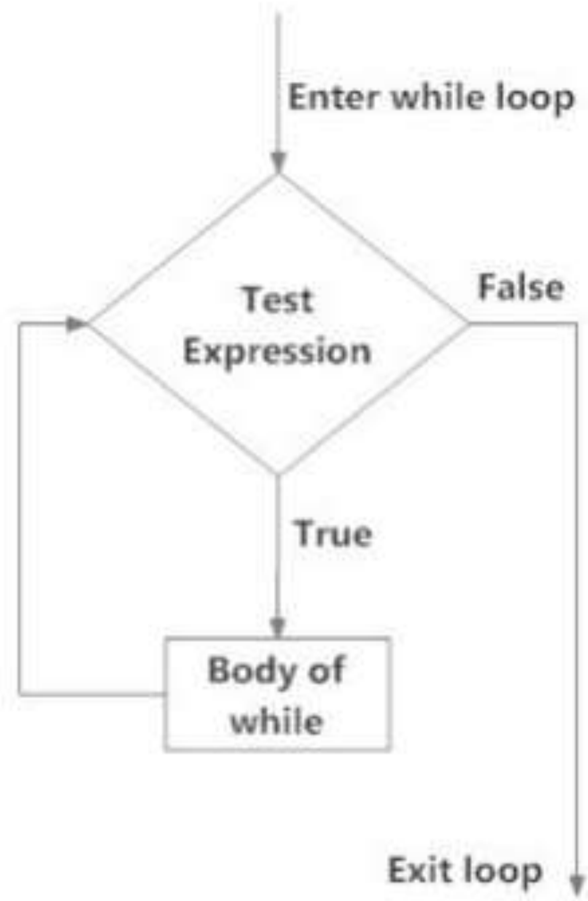


Fig: operation of while loop

- **Example: Python while Loop**
- **# Program to add natural**
- **# numbers upto**
- **# sum = 1+2+3+...+n**
- **#To take input from the user,**
- **n = int(input("Enter n: ")) n = 10**
- **# initialize sum and counter**
- **sum = 0**
- **i = 1**
- **while i <= n:**
- **sum = sum + I**
- **i = i+1**
- **# update counter**
- **# print the sum**
- **print("The sum is", sum)** **output:** Enter n: 10
- The sum is 55

- **3. Nested loops:**
- Python programming language allows to use one loop inside another loop. Following section shows few examples to illustrate the concept.
- Python Nested if Example
- # In this program, we input a number
- # check if the number is positive or
- # negative or zero and display
- # an appropriate message
- # This time we use nested if

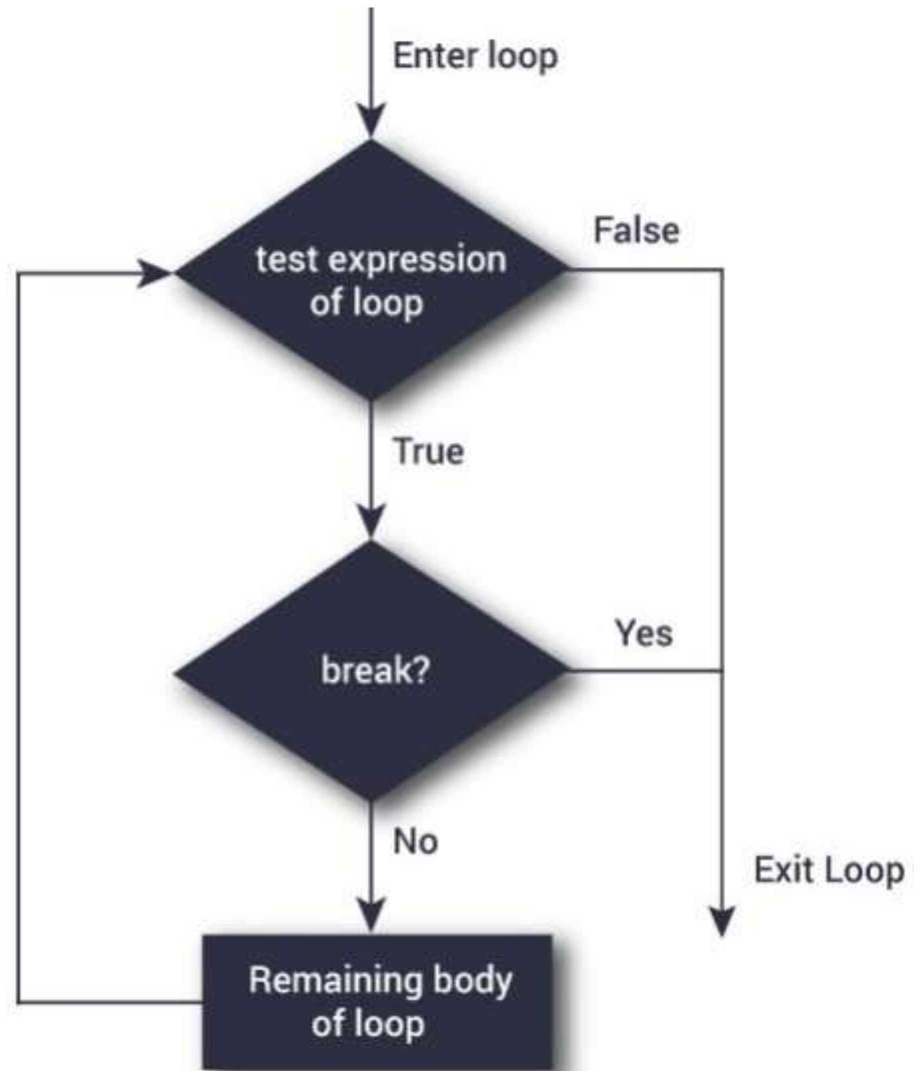
- num = float(input("Enter a number: "))
- if num >= 0:
- if num == 0:
- print("Zero")
- else:
- print("Positive number")
- else:
- print("Negative number")

- **Special note: while using nested if else statement make sure inner block has proper indentation prior to outer block .**

- **Control statements:**
- **1. Terminating loops:**
- The break statement terminates the loop containing it. Control of the program flows to the statement immediately after the body of the loop.

- If break statement is inside a nested loop (loop inside another loop), break will terminate the innermost loop.

- Syntax of break
- **break**


Enter loop
test expression
of loop
False
True
break?
Yes
No
Exit Loop
Remaining body
of loop

```
for var in sequence:
    # codes inside for loop
    if  condition:
        break
    # codes inside for loop

# codes outside for loop
```

```
while test expression:
    # codes inside while loop
    if  condition:
        break
    # codes inside while loop

# codes outside while loop
```

Example: Python break

```
# Use of break statement inside loop

for val in "string":
    if val == "i":
        break
    print(val)

print("The end")
```

Output

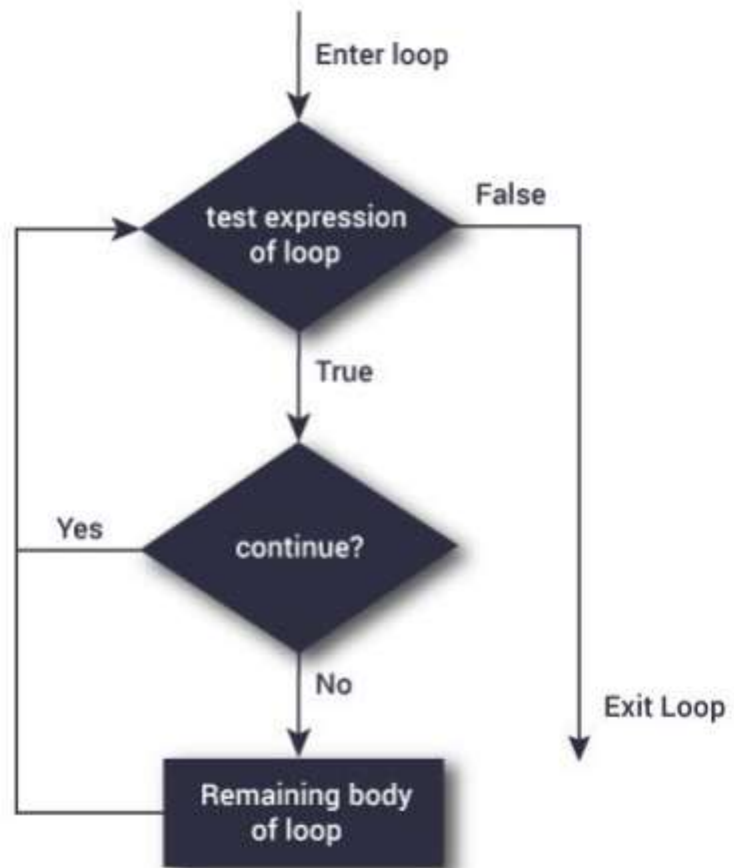
```
s

t

r

The end
```

- **2. Skipping specific conditions:**
- The continue statement is used to skip the rest of the code inside a loop for the current iteration only. Loop does not terminate but continues on with the next iteration.
- **Syntax of Continue**
- **continue**
-
- **Example: # Program to show the use of continue statement inside loops**
- ***for val in "string":***
- ***if val == "i":***
- ***continue***
- ***print(val) print("The end")***
- Output :
- s
- t
- r
- n
- g
- The end

- This program is same as the above example except the break statement has been replaced with continue. We continue with the loop, if the string is "i", not executing the rest of the block. Hence, we see in our output that all the letters except "i" gets printed.

- **NOTE: while performing practicals keep proper indentation in between program**

For better experience use editiors like jyupiter notebook or anaconda navigator .


Enter loop
test expression
of loop
False
True
Yes
continue?
No
Exit Loop
Remaining body
of loop

```
for var in sequence:
    # codes inside for loop
    if  condition:
        continue
    # codes inside for loop

# codes outside for loop
```

```
while test expression:
    # codes inside while loop
    if  condition:
        continue
    # codes  inside while loop

# codes outside while loop
```

- Thank you
- Visit https://www.profajaypashankar.com
- for more study material and notes .
- Visit https://www.youtube.com/user/ajaypashankar7 for more lectures .

PYTHON PROGRAMMING

Functions

- In python we have two kinds of functions .
- First is built –in functions .
- User defined functions (with def keyword).
- This functions (built-in functions) can also be categories into two types i.e.

1. Fruitful Function(returns value)
2. Void Function (doesn’t return any value)

- Special Note :
- You can some use built-in functions in python without any import .(for e.g . abs)
- You can use that on command prompt directly also.
- For some functions you need to import that function based module for e.g suppose you want to calculate ceiling value of a number using ceil function you need to import math module first .

```
import math # This will import math module
print ("math.ceil(-45.17) : ", math.ceil(-45.17))
```

- If you use this kind functions directly with its module importing you will get error .

- The syntax of a function call is simply
- FUNCTION NAME(ARGUMENTS)
- Not all functions take an argument, and some take more than one (in which case the arguments are separated by commas).
- The value or variable, which is called the argument of the function, has to be enclosed in parentheses.
- It is common to say that a function “takes” an argument and “returns” a result.
- The result is called the return value.

- Another useful function is len. It takes a Python sequence as an argument.

- The only Python sequence we have met so far is a string.
- A string is a sequence of characters.

- For a string argument, len returns the number of characters the string contains.

```
>>> my_str = "Hello world"
>>> len(my_str)
11
```

- Math Functions:

Function	Returns (description)
abs(x)	The absolute value of x: the (positive) distance between x and zero.
ceil(x)	The ceiling of x: the smallest integer not less than x
cmp(x, y)	-1 if x < y, 0 if x == y, or 1 if x > y
exp(x)	The exponential of x: e^x
fabs(x)	The absolute value of x.
floor(x)	The floor of x: the largest integer not greater than x

log(x)	The natural logarithm of x, for x> 0
log10(x)	The base-10 logarithm of x for x> 0.
max(x1, x2,...)	The largest of its arguments: the value closest to positive infinity
min(x1, x2,...)	The smallest of its arguments: the value closest to negative infinity
modf(x)	The fractional and integer parts of x in a two-item tuple. Both parts have the same sign as x. The integer part is returned as a float.
pow(x, y)	The value of x**y.
round(x [,n])	x rounded to n digits from the decimal point. Python rounds away from zero as a tie-breaker: round(0.5) is 1.0 and round(-0.5) is -1.0.

- **abs() Method :**
- **Description**: The abs() method returns the absolute value of x i.e. the positive distance between x and zero.
- **Syntax** : Following is the syntax for abs() method-
- abs(x)
- **Parameters :**
- x - This is a numeric expression.
- **Return** : This method returns the absolute value of x.
- The following example shows the usage of the abs() method.
- abs(-45): 45
- abs(100.12) : 100.12

- **ceil() Method :**
- **Description**: The ceil() method returns the ceiling value of x i.e. the smallest integer not less than x.
- **Syntax**: Following is the syntax for the ceil() method
- import math
- math.ceil(x)
- **Parameters**
- x - This is a numeric expression.
- Return Value
- This method returns the smallest integer not less than x.
- *Note: This function is not accessible directly, so we need to import math module and then we need to call this function using the math static object.*

- **exp() Method**
- Description
- The exp() method returns exponential of x: ex.
- Syntax
- Following is the syntax for the exp() method
- import math
- math.exp(x)
- Parameters
- X - This is a numeric expression.
- Return Value
- This method returns exponential of x: ex.
- ***Note: This function is not accessible directly. Therefore, we need to import the math module and then we need to call this function using the math static object.***

- **fabs() Method**
- Description
- The fabs() method returns the absolute value of x. Although similar to the abs()
- function, there are differences between the two functions. They are-
- • abs() is a built in function whereas fabs() is defined in math module.
- • fabs() function works only on float and integer whereas abs() works with complex number also.
- **Syntax**
- Following is the syntax for the fabs() method
- import math
- math.fabs(x)
- ***Note: This function is not accessible directly, so we need to import the math module And then we need to call this function using the math static object.***

- **floor() Method**
- **Description**
- The floor() method returns the floor of x i.e. the largest integer not greater than x.
- **Syntax**
- Following is the syntax for the floor() method
- import math
- math.floor(x)
- **Parameters**
- x - This is a numeric expression.
- **Return Value**
- This method returns the largest integer not greater than x.
- The following example shows the usage of the floor() method.
- ***Note: This function is not accessible directly, so we need to import the math module and then we need to call this function using the math static object.***

- **log() Method**
- **Description**
- The log() method returns the natural logarithm of x, for x > 0.
- **Syntax**
- Following is the syntax for the log() method
- import math
- math.log(x)
- **Parameter:**
- x - This is a numeric expression.
- **Return Value**
- This method returns natural logarithm of x, for x > 0.
- ***Note: This function is not accessible directly, so we need to import the math module and then we need to call this function using the math static object.***

- **log 10() Method**
- **Description**
- The log10() method returns base-10 logarithm of x for x > 0.
- **Syntax**
- Following is the syntax for log10() method
- import math
- math.log10(x)
- **Parameters**
- x - This is a numeric expression.
- **Return Value**
- This method returns the base-10 logarithm of x for x > 0.
- ***Note: This function is not accessible directly, so we need to import the math module and then we need to call this function using the math static object.***

- **max() Method**
- **Description**
- The max() method returns the largest of its arguments i.e. the value closest to positive infinity.
- **Syntax**
- Following is the syntax for max() method
- max(x, y, z, ...)
- **Parameters**
- • x - This is a numeric expression.
- • y - This is also a numeric expression.
- • z - This is also a numeric expression.
- **Return Value**
- This method returns the largest of its arguments.

- **min() Method**
- **Description**
- The method min() returns the smallest of its arguments i.e. the value closest to negative infinity.
- **Syntax**
- Following is the syntax for the min() method
- min(x, y, z, ...)
- **Parameters**
- • x - This is a numeric expression.
- • y - This is also a numeric expression.
- • z - This is also a numeric expression.
- **Return Value**
- This method returns the smallest of its arguments.

- **modf() Method**
- **Description**
- The modf() method returns the fractional and integer parts of x in a two-item tuple.
- Both parts have the same sign as x. The integer part is returned as a float.
- **Syntax**
- Following is the syntax for the modf() method
- import math
- math.modf(x)
- **Parameters**
- x - This is a numeric expression.
- **Return Value**
- This method returns the fractional and integer parts of x in a two-item tuple. Both the parts have the same sign as x. The integer part is returned as a float.

- **pow() Method**
- Return Value
- This method returns the value of xy.
- Example
- The following example shows the usage of the pow() method.
- import math # This will import math module
- print ("math.pow(100, 2) : ", math.pow(100, 2))
- print ("math.pow(100, -2) : ", math.pow(100, -2))
- print ("math.pow(2, 4) : ", math.pow(2, 4))
- print ("math.pow(3, 0) : ", math.pow(3, 0))
- ***NOTE: make sure you pass two arguments to pow ()***
- ***Otherwise it will raise an exception .***

Python 3.8.5 Shell

File Edit Shell Debug Options Window Help

```
Python 3.8.5 (tags/v3.8.5:580fbb0, Jul 20 2020, 15:43:08) [MSC v.1926 32 bit (In
tel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
=== RESTART: C:/Users/admin/AppData/Local/Programs/Python/Python38-32/ghh.py ===
Traceback (most recent call last):
  File "C:/Users/admin/AppData/Local/Programs/Python/Python38-32/ghh.py", line 2
, in <module>
    print ("math.pow(100, 2) : ", math.pow(100))
TypeError: pow expected 2 arguments, got 1
>>>
```

Activate Windows
Go to Settings to activate Windows.

Show all

ENG 9:51 AM

- **round() Method**
- **Description**
- round() is a built-in function in Python. It returns x rounded to n digits from the decimal point.
- **Syntax**
- Following is the syntax for the round() method
- round(x [, n])
- **Parameters**
- • x - This is a numeric expression.
- • n - Represents number of digits from decimal point up to which x is to be rounded.
- Default is 0.
- **Return Value**
- This method returns x rounded to n digits from the decimal point.

- **sqrt() Method**
- The sqrt() method returns the square root of x for x > 0.
- **Syntax**
- Following is the syntax for sqrt() method
- import math
- math.sqrt(x)
- **Parameters**
- x - This is a numeric expression.
- **Return Value**
- This method returns square root of x for x > 0.
- ***Note: This function is not accessible directly, so we need to import the math module and then we need to call this function using the math static object.***

- **Adding New Functions**
- A new function can be created in python using keyword def followed by the function name and arguments in parathesis and statements to be executed in function
- Example:
- def requiredArg (str,num):
- Statements
- **Function definitions and use**
- As well as the built-in functions provided by Python you can define your own functions.
- In the context of programming, a function is a named sequence of statements that performs a desired operation. This operation is specified in a function definition. In Python, the syntax for a function definition is:

- def NAME(LIST OF PARAMETERS):
- STATEMENTS

- There can be any number of statements inside the function, but they have to be you.
- indented from the def. In the examples in this book, we will use the standard
- indentation of four spaces3. IDLE automatically indents compound statements for
- Function definitions are the first of several compound statements we will see, all
- of which have the same pattern:

- 1. A header, which begins with a keyword and ends with a colon.
- 2. A body consisting of one or more Python statements, each indented the same
- amount – 4 spaces is the Python standard – from the header.
- In a function definition, the keyword in the header is def, which is followed by the
- list name of the function and a list of parameters enclosed in parentheses. The
- parameter may be empty, or it may contain any number of parameters. In either
- case, the parentheses are required. The first couple of functions we are going to no
- write have parameters, so the syntax looks like this:

- This function is named new_line. The empty parentheses indicate that it has no which parameters (that is it takes no arguments). Its body contains only a single statement, outputs a newline character. (That's what happens when you use a print command without any arguments.)
- Defining a new function does not make the function run. To do that we need a by a function call. Function calls contain the name of the function to be executed
- followed list of values, called arguments, which are assigned to the parameters in the function definition. Our first examples have an empty parameter list, so the do function calls not take any arguments. Notice, however, that the parentheses are required in the function call

- Flow of Execution:
- In order to ensure that a function is defined before its first use, you have to know the order in which statements are executed, which is called the flow of execution.
- Execution always begins at the first statement of the program. Statements are executed 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 are not executed until the function is called.
-
- Although it is not common, you can define one function inside another. In this case, the inner definition isn't executed until the outer function is called.

- **Parameters and Arguments:**
- Most functions require arguments.
- **Arguments** are values that are input to the function and these contain the data that the function works on.
- Some functions take more than one argument.
- Inside the function, the values that are passed get assigned to variables called **parameters.**

- Fruitful functions and Void functions:
- **The return statement :**
- The return statement allows you to terminate the execution of a function before you reach the end. One reason to use it is if you detect an error condition:

```
def print_square_root(x):
    if x < 0:
        print("Warning: cannot take square root of a negative number.")
    return
    result = x**0.5
    print("The square root of x is", result)
```

The function print square root has a parameter named x. The first thing it does is check whether x is less than 0, in which case it displays an error message and then uses return to exit the function. The flow of execution immediately returns to the caller, and the remaining lines of the function are not executed.

- **Fruitful Functions :**
- The functions that returns some value is known as fruitful functions.

- **Void Functions :**
- The functions that don't return any value is known as Void Functions.

- you can call one function from within another. This ability is called **composition**

- **Boolean functions:**
- Functions can return boolean values, which is often convenient for hiding complicated tests inside functions. For example:

```
>>> def is_divisible(x, y):
        if x % y == 0:
            return True
        else:
            return False
```

- **Void Functions:**
- Void functions are functions, like ‘print_twice’ (that we defined earlier), that perform an action (either display something on the screen or perform some other action). However, they do not return a value.

- **Importing with from:**
- We can use functions in modules in three different ways:
- **Import a module object in Python:**
- If you import math, you get a module object named math. The module object contains constants like pi and functions like sin and exp.

```
>>> import math

>>> print(math)

<module 'math' (built-in)>

>>> print(math.pi)

3.141592653589793
```

- Import an object **from** a module in Python:

```
>>> print(math.pi)
3.141592653589793
```

Now you can access pi directly, without dot notation.

```
>>> print(pi)


3.141592653589793
```

- Import all objects from a module in Python :
- >>> from math import*
- The advantage of importing everything from the math module is that your code can be more concise.
-
- The disadvantage is that there might be conflicts between names defined in different modules, or between a name from a module and one of your variables.

- **Recursion:**
- Recursion is a way of programming or coding a problem, in which a function calls itself one or more times in its body. Usually, it is returning the return value of this function call. If a function definition fulfils the condition of recursion, we call this function a recursive function.
- **Termination condition:** A recursive function has to terminate to be used in a program.
- A recursive function terminates, if with every recursive call the solution of the problem is downsized and moves towards a base case.
- A base case is a case, where the problem can be solved without further recursion.
- A recursion can lead to an infinite loop, if the base case is not met in the calls.

- **Recursion functions in Python:**
- Now we come to implement the factorial in Python. It's as easy and elegant as the mathematical definition.

```
def factorial(n):
    print("factorial has been called with n = " + str(n))
    if n == 1:
        return 1
    else:
        res = n * factorial(n-1)
    print("intermediate result for ", n, " * factorial(" ,n-1, "): ",res)
    return res
```

```
>>> print(factorial(5))
```

- **STRINGS:**
- A String Is A sequence of Characters Strings in Python are identified as a contiguous set of characters represented in the quotation marks.

- Python allows either pair 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 to the end.

- The plus (+) sign is the string concatenation operator and the asterisk (*) is repetition operator

```
str = 'Hello World!'
print (str) # Prints complete string
print (str[0]) # Prints first character of the string
print (str[2:5]) # Prints characters starting from 3rd to 5th
print (str[2:]) # Prints string starting from 3rd character
print (str * 2) # Prints string two times
print (str + "TEST") # Prints concatenated string
```

This will produce the following result

```
Hello World!
 H
 Llo
 llo World!
 Hello World!Hello World!
 Hello World!TEST
```

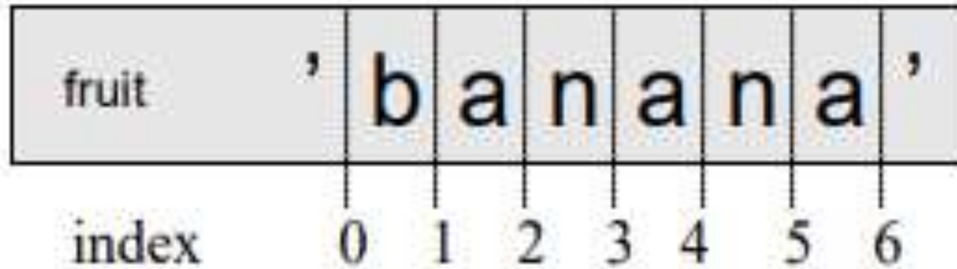
- **Traversal as a For Loop:**
- A lot of computations involve processing a string one character at a time.
- Often they start at the beginning, select each character in turn, do something to it, and continue until the end.
- This pattern of processing is called a traversal.
- Python provides a very useful language feature for traversing many compound types— the for loop:

```
>>> fruit ='banana'
>>> for char in fruit:
        print(char)
```

- The above piece of code can be understood as an abbreviated version of an English sentence: “For each character in the string fruit, print out the character”. The for loop is an example of an iterator: something that visits or selects every element in a structure (in this case a string), usually in turn. The for loop also works on other compound types such as lists and tuples, which we will look at later.

- prefixes = "JKLMNOPQ“
- suffix = "ack“
- for letter in prefixes:
- print letter + suffix
- The output of this program is:
- Jack
- Kack
- Lack
- Mack
- Nack
- Nack
- Oack
- Pack
- Qack

- **String Slices:**
- A substring of a string is called a slice.
- Selecting a slice is similar to selecting a character:
- >>> s = "Peter, Paul, and Mary"
- >>> print(s[0:5])
- Peter
- >>> print(s[7:11])
- Paul
- >>> print(s[17:21])
- Mary
- *Note :The operator [n:m] returns the part of the string from the nth character to the mth character, including the first but excluding the last.*



If you omit the first index (before the colon), the slice starts at the beginning of the string. If you omit the second index, the slice goes to the end of the string. Thus:

```
>>> fruit= "banana"
>>> fruit[0:3]
'ban'
>>> fruit[3:]
'ana'
```

- ***Strings Are Immutable:***
- Strings are immutable, which means you can’t change an existing string.
- The best you can do is create a new string that is a variation on the original:
- >>> greeting = "Hello, world!"
- >>> newGreeting = "J" + greeting[1:]
- >>> print(newGreeting)
- Jello, world!

- ***The solution here is to concatenate a new first letter onto a slice of greeting.***
- ***This operation has no effect on the original string.***

- ***Searching within a string:***
- It determines if string str occurs in string, or in a substring of string if starting **index beg** and ending **index end** are given.

Syntax

```
str.find(str, beg=0, end=len(string))
```

Parameters

- **str** -- This specifies the string to be searched.
- **beg** -- This is the starting index, by default its 0.
- **end** -- This is the ending index, by default its equal to the length of the string.

Return Value

Index if found and -1 otherwise.

- >>> str1 = "this is string example....wow!!!"
- >>> str2 = "exam"
- >>> print(str1.find(str2))
- 15
- >>> print(str1.find(str2, 10))
- 15
- >>> print(str1.find(str2, 40))
- -1

- ***Note : if the required sub string is not found in available string i.e. no index found so it will always return -1.***

- **String Methods:**
- In addition to the functions that we have seen so far there is also a special type of function called a method.
- You can think of a method as a function which is attached to a certain type of variable (e.g. a string).
- When calling a function you just need the name of the function followed by parentheses (possibly with some arguments inside).
- VARIABLE.METHODNAME(ARGUMENTS)

- **The in operator:**
- The in operator tests if one string is a substring of another:

```
>>> "p" in "apple"
True
>>> "i" in "apple"
False
>>> "ap" in "apple"
True
>>> "pa" in "apple"
False
```

Note that a string is a substring of itself:

```
>>> "a" in "a"
```

```
True
>>> "apple" in "apple"
True
```

- **String Comparison:**

The comparison operators work on strings. To see if two strings are equal:

```
>>> if word < "banana":
      print("Your word," + word + ", comes before banana.")
elif word > "banana":
      print("Your word," + word + ", comes after banana.")
else:
      print("Yes, we have no bananas!")
```

You should be aware, though, that Python does not handle upper- and lowercase letters the same way that people do. All the uppercase letters come before all the lowercase letters. As a result:

```
Your word,zebra, comes after banana.
```

A common way to address this problem is to convert strings to a standard format, such as all lowercase, before performing the comparison. A more difficult problem is making the program realize that zebras are not fruit.

- **String Operations:**

S. No.	Methods with Description
1	**capitalize()** Capitalizes first letter of string
2	**center(width, fillchar)** Returns a string padded with ***fillchar*** with the original string centered to a total of ***width*** columns.
3	**count(str, beg= 0,end=len(string))** Counts how many times str occurs in string or in a substring of string if starting index beg and ending index end aregiven.
4	**decode(encoding='UTF-8',errors='strict')** Decodes the string using the codec registered for encoding. encoding defaults to the default string encoding.
5	**encode(encoding='UTF-8',errors='strict')** Returns encoded string version of string; on error, default is to raise a ValueError unless errors is given with 'ignore' or 'replace'.

6	**endswith(suffix, beg=0, end=len(string))** Determines if string or a substring of string (if starting index beg and ending index end are given) ends with suffix; returns true if so and false otherwise.
7	**expandtabs(tabsize=8)** Expands tabs in string to multiple spaces; defaults to 8 spaces per tab if tabsize not provided.
8	**find(str, beg=0 end=len(string))** Determine if str occurs in string or in a substring of string if starting index beg and ending index end are given returns index if found and -1 otherwise.
9	**index(str, beg=0, end=len(string))** Same as find(), but raises an exception if str not found.
10	**isalnum()** Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise.

11	**isalpha()** Returns true if string has at least 1 character and all characters are alphabetic and false otherwise.
12	**isdigit()** Returns true if the string contains only digits and false otherwise.
13	**islower()** Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise.
14	**isnumeric()** Returns true if a unicode string contains only numeric characters and false otherwise.
15	**isspace()** Returns true if string contains only whitespace characters and false otherwise.

16	**istitle()** Returns true if string is properly "titlecased" and false otherwise.
17	**isupper()** Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise.
18	**join(seq)** Merges (concatenates) the string representations of elements in sequence seq into a string, with separator string.
19	**len(string)** Returns the length of the string

20	**ljust(width[, fillchar])** Returns a space-padded string with the original string left-justified to a total of width columns.
21	**lower()** Converts all uppercase letters in string to lowercase.
22	**lstrip()** Removes all leading whitespace in string.
23	**maketrans()** Returns a translation table to be used in translate function.
24	**max(str)** Returns the max alphabetical character from the string str.
25	**min(str)** Returns the min alphabetical character from the string str.

26	**replace(old, new [, max])** Replaces all occurrences of old in string with new or at most max occurrences if max given.
27	**rfind(str, beg=0,end=len(string))** Same as find(), but search backwards in string.
28	**rindex(str, beg=0, end=len(string))** Same as index(), but search backwards in string.
29	**rjust(width,[, fillchar])** Returns a space-padded string with the original string right-justified to a total of width columns.
30	**rstrip()** Removes all trailing whitespace of string.
31	**split(str="", num=string.count(str))** Splits string according to delimiter str (space if not provided) and returns list of substrings; split into at most num substrings if given.

32	**splitlines(num=string.count('\n'))** Splits string at all (or num) NEWLINEs and returns a list of each line with NEWLINEs removed.
33	**startswith(str, beg=0,end=len(string))** Determines if string or a substring of string (if starting index beg and ending index end are given) starts with substring str; returns true if so and false otherwise.
34	**strip([chars])** Performs both lstrip() and rstrip() on string
35	**swapcase()** Inverts case for all letters in string.
36	**title()** Returns "titlecased" version of string, that is, all words begin with uppercase and the rest are lowercase.

37	**translate(table, deletechars="")** Translates string according to translation table str(256 chars), removing those in the del string.
38	**upper()** Converts lowercase letters in string to uppercase.
39	**zfill (width)** Returns original string leftpadded with zeros to a total of width characters; intended for numbers, zfill() retains any sign given (less one zero).
40	**isdecimal()** Returns true if a unicode string contains only decimal characters and false otherwise.

- **QUICK REVISION :**
- In python we have two kinds of functions .
- First is built –in functions .
- User defined functions (with def keyword).
- This functions (built-in functions) can also be categories into two types i.e.

1. Fruitful Function(returns value)
2. Void Function (doesn't return any value).

- A new function can be created in python using keyword **def** followed by the function name and arguments in parathesis and statements to be executed in function.

- **Arguments** are values that are input to the function and these contain the data that the function works on.
- Inside the function, the values that are passed get assigned to variables called **parameters.**
- **Fruitful Functions :**
- The functions that returns some value is known as fruitful functions.
- **Void Functions :**
- The functions that don’t return any value is known as Void Functions.
- **Strings are immutable**, which means you can’t change an existing string.
- The best you can do is create a new string that is a variation on the original:

- **VISIT**
- **https://www.profajaypashankar.com**
- **For more study material and notes .**

- **VISIT**
- https://www.youtube.com/channel/UCu4Bd22zM6RpvHWC9YHBh5Q?view_as=subscriber
- **For more lectures .**

- **VISIT : FOR PRACTICAL MANUAL**
- https://www.profajaypashankar.com/python-programming-practical-manual/
- **Password:STUDYHARD**

PYTHON PROGRAMMING

- **Lists :**
- **A list is an ordered set of values, where each value is identified by an index.**
- **The values that make up a list are called its elements .**
- **Lists are similar to strings, which are ordered sets of characters, except that the elements of a list can have any type.**
- **Lists and strings—and other things that behave like ordered sets—are called sequences .**
- **The list is the most versatile datatype available in Python, which can be written as a list of comma-separated values (items) between square brackets.**
- ***Important thing about a list is that the items in a list need not be of the same type.***
- There are several ways to create a new list; the simplest is to enclose the elements in square brackets ([and]):
- [10, 20, 30, 40] ["spam", "bungee", "swallow"]

- A list within another list is said to be nested .
- Finally, there is a special list that contains no elements.
- It is called the empty list, and is denoted []. Like numeric 0 values and the empty string, the empty list is false in a boolean expression:

- **Values and Accessing Elements:**
- **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.**

- #!/usr/bin/python3
- list = ['abcd', 786 , 2.23, 'john',
- 70.2] tinylist = [123, 'john']

- print (list)
- print (list[0]) print (list[1:3])
- # Prints complete list
- # Prints first element of the list
- # Prints elements starting from 2nd
- till 3rd print (list[2:])
- # Prints elements starting
- from 3rd element print (tinylist * 2) # Prints list two
- times
- print (list + tinylist) # Prints concatenated lists

- **Lists are mutable :**
- Unlike strings lists are **mutable** , which means we can change their elements.
- Using the bracket operator on the left side of an assignment, we can update one of the elements:
- >>> fruit = ["banana", "apple", "quince"]
- >>> fruit[0] = "pear"
- >>> fruit[-1] = "orange"
- >>> print fruit
- [’pear’, ’apple’, ’orange’]

- **Explanation:**
- The bracket operator applied to a list can appear anywhere in an expression. When it appears on the left side of an assignment, it changes one of the elements in the list, so the first element of fruit has been changed from "banana" to "pear", and the last from "quince" to "orange".
- **An assignment to an element of a list is called item assignment.**

- Item assignment does not work for strings:
- >>> my_string = "TEST"
- >>> my_string[2] = "X"
- Traceback (most recent call last): File "<stdin>", line 1, in
- <module>
- TypeError: ’str’ object does not support item assignment
-
- but it does for lists:
- >>> my_list = ["T", "E", "S", "T"]
- >>> my_list[2] = "X"
- >>> my_list
- [’T’, ’E’, ’X’, ’T’]

- **With the slice operator we can update several elements at once:**
- >>> a_list = ["a", "b", "c", "d", "e", "f"]
- >>> a_list[1:3] = ["x", "y"]
- >>> print a_list
- ['a', 'x', 'y', 'd', 'e', 'f']

- **We can also remove elements from a list by assigning the empty list to them:**
- >>> a_list = ["a", "b", "c", "d", "e", "f"]
- >>> a_list[1:3] = []
- >>>
- print a_list ['a',
- 'd',
- 'e',
- 'f']

- **And we can add elements to a list by squeezing them into an empty slice at the desired location:**
- >>> a_list = ["a", "d", "f"]
- >>> a_list[1:1] = ["b", "c"]
- >>> print a_list
- ['a', 'b', 'c', 'd', 'f']
- >>> a_list[4:4] = ["e"]
- >>> print a_list
- ['a', 'b', 'c', 'd', 'e', 'f']
- **--**

- **Deleting elements from List :**
- To remove a list element, you can use either the **del** statement if you know exactly which element(s) you are deleting.
- You can use the remove() method if you do not know exactly which items to delete.
- #!/usr/bin/python3
- list = ['physics', 'chemistry', 1997, 2000] print (list)
- del list[2]
- print ("After deleting value at index 2 : ", list)
- *When the above code is executed, it produces the following result-*
- ['physics', 'chemistry', 1997, 2000]
- After deleting value at index 2 : ['physics', 'chemistry', 2000]
- **Note:** remove() method is discussed in subsequent section.

- **Built-in List Operators, Concatenation, Repetition, In Operator :**
- Lists respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new list, not a string.

Python Expression	Results	Description
len([1, 2, 3])	3	Length
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	Concatenation
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	Repetition
3 in [1, 2, 3]	True	Membership
for x in [1,2,3] : print (x,end=' ')	1 2 3	Iteration

- **Built-in List functions and methods :**
- Python includes the following list functions :

SN	Function with Description
1	**cmp(list1, list2) :** No longer available in Python 3.
2	**len(list) :** Gives the total length of the list.
3	**max(list) :** Returns item from the list with max value.
4	**min(list) :** Returns item from the list with min value.
5	**list(seq) :** Converts a tuple into list.

- **Listlen()Method**
- ***Description***
- The **len()** method returns the number of elements in the list.
- <u>Syntax</u>
- Following is the syntax for len() method-
- len(list)
- **Parameters**
- **list** - This is a list for which, number of elements are to be counted.
- **Return Value**
- This method returns the number of elements in the list.
- ***Example***
- The following example
- list1 = ['physics', 'chemistry', 'maths']
- print (len(list1))
- list2=list(range(5)) #creates list of numbers between 0-4 print (len(list2)) shows the usage of len() method.

- **List max() Method :**
- **Description**
- The **max()** method returns the elements from the list with maximum value.
- **Syntax**
- Following is the syntax for max() method-
- max(list)
- **Parameters**
- **list** - This is a list from which max valued element are to be returned.
- **Return Value**
- This method returns t
- **Example**
- The following example shows the usage of max() method.
- list1, list2 = ['C++','Java', 'Python'], [456, 700, 200]
- print ("Max value element : ", max(list1))
- print ("Max value element : ", max(list2))

- **List min() Method :**
- **Description**
- The method min() returns the elements from the list with minimum value.
- **Syntax**
- Following is the syntax for min() method-
- min(list)
- **Parameters**
- **list** - This is a list from which min valued element is to be returned.
- **Return Value**
- This method returns the elements from the list with minimum value.

- list1, list2 = ['C++','Java', 'Python'], [456, 700, 200]
- print ("min value element : ", min(list1))
- print ("min value element : ", min(list2))

- **List list() Method :**
- **Description**
- The **list()**method takes sequence types and converts them to lists. This is used to convert a given tuple into list.
- ***Note:*** *Tuple are very similar to lists with only difference that element values of a tuple can not be changed and tuple elements are put between parentheses instead of square bracket. This function also converts characters in a string into a list.*
- ***Syntax***
- Following is the syntax for list() method-
- list(seq)
- **Parameters**
- **seq** - This is a tuple or string to be converted into list.
- **Return Value** : This method returns the list.
- aTuple = (123, 'C++', 'Java', 'Python') list1 = list(aTuple)
- print ("List elements : ", list1)
- str="Hello World" list2=list(str) print ("List elements : ", list2)

- Python includes the following list methods- :

SN	Methods with Description
1	**list.append(obj)** Appends object obj to list
2	**list.count(obj)** Returns count of how many times obj occurs in list
3	**list.extend(seq)** Appends the contents of seq to list
4	**list.index(obj)** Returns the lowest index in list that obj appears
5	**list.insert(index, obj)** Inserts object obj into list at offset index
6	**list.pop(obj=list[-1])** Removes and returns last object or obj from list
7	**list.remove(obj)** Removes object obj from list
8	**list.reverse()** Reverses objects of list in place
9	**list.sort([func])** Sorts objects of list, use compare func if given

- **List append() Method :**
- **Description**
- The **append()** method appends a passed obj into the existing list.
- **Syntax**
- Following is the syntax for append() method-
- list.append(obj)
- **Parameters**
- **obj** - This is the object to be appended in the list.
- **Return Value**
- This method does not return any value but updates existing list.
- **Example**
- list1 = ['C++', 'Java', 'Python']
- list1.append('C#')
- print ("updated list : ", list1)
- **NOTE: append method used to add element in list at last position**

- **List count()Method**
- **Description**
- The **count()** method returns count of how many times obj occurs in list.
- **Syntax**
- Following is the syntax for count() method-
- list.count(obj)
- **Parameters**
- **obj** - This is the object to be counted in the list.
- **Return Value**
- This method returns count of how many times obj occurs in list.
- aList = [123, 'xyz', 'zara', 'abc', 123]; print ("Count for 123 : ", aList.count(123))
- print ("Count for zara : ", aList.count('zara'))
- O/P
- Count for 123 : 2
- Count for zara : 1

- **Listextend()Method**
- **Description**
- The **extend()** method appends the contents of seq to list.
- **Syntax**
- list.extend(seq)
- **Parameters**
- **seq** - This is the list of elements
- **Return Value**
- This method does not return any value but adds the content to an existing list.

- list1 = ['physics', 'chemistry', 'maths'] list2=list(range(5)) #creates list of numbers between 0-4 list1.extend('Extended List :', list2)
- print (list1)

- **List index() Method**
- The **index()** method returns the lowest index in list that obj appears.
- **Syntax**
- Following is the syntax for index() method-
- list.index(obj)
- **Parameters**
- **obj** - This is the object to be find out.
- **Return Value**
- This method returns index of the found object otherwise raises an exception indicating that the value is not found.

- list1 = ['physics', 'chemistry', 'maths']
- print ('Index of chemistry', list1.index('chemistry')) print ('Index of C#', list1.index('C#'))

- **List insert() Method**
- **Description**
- The **insert() method** inserts object obj into list at offset index.
- **Syntax**
- Following is the syntax for insert() method-
- list.insert(index, obj)
- **Parameters**
- • **index** - This is the Index where the object obj need to be inserted.
- • **obj** - This is the Object to be inserted into the given list.

- **Return Value**
- This method does not return any value but it inserts the given element at the given index.
- list1 = ['physics', 'chemistry', 'maths'] list1.insert(1, 'Biology')
- print ('Final list : ', list1)
- Final list : ['physics', 'Biology', 'chemistry', 'maths']

- **List pop() Method :**
- **Description**
- The **pop()** method removes and returns last object or obj from the list.
- **Syntax**
- Following is the syntax for pop() method-
- list.pop(obj=list[-1])
- **Parameters**
- **obj** - This is an optional parameter, index of the object to be removed from the list.
- **Return Value**
- This method returns the removed object from the list.
- list1 = ['physics', 'Biology', 'chemistry', 'maths'] list1.pop()
- print ("list now : ", list1) list1.pop(1)
- print ("list now : ", list1)
- o/p
- ['physics', 'Biology', 'chemistry']
- ['physics', 'chemistry']

- **Listremove()Method :**
- **Parameters**
- **obj** - This is the object to be removed from the list.

- **Return Value**
- This method does not return any value but removes the given object from the list.

- Example :
- list1 = ['physics', 'Biology', 'chemistry', 'maths'] list1.remove('Biology')
- print ("list now : ", list1) list1.remove('maths')
- print ("list now : ", list1)

- list now :
- ['physics', 'chemistry', 'maths']
- ['physics', 'chemistry']

- **Listreverse()Method**
- **Description**
- The **reverse()** method reverses objects of list in place.
- **Syntax**
- Following is the syntax for reverse() method-
- list.reverse()
- **Parameters**
- NA
- **Return Value**
- This method does not return any value but reverse the given object from the list. \

- list1 = ['physics', 'Biology', 'chemistry', 'maths'] list1.reverse()
- print ("list now : ", list1)
- o/p
- list now :['maths', 'chemistry', 'Biology', 'physics']

- **List sort() Method :**
- **Description**
- The **sort()** method sorts objects of list, use compare function if given.
- **Syntax**
- Following is the syntax for sort() method-
- list.sort([func])
- **Parameters**
- NA
- **Return Value**
- This method does not return any value but reverses the given object from the list.
- list1 = ['physics', 'Biology', 'chemistry', 'maths'] list1.sort()
- print ("list now : ", list1)
- list now : ['Biology', 'chemistry', 'maths', 'physics']
- **Note: this method sorts the list as alphabetically , incase of numbers it will sort according to its value**

- **Tuples and Dictionaries :**
- tuple is a sequence of immutable Python objects.
- Tuples are sequences, just like lists.
- The main difference between the tuples and the lists is that the tuples cannot be changed unlike lists. Tuples use parentheses, whereas lists use square brackets .
- Creating a tuple is as simple as putting different comma-separated values. Optionally, you can put these comma-separated values between parentheses also.
- For example-
- tup1 = ('physics', 'chemistry', 1997, 2000)
- tup2 = (1, 2, 3, 4, 5)
- tup3 = "a", "b", "c", "d"
- The empty tuple is written as two parentheses containing nothing.
- tup1 = ();
- To write a tuple containing a single value you have to include a comma, even though there is only one value.
- tup1 = (50,) Like string indices, tuple indices start at 0, and they can be sliced, concatenated, and so on.

- **Accessing values in Tuples :**
- To access values in tuple, use the square brackets for slicing along with the index or indices to obtain the value available at that index.
- tup1 = ('physics', 'chemistry', 1997, 2000)
- tup2 = (1, 2, 3, 4, 5, 6, 7)
- print ("tup1[0]: ", tup1[0])
- print ("tup2[1:5]: ", tup2[1:5])
- When the above code is executed, it produces the following result-
- tup1[0] : physics
- tup2[1:5] : [2, 3, 4, 5]

- **Tuple Assignment :**
- Once in a while, it is useful to perform multiple assignments in a single statement and this can be done with tuple assignment **:**
- >>> a,b = 3,4
- >>> print a 3
- >>> print b 4
- >>> a,b,c = (1,2,3),5,6
- >>> print a (1, 2, 3)
- >>> print b 5
- >>> print c
- The left side is a tuple of variables; the right side is a tuple of values. Each value is assigned to its respective variable. All the expressions on the right side are evaluated before any of the assignments. This feature makes tuple assignment quite versatile. Naturally, the number of variables on the left and the number of values on the right have to be the same:
- Such statements can be useful shorthand for multiple assignment statements, but care should be taken that it doesn't make the code more difficult to read.
- One example of tuple assignment that improves readibility is when we want to swap the values of two variables. With conventional assignment statements, we have to use a temporary variable. For example, to swap aand b:

- **Tuples as return values :**
- Functions can return tuples as return values. For example, we could write a function that swaps two parameters :
- **def** swap(x, y):
- **return y, x**
- Then we can assign the return value to a tuple with two variables:
- a, b = swap(a, b)
- **Basic tuples operations, Concatenation, Repetition, in Operator, Iteration :**
- Tuples respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new tuple, not a string.

Python Expression	Results	Description
len((1, 2, 3))	3	Length
(1, 2, 3) + (4, 5, 6)	(1, 2, 3, 4, 5, 6)	Concatenation
('Hi!',) * 4	('Hi!', 'Hi!', 'Hi!', 'Hi!')	Repetition
3 in (1, 2, 3)	True	Membership

- **Built-in Tuple Functions :**
- Python includes the following tuple functions-

SN	Function with Description
1	cmp(tuple1, tuple2) No longer available in Python 3.
2	len(tuple) Gives the total length of the tuple.
3	max(tuple) Returns item from the tuple with max value.
4	min(tuple) Returns item from the tuple with min value.
5	tuple(seq) Converts a list into tuple.

- **Tuplelen()Method**
- **Description**
- The **len()** method returns the number of elements in the tuple.
- **Syntax**
- Following is the syntax for len() method-
- len(tuple)
- **Parameters**
- **tuple** - This is a tuple for which number of elements to be counted.
- **Return Value**
- This method returns the number of elements in the tuple.
- tuple1, tuple2 = (123, 'xyz', 'zara'), (456, 'abc') print ("First tuple length : ", len(tuple1))
- print ("Second tuple length : ", len(tuple2))

- **Tuplemax()Method**
- **Description**
- The **max()** method returns the elements from the tuple with maximum value.
- **Syntax**
- Following is the syntax for max() method-
- max(tuple)
- **Parameters**
- **tuple** - This is a tuple from which max valued element to be returned.
- **Return Value**
- This method returns the elements from the tuple with maximum value.
- **Example**
- The following example shows the usage of max() method.
- When we run the above program, it produces the following result-
- tuple1, tuple2 = ('maths', 'che', 'phy', 'bio'), (456, 700, 200)
- print ("Max value element : ", max(tuple1)) print ("Max value element : ", max(tuple2)

- **Tuple min() Method**
- **Description**
- The **min()** method returns the elements from the tuple with minimum value.
- **Syntax**
- Following is the syntax for min() method-
- min(tuple)
- **Parameters**
- **tuple** - This is a tuple from which min valued element is to be returned.
- **Return Value**
- This method returns the elements from the tuple with minimum value.
- tuple1, tuple2 = ('maths', 'che', 'phy', 'bio'), (456, 700, 200)
- print ("min value element : ", min(tuple1)) print ("min value element : ", min(tuple2))

- **Tupletuple()Method**
- **Description**
- The **tuple()** method converts a list of items into tuples.
- **Syntax**
- Following is the syntax for tuple() method-
- tuple(seq)
- **Parameters**
- **seq** - This is a tuple to be converted into tuple.
- **Return Value**
- This method returns the tuple.
- list1= ['maths', 'che', 'phy', 'bio'] tuple1=tuple(list1)
- print ("tuple elements : ", tuple1)
- o/p
- tuple elements : ('maths', 'che', 'phy', 'bio')

- **Dictionary**
- Each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces. An empty dictionary without any items is written with just two curly braces, like this:
- { }.
- Keys are unique within a dictionary while values may not be. The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.

- **Accessing Values in a dictionary :**
- To access dictionary elements, you can use the familiar square brackets along with the key to obtain its value. Following is a simple example.
- dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
- print ("dict['Name']: ", dict['Name'])
- print ("dict['Age']: ", dict['Age'])
- o/p
- dict['Name']: Zara
- dict['Age']: 7

- **Updating Dictionary :**
- You can update a dictionary by adding a new entry or a key-value pair, modifying an existing entry, or deleting an existing entry as shown in a simple example given below.
- dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'} dict['Age'] = 8; # update existing entry dict['School'] = "DPS School" # Add new entry
- print ("dict['Age']: ", dict['Age'])
- print ("dict['School']: ", dict['School'])
- **o/p**
- dict['Age']: 8 dict['School']: DPS School
- --
- **Deleting Elements from Dictionary :**
- You can either remove individual dictionary elements or clear the entire contents of a dictionary. You can also delete entire dictionary in a single operation.
- To explicitly remove an entire dictionary, just use the **del** statement. Following is a simple example-
- dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
- del dict['Name'] # remove entry with key 'Name' dict.clear() # remove all entries in dict del dict # delete entire dictionary
- print ("dict['Age']: ", dict['Age'])
- print ("dict['School']: ", dict['School'])
- Note: An exception is raised because after del dict, the dictionary does not exist anymore.

- **Properties of Dictionary keys :**
- Dictionary values have no restrictions. They can be any arbitrary Python object, either standard objects or user-defined objects. However, same is not true for the keys.
- There are two important points to remember about dictionary keys-
- A. **More than one entry per key is not allowed**. This means no duplicate key is allowed. When duplicate keys are encountered during assignment, the last assignment wins.
- dict = {'Name': 'Zara', 'Age': 7, 'Name': 'Manni'}
- print ("dict['Name']: ", dict['Name'])
- o/p
- dict['Name']: Manni
- B. **Keys must be immutable**. This means you can use strings, numbers or tuples as dictionary keys but something like ['key'] is not allowed.
- dict = {['Name']: 'Zara', 'Age': 7}
- print ("dict['Name']: ", dict['Name'])
- o/p
- Traceback (most recent call last): File "test.py", line 3, in <module>
- dict = {['Name']: 'Zara', 'Age': 7}
- TypeError: list objects are unhashable

- **Operations in Dictionary :**
- The del statement removes a key-value pair from a dictionary. For example, the following dictionary
- contains the names of various fruits and the number of each fruit in stock:
-
- >>> del inventory["pears"]
- >>> print inventory
- {’oranges’: 525, ’apples’: 430, ’bananas’: 312}
- Or if we’re expecting more pears soon, we might just change the value associated with pears:
- >>> inventory["pears"] = 0
- >>> print inventory
- {’oranges’: 525, ’apples’: 430, ’pears’: 0, ’bananas’: 312}
- The len function also works on dictionaries; it returns the number of key-value pairs:
- >>> len(inventory) 4

- **Built-In Dictionary Functions & Methods :**
- Python includes the following dictionary functions-

SN	Functions with Description
1	**cmp(dict1, dict2)** No longer available in Python 3.
2	**len(dict)** Gives the total length of the dictionary. This would be equal to the number of items in the dictionary.
3	**str(dict)** Produces a printable string representation of a dictionary.
4	**type(variable)** Returns the type of the passed variable. If passed variable is dictionary, then it would return a dictionary type.

- **Dictionarylen()Method**
- **Description**
- The method len() gives the total length of the dictionary. This would be equal to the number of items in the dictionary.

- **Syntax**
- Following is the syntax for len() method-
- len(dict)

- **Example**
- The following example shows the usage of len() method.
- #!/usr/bin/python3
- dict = {'Name': 'Manni', 'Age': 7, 'Class': 'First'} print ("Length : %d" % len (dict))
- When we run the above program, it produces the following result-
- Length : 3

- **Dictionarystr()Method**
- **Description**
- The method **str()** produces a printable string representation of a dictionary.
- **Syntax**
- Following is the syntax for str() method −
- str(dict)
- **Parameters**
- **dict** - This is the dictionary.
- **Return Value**
- This method returns string representation.
- **Example**
- The following example shows the usage of str() method.
- #!/usr/bin/python3
- dict = {'Name': 'Manni', 'Age': 7, 'Class': 'First'} print ("Equivalent String : %s" % str (dict))
- Equivalent String : {'Name': 'Manni', 'Age': 7, 'Class': 'First'}

Dictionary type() Method

- **Description**
- The method **type()** returns the type of the passed variable. If passed variable is dictionary then it would return a dictionary type.
- **Syntax**
- Following is the syntax for type() method-
- type(dict)
- **Parameters**
- **dict** - This is the dictionary.
- **Return Value**
- This method returns the type of the passed variable.
- dict = {'Name': 'Manni', 'Age': 7, 'Class': 'First'} print ("Variable Type : %s" % type (dict))
- o/p
- Variable Type : <type 'dict'>

- **Python includes the following dictionary methods-**

SN	Methods with Description
1	**dict.clear()** Removes all elements of dictionary *dict*.
2	**dict.copy()** Returns a shallow copy of dictionary *dict*.
3	**dict.fromkeys()** Create a new dictionary with keys from seq and values *set* to *value*.
4	**dict.get(key, default=None)** For *key* key, returns value or default if key not in dictionary.
5	**dict.has_key(key)** Removed, use the **in** operation instead.
6	**dict.items()** Returns a list of *dict*'s (key, value) tuple pairs.

7	**dict.keys()** Returns list of dictionary dict's keys.
8	**dict.setdefault(key, default=None)** Similar to get(), but will set dict[key]=default if *key* is not already in dict.
9	**dict.update(dict2)** Adds dictionary *dict2*'s key-values pairs to *dict*.
10	**dict.values()** Returns list of dictionary *dict*'s values.

- **Dictionaryclear()Method**
- **Description**
- The method **clear()** removes all items from the dictionary.
- **Syntax**
- Following is the syntax for clear() method-
- dict.clear()
- **Parameters**
- NA
- **Return Value**
- This method does not return any value.
- Example :
- dict = {'Name': 'Zara', 'Age': 7} print ("Start Len : %d" % len(dict)) dict.clear()
- print ("End Len : %d" % len(dict))
- o/p
- Start Len : 2
- End Len : 0

- **Dictionary copy() Method**
- **Description**
- The method **copy()** returns a shallow copy of the dictionary.
- **Syntax**
- Following is the syntax for copy() method-
- dict.copy()
- **Parameters**
- NA
- **Return Value**
- This method returns a shallow copy of the dictionary.
- dict1 = {'Name': 'Manni', 'Age': 7, 'Class': 'First'} dict2 = dict1.copy()
- print ("New Dictionary : ",dict2)
- o/p:
- New dictionary : {'Name': 'Manni', 'Age': 7, 'Class':'First'}

- **NOTE:**
- **Python provides two types of copy i.e. 1)shallow 2)deep**

- **Shallow Copy**

- A shallow copy creates a new object which stores the reference of the original elements.
- So, a shallow copy doesn't create a copy of nested objects, instead it just copies the reference of nested objects. This means, a copy process does not recurse or create copies of nested objects itself.
-

- **Deep Copy**
- A deep copy creates a new object and recursively adds the copies of nested objects present in the original elements.
- The deep copy creates independent copy of original object and all its nested objects.

- **Dictionary fromkeys() Method :**
- **Description**
- The method fromkeys() creates a new dictionary with keys from seq and values set to value.
- **Syntax**
- Following is the syntax for fromkeys() method-
- dict.fromkeys(seq[, value]))
- **Parameters**
- • **seq** - This is the list of values which would be used for dictionary keys preparation.
- • **value** - This is optional, if provided then value would be set to thisvalue
- **Return Value**
- This method returns the list.
- Example:
- seq = ('name', 'age', 'sex') dict = dict.fromkeys(seq)
- print ("New Dictionary : %s" % str(dict)) dict = dict.fromkeys(seq, 10)
- print ("New Dictionary : %s" % str(dict)

- **Dictionary get() Method**
- **Description**
- The method **get()** returns a value for the given key. If the key is not available then returns default value None.
- **Syntax**
- Following is the syntax for get() method-
- dict.get(key, default=None)
- **Parameters**
- • **key** - This is the Key to be searched in the dictionary.
- • **default** - This is the Value to be returned in case key does not exist.
- **Return Value**
- This method returns a value for the given key. If the key is not available, then returns default value as None.
- dict = {'Name': 'Zara', 'Age': 27}
- print ("Value : %s" % dict.get('Age'))
- print ("Value : %s" % dict.get('Sex', "NA"))

- **Dictionary items() Method**
- **Description**
- The method items() returns a list of dict's (key, value) tuple pairs.
- **Syntax :**
- Following is the syntax for items() method-
- dict.items()
- **Parameters**
- NA
- **Return Value**
- This method returns a list of tuple pairs.
- dict = {'Name': 'Zara', 'Age': 7}
- print ("Value : %s" % dict.items())
- o/p
- Value : [('Age', 7), ('Name', 'Zara')]

- **Dictionary keys() Method :**
- **Description**
- The method **keys()** returns a list of all the available keys in the dictionary.
- **Syntax**
- Following is the syntax for keys() method-
- dict.keys()
- **Parameters**
- NA
- **Return Value**
- This method returns a list of all the available keys in the dictionary.
- dict = {'Name': 'Zara', 'Age': 7}
- print ("Value : %s" % dict.keys())
- o/p
- Value : ['Age', 'Name']

Dictionary setdefault() Method :

- The method setdefault() is similar to get(), but will set dict[key]=default if the key is not already in dict.
- **Syntax**
- Following is the syntax for setdefault() method-
- dict.setdefault(key, default=None)
- **Parameters**
- • **key** - This is the key to be searched.
- • **default** - This is the Value to be returned in case key is not found.
- **Return Value**
- This method returns the key value available in the dictionary and if given key is not available then it will return provided default value.
- dict = {'Name': 'Zara', 'Age': 7}
- print ("Value : %s" % dict.setdefault('Age', None)) print ("Value : %s" % dict.setdefault('Sex', None)) print (dict)

- **Dictionary update() Method**
- **Description**
- The method **update()** adds dictionary dict2's key-values pairs in to dict. This function does not return anything.
- **Syntax**
- Following is the syntax for update() method-
- dict.update(dict2)
- **Parameters**
- **dict2** - This is the dictionary to be added into dict.
- **Return Value**
- This method does not return any value.
- dict = {'Name': 'Zara', 'Age': 7}
- dict2 = {'Sex': 'female' } dict.update(dict2)
- print ("updated dict : ", dict)
- o/p
- updated dict : {'Sex': 'female', 'Age': 7, 'Name': 'Zara'}

- **Dictionaryvalues()Method**
- **Description**
- The method **values()** returns a list of all the values available in a given dictionary.
- **Syntax**
- Following is the syntax for values() method-
- dict.values()
- **Parameters**
- NA
- **Return Value**
- This method returns a list of all the values available in a given dictionary.
- **Example**
- The following example shows the usage of values() method.
- #!/usr/bin/python3
- dict = {'Sex': 'female', 'Age': 7, 'Name': 'Zara'} print ("Values : ", list(dict.values()))
- When we run above program, it produces following result-
- Values : ['female', 7, 'Zara']

- **Files**
- Python provides basic functions and methods necessary to manipulate files by default. You can do most of the file manipulation using a **file** object.
- **The open Function**
- Before you can read or write a file, you have to open it using Python's built-in open() function. This function creates a **file** object, which would be utilized to call other support methods associated with it.
- **Syntax**
- file object = open(file_name [, access_mode][, buffering])
- Here are parameter details-
- ☐ **file_name:** The file_name argument is a string value that contains the name of the file that you want to access.
- ☐ **access_mode:** The access_mode determines the mode in which the file has to be opened, i.e., read, write, append, etc.
- A complete list of possible values is given below in the table. This is an optional parameter and the default file access mode is read (r).
- ☐ **buffering:** If the buffering value is set to 0, no buffering takes place. If the buffering value is 1, line buffering is performed while accessing a file. If you specify the buffering value as an integer greater than 1, then buffering action is performed with the indicated buffer size. If negative, the buffer size is the system default (default behavior).

Modes	Description
r	Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
rb	Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.
r+	Opens a file for both reading and writing. The file pointer placed at the beginning of the file.
rb+	Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file.
w	Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
wb	Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.

w+	Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading andwriting.
wb+	Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
a	Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
ab	Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
a+	Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.
ab+	Opens a file for both appending and reading in binary format. The filepointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

- **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 the attributes related to a file object-

Attribute	Description
file.closed	Returns true if file is closed, false otherwise.
file.mode	Returns access mode with which file was opened.
file.name	Returns name of the file.

Note: softspace attribute is not supported in Python 3.x

- **The close() Method**
- The close() method of a file object flushes any unwritten information and closes the file object, after which no more writing can be done.
- Python automatically closes a file when the reference object of a file is reassigned to another file. It is a good practice to use the close() method to close a file.
- **Syntax**
- fileObject.close();
- **Example**
- #!/usr/bin/python3
- # Open a file
- fo = open("foo.txt", "wb")
- print ("Name of the file: ", fo.name)
- # Close opened file fo.close()
- This produces the following result-
- Name of the file: foo.txt

- **Reading and Writing Files**
- The file object provides a set of access methods to make our lives easier. We would see how to use read() and write() methods to read and write files.
- --
- **The write() Method**
- The write() method writes any string to an open file. It is important to note that Python strings can have binary data and not just text.
- The write() method does not add a newline character ('\n') to the end of the string-
- **Syntax**
- fileObject.write(string);
- Here, passed parameter is the content to be written into the opened file.
- # Open a file
- fo = open("foo.txt", "w")
- fo.write("Python is a great language.\nYeah its great!!\n")
- # Close opend file fo.close()
- The above method would create foo.txt file and would write given content in that file and finally it would close that file. If you would open this file, it would have the following content-

- **The read() Method**
- The read() method reads a string from an open file. It is important to note that Python strings can have binary data apart from the text data.
- **Syntax**
- fileObject.read([count]);
- Here, passed parameter is the number of bytes to be read from the opened file. This method starts reading from the beginning of the file and if count is missing, then it tries to read as much as possible, maybe until the end of file.
- **Example**
- Let us take a file foo.txt, w
- # Open a file
- fo = open("foo.txt", "r+") str = fo.read(10)
- print ("Read String is : ", str)
- # Close opened file fo.close() which we created above.

- **File Positions :**

- 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.

- 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, the beginning of the file is used as the reference position. If it is set to 1, the current position is used as the reference position. If it is set to 2 then the end of the file would b

- # Open a file
- fo = open("foo.txt", "r+") str = fo.read(10)
- print ("Read String is : ", str)
- # Check current positione taken as the reference position.
- position = fo.tell()
- print ("Current file position : ", position)
- # Reposition pointer at the beginning once again position = fo.seek(0, 0)
- str = fo.read(10)
- print ("Again read String is : ", str)
- # Close opened file fo.close()

- This produces the following result-
- Read String is : Python is Current file position : 10
- Again read String is : Python is

- **Renaming and Deleting Files**
- Python **os** module provides methods that help you perform file-processing operations, such as renaming and deleting files.

- **The rename()Method**
- The rename() method takes two arguments, the current filename and the new filename.
- **Syntax**
- os.rename(current_file_name, new_file_name)
- **Example**
- Following is an example to rename an existing file *test1.txt*-
- #!/usr/bin/python3 import os
- # Rename a file from test1.txt to test2.txt os.rename("test1.txt", "test2.txt")

- **The remove() Method**
- You can use the remove() method to delete files by supplying the name of the file to be deleted as the argument.
- **Syntax**
- os.remove(file_name)
- **Example**
- Following is an example to delete an existing file test2.txt-
- #!/usr/bin/python3 import os
- # Delete file test2.txt os.remove("text2.txt")

- **Directories :**
- All files are contained within various directories, and Python has no problem handling these too.
- The **os** module has several methods that help you create, remove, and change directories.

- **The mkdir() Method**
- You can use the mkdir() method of the **os** module to create directories in the current directory. You need to supply an argument to this method, which contains the name of the directory to be created.
- **Syntax**
- os.mkdir("newdir")

- #!/usr/bin/python3 import os
- # Create a directory "test" os.mkdir("test")

- **The chdir() Method**
- You can use the chdir() method to change the current directory.
- The chdir() method takes an argument, which is the name of the directory that you want to make the current directory.

- **Syntax**
- os.chdir("newdir")
- **Example**
- Following is an example to go into "/home/newdir" directory-
- #!/usr/bin/python3 import os
- # Changing a directory to "/home/newdir" os.chdir("/home/newdir")

- **The getcwd() Method**
- The getcwd() method displays the current working directory.
- **Syntax :**
- os.getcwd()
- #!/usr/bin/python3 import os
- # This would give location of the current directory os.getcwd()

- --
- **The rmdir() Method**
- The rmdir() method deletes the directory, which is passed as an argument in the method. Before removing a directory, all the contents in it should beremoved.
- **Syntax :**
- os.rmdir('dirname')
- import os
- # This would remove "/tmp/test" directory.
- os.rmdir("/tmp/test")

- **Exceptions :**
- An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error.
- When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.
- Python provides two types of exceptions i.e. 1)built-in 2)user defined
- **Built-in Exceptions**
- :

EXCEPTION NAME	DESCRIPTION
Exception	Base class for all exceptions
StopIteration	Raised when the next() method of an iterator does not point to any object.
SystemExit	Raised by the sys.exit() function.
StandardError	Base class for all built-in exceptions except StopIteration and SystemExit.

ArithmeticError	Base class for all errors that occur for numeric calculation.
OverflowError	Raised when a calculation exceeds maximum limit for a numeric type.
FloatingPointError	Raised when a floating point calculation fails.
ZeroDivisonError	Raised when division or modulo by zero takes place for all numeric types.
AssertionError	Raised in case of failure of the Assert statement.
AttributeError	Raised in case of failure of attribute reference or assignment.
EOFError	Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
ImportError	Raised when an import statement fails.
KeyboardInterrupt	Raised when the user interrupts program execution, usually by pressing Ctrl+c.
LookupError	Base class for all lookup errors.

IndexError	Raised when an index is not found in a sequence.
KeyError	Raised when the specified key is not found in the dictionary.
NameError	Raised when an identifier is not found in the local or global namespace.
UnboundLocalError	Raised when trying to access a local variable in a function or method but no value has been assigned to it.
EnvironmentError	Base class for all exceptions that occur outside the Python environment.
IOError	Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.
OSError	Raised for operating system-related errors.
SyntaxError	Raised when there is an error in Python syntax.

IndentationError	Raised when indentation is not specified properly.
SystemError	Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.
SystemExit	Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.
TypeError	Raised when an operation or function is attempted that is invalid for the specified data type.
ValueError	Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
RuntimeError	Raised when a generated error does not fall into any category.
NotImplementedError	Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.

- **Handling Exceptions :**
- If you have some *suspicious* code that may raise an exception, you can defend your program by placing the suspicious code in a **try:** block. After the try: block, include an **except:** statement, followed by a block of code which handles the problem as elegantly as possible.
- **Syntax**
- Here is simple syntax of
- try....except...else blocks-

```
try:
        You do your operations here
        ......................
except ExceptionI:
        If there is ExceptionI, then execute this block.
except ExceptionII:
        If there is ExceptionII, then execute this block.
......................
else:
        If there is no exception then execute this block.
```

- Here are few important points about the above-mentioned syntax-
- • A single try statement can have multiple except statements. This is useful when the try block contains statements that may throw different types of exceptions.
- • You can also provide a generic except clause, which handles any exception.
- • After the except clause(s), you can include an else-clause. The code in the else- block executes if the code in the try: block does not raise an exception.
- • The else-block is a good place for code that does not need the try: block's protection.

- Example:
- #!/usr/bin/python3
- try:
- fh = open("testfile", "w")
- fh.write("This is my test file for exception handling!!")
- except IOError:
- print ("Error: can\'t find file or read data")
- else:
- print ("Written content in the file successfully") fh.close()

- **Exception with Arguments :**
- An exception can have an *argument*, which is a value that gives additional information about the problem. The contents of the argument vary by exception. You capture an exception's argument by supplying a variable in the except clause as follows-

```
try:
    You do your operations here
    ......................
except ExceptionType as Argument:
    You can print value of Argument here...
```

If you write the code to handle a single exception, you can have a variable follow the name of the exception in the except statement. If you are trapping multiple exceptions, you can have a variable follow the tuple of the exception.
This variable receives the value of the exception mostly containing the cause of the exception. The variable can receive a single value or multiple values in the form of a tuple. This tuple usually contains the error string, the error number, and an errorlocation.

- **Example**
- Following is an example for a single exception-
- #!/usr/bin/python3
- # Define a function here. def temp_convert(var): try:
- returnint(var)
- except ValueError as Argument:
- print("The argument does not contain numbers\n",Argument)
- # Call above function here. temp_convert("xyz")
- o/p

- The argument does not contain numbers invalid literal for int() with base 10: 'xyz'

- **<u>User-defined Exceptions :</u>**
- Python also allows you to create your own exceptions byderiving classes fromthe standard built-in exceptions.
- Here is an example related to *RuntimeError*. Here, a class is created that is subclassed from *RuntimeError*. This is useful when you need to display more specific information when an exception is caught.
- In the try block, the user-defined exception is raised and caught in the except block. The variable **e** is used to create an instance of the class *Networkerror*.

```
class Networkerror(RuntimeError):
        def init (self, arg):
                self.args = arg
```

- So once you have defined the above class, you can raise the exception as follows-

```
try:
        raise Networkerror("Bad hostname")
except Networkerror,e:
        print e.args
```

- **QUICK REVISION OF UNIT III :**
- **A list is an ordered set of values, where each value is identified by an index.**
- ***Important thing about a list is that the items in a list need not be of the same type.***
- **Lists are mutable .**
- **An assignment to an element of a list is called item assignment**
- ***Tuple are very similar to lists with only difference that element values of a tuple can not be changed and tuple elements are put between parentheses instead of square bracket.***
- **append method used to add element in list at last position .**
- **tuple is a sequence of immutable Python objects.**
- **Dictionary :Each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces.**
- **More than one entry per key is not allowed.**
- **Keys must be immutable.**
- **An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions.**
- **Python provides two types of exceptions i.e. 1)built-in 2)user defined**

- VISIT
- https://www.profajaypashankar.com
- For more study material and notes .

- VISIT
- https://www.youtube.com/channel/UCu4Bd22zM6RpvHWC9YHBh5Q?view_as=subscriber
- For more lectures .

- VISIT : FOR PRACTICAL MANUAL
- https://www.profajaypashankar.com/python-programming-practical-manual/
- Password:STUDYHARD

18. Python - GUI Programming (Tkinter)

Python provides various options for developing graphical user interfaces (GUIs). Most important are listed below:

- **Tkinter:** Tkinter is the Python interface to the Tk GUI toolkit shipped with Python. We would look this option in this tutorial.
- **wxPython:** This is an open-source Python interface for wxWindows http://wxpython.org.
- **JPython:** JPython is a Python port for Java, which gives Python scripts seamless access to Java class libraries on the local machine http://www.jython.org.

Tkinter Programming:

- Tkinter is the standard GUI library for Python. Python when combined with Tkinter provides a fast and easy way to create GUI applications. Tkinter provides a powerful object-oriented interface to the Tk GUI toolkit.
- Creating a GUI application using Tkinter is an easy task. All you need to do is perform the following steps:
  - **Example:**Import the *Tkinter* module.
  - Create the GUI application main window.
  - Add one or more of the above mentioned widgets to the GUI application.
  - Enter the main event loop to take action against each event triggered by the user.

```
import Tkinter
top = Tkinter.Tk()
# Code to add widgets will go here...
top.mainloop()
```

Python – Tkinter Button

The Button widget is used to add buttons in a Python application. These buttons can display text or images that convey the purpose of the buttons. You can attach a function or a method to a button, which is called automatically when you click the button.

Syntax:

```
w = Button ( master, option=value, ... )
```

Parameters:

- **master**: This represents the parent window.
- **options**: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Example:

```
import Tkinter
import tkMessageBox
top = Tkinter.Tk()
def helloCallBack():
   tkMessageBox.showinfo( "Hello Python", "Hello World")
B = Tkinter.Button(top, text ="Hello", command =
   helloCallBack)
B.pack()
top.mainloop()
```

Python – Tkinter Canvas

The Canvas is a rectangular area intended for drawing pictures or other complex layouts. You can place graphics, text, widgets, or frames on a Canvas.

Syntax:

```
w = Canvas ( master, option=value, ... )
```

- **Parameters**:
  - **master**: This represents the parent window.
  - **options**: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

The Canvas widget can support the following standard items:

- **arc .** Creates an arc item.

```
coord = 10, 50, 240, 210
arc = canvas.create_arc(coord, start=0, extent=150,
  fill="blue")
```

- **image .** Creates an image item, which can be an instance of either the BitmapImage or the PhotoImage classes.

```
filename = PhotoImage(file = "sunshine.gif")
image = canvas.create_image(50, 50, anchor=NE,
  image=filename)
```

- **line .** Creates a line item.

```
line = canvas.create_line(x0, y0, x1, y1, ..., xn, yn,
  options)
```

- **oval .** Creates a circle or an ellipse at the given coordinates. `oval = canvas.create_oval(x0, y0, x1, y1, options)`

- **polygon .** Creates a polygon item that must have at least three vertices.

```
oval = canvas.create_polygon(x0, y0, x1, y1,...xn, yn,
  options)
```

Example:

```
import Tkinter
import tkMessageBox
top = Tkinter.Tk()
C = Tkinter.Canvas(top, bg="blue", height=250,
  width=300)
coord = 10, 50, 240, 210
arc = C.create_arc(coord, start=0, extent=150,
  fill="red")
C.pack()
top.mainloop()
```

Example:

```
import Tkinter
import tkMessageBox

top = Tkinter.Tk()

C = Tkinter.Canvas(top, bg="blue", height=250,
  width=300)

coord = 10, 50, 240, 210
arc = C.create_arc(coord, start=0, extent=150,
  fill="red")

C.pack()
top.mainloop()
```

Python – Tkinter Checkbutton

The Checkbutton widget is used to display a number of options to a user as toggle buttons. The user can then select one or more options by clicking the button corresponding to each option.

You can also display images in place of text.

Syntax:

```
w = Checkbutton ( master, option, ... )
```

- **Parameters**:
  - **master**: This represents the parent window.
  - **options**: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Example:

```
from Tkinter import *
import tkMessageBox
import Tkinter

top = Tkinter.Tk()
CheckVar1 = IntVar()
CheckVar2 = IntVar()
C1 = Checkbutton(top, text = "Music", variable =
   CheckVar1, \
     onvalue = 1, offvalue = 0, height=5, width = 20)
C2 = Checkbutton(top, text = "Video", variable =
   CheckVar2, \
     onvalue = 1, offvalue = 0, height=5, width = 20)
C1.pack()
C2.pack()
top.mainloop()
```

Python – Tkinter Entry:

- The Entry widget is used to accept single-line text strings from a user.
- If you want to display multiple lines of text that can be edited, then you should usethe *Text* widget.
- If you want to display one or more lines of text that cannot be modified by the user then you should use the *Label* widget.

Syntax:

Here is the simple syntax to create this widget:

```
w = Entry( master, option, ... )
```

Parameters:

- **master**: This represents the parent window.
- **options**: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Example:

```
from Tkinter import *

top = Tk()
L1 = Label(top, text="User Name")
L1.pack( side = LEFT)
E1 = Entry(top, bd =5)

E1.pack(side = RIGHT)

top.mainloop()
```

Python – Tkinter Frame

- The Frame widget is very important for the process of grouping and organizing other widgets in a somehow friendly way. It works like a container, which is responsible for arranging the position of other widgets.
- It uses rectangular areas in the screen to organize the layout and to provide padding of these widgets. A frame can also be used as a foundation class to implement complex widgets.

Syntax:

Here is the simple syntax to create this widget:

```
w = Frame ( master, option, ... )
```

Parameters:

- **master**: This represents the parent window.
- **options**: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Example:

```
from Tkinter import *
root = Tk()
frame = Frame(root)
frame.pack()
bottomframe = Frame(root)
bottomframe.pack( side = BOTTOM )
redbutton = Button(frame, text="Red", fg="red")
redbutton.pack( side = LEFT)
greenbutton = Button(frame, text="Brown", fg="brown")
greenbutton.pack( side = LEFT )
bluebutton = Button(frame, text="Blue", fg="blue")
bluebutton.pack( side = LEFT )
blackbutton = Button(bottomframe, text="Black",
   fg="black")
blackbutton.pack( side = BOTTOM)
root.mainloop()
```

Python – Tkinter Label

- This widget implements a display box where you can place text or images. The text displayed by this widget can be updated at any time you want.
- It is also possible to underline part of the text (like to identify a keyboard shortcut), and span the text across multiple lines.

Syntax:

Here is the simple syntax to create this widget:

```
w = Label ( master, option, ... )
```

Parameters:

- **master**: This represents the parent window.
- **options**: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Example:

```
from Tkinter import *

root = Tk()

var = StringVar()
label = Label( root, textvariable=var, relief=RAISED )

var.set("Hey!? How are you doing?")
label.pack()
root.mainloop()
```

Python – Tkinter Listbox

- The Listbox widget is used to display a list of items from which a user can select a number of items

Syntax:

- Here is the simple syntax to create this widget:

```
w = Listbox ( master, option, ... )
```

Parameters:

  - **master**: This represents the parent window.
  - **options**: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Example:

```
from Tkinter import *
import tkMessageBox
import Tkinter

top = Tk()

Lb1 = Listbox(top)
Lb1.insert(1, "Python")
Lb1.insert(2, "Perl")
Lb1.insert(3, "C")
Lb1.insert(4, "PHP")
Lb1.insert(5, "JSP")
Lb1.insert(6, "Ruby")

Lb1.pack()
top.mainloop()
```

Python – Tkinter Menubutton

- A menubutton is the part of a drop-down menu that stays on the screen all the time. Every menubutton is associated with a Menu widget that can display the choices for that menubutton when the user clicks on it.

Syntax:

Here is the simple syntax to create this widget:

```
w = Menubutton ( master, option, ... )
```

Parameters:

- **master**: This represents the parent window.
- **options**: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Example:

```
from Tkinter import *
import tkMessageBox
import Tkinter
top = Tk()
mb=  Menubutton ( top, text="condiments", relief=RAISED )
mb.grid()
mb.menu  =  Menu ( mb, tearoff = 0 )
mb["menu"]  =  mb.menu
mayoVar  = IntVar()
ketchVar = IntVar()
mb.menu.add_checkbutton ( label="mayo“, variable=mayoVar )
mb.menu.add_checkbutton ( label="ketchup“,
                                 variable=ketchVar )

mb.pack()
top.mainloop()
```

Python – Tkinter Message

- This widget provides a multiline and noneditable object that displays texts, automatically breaking lines and justifying their contents.
- Its functionality is very similar to the one provided by the Label widget, except that it can also automatically wrap the text, maintaining a given width or aspect ratio.

Syntax:

Here is the simple syntax to create this widget:

```
w = Message ( master, option, ... )
```

Parameters:

- **master**: This represents the parent window.
- **options**: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Example:

```
from Tkinter import *

root = Tk()

var = StringVar()
label = Message( root, textvariable=var, relief=RAISED )

var.set("Hey!? How are you doing?")
label.pack()
root.mainloop()
```

Python – Tkinter Radiobutton

- This widget implements a multiple-choice button, which is a way to offer many possible selections to the user, and let user choose only one of them.
- In order to implement this functionality, each group of radiobuttons must be associated to the same variable, and each one of the buttons must symbolize a single value. You can use the Tab key to switch from one radionbutton to another.

Syntax:

Here is the simple syntax to create this widget:

```
w = Radiobutton ( master, option, ... )
```

Parameters:

- **master**: This represents the parent window.
- **options**: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Example:

```
from Tkinter import *
def sel():
   selection = "You selected the option " + str(var.get())
   label.config(text = selection)
root = Tk()
var = IntVar()
R1 = Radiobutton(root, text="Option 1", variable=var,
                          value=1, command=sel)
R1.pack( anchor = W )
R2 = Radiobutton(root, text="Option 2", variable=var,
                          value=2,command=sel)
R2.pack( anchor = W )
R3 = Radiobutton(root, text="Option 3", variable=var,
                          value=3,command=sel)
R3.pack( anchor = W)
label = Label(root)
label.pack()
root.mainloop()
```

Python - Tkinter Scale

- The Scale widget provides a graphical slider object that allows you to select values from a specific scale.

Syntax:

Here is the simple syntax to create this widget:

```
w = Scale ( master, option, ... )
```

Parameters:

- **master**: This represents the parent window.
- **options**: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Example:

```
from Tkinter import *
def sel():
   selection = "Value = " + str(var.get())
   label.config(text = selection)

root = Tk()
var = DoubleVar()
scale = Scale( root, variable = var )
scale.pack(anchor=CENTER)

button = Button(root, text="Get Scale Value", command=sel)
button.pack(anchor=CENTER)

label = Label(root)
label.pack()
root.mainloop()
```

Python – Tkinter Scrollbar

- This widget provides a slide controller that is used to implement vertical scrolled widgets, such as Listbox, Text, and Canvas. Note that you can also create horizontal scrollbars on Entry widgets.

Syntax:

Here is the simple syntax to create this widget:

```
w = Scrollbar ( master, option, ... )
```

Parameters:

- **master:** This represents the parent window.
- **options:** Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Example:

```
from Tkinter import *

root = Tk()
scrollbar = Scrollbar(root)
scrollbar.pack( side = RIGHT, fill=Y )

mylist = Listbox(root, yscrollcommand = scrollbar.set )
for line in range(100):
   mylist.insert(END, "This is line number " + str(line))

mylist.pack( side = LEFT, fill = BOTH )
scrollbar.config( command = mylist.yview )

mainloop()
```

Python – Tkinter Text

- Text widgets provide advanced capabilities that allow you to edit a multiline text and format the way it has to be displayed, such as changing its color and font.
- You can also use elegant structures like tabs and marks to locate specific sections of the text, and apply changes to those areas. Moreover, you can embed windows and images in the text because this widget was designed to handle both plain and formatted text.

Syntax:

Here is the simple syntax to create this widget:

```
w = Text ( master, option, ... )
```

Parameters:

- **master**: This represents the parent window.
- **options**: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Example:

```
from Tkinter import *
def onclick():
   pass
root = Tk()
text = Text(root)
text.insert(INSERT, "Hello.....")
text.insert(END, "Bye Bye.....")
text.pack()

text.tag_add("here", "1.0", "1.4")
text.tag_add("start", "1.8", "1.13")
text.tag_config("here", background="yellow",
   foreground="blue")
text.tag_config("start", background="black",
   foreground="green")
root.mainloop()
```

Python – Tkinter Toplevel

- Toplevel widgets work as windows that are directly managed by the window manager. They do not necessarily have a parent widget on top of them.
- Your application can use any number of top-level windows.

Syntax:

Here is the simple syntax to create this widget:

```
w = Toplevel ( option, ... )
```

Parameters:

- **options:** Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Example:

```
from Tkinter import *

root = Tk()
top = Toplevel()

top.mainloop()
```

Python – Tkinter Spinbox

- The Spinbox widget is a variant of the standard Tkinter Entry widget, which can be used to select from a fixed number of values.

Syntax:

Here is the simple syntax to create this widget:

```
w = Spinbox( master, option, ... )
```

Parameters:

- **master:** This represents the parent window.
- **options**: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Example:

```
from Tkinter import *

master = Tk()

w = Spinbox(master, from_=0, to=10)
w.pack()

mainloop()
```

Python – Tkinter PanedWindow

- A PanedWindow is a container widget that may contain any number of panes, arranged horizontally or vertically.
- Each pane contains one widget, and each pair of panes is separated by a moveable (via mouse movements) sash. Moving a sash causes the widgets on either side of the sash to be resized.

Syntax:

Here is the simple syntax to create this widget:

```
w = PanedWindow( master, option, ... )
```

Parameters:

- **master**: This represents the parent window.
- **options**: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Example:

```
from Tkinter import *
m1 = PanedWindow()
m1.pack(fill=BOTH, expand=1)
left = Label(m1, text="left pane")
m1.add(left)
m2 = PanedWindow(m1, orient=VERTICAL)
m1.add(m2)
top = Label(m2, text="top pane")
m2.add(top)
bottom = Label(m2, text="bottom pane")
m2.add(bottom)
mainloop()
```

Python – Tkinter LabelFrame

- A labelframe is a simple container widget. Its primary purpose is to act as a spacer or container for complex window layouts.
- This widget has the features of a frame plus the ability to display a label.

Syntax:

Here is the simple syntax to create this widget:

```
w = LabelFrame( master, option, ... )
```

Parameters:

- **master**: This represents the parent window.
- **options**: Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Example:

```
from Tkinter import *

root = Tk()

labelframe = LabelFrame(root, text="This is a LabelFrame")
labelframe.pack(fill="both", expand="yes")

left = Label(labelframe, text="Inside the LabelFrame")
left.pack()

root.mainloop()
```