# UNIT-I

## Introduction to Python & Operators

## 1. History of Python

Python is a widely used general-purpose, high-level programming language. It was initially designed by Guido van Rossum in 1991 and developed by Python Software Foundation (PSF) at Centrum Wiskunde & Informatica (CWI) in the Netherlands as a successor to the ABC language. It was mainly developed for emphasis on code readability, and its syntax allows programmers to express concepts in fewer lines of code.

PSF supports two versions, Python 2.x & Python 3.x. Python 2.0 was released in october 2000 and includes a large number of features. PSF continues to support version Python 2 because a large body of existing code could not be forward ported to Python 3. So, they will support Python 2 until 2020.

Python 3.0 was released on December 3rd, 2008. It was designed to rectify certain flaws in earlier version. This version is not completely backward-compatible with previous versions. However, many of its major features have since been back-ported to the Python 2.6.x and 2.7.x version series. Releases of Python 3 include 2 to 3 utilities to facilitate the automation of translation of Python 2 code to Python 3.

### 1.1 Versions of Python

| Version | Release Date | Important Features |
|---|---|---|
| Python 0.9.0 | February 1991 | ❖ Classes with inheritance exception handling<br>❖ Functions<br>❖ Modules |
| Python 1.0 | January 1994 | ❖ Functional programming tools (lambda, map, filter and reduce).<br>❖ Support for complex numbers.<br>❖ Functions with keyword arguments |
| Python 2.0 | October 2000 | ❖ List comprehension.<br>❖ Cycle-detecting garbage collector.<br>❖ Support for Unicode. Unification of data types and classes |
| Python 2.7.0 - Current version | July 2010 | |
| Python 2.7.15 - Current sub-version | May 2018 | |
| Python 3 | December 2008 | ❖ Backward incompatible.<br>❖ print keyword changed to print () function<br>❖ raw_input() function depreciated<br>❖ Unified str/Unicode types.<br>❖ Utilities for automatic conversion of Python 2.x code |
| Python 3.6 | December 2016 | |
| Python 3.6.5 | March 2018 | |

| Version | Release Date | Important Features |
|---|---|---|
| Python 3.7.0 - Current Version | May 2018 | ❖ New CAPI for thread-local storage<br>❖ Built-in breakpoint ()<br>❖ Data classes<br>❖ Context variables |

## 1.2 Python Features:

- Python is an interpreter-based language, which allows execution of one instruction at a time.
- Extensive basic data types are supported e.g. numbers (floating point, complex, and unlimited-length long integers), strings (both ASCII and Unicode), lists, and dictionaries.
- Variables can be strongly typed as well as dynamic typed.
- Supports object-oriented programming concepts such as class, inheritance, objects, module, namespace etc.
- Cleaner exception handling support.
- Supports automatic memory management.

## 1.3 Python Advantages

- Python provides enhanced readability. For that purpose, uniform indents are used to delimit blocks of statements instead of curly brackets, like in many languages such as C, C++ and Java.
- Python is free and distributed as open-source software. A large programming community is actively involved in the development and support of Python libraries for various applications such as web frameworks, mathematical computing and data science.
- Python is a cross-platform language. It works equally on different OS platforms like Windows, Linux, Mac OSX etc. Hence Python applications can be easily ported across OS platforms.
- Python supports multiple programming paradigms including imperative, procedural, object-oriented and functional programming styles.
- Python is an extensible language. Additional functionality (other than what is provided in the core language) can be made available through modules and packages written in other languages (C, C++, Java etc.)

- A standard DB-API for database connectivity has been defined in Python. It can be enabled using any data source (Oracle, MySQL, SQLite etc.) as a backend to the Python program for storage, retrieval and processing of data.
- Python can be integrated with other popular programming technologies like C, C++, Java, ActiveX and CORBA.

## 1.4 Python Application Types

Even though Python started as a general-purpose programming language with no particular application as its focus, over last few years it has emerged as the language of choice for developers in some application areas. Some important applications of Python are given below:

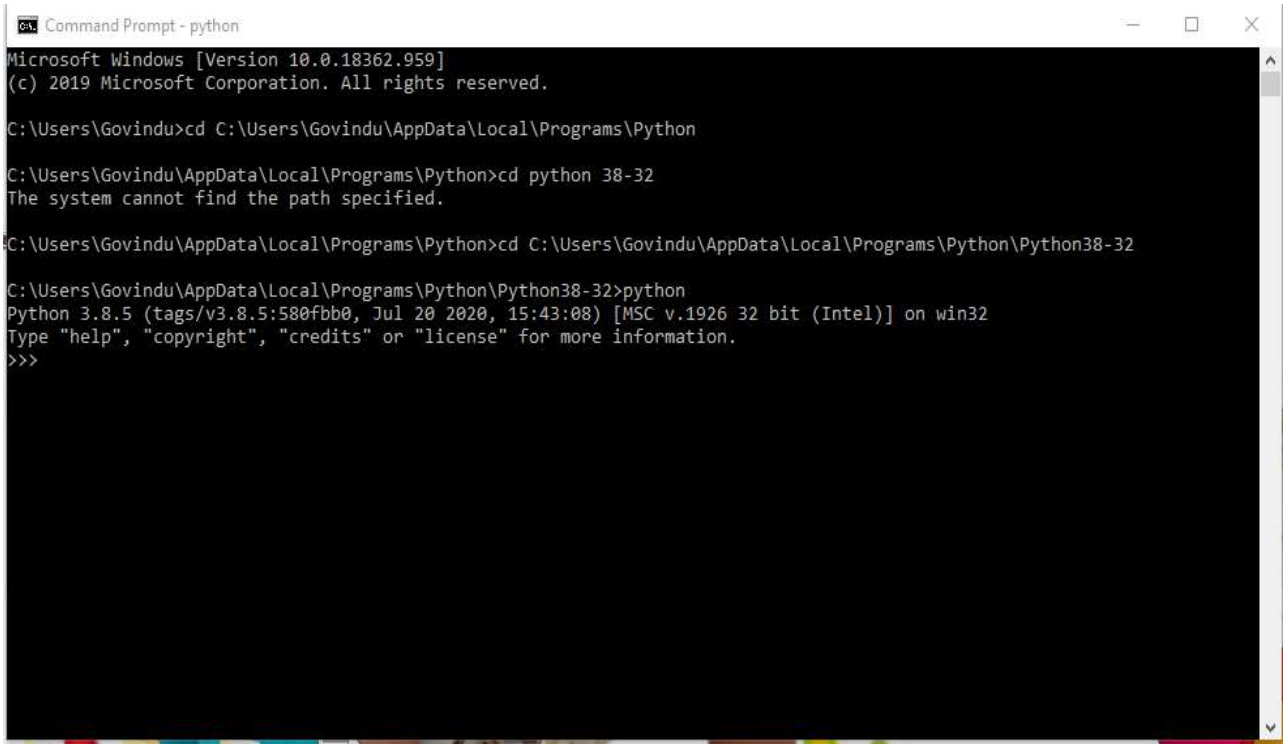
- Data Science
- Machine Learning
- Web Development
- Image Processing
- Game Development
- Embedded Systems and IoT
- Android Apps

## 2 Usage of Python Interpreter, Python Shell

Python is an interpreter language. It means it executes the code line by line. Python provides a Python Shell (also known as Python Interactive Shell) which is used to execute a single Python command and get the result.

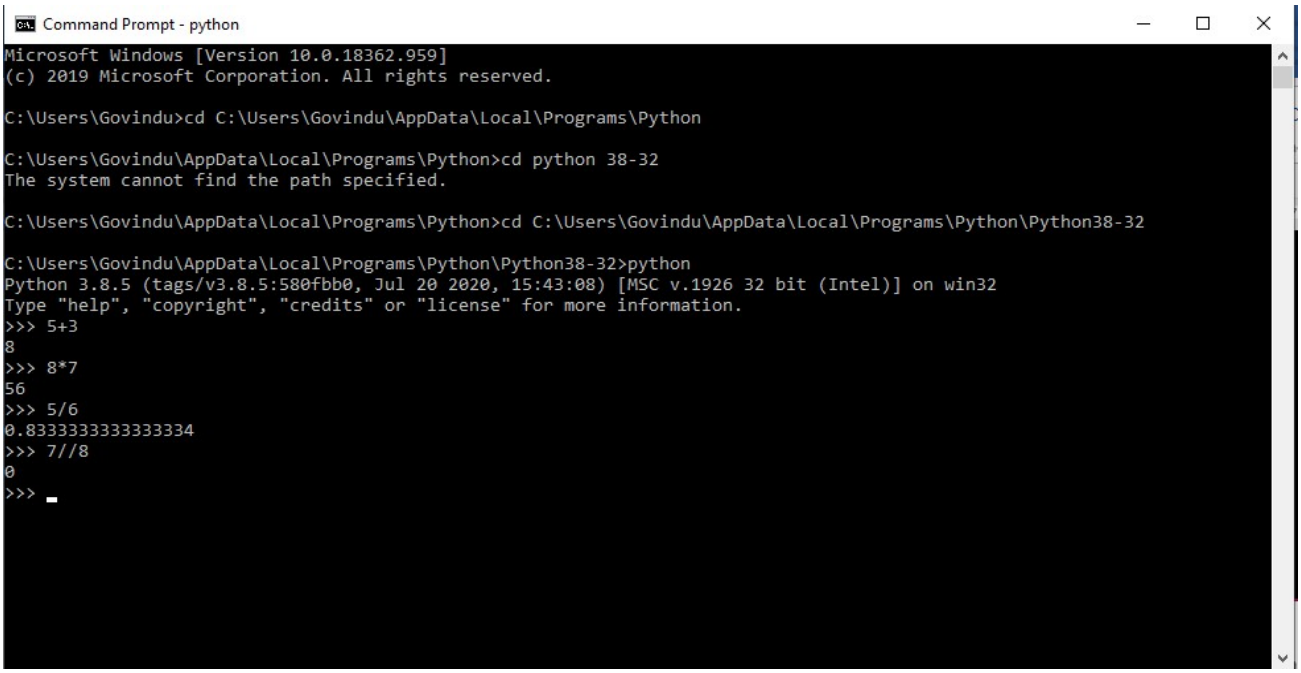
Python Shell waits for the input command from the user. As soon as the user enters the command, it executes it and displays the result.

To open the Python Shell on Windows, open the command prompt, write **python** and press **enter**.




**Python Shell**

As you can see, a Python Prompt comprising of three Greater Than symbols (>>>) appears. Now, you can enter a single statement and get the result. For example, enter a simple expression like `3 + 2`, press enter and it will display the result in the next line, as shown below.



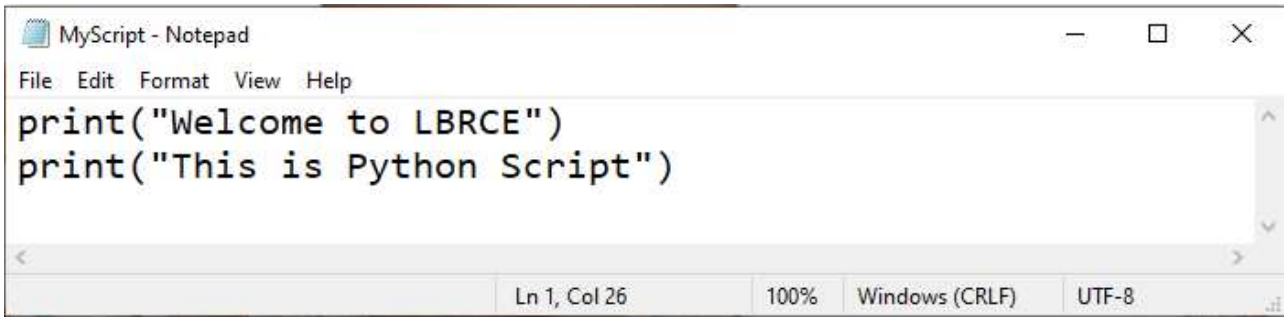

**Command Execution on Python Shell**

## 2.1 Execute Python Script using Command Prompt

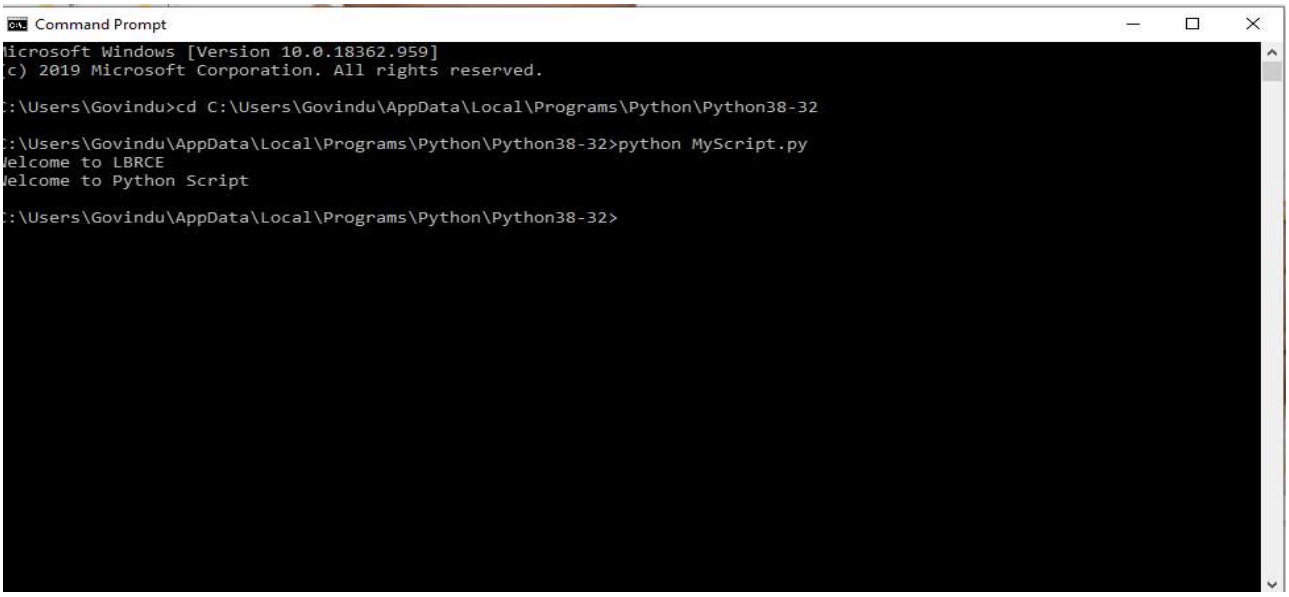
As you have seen above, Python Shell executes a single statement. To execute multiple statements, create a Python file with extension .py, and write Python scripts (multiple statements).

For example, enter the following statement in a text editor such as Notepad.

Example: MyScript.py




Save it as MyScript.py, navigate command prompt to the folder where you have saved this file and execute the python MyScript.py command, as shown below.




## 2.2 Python - IDLE

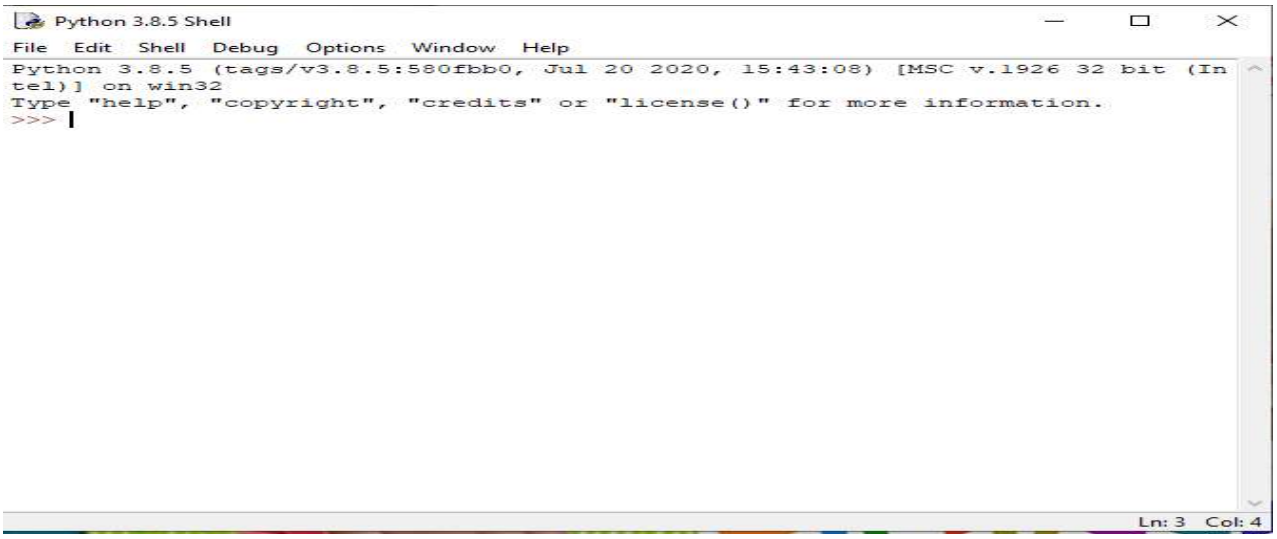
IDLE (Integrated Development and Learning Environment) is an integrated development environment (IDE) for Python. The Python installer for Windows contains the IDLE module by default.

IDLE is not available by default in Python distributions for Linux. It needs to be installed using the respective package managers. For example, in case of Ubuntu:

$ sudo apt-get install idle

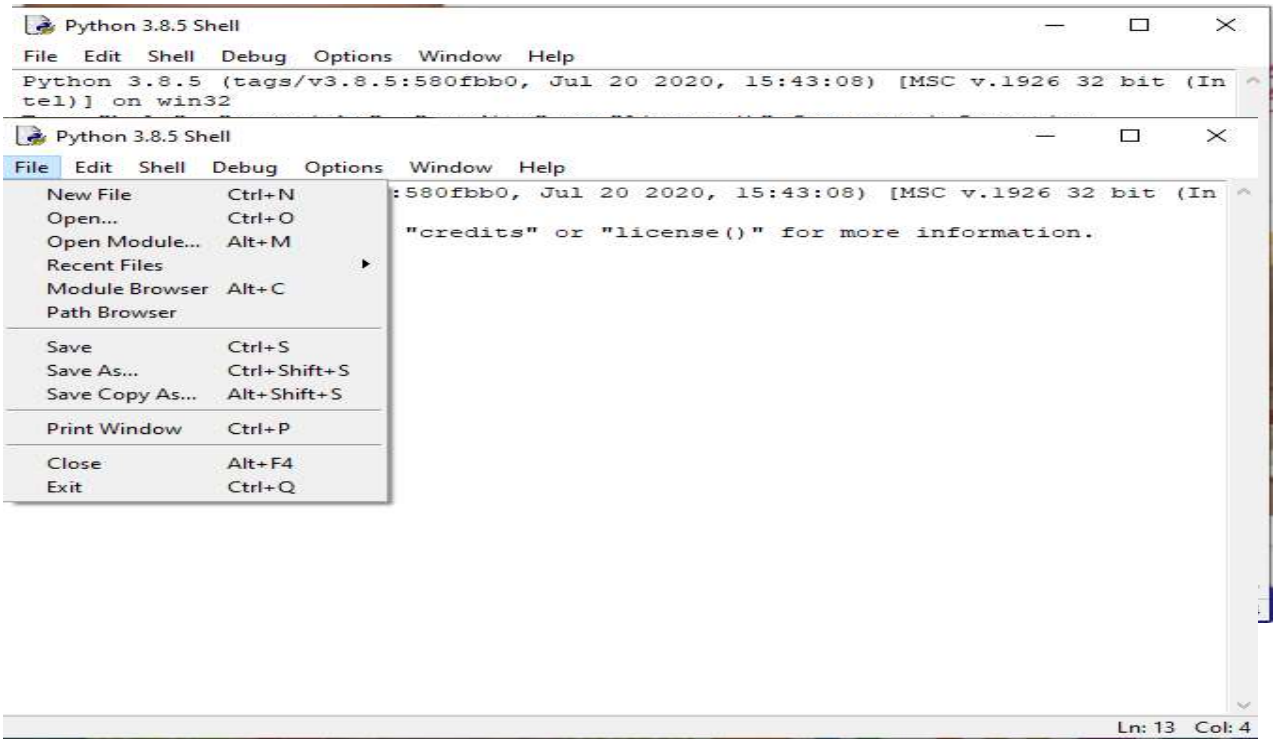
IDLE can be used to execute a single statement just like Python Shell and also to create, modify and execute Python scripts. IDLE provides a fully-featured text editor to create Python scripts that includes features like syntax highlighting, auto completion and smart indent. It also has a debugger with stepping and breakpoints features.

To start IDLE interactive shell, search for the IDLE icon in the start menu and double click on it.



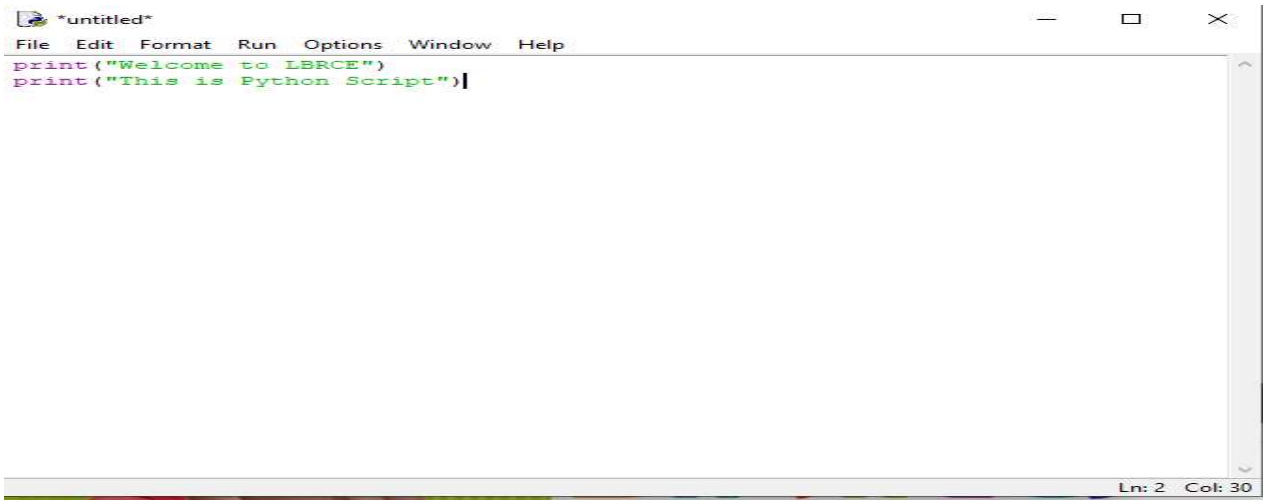

**Python IDLE**

Now, you can execute Python statements same as in Python Shell as shown below.



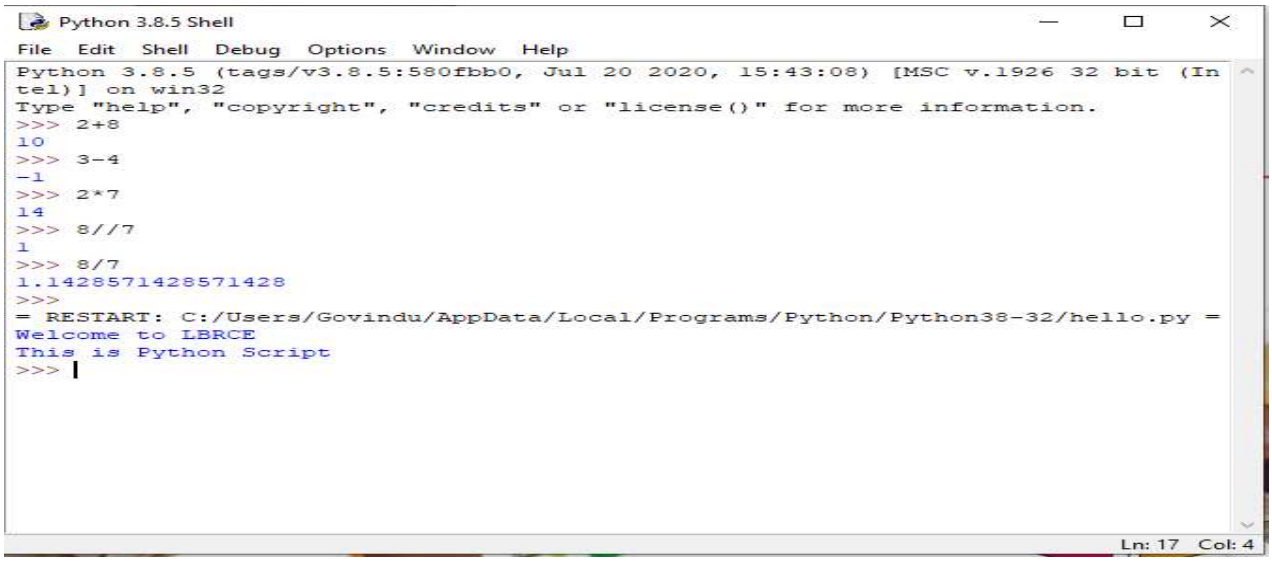

To execute a Python script, create a new file by selecting File -> New File from the menu.

Enter multiple statements and save the file with extension .py using File -> Save. For example, save the following code as hello.py.




**Python Script in IDLE**

Now, press F5 to run the script in the editor window. The IDLE shell will show the output.

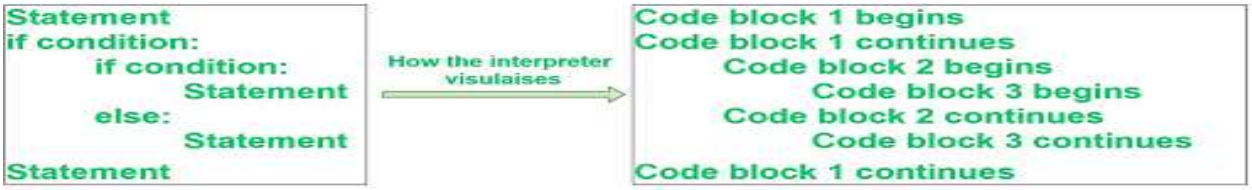



**Python Script Execution Result in IDLE**

## 2.3 Python Indentation

Indentation refers to the spaces at the beginning of a code line. Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important. Python uses indentation to indicate a block of code.


Let us consider the following example.




In the above example,

- Statement (line 1), if condition (line 2), and statement (last line) belongs to the same block which means that after statement 1, if condition will be executed. and suppose the if condition becomes False then the Python will jump to the last statement for execution.
- The nested if-else belongs to block 2 which means that if nested if becomes False, then Python will execute the statements inside the else condition.
- Statements inside nested if-else belongs to block 3 and only one statement will be executed depending on the if-else condition.

**Python program with Indentation:**

1) 
```
if 5 > 2:
  print("Five is greater than two!")
```

The above program will not give any error because it follows indentation.

2) 
```
from math import sqrt
n = input ("Maximum Number? ")
n = int(n)+1
for a in range(1,n):
    for b in range (a, n):
        c_square = a**2 + b**2
        c = int(sqrt(c_square))
        if ((c_square - c**2) == 0):
            print(a, b, c)
```

The above program will not give any error because it follows indentation.

3) 
```
if 5 > 2:
print("Five is greater than two!")
```

The above program will give error because it does not follow indentation, i.e., if is a block and the print() belongs to if. So, we should include print() in if block. In above program the print() does not belongs to if block so, it produces error.

## 2.4 Python Built-in types

Data types are the classification or categorization of data items. Data types represent a kind of value which determines what operations can be performed on that data. Numeric, non-numeric and Boolean (true/false) data are the most used data types. However, each programming language has its own classification largely reflecting its programming philosophy.

Python has the following standard or built-in data types:

### 2.4.1 Numeric:

A numeric value is any representation of data which has a numeric value. Python identifies three types of numbers:

- **Integer:** Positive or negative whole numbers (without a fractional part).
- **Float:** Any real number with a floating point representation in which a fractional component is denoted by a decimal symbol or scientific notation.
- **Complex number:** A number with a real and imaginary component represented as x+yj. x and y are floats and j is -1(square root of -1 called an imaginary number).

### 2.4.2 Boolean:

Data with one of two built-in values True or False. Notice that 'T' and 'F' are capital. true and false are not valid booleans and Python will throw an error for them.

### 2.4.3 Sequence Type:

A sequence is an ordered collection of similar or different data types. Python has the following built-in sequence data types:

- **String**: A string value is a collection of one or more characters put in single, double or triple quotes.
- **List**: A list object is an ordered collection of one or more data items, not necessarily of the same type, put in square brackets.
- **Tuple**: A Tuple object is an ordered collection of one or more data items, not necessarily of the same type, put in parentheses.

### 2.4.4 Dictionary:

- A dictionary object is an unordered collection of data in a key: value pair form.
- A collection of such pairs is enclosed in curly brackets. For example: {1:"Steve", 2:"Bill", 3:"Ram", 4: "Farha"}

### 2.4.5 type () function:

Python has an in-built function **type()** to ascertain the data type of a certain value. For example, enter type(1234) in Python shell and it will return <class 'int'>, which means 1234 is an integer value. Try and verify the data type of different values in Python shell, as shown below.

1) >>> type(1234)
   <class 'int'>
2) >>> type(67.5)
   <class 'float'>
3) >>> type(6+4j)
   <class 'complex'>
4) >>> type("hello")
   <class 'str'>

### 2.4.6 Mutable and Immutable Objects:

Data objects of the above types are stored in a computer's memory for processing. Some of these values can be modified during processing, but contents of others can't be altered once they are created in the memory.

Number values, strings, and tuple are immutable, which means their contents can't be altered after creation.

On the other hand, collection of items in a List or Dictionary object can be modified. It is possible to add, delete, insert, and rearrange items in a list or dictionary. Hence, they are mutable objects.

### Python - Number Types

Python includes three numeric types to represent numbers: integer, float, and complex.

**Integer**: Zero, positive and negative whole numbers without a fractional part and having unlimited precision, e.g. 1234, 0, -456.

- A number having **0o** or **0O** as prefix represents an octal number. For example: 0O12: equivalent to 10 (ten) in the decimal number system.

- A number with **0x** or **0X** as prefix represents **hexadecimal** number. For Example: 0x12: equivalent to 18 (Eighteen) in the decimal number system.

**Float:** Positive and negative real numbers with a fractional part denoted by the decimal symbol or the scientific notation using E or e, e.g. 1234.56, 3.142, -1.55, 0.23. Scientific notation is used as a short representation to express floats having many digits.

For Example: 345600000000 is represented as 3.456e11 or 3.456E11

**Complex:** A complex number is a number with real and imaginary components. For example, 5 + 6j is a complex number where 5 is the real component and 6 multiplied by j is an imaginary component.

Examples: 1+2j, 10-5.5J, 5.55+2.33j, 3.11e-6+4j

## Python - String

A string object is one of the sequence data types in Python. It is an immutable sequence of Unicode characters. Strings are objects of Python's built-in class 'str'. String literals are written by enclosing a sequence of characters in single quotes ('hello'), double quotes ("hello") or triple quotes ('''hello''' or """hello""").

Example:
```
>> str1='hello'
>>> str1
'hello'
>>> str2="hello"
>>> str2
'hello'
>>> str3="""hello"""
>>> str3
'hello'
```

Note that the value of all the strings, as displayed by the Python interpreter, is the same ('hello'), irrespective of whether single, double or triple quotes were used for string formation. If it is required to embed double quotes as part of a string, the string itself should be put in single quotes. On the other hand, if a single-quoted text is to be embedded, the string should be written in double quotes.

```
>>> str='Welcome to "Python Programming" '
>>> str
'Welcome to "Python Programming" '
>>> str1="Welcome to 'Python Programming' "
>>> str1
"Welcome to 'Python Programming' "
```

A sequence is defined as an ordered collection of items. Hence, a string is an ordered collection of characters. The sequence uses an index (starting with zero) to fetch a certain item (a character in case of a string) from it.

```
>>> mystring="Good Morning"
>>> mystring[0]
'G'
>>> mystring[8]
'n'
```

The string is an immutable object. Hence, it is not possible to modify it. The attempt to assign different characters at a certain index results in errors.

```
>>> mystring
'Good Morning'
>>> mystring[1]='$'
TypeError: 'str' object does not support item assignment
```

**Triple Quoted String:**

The triple quoted string is useful when a multi-line text is to be defined as a string literal.

```
>>> str="""Welcome to
Python
Programming"""
>>> str
'Welcome to\nPython\nProgramming'
```

**Escape Sequences**

The escape character is used to invoke an alternative implementation of the subsequent character in a sequence. In Python backslash \ is used as an escape character. Here is a list of escape sequences and their purpose.

| Escape sequence | Description | Example | Result |
|---|---|---|---|
| \a | Bell or alert | "\a" | Bell sound |
| \b | Backspace | "ab\bc" | ac |
| \f | Form feed | "hello\fworld" | hello world |
| \n | Newline | "hello\nworld" | Hello |
| \nnn | Octal notation, where n is in the range 0-7 | '\101' | A |
| \t | Tab | 'Hello\tPython' | Hello Python |
| \xnn | Hexadecimal notation, where n is in the range 0-9, a-f, or A-F | '\x41' | A |

**Python - List**

In Python, the list is a collection of items of different data types. It is an ordered sequence of items. A list object contains one or more items, not necessarily of the same type, which are separated by comma and enclosed in square brackets [].

Syntax: list= [value-1, value-2, value-3,....., value-N]

Example:

1) names= [“Jeff”, “Bill”, “Steve”, “Mohan”]
2) orderItem= [1, “Jeff”,56.8, True]

## Python - Tuple

Tuple is a collection of items of any Python data type, same as the list type. Unlike the list, tuple is immutable. The tuple object contains one or more items, of the same or different types, separated by comma and enclosed in parentheses ().

Syntax: list= (value-1, value-2, value-3,....., value-N)

Example:

1) names= (“Jeff”, “Bill”, “Steve”, “Mohan”)
2) orderItem= (1, “Jeff”,56.8, True)

It is however not necessary to enclose the tuple elements in parentheses. The tuple object can include elements separated by comma without parentheses.

Example:

names= “Jeff”, “Bill”, “Steve”, “Mohan”

## Python - Set

A set is a collection of data types in Python, same as the list and tuple. However, it is not an ordered collection of objects. The set is a Python implementation of the set in Mathematics. A set object has suitable methods to perform mathematical set operations like union, intersection, difference, etc.
A set object contains one or more items, not necessarily of the same type, which are separated by comma and enclosed in curly brackets { }.

Syntax: set = {value-1, value-2, value-3,...value-N}

Example: S1={1, "Bill", 75.50}

A set doesn't store duplicate objects. Even if an object is added more than once inside the curly brackets, only one copy is held in the set object. Hence, indexing and slicing operations cannot be done on a set object.

Example:

```
>>>S1={1,2,2,3,4,4,5,5}
>>>S1
{1, 2, 3, 4, 5}
```

## Python - Dictionary

Like the list and the tuple, dictionary is also a collection type. However, it is not an ordered sequence, and it contains key-value pairs. One or more key: value pairs separated by commas are put inside curly brackets to form a dictionary object.

Syntax: dict = { key1:value1, key2:value2,...keyN:valueN }

Example: capitals={"USA":"Washington, D.C.", "France":"Paris", "India":"New Delhi"}

## 2.5 Python - Variables

Any value of certain type is stored in the computer's memory for processing. Out of available memory locations, one is randomly allocated for storage. In order to conveniently and repeatedly refer to the stored value, it is given a suitable name. A value is bound to a name by the assignment operator '='.

**>>> myVar=21**

Here myVar is an identifier (name) referring to integer value 21 (Python treats data value as a value). However, the same identifier can be used to refer to another value. For example, the below code will assign myVar as the name of a string value, as shown below.

>>> myVar=" LBRCE"

So, the value being referred can change (or vary), hence it is called a variable. It is important to remember that a variable is a name given to a value, and not to a memory location storing the value.

Variables can be accessed by using their identifier (name) as shown below.

```
>>> myVar= "LBRCE"
>>> myVar
'LBRCE'
```

### 2.5.1 Dynamic Typing

One of the important features of Python is that it is a dynamically-typed language. Programming languages such as C, C++, Java, and C# are **statically-typed languages**. A variable in these languages is a user-friendly name given to a memory location and is intended to store the value of a particular data type.

A variable in Python is not bound permanently to a specific data type. Instead, it only serves as a label to a value of a certain type. Hence, the prior declaration of variable's data type is not possible. In Python, the data assigned to a variable decides its data type and not the other way around.

In the following Python statements, a string value is assigned to a variable 'name'. We can test its type using the type () function.

```
>>> myVar= "LBRCE"
>>>type(myVar)
<class 'str'>
```

Now the Python interpreter won't object if the same variable is used to store a reference to an integer.

```
>>> name=1234
>>> type(name)
<class 'int'>
```

We can see that the data type of a variable has now been changed to integer. This is why Python is called a **dynamically-typed language**.

### 2.5.2 Naming Conventions

Any suitable identifier can be used as a name of a variable, based on the following rules:

1. The name of the variable should start with either an alphabet letter (lower or upper case) or an underscore (_), but it cannot start with a digit.
2. More than one alpha-numeric characters or underscores may follow.
3. The variable name can consist of alphabet letter(s), number(s) and underscore(s) only. For example, myVar, MyVar, _myVar, MyVar123 are valid variable names but m*var, my-var, 1myVar are invalid variable names.
4. Identifiers in Python are case sensitive. So, NAME, name, nAME, and nAmE are treated as different variable names.

Starting the name with a single or double underscore has a special meaning in Python. More about this will be discussed in the chapter on object-oriented Programming.

## 2.6 Python assignment statements

Assignment statement is used to assign objects to names. The target of an assignment statement is written on the left side of the equal sign (=), and the object on the right can be an arbitrary expression that computes an object.

### 2.6.1 Assignment statement forms: -

1) **Basic form**: This form is the most commonly used form for assignment.

Example**:**
```
>>> str=" Hello"
>>> print(str)
Hello
```

**2) Tuple Assignment:**

```
>>> x,y=(50,100)
>>> print('x=',x)
x= 50
>>> print("y=",y)
y= 100
```

When we code a tuple on the left side of the =, Python pairs objects on the right side with targets on the left by position and assigns them from left to right. Therefore, the values of x and y are 50 and 100 respectively.

**3) List Assignment:** This works in the same way as the tuple assignment.

```
>>> x,y=[3,5]
>>> print('x=',x)
x= 3
>>> print("y=",y)
y= 5
```

4) **Sequence Assignment**: Any sequence of names can be assigned to any sequence of values, and Python assigns the items one at a time by position.

```
>>> a,b,c='hai'
>>> a
'h'
>>> b
'a'
>>> c
'i'
```

**5) Extended Sequence unpacking:** It allows us to be more flexible in how we select portions of a sequence to assign.

**>>> p, *q="Hello"**

Here, p is matched with the first character in the string on the right and q with the rest. The starred name (*q) is assigned a list, which collects all items in the sequence not assigned to other names.

```
>>> p
'H'
>>> q
['e', 'l', 'l', 'o']
```

**6) Multiple target assignment:** In this form, Python assigns a reference to the same object (the object which is rightmost) to all the target on the left.

```
>>> a=b=76
>>> print(a)
76
>>> print(b)
76
```

**7) Augmented assignment:** The augmented assignment is a shorthand assignment that combines an expression and an assignment.

```
>>> x=8
>>> x+=1
>>> print(x)
9
```

There are several other augmented assignment forms: -=, **=, &=, etc.

## 2.7 Input and Output Statements

A Program needs to interact with the user to accomplish the desired task; this is done using **Input-Output** facility. Input means the data entered by the user of the program. In python, we have input() function available for **Input**.

### 2.7.1 Input Statement:

**In** Python to read the data from the key board we have input() function.

**Syntax:** input (expression)

If prompt is present, it is displayed on monitor, after which the user can provide data from keyboard. Input takes whatever is typed from the keyboard and evaluates it. As the input provided is evaluated, it expects valid python expression. If the input provided is not correct then either syntax error or exception is raised by python.

```
Example: >>>x=input("Enter value for X")
          Enter value for X6
         >>> print("Entered value is",x)
         Entered value is 6
```

- The input( ) function is used to read data from keyboard.
- It reads data into string format.
- Programmer have to convert data into specific format before using them.
- To convert use the int, float and str functions
  - int(2.3) returns 2, int("123") returns 123
  - float(6) returns 6.0, float("3.14") returns 3.14
  - str(123) returns "123" , str(4.5) returns "4.5"
- If the conversion isn't possible, there will be an error.
  Example : int("abc"), int("two")

Example-1: Reading Single Input without Request Message

```
a=input()
Input: 10
```

Example-2: Reading Single Input with Request Message

```
a=input("Enter the value of a:")
Input: Enter the value of a:
10
```

Example-3: Reading Multiple Inputs in a Single Line Using Split( )

```
a=input( ).split(" ")
Input : 10 20 30 40 50
```

### 2.7.2 Output Statement

In Python, we have print() function for output. The print( ) function prints the specified text or value to the screen. print evaluates the expression before printing it on the monitor. Print statement outputs an entire (complete) line and

then goes to next line for subsequent output (s). To print more than one item on a single line, comma (,) may be used.

Syntax: print (expression/constant/variable)

**Example-1**: print("Hello World")
Output: Hello World

**Example-2**: Add A New Line or Vertical Space Between Two Outputs

```
print("Hello!\n")
print("Welcome to Python Programming")
```

**Output:** Hello!

Welcome to Python Programming

**Example-3:** OPTION KEYWORD ARGUMENT "sep= "

```
print("Welcome"," To","Python","Programming")
```

Output: Welcome To Python Programming

```
print ("Welcome"," To"," Python"," Programming", sep=" \n ")
```

Output: Welcome

To

Python

Programming

**Example-4:** Using Keyword Argument "end="

- "end= " is a string appended after the last value, defaults to a new line.
- It allows the programmer to define a custom ending character for each print call other than the default newline or \n.

```
print("Hello!",end=" ")
print("Welcome to python programming")
```

Output: Hello! Welcome to python programming

**Example-5:** Using Format Specifiers

```
a=10
b=2.5
print("%d is an integer and %f is a float."%(a,b))
```

Output: 10 is an integer and 2.5 is a float.

**Example-6:** Without Using Format Specifiers

```
a=10
```

b=2.5

print(a,"is an integer and",b,"is a float.")

Output : 10 is an integer and 2.5 is a float.

## 2.8 Python Identifiers:

An identifier is a name given to entities like class, functions, variables, etc. It helps to differentiate one entity from another.

**Rules for writing identifiers**

- Identifiers can be a combination of letters in lowercase **(a to z)** or uppercase **(A to Z)** or digits **(0 to 9)** or an underscore _.
- Names like myClass, var_1 and print_this_to_screen, all are valid example.
- An identifier cannot start with a digit. 1variable is invalid, but variable1 is a valid name.
- An identifier can be of any length.
- Keywords cannot be used as identifiers.

  Example: >>> global = 1
  Output: Syntax Error: invalid syntax

- We cannot use special symbols like !, @, **#**, **$**, **%** etc. in our identifier.

  Example: >>> a@ = 1
  Output: Syntax Error: invalid syntax

## 2.9 Python Keywords

- Keywords are the reserved words in Python.
- We cannot use a keyword as a variable name, function name or any other identifier. They are used to define the syntax and structure of the Python language.
- In Python, keywords are case sensitive.
- There are 33 keywords in Python 3.7.
- All the keywords except True, False and None are in lowercase and they must be written as they are.

| Keywords in Python | | | | |
|---|---|---|---|---|
| False | class | finally | is | return |
| None | continue | for | lambda | try |
| True | def | from | nonlocal | while |
| and | del | global | not | with |
| as | elif | if | or | yield |
| assert | else | import | pass | |
| break | except | in | raise | |

## 2.10 Python Literals

Literal is a raw data given in a variable or constant. In Python, there are various types of literals they are as follows:

### 2.10.1 Numeric Literals

Numeric Literals are immutable (unchangeable). Numeric literals can belong to 3 different numerical types: Integer, Float, and Complex.

**Example:**

```
a = 0b1010 #Binary Literals
b = 100 #Decimal Literal
c = 0o310 #Octal Literal
d = 0x12c #Hexadecimal Literal
#Float Literal
float_1 = 10.5
float_2 = 1.5e2
#Complex Literal
x = 3.14j
print(a, b, c, d)
print(float_1, float_2)
print(x, x.imag, x.real)
```

**Output:**
```
10 100 200 300
10.5 150.0
3.14j 3.14 0.0
```

### 2.10.2 String literals

A string literal is a sequence of characters surrounded by quotes. We can use both single, double, or triple quotes for a string. And, a character literal is a single character surrounded by single or double quotes.

**Example:**

```
strings = "This is Python"
char = "C"
multiline_str = """This is a
                    multiline string with
                        more than one-line code."""
unicode = u"\u00dcnic\u00f6de"
raw_str = r"raw \n string"
print(strings)
print(char)
print(multiline_str)
print(unicode)
print(raw_str)
```

**Output:**
```
This is Python
C
This is a
                    multiline string with
                        more than one-line code.
Ünicöde
raw \n string
```

### 2.10.3 Boolean literals

A Boolean literal can have any of the two values: True or False.

**Example:**

```
x = (1 == True)
y = (1 == False)
a = True + 4
b = False + 10
print("x is", x)
print("y is", y)
print("a:", a)
print("b:", b)
```

**Output:**
```
x is True
y is False
a: 5
b: 10
```

### 2.10.4 Special literals

Python contains one special literal i.e. None. We use it to specify that the field has not been created.

**Example:**

```
drink = "Available"
food = None
```

```
def menu(x):
    if x == drink:
        print(drink)
    else:
        print(food)
menu(drink)
menu(food)
```

**Output:** Available
None

### 2.10.5 Collection Literals

There are four different literal collections List literals, Tuple literals, Dictionary literals, and Set literals.

**Example:**

```
fruits = ["apple", "mango", "orange"] #list
numbers = (1, 2, 3) #tuple
alphabets = {'a':'apple', 'b':'ball', 'c':'cat'} #dictionary
vowels = {'a', 'e', 'i' , 'o', 'u'} #set

print(fruits)
print(numbers)
print(alphabets)
print(vowels)
```

**Output:** ['apple', 'mango', 'orange']
(1, 2, 3)
{'a': 'apple', 'b': 'ball', 'c': 'cat'}
{'o', 'i', 'a', 'e', 'u'}

## 2.11 Python Operators

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

For example:

```
>>> 2+3
5
```

Here, + is the operator that performs addition. 2 and 3 are the operands and 5 is the output of the operation. Python provides following operators.

1) Arithmetic Operators
2) Comparison Operators
3) Logical Operators
4) Bitwise Operators
5) Assignment Operators
6) Special Operators

## 2.11.1 Arithmetic operators

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

| Operator | Meaning | Example |
|---|---|---|
| + | Add two operands or unary plus | x + y+ 2 |
| - | Subtract right operand from the left or unary minus | x - y- 2 |
| * | Multiply two operands | x * y |
| / | Divide left operand by the right one (always results into float) | x / y |
| % | Modulus - remainder of the division of left operand by the right | x % y (remainder of x/y) |
| // | Floor division - division that results into whole number adjusted to the left in the number line | x // y |
| ** | Exponent - left operand raised to the power of right | x**y (x to the power y) |

**Example:**

```
x = 15
y = 4

print('x + y =',x+y)
```

**Output:** x + y = 19

```
print('x - y =',x-y)
```

**Output:** x - y = 11

```
print('x * y =',x*y)
```

**Output:** x * y = 60

```
print('x / y =',x/y)
```

**Output:** x / y = 3.75

```
print('x // y =',x//y)
```

**Output:** x // y = 3

```
print('x ** y =',x**y)
```

**Output:** x ** y = 50625

## 2.11.2 Comparison operators

Comparison operators are used to compare values. It returns either True or False according to the condition.

| Operator | Meaning | Example |
|---|---|---|
| > | Greater than - True if left operand is greater than the right | x > y |
| < | Less than - True if left operand is less than the right | x < y |
| == | Equal to - True if both operands are equal | x == y |
| != | Not equal to - True if operands are not equal | x != y |
| >= | Greater than or equal to - True if left operand is greater than or equal to the right | x >= y |
| <= | Less than or equal to - True if left operand is less than or equal to the right | x <= y |

**Example:**

```
x = 10
y = 12

# Output: x > y is False
print('x > y is',x>y)

# Output: x < y is True
print('x < y is',x<y)

# Output: x == y is False
print('x == y is',x==y)

# Output: x != y is True
print('x != y is',x!=y)

# Output: x >= y is False
print('x >= y is',x>=y)

# Output: x <= y is True
print('x <= y is',x<=y)
```

**Output:**

```
x > y is False
x < y is True
x == y is False
x != y is True
x >= y is False
x <= y is True
```

### 2.11.3 Logical operators

Logical operators are the and, or, not operators.

| Operator | Meaning | Example |
|---|---|---|
| and | True if both the operands are true | x and y |
| or | True if either of the operands is true | x or y |
| not | True if operand is false (complements the operand) | not x |

**Example:**
```
x = True
y = False

print('x and y is',x and y)
print('x or y is',x or y)
print('not x is',not x)
```

**Output:**
```
x and y is False
x or y is True
not x is False
```

### 2.11.4 Bitwise operators

Bitwise operators act on operands as if they were strings of binary digits. They operate bit by bit, hence the name.

In the table below: Let x = 10 (0000 1010 in binary) and y = 4 (0000 0100 in binary)

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

## 2.11.5 Assignment operators

- Assignment operators are used in Python to assign values to variables.
- a = 5 is a simple assignment operator that assigns the value 5 on the right to the variable a on the left.
- There are various compound operators in Python like a += 5 that adds to the variable and later assigns the same. It is equivalent to a = a + 5.

| Operator | Example | Equivalent to |
|---|---|---|
| = | x = 5 | x = 5 |
| += | x += 5 | x = x + 5 |
| -= | x -= 5 | x = x - 5 |
| *= | x *= 5 | x = x * 5 |
| /= | x /= 5 | x = x / 5 |
| %= | x %= 5 | x = x % 5 |
| //= | x //= 5 | x = x // 5 |
| **= | x **= 5 | x = x ** 5 |
| &= | x &= 5 | x = x & 5 |
| \|= | x \|= 5 | x = x \| 5 |
| ^= | x ^= 5 | x = x ^ 5 |
| >>= | x >>= 5 | x = x >> 5 |
| <<= | x <<= 5 | x = x << 5 |

## 2.11.6 Special operators

Python language offers some special types of operators like the identity operator or the membership operator. They are described below with examples.

**1) Identity operators**

- is and is not are the identity operators in Python. They are used to check if two values (or variables) are located on the same part of the memory. Two variables that are equal does not imply that they are identical.

| Operator | Meaning | Example |
|---|---|---|
| is | True if the operands are identical (refer to the same object) | x is True |
| is not | True if the operands are not identical (do not refer to the same object) | x is not True |

**Example:**

```
x1 = 5
y1 = 5
x2 = 'Hello'
y2 = 'Hello'
x3 = [1,2,3]
y3 = [1,2,3]

# Output: False
print(x1 is not y1)
# Output: True
print(x2 is y2)
# Output: False
print(x3 is y3)
```

**Output:**

```
False
True
False
```

Here, we see that x1 and y1 are integers of the same values, so they are equal as well as identical. Same is the case with x2 and y2 (strings). But x3 and y3 are lists. They are equal but not identical. It is because the interpreter locates them separately in memory although they are equal.

## 2) Membership operators

- in and not in are the membership operators in Python. They are used to test whether a value or variable is found in a sequence (string, list, tuple, set and dictionary).

In a dictionary we can only test for presence of key, not the value.

| Operator | Meaning | Example |
|---|---|---|
| in | True if value/variable is found in the sequence | 5 in x |
| not in | True if value/variable is not found in the sequence | 5 not in x |

**Example:**

```
x = 'Hello world'
y = {1:'a',2:'b'}

# Output: True
print('H' in x)

# Output: True
print('hello' not in x)

# Output: True
print(1 in y)

# Output: False
print('a' in y)
```

**Output:**

```
True
True
True
False
```

Here, 'H' is in x but 'hello' is not present in x (remember, Python is case sensitive). Similarly, 1 is key and 'a' is the value in dictionary y. Hence, 'a' in y returns False.

## 2.12 Precedence of Python Operators

The combination of values, variables, operators, and function calls is termed as an expression. The Python interpreter can evaluate a valid expression.

For example: >>> 5-7

-2

Here 5 - 7 is an expression. There can be more than one operator in an expression. To evaluate these types of expressions there is a rule of precedence in Python. It guides the order in which these operations are carried out.

For example, multiplication has higher precedence than subtraction.

>>>10-4*2

2

| Operator | Description |
|---|---|
| ** | Exponentiation (raise to the power) |
| ~ + - | Complement, unary plus and minus (method names for the last two are +@ and -@) |
| * / % // | Multiply, divide, modulo and floor division |
| + - | Addition and subtraction |
| >> << | Right and left bitwise shift |
| & | Bitwise 'AND' |
| ^ \| | Bitwise exclusive `OR' and regular `OR' |
| <= < > >= | Comparison operators |
| <> == != | Equality operators |
| = %= /= //= -= += *= **= | Assignment operators |
| is is not | Identity operators |
| in not in | Membership operators |
| not or and | Logical operators |

Operator precedence affects how an expression is evaluated.

For example, x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has higher precedence than +, so it first multiplies 3*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom.

# UNIT-II

## Control Structures and Lists

1. **Conditional Statements:** By default, statements in the script are executed sequentially from the first to the last. If the processing logic requires so, the sequential flow can be altered in two ways:

   - **Conditional execution:** a block of one or more statements will be executed if a certain expression is true.
   - **Repetitive execution:** a block of one or more statements will be repetitively executed as long as a certain expression is true.

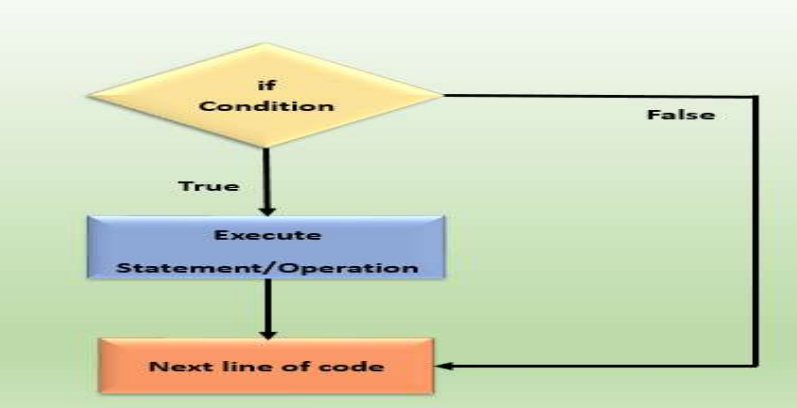
   Python supports the 3 types of condition statements. Those are
   - Simple if
   - If-else
   - If-elif-else ladder

**1.1 Simple if:** This is also called as one-way selection statement. Python uses the if keyword to implement decision control. Python's syntax for executing a block conditionally is as below:

**Syntax:**

```
if [boolean expression]:
    statement1
    statement2
    ...
    statement
```

**Flow-chart:**




Any Boolean expression evaluating to True or False appears after the if keyword. Use the: symbol and press Enter after the expression to start a block with increased indent. One or more statements written with the same level of indent will be executed if the Boolean expression evaluates to True.

To end the block, decrease the indentation. Subsequent statements after the block will be executed out of the if condition. The following example demonstrates the if condition.

**Example:**
```
>>> if 10<100:
        print ("10 is less than 100")
```

In the above example, the expression 10<100 evaluates to True, so it will execute the block. The if block starts from new line after:. All the statements under the if condition start with an increased indentation. Above, if block contains only one statement.

**Example:** write a Python program that calculates the amount payable from price and quantity inputs by the user and applies a 10% discount if the amount exceeds 1000.

Calculation and application of the discount is to be done only if the amount is greater than 1000, hence, the process is placed in a block with increased indent, following the conditional expression.

The print() statement is written after the conditional block is over, hence, it will be executed if the expression is false (i.e. the amount is not greater than 1000), as well as after applying the discount if the expression is true (i.e. the amount is greater than 1000).

```
price=int(input("Enter Price: "))
qty=int(input("Enter Quantity: "))
amt=price*qty
if amt>1000:
    print ("10% discount is applicable")
    discount=amt*10/100
    amt=amt-discount
print ("Amount payable: ",amt)
```

**Output:**
```
Enter Price: 100
Enter Quantity: 20
10% discount is applicable
Amount payable:  1800.0
```

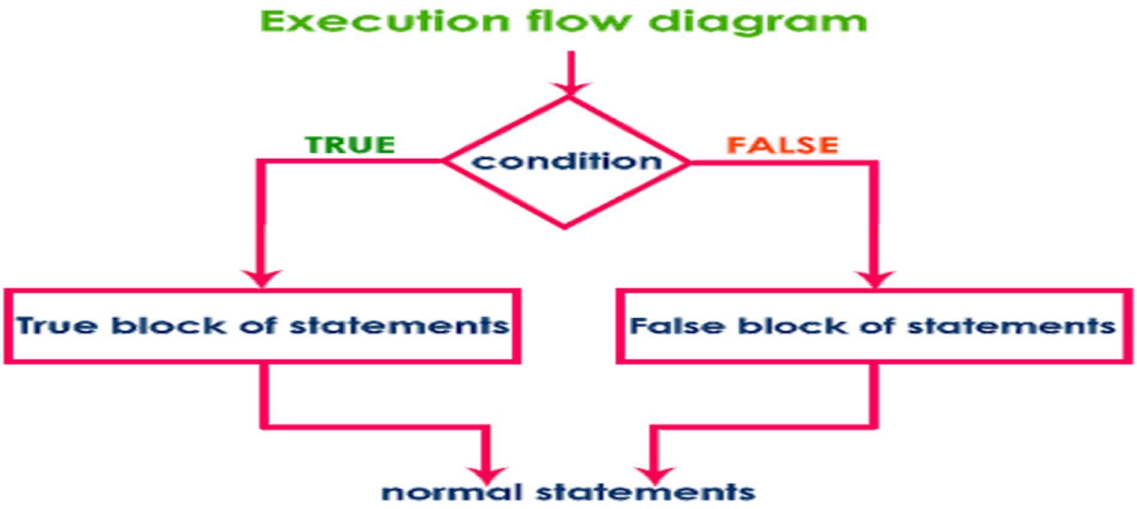
**1.2 if-else Statement:** This is also called as two-way selection statement. Along with the if statement, the else condition can be optionally used to define an alternate block of statements to be executed if the Boolean expression in the if condition is not true.

**Syntax:**
```
if [Boolean expression]:
        statement1
        statement2
        ...
```

```
        statementN
else:
        statement1
        statement2
        ...
        statementN
```

The indented block starts after the : symbol, i.e., after the Boolean expression. It will get executed when the condition is true. We have another block that should be executed when the if condition is false.

**Flow-chart:**

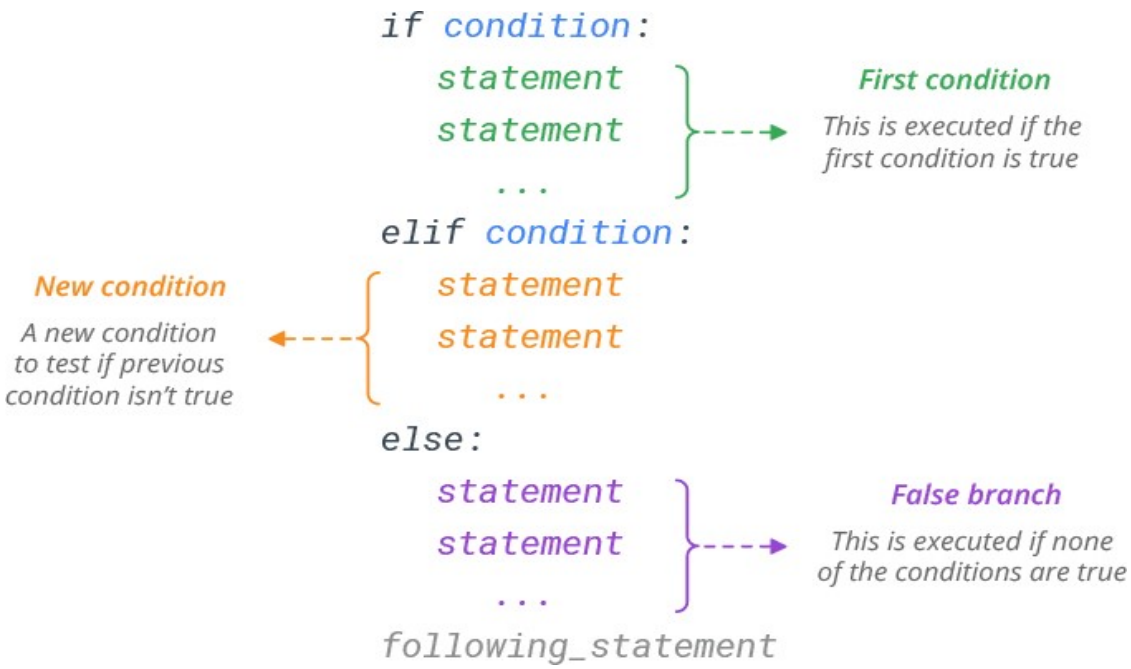



**Example:**

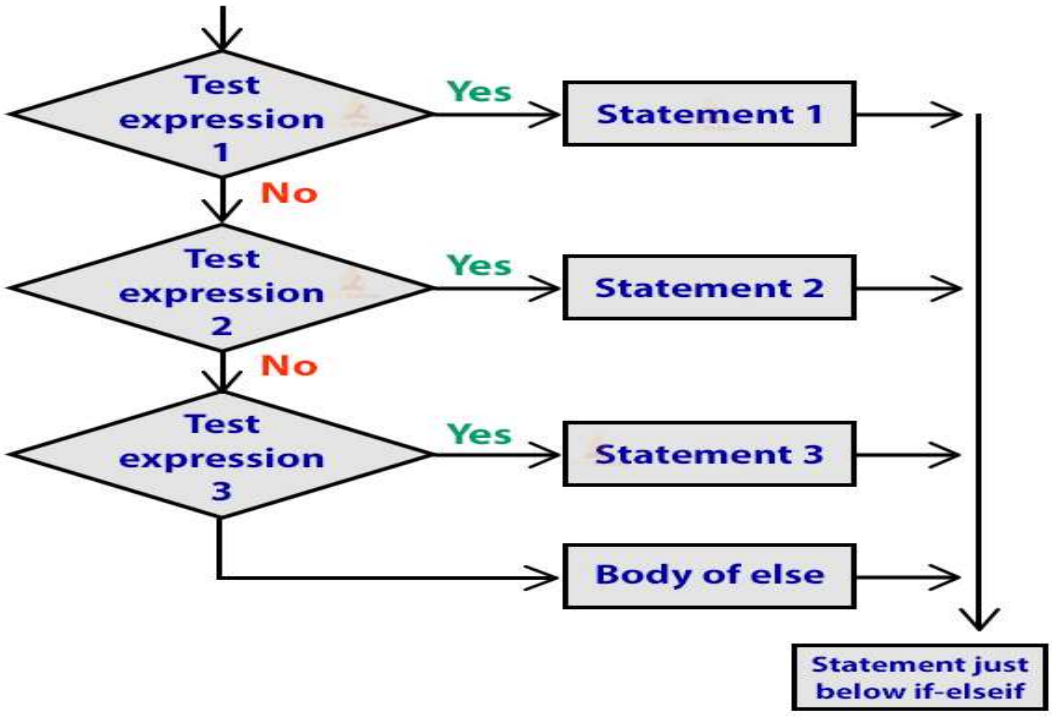
```
num = int(input("Enter a number: "))
mod = num % 2
if mod > 0:
        print("This is an odd number.")
else:
        print("This is an even number.")
```

**1.3 if-elif-else Statement:** Use the elif condition is used to include multiple conditional expressions between if and else. We can include as many number of elif statements based on the requirement.

**Syntax:**




**Flow-chart:**




**Example:**

```
x=10
if x==1:
    print('X is 1')
elif x==5:
    print('X is 5')
elif x==10:
```

```
    print('X is 10')
else:
    print('X is something else')
```

**1.4** **Nested if-else statements:** In this statements we can include if-else or if statements in another if-else. In this type of statements, we will check more number of conditions.

**Syntax:**

```
if   Test_expression_1:
        statement (s)
        if Test_expression_2:
                statement(s)
        else:
                statement(s)
else:
        statement(s)
```

**Example:**

```
#Program to check whether the given year is leap year or not
year=int(input("Enter the year "))
if year%4==0:
    if year%100==0:
        if year%400==0:
            print("Leap year")
        else:
            print("Not a leap year")
    else:
        print("Leap year")
else:
    print("Not a leap year")
```

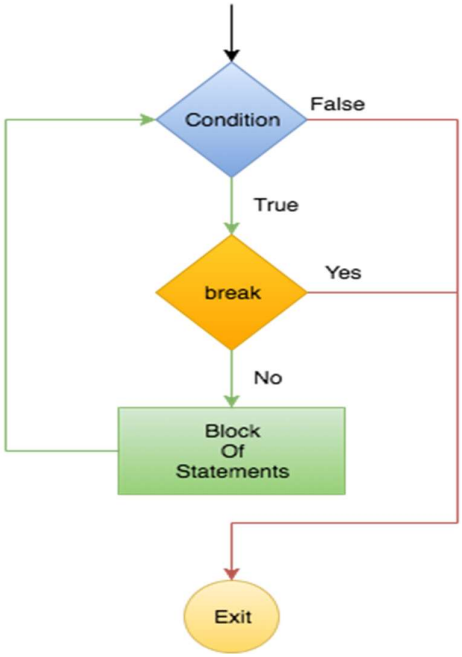
**2. Jumping Statements:** Jump statements in python are used to alter the flow of a loop like you want to skip a part of a loop or terminate a loop. Basically Python provides 3 types of jumping statements. Those are

- break
- continue
- pass

**2.1 break:** break Statement in Python is used to terminate the loop. Break statement can be used in loops only.

**Syntax of break Statement:** break

**Flow-chart:**




**Example:**

```
for i in range(10):
    print(i)
    if(i == 3):
        print('break')
        break
```

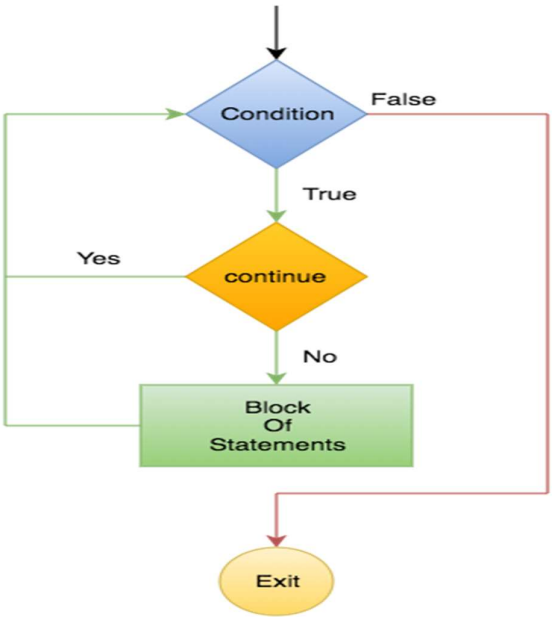
**Output:**

```
0
1
2
break
```

**2.2 continue:** continue Statement in Python is used to skip all the remaining statements in the loop and move controls back to the top of the loop.

**Syntax for continue:** continue

**Flow-chart for continue:**




**Example:**

```
for i in range(6):
    if(i==3):
        continue
    print(i)
```

**Output:**

```
0
1
2
4
5
```

when 'i' is equal to 3 continue statement will be executed which skip the print statement.

**2.3 pass:** pass Statement in Python does nothing. You use pass statement when you create a method that you don't want to implement, yet.

**Syntax:** pass

**Example without using pass statement:**

```
def myMethod():
print('hello')
```

**Output:**

```
Traceback (most recent call last):
File "python", line 3
  print('hello')
Indentation Error: expected an indented block
```

**Example using pass statement:**

```
def myMethod():
    pass
print('hello')
```

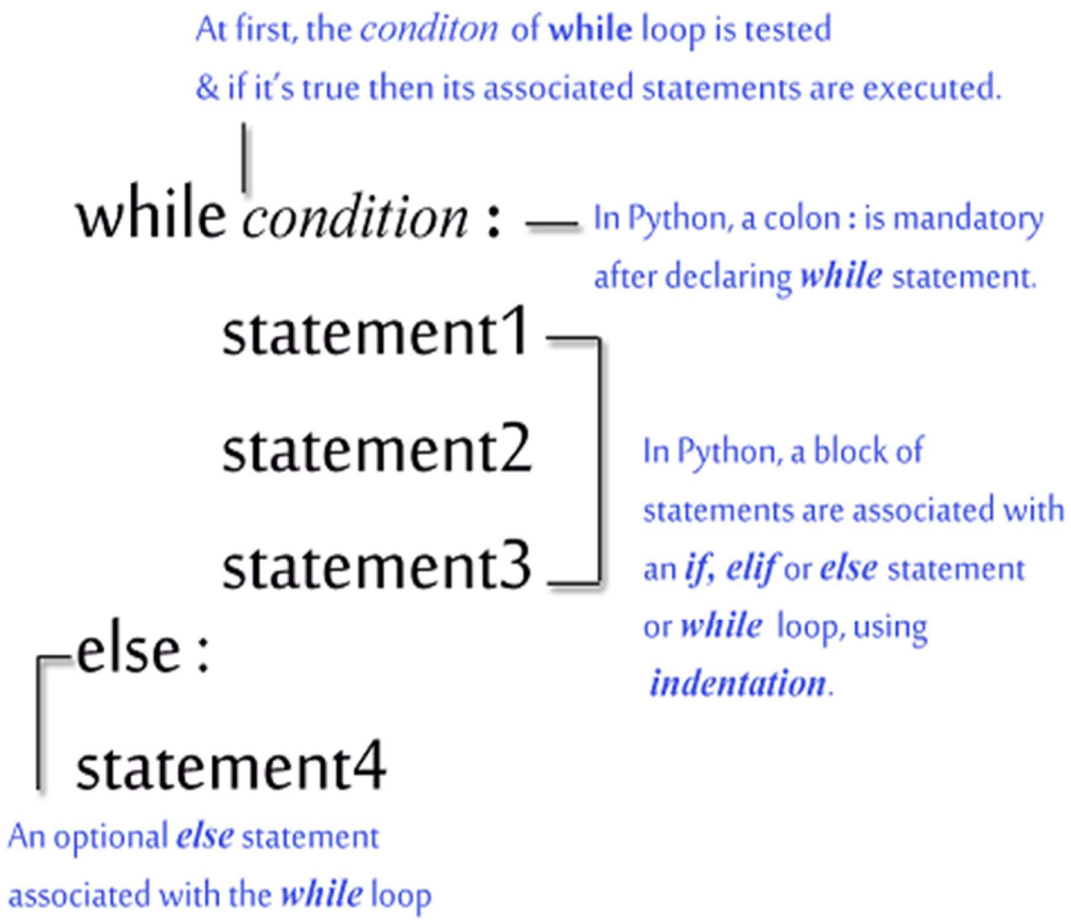
**Output:** hello

3. **Python Loops:** Loops are used to execute one or more number of statements multiple number of times based on the given condition. In this the statements are executed until the condition false. Python supports 2 types of loops. Those are

   - **while loop**
   - **for loop**

**3.1 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 the number of times to iterate beforehand.

**Syntax:**

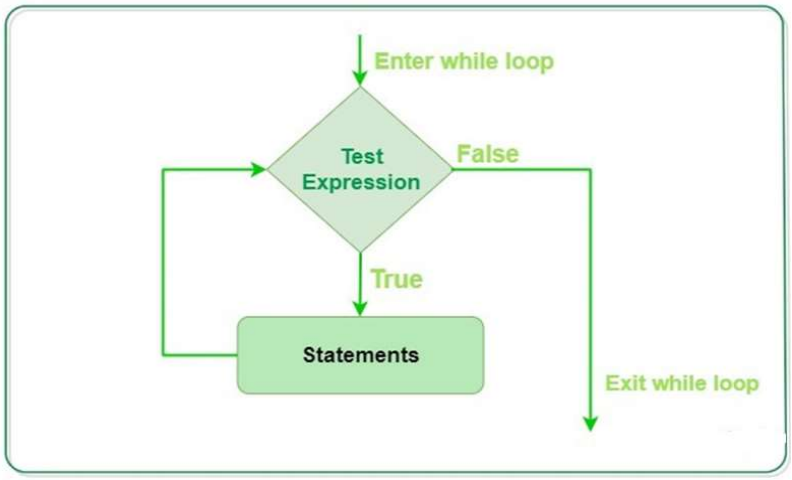



- In the while loop, condition or test_expression is checked first. The body of the loop is entered only if the test_expression evaluates to True. After one

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.
- The 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.

**Flow-chart for while loop:**



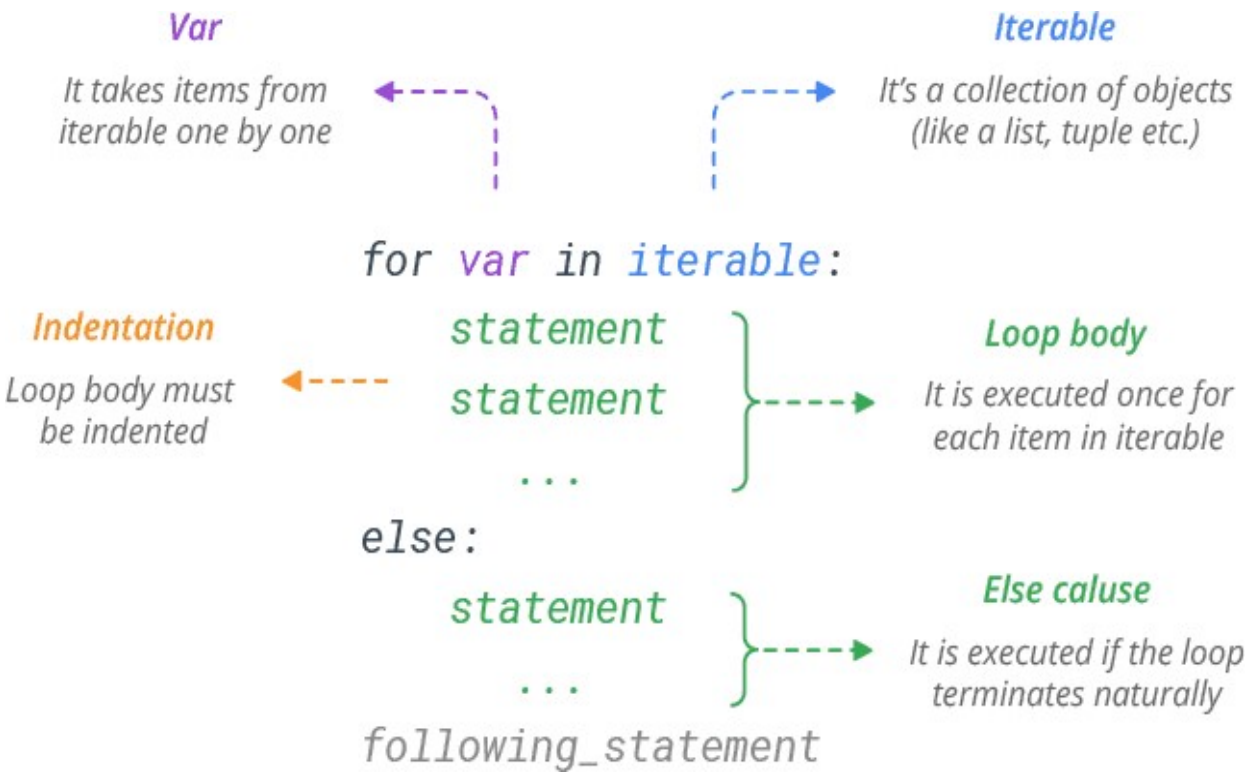

**Example:**

```
# Program to add natural numbers up to n
# sum = 1+2+3+...+n
n = int(input("Enter n: "))        # To take input from the user,
sum = 0                            # initialize sum and counter
i = 1
while i <= n:
    sum = sum + i
    i = i+1          # update counter
print("The sum is", sum) # print the sum
```

**Output:** Enter n: 10
The sum is 55

**3.2 for loop:** 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 for for loop:**




**Flow-chart:**

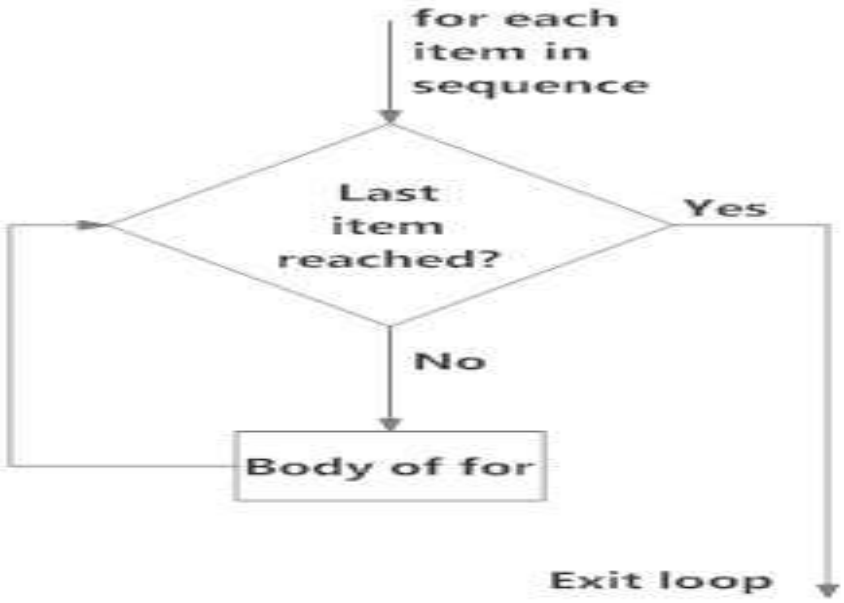



Fig: operation of for loop

**Example:** # Program to find the sum of all numbers stored in a list

```
numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]
sum = 0        # variable to store the sum
for val in numbers:                # iterate over the list
        sum = sum+val
print("The sum is", sum)
```

**Output:** The sum is 48

**3.3 Python nested loops:** Python programming language allows to use one loop inside another loop.

**Syntax:**

```
for iterating_var in sequence:
   for iterating_var in sequence:
      statements(s)
   statements(s)
```

The syntax for a **nested while loop** statement in Python programming language is as follows −

```
while expression:
   while expression:
      statement(s)
   statement(s)
```

**Note:** you can put any type of loop inside of any other type of loop. For example, a for loop can be inside a while loop or vice versa.

**Example:** The following program uses a nested while loop to find the prime numbers from 2 to 100.

```
i = 2
while(i < 10):
   j = 2
   while(j <= (i/j)):
      if not(i%j): break
      j = j + 1
   if (j > i/j) : print( i, " is prime")
   i = i + 1

print "Good bye!"
```

**Output:**

```
2 is prime
3 is prime
5 is prime
7 is prime
Good bye!
```

4. **Mathematical Functions:** Python supports several mathematical functions and these functions are available in **math** module. The **math** module is a standard module in Python and is always available. To use mathematical functions under this module, you have to import the module using import math.

**List of Mathematical Functions:**

| Function | Description |
|---|---|
| ceil(x) | Returns the smallest integer greater than or equal to x. |
| copysign(x, y) | Returns x with the sign of y |
| fabs(x) | Returns the absolute value of x |
| factorial(x) | Returns the factorial of x |
| floor(x) | Returns the largest integer less than or equal to x |
| fmod(x, y) | Returns the remainder when x is divided by y |
| frexp(x) | Returns the mantissa and exponent of x as the pair (m, e) |
| fsum(iterable) | Returns an accurate floating point sum of values in the iterable |

| | |
|---|---|
| isfinite(x) | Returns True if x is neither an infinity nor a NaN (Not a Number) |
| isinf(x) | Returns True if x is a positive or negative infinity |
| isnan(x) | Returns True if x is a NaN |
| trunc(x) | Returns the truncated integer value of x |
| exp(x) | Returns e**x |
| log(x[, base]) | Returns the logarithm of x to the base (defaults to e) |
| pow(x, y) | Returns x raised to the power y |
| sqrt(x) | Returns the square root of x |
| acos(x) | Returns the arc cosine of x |
| asin(x) | Returns the arc sine of x |
| atan(x) | Returns the arc tangent of x |
| cos(x) | Returns the cosine of x |
| sin(x) | Returns the sine of x |
| tan(x) | Returns the tangent of x |
| degrees(x) | Converts angle x from radians to degrees |
| radians(x) | Converts angle x from degrees to radians |
| pi | Mathematical constant, the ratio of circumference of a circle to it's diameter (3.14159...) |
| e | mathematical constant e (2.71828...) |
| inf | Mathematical constant to represent infinite value |
| nan | Mathematical constant to represent not a number value |

**4.1 ceil() method:** The method ceil() in Python returns ceiling value of x i.e., the smallest integer not less than x.

**Syntax:**

import math
math.ceil(x)

**Parameter:**

x: This is a numeric expression.

**Returns:**

Smallest integer not less than x.

**Example:**

```
import math    # This will import math module
# prints the ceil using ceil() method
print( "math.ceil(-23.11) : ", math.ceil(-23.11) )
print ("math.ceil(300.16) : ", math.ceil(300.16) )
print ("math.ceil(300.72) : ", math.ceil(300.72) )
```

**Output:**

```
math.ceil(-23.11) :  -23.0
math.ceil(300.16) :  301.0
math.ceil(300.72) :  301.0
```

**4.2 floor() method:** floor() method in Python returns floor of x i.e., the largest integer not greater than x.

**Syntax:**

import math
math.floor(x)

**Parameter:**

x-numeric expression.

**Returns:**

largest integer not greater than x.

**Example:**

```
import math        # This will import math module
# prints the ceil using floor() method
print "math.floor(-23.11) : ", math.floor(-23.11)
print "math.floor(300.16) : ", math.floor(300.16)
print "math.floor(300.72) : ", math.floor(300.72)
```

**Output:**

```
math.floor(-23.11) :  -24.0
math.floor(300.16) :  300.0
math.floor(300.72) :  300.0
```

**4.3 copysign () method:** math.copysign() is a function exists in Standard math Library of Python. This function returns a float value consisting of magnitude from parameter x and the sign (+ve or -ve) from parameter y.

**Syntax:** math.copysign(x, y)

**Parameters :**

x : Integer value to be converted

y : Integer whose sign is required

**Returns:** float value consisting of magnitude from parameter x and the sign from parameter y.

**Example:**

```
# Python code to demonstrate copy.sign() function
import math
a = 5
b = -7
c = math.copysign(a, b)  # implementation of copysign
print (c)
```

**Output:** 5.0

**4.4 Python | fabs() vs abs():** Both the abs() and the fabs() function is used to find the absolute value of a number.

**Syntax of abs():** abs(number)

**Syntax of fabs():** math.fabs(number)

Both will return the absolute value of number.

The difference is that math.fabs(number) will always return a floating point number even if the argument is integer, whereas abs() will return a floating point or an integer depending upon the argument.

**Example:**

```
import math
number = -12.08
print(abs(number))
print(math.fabs(number))
```

**Output:**

```
12.08
12.08
```

**4.5 factorial(x):** This method is used to find out the factorial of the given value.

**Syntax:** factorial(number)

**Example:**

```
import math
n=int(input("Enter the value for n: "))
print(math.factorial(n))
```

**Output:**

```
Enter the value for n: 5
120
```

**4.6 Python | fmod() function:** fmod() function is one of the Standard math library function in Python, which is used to calculate the Module of the specified given arguments.

**Syntax:** math.fmod( x, y )

**Parameters:**

x any valid number (positive or negative).

y any valid number(positive or negative).

**Returns:** Return a floating point number value after calculating module of given parameters x and y.

**Example:**

```
# Python3 program to demonstrate errors in fmod() function
import math
print(math.fmod(0, 0))          # will give ValueError
print(math.fmod(2, 0))          # will give ValueError
print(math.fmod(10,2))
print(math.fmod('2', 3))        # it will give TypeError
```

**Output:**

```
ValueError: math domain error

ValueError: math domain error

0

TypeError: a float is required
```

**4.7 Python | frexp() Function:** frexp() function is one of the Standard math Library function in Python. It returns mantissa and exponent as a pair (m, e) of a given value x, where mantissa **m** is a floating point number and **e** exponent is an integer value. **m** is a float and **e** is an integer such that **x == m * 2**e** exactly.

If x is zero, returns (0.0, 0), otherwise 0.5 <= abs(m) < 1.

**Syntax:** math.frexp( x )

**Parameters:** Any valid number (positive or negative).

**Returns:** Returns mantissa and exponent as a pair (m, e) value of a given number x.

**Exception:** If **x** is not a number, function will return TypeError.

**Example:**

```
# importing math library
import math
print(math.frexp(3)) # calculating mantissa and exponent of given integer
print(math.frexp(15.7))
print(math.frexp(-15))
```

**Output:**

(0.75, 2)

(0.98125, 4)

(-0.9375, 4)

**4.8 Python | fsum() function**: fsum() is inbuilt function in Python, used to find sum between some range or an iterable. To use this function we need to import the math library.

**Syntax :** maths.fsum( iterable )
**Parameter : iterable :** Here we pass some value which is iterable like arrays, list.
**Return Type :** The function return a floating point number.
**Example:**

```
# Python code to demonstrate use of math.fsum() function
# fsum() is found in math library
import math
print(math.fsum(range(10)))
arr = [1, 4, 6]            # Integer list
print(math.fsum(arr))
arr = [2.5, 2.4, 3.09]     # Floating point list
print(math.fsum(arr))
```

Output :

45.0

11.0

7.99

**4.9 python | isfinite() function:** The math.isfinite() method checks whether a value is finite, or not. This method returns a Boolean value: True if the value is finite, otherwise False.

**Syntax :** math.isfinite(x)

**Parameter Values:** x. The value to check if it is finite or not.

**Return value:** A bool value, True if the value is finite, otherwise False.

**Example:**

```
# Import math Library
import math
# Check whether some values are finite
```

```
print (math.isfinite (56))
print (math.isfinite (-45.34))
print (math.isfinite (+45.34))
print (math.isfinite (math.inf))
print (math.isfinite (float("nan")))
print (math.isfinite (float("inf")))
print (math.isfinite (float("-inf")))
print (math.isfinite (-math.inf))
```

**Output:**

```
True
True
True
False
False
False
False
False
```

**4.10 python | isinf() function:** The math.isinf() method checks whether a value is positive or negative infinity, or not. This method returns a Boolean value: True if the value is infinity, otherwise False.

**Syntax:** math.isinf(x)

**Parameter Values :** x . The value to check for infinity.

**Return value:** A bool value, True if the value is finite, otherwise False.

**Example:**

```
# Import math Library
import math
# Check whether some values are infinite
print (math.isinf (56))
print (math.isinf (-45.34))
print (math.isinf (+45.34))
print (math.isinf (math.inf))
print (math.isinf (float("nan")))
print (math.isinf (float("inf")))
print (math.isinf (float("-inf")))
print (math.isinf (-math.inf))
```

**Output:**

```
False
False
False
True
False
True
```

```
True
True
```

**4.11 Python math library | isnan() method:** Python has math library and has many functions regarding it. One such function is isnan(). This method is used to check whether a given parameter is a valid number or not.

**Syntax :** math.isnan(x)
**Parameters : x** [Required] : It is any valid python data type or any number.
**Returns:** Return type is boolean.
-> Returns **False** if the given parameter is any number(positive or negative)
-> Returns **True** if given parameter is NaN (Not a Number).

**Example:**

```
import math
test_int = 4
test_neg_int = -3
test_float = 0.00
print (math.isnan(test_int))
print (math.isnan(test_neg_int))
print (math.isnan(test_float))
```

**Output:**

```
False

False

False
```

**4.12 trunc() in Python: truncate** behaves as a ceiling function for negative number and floor function for positive number.

**Example:**

```
# Python program to show output of floor(), ceil()  truncate() for a positive
number.
import math
print math.floor(3.5) # floor
print math.trunc(3.5) # work as floor
print math.ceil(3.5)  # ceil
```

**Output:**

```
3.0

3

4.0
```

**4.13 Python math library | exp() method:** Python has math library and has many functions regarding it. One such function is exp(). This method is used to calculate the power of e i.e. e^y or we can say exponential of y. The value of e is approximately equal to 2.71828.....

**Syntax :** math.exp(y)
**Parameters :** y [Required] – It is any valid python number either positive or negative. Note that if y has value other than number then its return error.
**Returns:** Returns floating point number by calculating e^y.
**Example:**

```
# Python3 code to demonstrate the working of exp()
import math
# initializing the value
test_int = 4
test_neg_int = -3
test_float = 0.00
# checking exp() values with different numbers
print (math.exp(test_int))
print (math.exp(test_neg_int))
print (math.exp(test_float))
```

**Output:**

54.598150033144236

0.049787068367863944

1.0

**4.14 Log functions in Python:** Python offers many inbuild logarithmic functions under the module "**math**" which allows us to compute logs using a single line. There are 4 variants of logarithmic functions.

**1. log(a,(Base)) :** This function is used to compute the **natural logarithm** (Base e) of a. If 2 arguments are passed, it computes the logarithm of desired base of argument a, numerically value of **log(a)/log(Base)**.

**Syntax:** math.log(a,Base)
**Parameters: a:** The numeric value
**Base:** Base to which the logarithm has to be computed.
**Return Value:** Returns natural log if 1 argument is passed and log with specified base if 2 arguments are passed.
**Exceptions:** Raises Value Error is a negative no. is passed as argument.

```
# Python code to demonstrate the working of   log(a,Base)
import math
# Printing the log base e of 14
print ("Natural logarithm of 14 is : ", end="")
print (math.log(14))
# Printing the log base 5 of 14
print ("Logarithm base 5 of 14 is : ", end="")
print (math.log(14,5))
```

**Output :**

Natural logarithm of 14 is : 2.6390573296152584

Logarithm base 5 of 14 is : 1.6397385131955606

**2. log2(a) :** This function is used to compute the **logarithm base 2** of a. Displays more accurate result than log(a,2).

**Syntax :** math.log2(a)

**Parameters : a :** The numeric value

**Return Value :** Returns logarithm base 2 of a

**Exceptions :** Raises ValueError is a negative no. is passed as argument.

**Example:**

```
# Python code to demonstrate the working of  log2(a)
 import math
# Printing the log base 2 of 14
print ("Logarithm base 2 of 14 is : ", end="")
print (math.log2(14))
```

**Output :**

Logarithm base 2 of 14 is : 3.807354922057604

**3. log10(a) :** This function is used to compute the **logarithm base 10** of a. Displays more accurate result than log(a,10).

**Syntax :** math.log10(a)

**Parameters : a :** The numeric value

**Return Value :** Returns logarithm base 10 of a

**Exceptions :** Raises ValueError is a negative no. is passed as argument.

**Example:**

```
# Python code to demonstrate the working of  log10(a)
import math
# Printing the log base 10 of 14
print ("Logarithm base 10 of 14 is : ", end="")
print (math.log10(14))
```

**Output :** Logarithm base 10 of 14 is : 1.146128035678238

**4. log1p(a) :** This function is used to compute **logarithm(1+a)** .

**Syntax :** math.log1p(a)

**Parameters : a :** The numeric value
**Return Value :** Returns log(1+a)
**Exceptions :** Raises ValueError is a negative no. is passed as argument.

**Example:**

```
# Python code to demonstrate the working of log1p(a)
import math
# Printing the log(1+a) of 14
print ("Logarithm(1+a) value of 14 is : ", end="")
print (math.log1p(14))
```

**Output :** Logarithm(1+a) value of 14 is : 2.70805020110221

**4.15 pow() in Python:** Python offers to compute the power of a number and hence can make task of calculating power of a number easier. It has many-fold applications in day to day programming.

**1. float pow(x,y) :** This function **computes x**y**. This function first converts its arguments into float and then computes the power.

**Declaration :** float pow(x,y)
**Parameters : x :** Number whose power has to be calculated.
**y :** Value raised to compute power.
**Return Value :** Returns the value x**y in float.

**Example:**

```
# Python code to demonstrate pow()  version 1
print ("The value of 3**4 is : ",end="")
print (pow(3,4))    # Returns 81
```

**Output :** The value of 3**4 is : 81.0

**2. float pow(x,y,mod) :** This function **computes (x**y) % mod**. This function first converts its arguments into float and then computes the power.

**Declaration :** float pow(x,y,mod)
**Parameters : x :** Number whose power has to be calculated.
**y:** Value raised to compute power.
**mod :** Value with which modulus has to be computed.
**Return Value :** Returns the value (x**y) % mod in float.

**Example:**

```
# Python code to demonstrate pow()  version 2
print ("The value of (3**4) % 10 is : ",end="")
# Returns 1    # Returns 81%10
print (pow(3,4,10))
```

**Output :** The value of (3**4) % 10 is : 1

**Implementation Cases in pow() :**

| X | Y | Return Value |
|---|---|---|
| Non-negative | Non-negative | Integer |
| Non-negative | Negative | Float |
| Negative | Non-Negative | Integer |
| Negative | Negative | Float |

**4.16 Python math function | sqrt():**

**sqrt()** function is an inbuilt function in Python programming language that returns the square root of any number.

**Syntax: math.sqrt(x)**
**Parameter:** x is any number such that x>=0
**Returns:** It returns the square root of the number passed in the parameter.

**Example:**
```
# Python3 program to demonstrate the sqrt() method
 import math
print(math.sqrt(0))        # print the square root of  0
print(math.sqrt(4))        # print the square root of 4
print(math.sqrt(3.5))      # print the square root of 3.5
```

**Output:**

```
0.0
2.0
1.8708286933869707
```

**4.17 Python – math.acos() function:** **Math module** contains a number of functions which is used for mathematical operations. The **math.acos()** function returns the arc cosine value of a number. The value passed in this function should be in between **-1 to 1**.

**Syntax:** math.acos(x)
**Parameter:** This method accepts only single parameters.

- **x :** This parameter is the value to be passed to acos()

**Returns:** This function returns the arc cosine value of a number.
**Example:**

```
# Python code to implement # the acos()function
# importing "math" for mathematical operations
import math
a = math.pi / 6
# returning the value of arc cosine of pi / 6
print ("The value of arc cosine of pi / 6 is : ", end ="")
```

```
print (math.acos(a))
```

**Output:** The value of arc cosine of pi / 6 is : 1.0197267436954502

**4.18 Python – math.asin() function:** **Math module** contains a number of functions which is used for mathematical operations. The **math.asin()** function returns the arc sine value of a number. The value passed in this function should be **between -1 to 1**.

**Syntax:** math.asin(x)
**Parameter:** This method accepts only single parameters.

- **x:** This parameter is the value to be passed to asin()

**Returns:** This function returns the arc sine value of a number.
**Example:**

```
# Python code to implement the acos()function
# importing "math" for mathematical operations
import math
 a = math.pi / 6
# returning the value of arc sine of pi / 6
print ("The value of arc sine of pi / 6 is : ", end ="")
print (math.asin(a))
```

**Output:** The value of arc sine of pi / 6 is : 0.5510695830994463

**4.19 Python – math.atan() function:** **Math module** contains a number of functions which is used for mathematical operations. The **math.atan()** function returns the arctangent of a number as a value. The value passed in this function should be **between -PI/2 and PI/2** radians.

**Syntax:** math.atan(x)
**Parameter:** This method accepts only single parameters.

- **x: This** parameter is the value to be passed to atan()

**Returns:** This function returns the arctangent of a number as a value.
**Example:**

```
# Python code to implement the atan()function
# importing "math" for mathematical operations
import math
a = math.pi / 6
# returning the value of arctangent  of pi / 6
print ("The value of tangent of pi / 6 is : ", end ="")
print (math.atan(a))
```

**Output:** The value of tangent of pi / 6 is : 0.48234790710102493

**4.20 Python | math.sin() function:** In Python, math module contains a number of mathematical operations, which can be performed with ease using the module. **math.sin()** function returns the sine of value passed as argument. The value passed in this function should be in radians.

**Syntax:** math.sin(x)
**Parameter:** x : value to be passed to sin()
**Returns:** Returns the sine of value passed as argument

**Example:**

```
# Python code to demonstrate the working of sin()
# importing "math" for mathematical operations
import math
a = math.pi / 6
# returning the value of sine of pi / 6
print ("The value of sine of pi / 6 is : ", end ="")
print (math.sin(a))
```

**Output:** The value of sine of pi/6 is : 0.49999999999999994

**4.21 Python | math.cos() function:** In Python, math module contains a number of mathematical operations, which can be performed with ease using the module. **math.cos()** function returns the cosine of value passed as argument. The value passed in this function should be in radians.

**Syntax:** math.cos(x)
**Parameter:** x : value to be passed to cos()
**Returns:** Returns the cosine of value passed as argument
**Example:**

```
# Python code to demonstrate the working of cos()
# importing "math" for mathematical operations
import math
a = math.pi / 6
# returning the value of cosine of pi / 6
print ("The value of cosine of pi / 6 is : ", end ="")
print (math.cos(a))
```

**Output:** The value of cosine of pi/6 is : 0.8660254037844387

**4.22 Python | math.cos() function:** In Python, math module contains a number of mathematical operations, which can be performed with ease using the module. **math.cos()** function returns the cosine of value passed as argument. The value passed in this function should be in radians.

**Syntax:** math.cos(x)
**Parameter:** x : value to be passed to cos()
**Returns:** Returns the cosine of value passed as argument
**Example:**

```
# Python code to demonstrate the working of cos()
# importing "math" for mathematical operations
import math
a = math.pi / 6
# returning the value of cosine of pi / 6
print ("The value of cosine of pi / 6 is : ", end ="")
print (math.cos(a))
```

**Output:** The value of cosine of pi/6 is : 0.8660254037844387

**4.23 degrees() and radians() in Python:** degress() and radians() are methods specified in math module in Python 3 and Python 2.

**radians():** This function accepts the "**degrees**" as input and converts it into its radians equivalent.

**Syntax:** radians(deg)
**Parameters: deg :** The degrees value that one needs to convert into radians
**Returns :** This function returns the floating point radians equivalent of argument.
**Computational Equivalent :** 1 Radians = 180/pi Degrees.

**Example:** # Python code to demonstrate working of radians()

```
import math
# Printing radians equivalents.
print("180 / pi Degrees is equal to Radians : ", end ="")
print (math.radians(180 / math.pi))
print("180 Degrees is equal to Radians : ", end ="")
print (math.radians(180))
print("1 Degrees is equal to Radians : ", end ="")
print (math.radians(1))
```

**Output:**

180/pi Degrees is equal to Radians : 1.0

180 Degrees is equal to Radians: 3.141592653589793

1 Degrees is equal to Radians: 0.017453292519943295

**degrees ():** This function accepts the "**radians**" as input and converts it into its degrees equivalent.

**Syntax**: degrees(rad)
**Parameters: rad:** The radians value that one needs to convert into degrees.
**Returns :** This function returns the floating point degrees equivalent of argument.
**Computational Equivalent :** 1 Degrees = pi/180 Radians.

**Example:**

```
# Python code to demonstrate working of degrees()
import math
# Printing degrees equivalents.
print("pi / 180 Radians is equal to Degrees : ", end ="")
print (math.degrees(math.pi / 180))
print("180 Radians is equal to Degrees : ", end ="")
print (math.degrees(180))

print("1 Radians is equal to Degrees : ", end ="")
print (math.degrees(1))
```

**Output:**

```
pi/180 Radians is equal to Degrees : 1.0

180 Radians is equal to Degrees : 10313.240312354817

1 Radians is equal to Degrees : 57.29577951308232
```

**4.24 Python math.pi Constant:** The math.pi constant returns the value pi: 3.141592653589793. It is defined as the ratio of the circumference to the diameter of a circle. Mathematically pi is represented by **π**.

**Syntax:** math.pi

**Return Value:** A float value, 3.141592653589793, representing the mathematical constant **PI**

**Example:**

```
#Print the value of PI:
import math        # Import math Library
print (math.pi)    # Print the value of pi
```

**4.25 Python math.e Constant:** The math.e constant returns the eular's number: 2.718281828459045.

**Syntax:** math.e
**Return Value:** A float value, 2.718281828459045, representing the mathematical constant **e**.

**Example:**

```
#Print the value of Euler e.
import math            # Import math Library
print (math.e)         # Print the value of Euler e
```

**4.26 Python math.inf Constant:** The math.inf constant returns a floating-point positive infinity. For negative infinity, use -math.inf. The inf constant is equivalent to float("inf").

**Syntax:** math.inf
**Return Value:** A float value, representing the value of positive infinity

**Example:** Print the positive and negative infinity.

```
# Import math Library
import math
# Print the positive infinity
print (math.inf)
# Print the negative infinity
print (-math.inf)
```

**4.27 Python math.nan Constant:** The math.nan constant returns a floating-point nan (Not a Number) value. This value is not a legal number. The nan constant is equivalent to float("nan").

**Syntax:** math.nan
**Return Value:** A float value, nan (Not a Number)

**Example:** Print the value of nan.

```
# Import math Library
import math
# Print the value of nan
print (math.nan)
```

**5.Python - Random Module :** Functions in the **random** module depend on a pseudo-random number generator function random(), which generates a random float number between 0.0 and 1.0.

**1. random.random():** Generates a random float number between 0.0 to 1.0. The function doesn't need any arguments.

**Example:**

```
Import random
Random.random()
0.645173684807533
```

**2.random.randint():** Returns a random integer between the specified integers.

**Example:**

```
Import random
Random.radiant(1,100)
95
Random.radiant(1,100)
45
```

**3.random.randrange():** Returns a randomly selected element from the range created by the start, stop and step arguments. The value of start is 0 by default. Similarly, the value of step is 1 by default.

**Example:**

```
Import random
random.randrange(1,10)
2
random.randrange(1,10,2)
5
```

**4.random.choice():** Returns a randomly selected element from a non-empty sequence. An empty sequence as argument raises an IndexError.

**Example:**

```
Import random
random.choice('computer')
't'
random.choice([1,10,2])
2
```

**5.random.shuffle():** This functions randomly reorders the elements in a list.

**Example:**

```
Import random
numbers= [12,23,45,67,65,43]
random.shuffle(numbers)
print(numbers)
[23,12,43,65,67,45]
```

## Python Lists

**Definition of List:** list is a collection of items of different data types or similar data types. It is an ordered sequence of items. A list object contains one or more items, not necessarily of the same type, which are separated by comma and enclosed in square brackets [].

**Note:** List allows duplicate values and the list is mutable.

**Syntax:** list_name= [value-1, value-2, value-3……, value-n]

**Example:**

```
A= [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]      #list with similar data types
A= [1," Ram", "Raghu", 2.5, 6, 9]       # list with different data types
```

**Creating a List:** A list is created by placing all the items (elements) inside square brackets [], separated by commas. It can have any number of items and they may be of different types (integer, float, string etc.).

**Example:**

```
# empty list
my_list = []

# list of integers
my_list = [1, 2, 3]

# list with mixed data types
my_list = [1, "Hello", 3.4]
```

Note: A list can also have another list as an item. This is called a nested list.

**Example:**

```
# nested list
my_list = ["mouse", [8, 4, 6], ['a']]
```

**Accessing the list elements:** There are various ways in which we can access the elements of a list.

1) **List Index:** We can use the index operator [] to access an item in a list. In Python, indices start at 0. So, a list having 5 elements will have an index from 0 to 4.

Trying to access indexes other than these will raise an Index Error. The index must be an integer. We can't use float or other types; this will result in Type Error.

Nested lists are accessed using nested indexing.

**Example:**

```
my_list = ['p', 'r', 'o', 'b', 'e']          # List indexing
print(my_list[0])    # Output: p
print(my_list[2])    # Output: o
print(my_list[4])    # Output: e
n_list = ["Happy", [2, 0, 1, 5]]                # Nested List
print(n_list[0][1])  # Nested indexing
print(n_list[1][3])
print(my_list[4.0]) # Error! Only integer can be used for indexing
```

**Output**

```
p
o
e
a
5
Traceback (most recent call last):
  File "<string>", line 21, in <module>
TypeError: list indices must be integers or slices, not float
```

2) **Negative indexing:** Python allows negative indexing for its sequences. The index of

-1 refers to the last item, -2 to the second last item and so on.

**Example:**

```
# Negative indexing in lists
my_list = ['p','r','o','b','e']
print(my_list[-1])
print(my_list[-5])
```

**Output:**

```
e
p
```

**Updating the list:** Modifying or updating a list means to change a particular entry, add a new entry, or remove an existing entry. To perform these tasks, you must sometimes read an entry. The concept of modification is found within the acronym CRUD, which stands for Create, Read, Update, and Delete. Here are the list functions associated with CRUD:

append (): Adds a new entry to the end of the list.

clear (): Removes all entries from the list.

copy (): Creates a copy of the current list and places it in a new list.

extend (): Adds items from an existing list and into the current list.

insert (): Adds a new entry to the position specified in the list.

pop (): Removes an entry from the end of the list.

remove (): Removes an entry from the specified position in the list.

**append ():** Adds a new entry to the end of the list. Basically append () method is used to add an element to the end of the list and by using append () method you can add only one element to the list.

**Syntax**: listname. append(element)

**Example:**

```
list1= [1, 2, 3, 4, 5]
list1.append(6)
print(list1)
```

**Output:** [1, 2, 3, 4, 5, 6]

**clear ():** Removes all entries from the list.

**Syntax**: listname. clear()

**Example:**
```
list1= [1, 2, 3, 4, 5]
list1.clear()
print(list1)
```

**Output:** [ ]

**copy ():** Creates a copy of the current list and places it in a new list.

**Syntax**: listname. copy()

**Example:**
```
list1= [1, 2, 3, 4, 5]
list2=[]
list2=list1.copy()
print(list2)
```

**Output:** [1, 2, 3, 4, 5]

**extend ()**: Adds items from an existing list to the current list. By using extend() method you can add more than one elements to the existing list.

**Syntax**: listname. extend(one element or group of elements)

**Example:**
```
list1= [1, 2, 3, 4, 5]
list1.extend([10,11,12])
print(list1)
```

**Output:** [1, 2, 3, 4, 5, 10, 11, 12]

**insert ():** Adds a new entry to the position specified in the list. The insert() method takes 2 parameters. Those are index value and element want to insert.

**Syntax:** listname.insert(index_value, element_value)

**Example:**
```
list1= [1, 2, 3, 4, 5]
list1.insert(2,19)
print(list1)
```

**Output:** [1, 2, 19, 3, 4, 5]

**pop ():** Removes an entry from the end of the list. By using we can delete a specific element by indicating the index value.

**Syntax:** listname.pop(index_value)

**Example**:
```
list1= [1, 2, 3, 4, 5]
list1.pop(2)
print(list1)
```

**Output:** [1, 2, 4, 5]

- In pop() method if we didn't pass any argument then it removes the last element from the list.

**Example:**
```
list1= [1, 2, 3, 4, 5]
list1.pop ()
print(list1)
```
**Output:** [1, 2, 3, 4]

**remove ():** It removes the specified element from the list.

**Syntax:** listname. remove(element)

**Example:**
```
list1= [1, 2, 3, 4, 5]
list1.remove(5)
print(list1)
```
**Output:** [1, 2, 3, 4]

- A list can be updated by changing the value at the given location.

**Example**:
```
list1= [1, 2, 3, 4, 5]
list1[2]="hello"
print(list1)
```
**Output:** [1, 2, 'hello', 4, 5]

**Deleting lists:** Deleting the lists means we can delete one or more elements from the list. There are several ways to delete the elements from the list.

**Method-1:** using clear() method. In this by using the clear() method we can delete all the elements from the list.

**Syntax**: listname. clear()

**Example:**
```
list1= [1, 2, 3, 4, 5]
list1.clear()
print(list1)
```
**Output:** [ ]

**Method-2:** Reinitializing the list: In this method initializes the list with no value. i.e list of size 0.

**Syntax**: listname=[]

**Example:**
```
list1= [1, 2, 3, 4, 5]
list1=[]      #Re-initialization
print(list1)
```
**Output:** [ ]

**Method-3**: Using **"*= 0".** By using this method we can clear the entire list once.

**Syntax**: listname*=0

**Example:**
```
list1= [1, 2, 3, 4, 5]
list1*=0
```

```
print(list1)
```

**Output:** [ ]

**Method-4** : Using del : **del** can be used to clear the list elements in a range, if we don't give a range, all the elements are deleted.

**Syntax:** del listname[range]

**Example:**

```
list1= [1, 2, 3, 4, 5]
del list1[1:3]  #removes range of values i.e, 2,3
print(list1)
del list1[:]    #removes all the elements and it shows list is empty
print(list1)
del list1       #removes the entire list i.e., list completely removed
print(list1)
```

**Output:**

```
[1, 4, 5]
[]
Traceback (most recent call last):
  File
"C:/Users/Govindu/AppData/Local/Programs/Python/Python38-
32/11.py", line 7, in <module>
    print(list1)
NameError: name 'list1' is not defined
```

**Basic list operations:**

- 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.
- In fact, lists respond to all of the general sequence operations we used on strings

| 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, | 1 2 3 | Iteration |

**Reversing the list:** There are two ways to reverse the list. Those are

- By using slicing technique
- By using reverse() method

**Method-1: Using the slicing technique:**

In this technique, a copy of the list is created and that list is reversed i.e., original remains same. There is no change in original list. Creating a copy requires more space to hold all of the existing elements. This exhausts more memory.

**Example:**

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

print(list1[::-1]) #prints the list in reverse order

print(list1)    #original list
```

**Output:**

```
[5, 4, 3, 2, 1]

[1, 2, 3, 4, 5]
```

**Method-2: Using reverse() method:**

Using the reverse () method we can reverse the contents of the list object **in-place** i.e., we don't need to create a new list instead we just copy the existing elements to the original list in reverse order. This method directly modifies the original list.

**Example:**

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

list1.reverse()

print(list1) #prints the list in reverse order
```

**Output:** [5, 4, 3, 2, 1]

**Indexing**: index() is an inbuilt function in Python, which searches for a given element from the start of the list and returns the lowest index where the element appears.

**Syntax:** list_name. index (element, start, end)

**Parameters :**

element - The element whose lowest index will be returned.

start (Optional) - The position from where the search begins.

end (Optional) - The position from where the search ends.

**Returns:** Returns lowest index where the element appears.

**Error:**

If any element which is not present is searched,
it returns a ValueError

**Example:**

```
myList = [1, 2, 3,"EduCBA"]
print(myList.index('EduCBA'))          # searches in the whole list
print(myList.index('EduCBA', 0, 2))    # searches from 0th to 2nd position
```

**Output:**

```
3

Traceback (most recent call last):

  File      "C:/Users/Govindu/AppData/Local/Programs/Python/Python38-
32/11.py", line 3, in <module>

print(myList.index('EduCBA', 0, 2))     # searches from 0th to 2nd position

ValueError: 'EduCBA' is not in list
```

In general, there are 2 types of indexes in lists. Those are

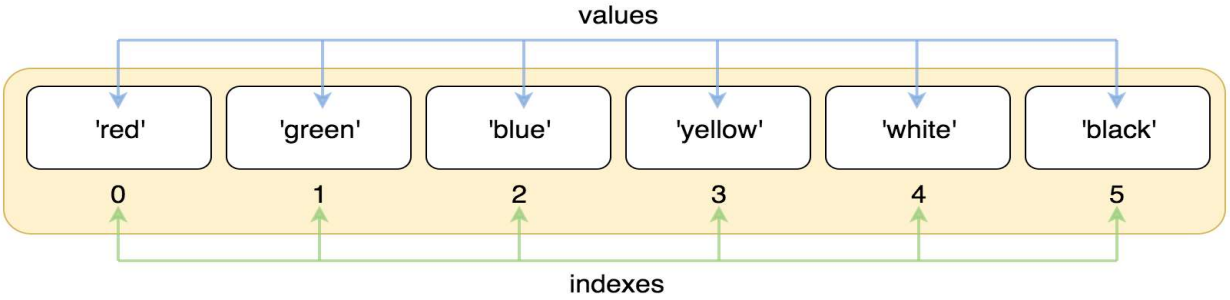
- Positive Indexing
- Negative Indexing

**Positive Indexing:** In this type of indexing index starts from 0 and ends at size_of _list-1**.** If the list is having 5 elements, then starting index is 0 and ending index is 4(5-1).

Let's take a simple example:

```
>>> colors = ['red', 'green', 'blue', 'yellow', 'white', 'black']
```

Here we defined a list of colors. Each item in the list has a value (color name) and an index (its position in the list). Python uses zero-based indexing. That means, the first element (value 'red') has an index 0, the second (value 'green') has index 1, and so on.



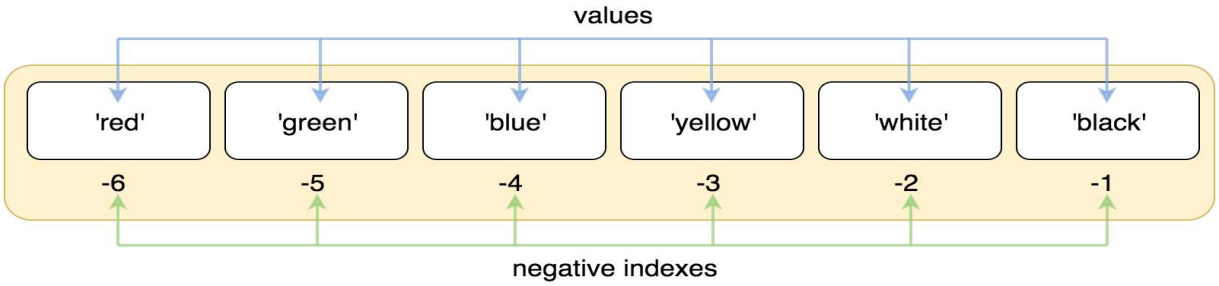

To access an element by its index we need to use square brackets:

```
>>> colors = ['red', 'green', 'blue', 'yellow', 'white', 'black']
>>> colors[0]
'red'
>>> colors[1]
'green'
>>> colors[5]
'black'
```

**Negative indexes:**

Using indexing we can easily get any element by its position. This is handy if we use position from the head of a list. But what if we want to take the last element of a list? Or the penultimate element? In this case, we want to enumerate elements from the tail of a list.

To address this requirement there is negative indexing. So, instead of using indexes from zero and above, we can use indexes from -1 and below.



In negative indexing system -1 corresponds to the last element of the list(value 'black'), -2 to the penultimate (value 'white'), and so on.

```
>>> colors = ['red', 'green', 'blue', 'yellow', 'white', 'black']
>>> colors[-1]
'black'
>>> colors[-2]
'white'
>>> colors[-6]
'red'
```

**Slicing:**

indexing allows you to access/change/delete only a single cell of a list. If we want to get a sub-list of the list or we want to update a bunch of cells at once? Or we want to extend a list with an arbitrary number of new cells in any position we will use slicing operation.

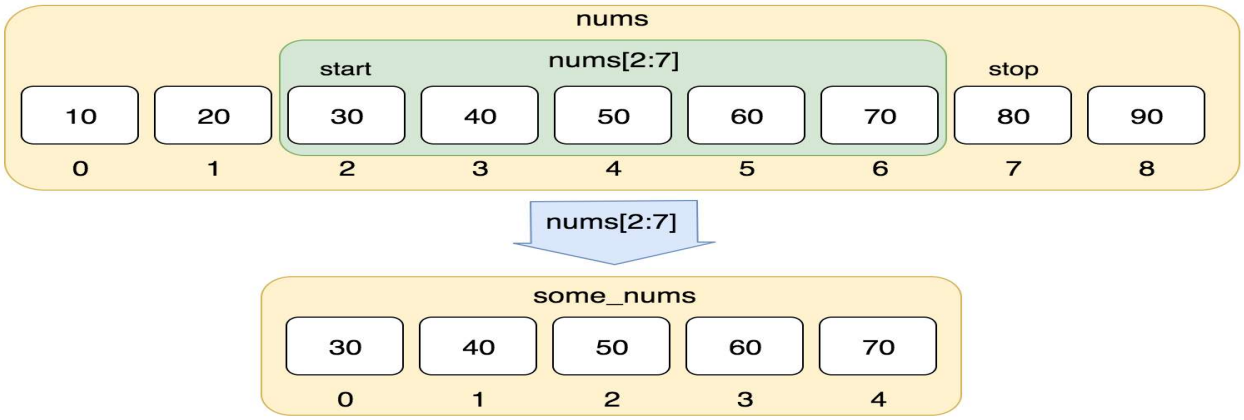
**Syntax:** list [*start:stop:step]*.

- **start** refers to the index of the element which is used as a start of our slice.
- **stop** refers to the index of the element we should stop just before to finish our slice.
- **step** allows you to take each nth-element within a *start:stop* range.

**Example:**

```
>>> nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
>>> some_nums = nums [2:7]
>>> some_nums
[30, 40, 50, 60, 70]
```

In the above example **start** equals 2, so our slice starts from value 30. **stop** is 7, so the last element of the slice is 70 with index 6. In the end, slice creates a new list (we named it some_nums) with selected elements.




We did not use step in our slice, so we didn't skip any element and obtained all values within the range. By default the step value is 1.

**Taking n first elements of a list:**

Slice notation allows you to skip any element of the full syntax. If we skip the start number, then it starts from 0 index:

```
>>> nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
>>> nums[:5]
[10, 20, 30, 40, 50]
```

So, *nums[:5]* is equivalent to *nums[0:5]*. This combination is a handy shortcut to take n first elements of a list.

**Taking n last elements of a list:**

Negative indexes allow us to easily take n-last elements of a list:

```
>>> nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
>>> nums[-3:]
[70, 80, 90]
```

Here, the stop parameter is skipped. That means you take from the start position, till the end of the list. We start from the third element from the end (value 70 with index -3) and take everything to the end.

We can freely mix negative and positive indexes in start and stop positions:

```
>>> nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
>>> nums[1:-1]
[20, 30, 40, 50, 60, 70, 80]
>>> nums[-3:8]
[70, 80]
>>> nums[-5:-1]
```

[50, 60, 70, 80]

**Taking all but n last elements of a list:**

```
>>> nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
>>> nums[:-2]
[10, 20, 30, 40, 50, 60, 70]
```

We take all but the last two elements of original list are ignored.

**Taking every nth-element of a list:**

What if we want to have only every 2-nd element of nums? This is where the step parameter comes into play:

```
>>> nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
>>> nums[::2]
[10, 30, 50, 70, 90]
```

Here we omit start/stop parameters and use only step. By providing start we can skip some elements:

**Slice and Copying**

One important thing to notice – is that list slice creates a shallow copy of the initial list. That means, we can safely modify the new list and it will not affect the initial list:

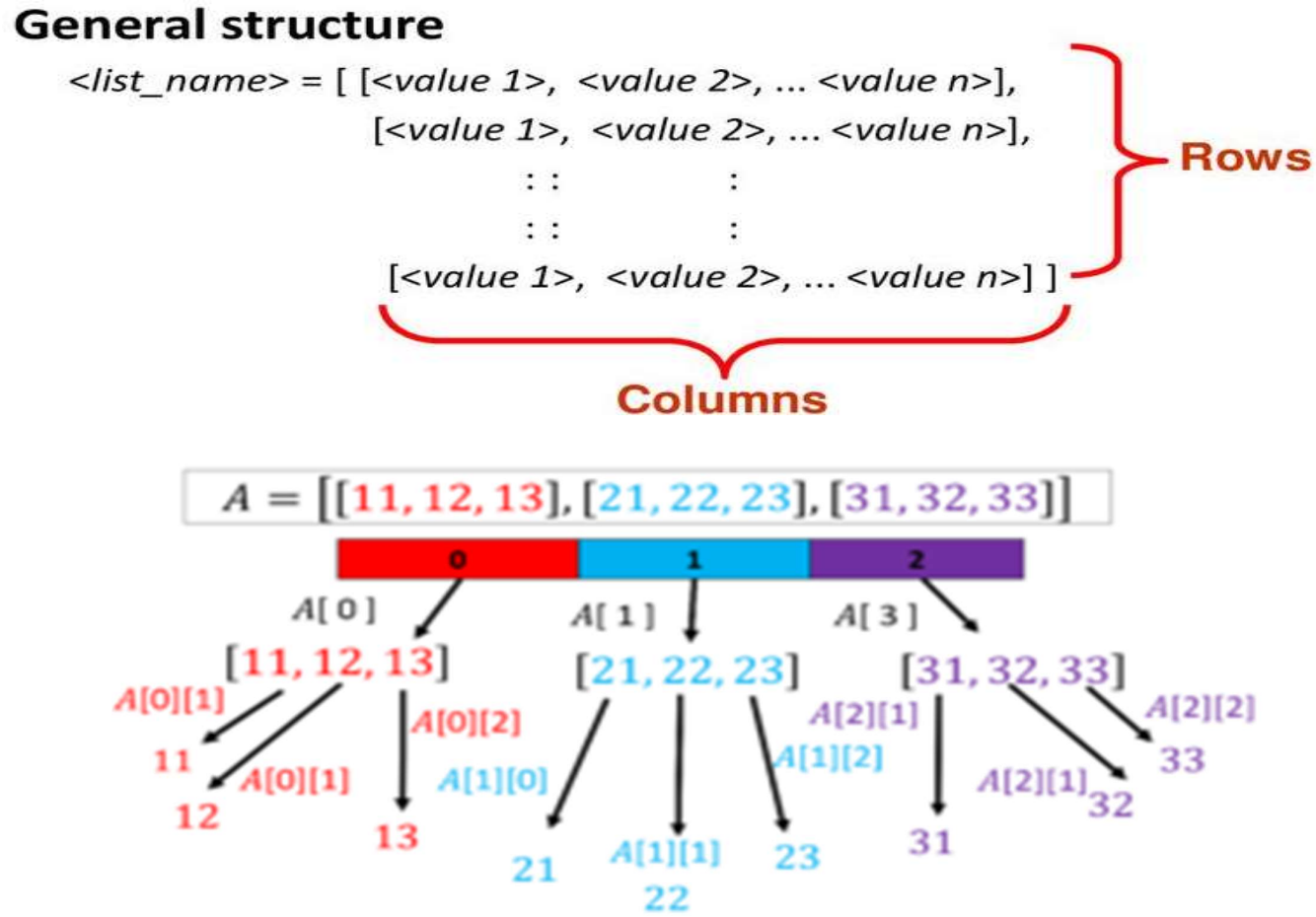
```
>>> nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
>>> first_five = nums[0:5]
>>> first_five[2] = 3
>>> first_five
[10, 20, 3, 40, 50]
>>> nums
[10, 20, 30, 40, 50, 60, 70, 80, 90]
```

**Substitute part of a list:**

Slice assignment allows you to update a part of a list with new values:

```
>>> nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
>>> nums[:4] = [1,2,3,4]
>>> nums
[1, 2, 3, 4, 50, 60, 70, 80, 90]
```

**Matrices:** Matrices allows us to store multiple elements in multiple dimensions. So, in order to implement matrices, we should implement multi-dimensional lists.




**Accessing a multidimensional list:**

**Method:1**

#Python program to demonstrate printing of complete multidimensional list

a = [[2, 4, 6, 8, 10], [3, 6, 9, 12, 15], [4, 8, 12, 16, 20]]

print(a)

**Output:**

[[2, 4, 6, 8, 10], [3, 6, 9, 12, 15], [4, 8, 12, 16, 20]]

**Method:2 Accessing with the help of loops:**

#Python program to demonstrate printing of complete multidimensional list row by row

a = [[2, 4, 6, 8, 10], [3, 6, 9, 12, 15], [4, 8, 12, 16, 20]]

for x in a:

    print(x)

**Output:**

[2, 4, 6, 8, 10]

[3, 6, 9, 12, 15]

[4, 8, 12, 16, 20]

**Method:3 Accessing array elements using square brackets:**

```
#Python program to demonstrate that accessing array elements using square brackets
a = [[2, 4, 6, 8, 10], [3, 6, 9, 12, 15], [4, 8, 12, 16, 20]]
for i in range(len(a)) :
    for j in range(len(a[i])) :
        print(a[i][j], end=" ")
    print()
```

**Output:**

2 4 6 8 10

3 6 9 12 15

4 8 12 16 20

**Methods on Multidimensional lists:**

1. **append():** Adds an element at the end of the list.

**Example:**

```
a = [[2, 4, 6, 8, 10], [3, 6, 9, 12, 15], [4, 8, 12, 16, 20]]
a.append([11,12,13,14,15])
print(a)
```

**Output:**

[[2, 4, 6, 8, 10], [3, 6, 9, 12, 15], [4, 8, 12, 16, 20], [11, 12, 13, 14, 15]]

2. **extend():** Add the elements of a list (or any iterable), to the end of the current list.

**Example:**

```
#extending a sublist
a = [[2, 4, 6, 8, 10], [3, 6, 9, 12, 15], [4, 8, 12, 16, 20]]
a[0].extend([11,12,13,14,15])
print(a)
```

**Output:**

[[2, 4, 6, 8, 10, 11, 12, 13, 14, 15], [3, 6, 9, 12, 15], [4, 8, 12, 16, 20]]

**3. reverse():** Reverses the order of the list.

**Example:**

```
#reversing a sublist
a = [[2, 4, 6, 8, 10], [3, 6, 9, 12, 15], [4, 8, 12, 16, 20]]
a[0].reverse()
print(a)
```

**Output:**

```
[[10, 8, 6, 4, 2], [3, 6, 9, 12, 15], [4, 8, 12, 16, 20]]
```

**Concatenation of lists:** It means merging of two or more lists into single list. There are several ways to perform concatenation operation.

**Method-1 : Using Naive Method**

In this method, we traverse the second list and keep appending elements in the first list, so that first list would have all the elements in both lists and hence would perform the append.

**Example:**

```
#Concatenating 2 Lists
a=[1,2,3,4,5]
b=[11,12,13,14,15,2]
for i in b:
    a.append(i)
print("After concatenation"+str(a))
```

**Output:**

```
After concatenation[1, 2, 3, 4, 5, 11, 12, 13, 14, 15, 2]
```

**Method #2 : Using + operator:**

The most conventional method to perform the list concatenation, the use of "+" operator can easily add the whole of one list behind the other list and hence perform the concatenation.

**Example:**

```
#Concatenating 2 Lists
a=[1,2,3,4,5]
```

```
b=[11,12,13,14,15,2]

print("After concatenation"+str(a+b))
```

**Output:**

```
After concatenation[1, 2, 3, 4, 5, 11, 12, 13, 14, 15, 2]
```

**Method #3 : Using extend():**

extend() is the function extended by lists in Python and hence can be used to perform this task. This function performs the inplace extension of first list.

**Example:**

```
#Concatenating 2 Lists

a=[1,2,3,4,5]

b=[11,12,13,14,15,2]

a.extend(b)

print("After concatenation"+str(a))
```

**Output:**

```
After concatenation[1, 2, 3, 4, 5, 11, 12, 13, 14, 15, 2]
```

**Method #4 : Using * operator:**

Using * operator, this method is the new addition to list concatenation and works only in Python 3.6+. Any no. of lists can be concatenated and returned in a new list using this operator.

**Example:**

```
#Concatenating 2 Lists

a=[1,2,3,4,5]

b=[11,12,13,14,15,2]

c=[*a,*b]

print("After concatenation"+str(c))
```

**Output:**

```
After concatenation[1, 2, 3, 4, 5, 11, 12, 13, 14, 15, 2]
```

**List comprehension:**

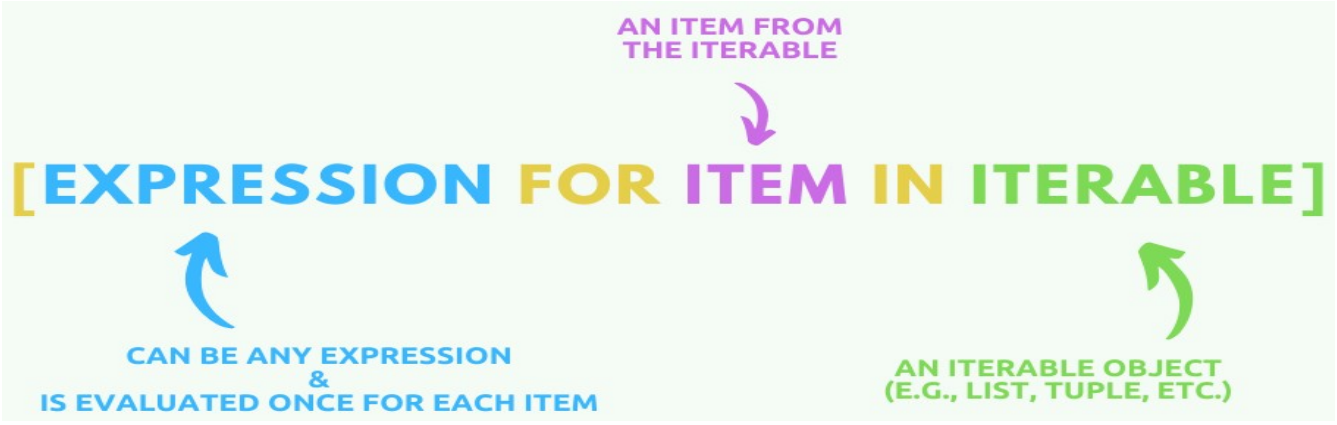
Python provides compact syntax for deriving one list from another. These expressions are called list comprehensions. List comprehensions are one of the most powerful tools in Python. Python's list comprehension is an example of the language's support for functional programming concepts.

The Python list comprehensions are a very easy way to apply a function or filter to a list of items. List comprehensions can be very useful if used correctly but very unreadable if you're not careful.

**Syntax:** [expr for element in iterable if condition]

Above is equivalent to −

```
for element in iterable:
  if condition:
    expr
```




For example, say you want to compute the square of each number in a list. You can do this by providing the expression for your computation and the input sequence to loop over.

```
>>> lst = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> squares = [x**2 for x in lst]
>>> print(squares)
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

**List Comprehension vs For Loop:**

Program for printing even numbers less than 10

**Example:**

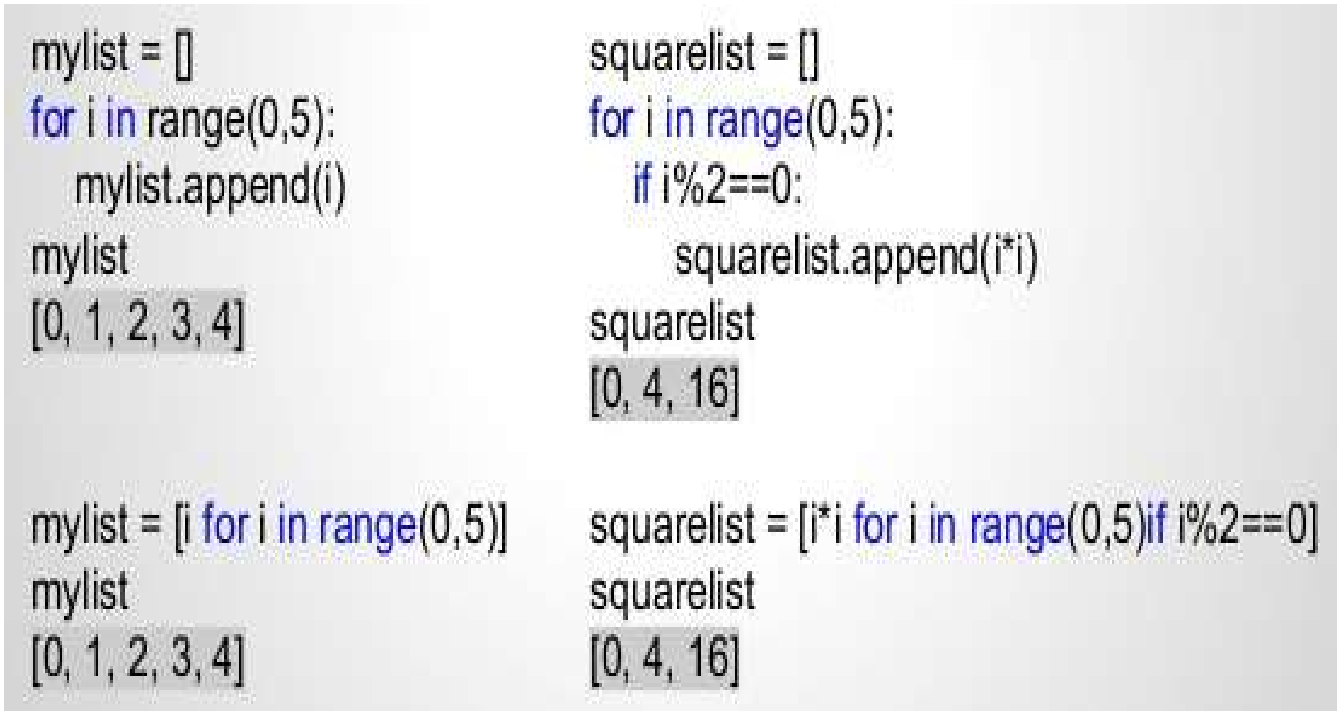
```
#USING FOR LOOP
evens = []
for i in range(10):
   if i % 2 == 0:
      evens. append(i)
print(evens)
```

**Output:** [0, 2, 4, 6, 8]

A better and faster way to write the above code is through list comprehension.

```
>>> [i for i in range(10) if i % 2 ==0]
[0, 2, 4, 6, 8]
```

Writing the code using list comprehension is a lot more efficient, it's shorter and involves lesser elements.




**LIST BUILT-IN FUNCTIONS / METHODS:** Python provides several built-in functions.

1) append()
2) extend()
3) insert()
4) remove()
5) pop()
6) Slice
7) reverse()
8) len()
9) min() & max()
10) count ()
11) index ()
12) sort ()
13) clear()

**1. append():**

The append() method is used to add elements at the end of the list. This method can only add a single element at a time. To add multiple elements, the append() method can be used inside a loop.

**Syntax:** listname.append(element)

**Example:**

```
myList = [1, 2, 3, 'EduCBA', 'makes learning fun!']
myList.append(4)
myList.append(5)
myList.append(6)
for i in range(7, 9):
        myList.append(i)
        print(myList)
```

**Output:**

[1, 2, 3, 'EduCBA', 'makes learning fun!', 4, 5, 6, 7, 8]

**2. extend()**

The extend() method is used to add more than one element at the end of the list. Although it can add more than one element unlike append(), it adds them at the end of the list like append().

**Syntax:** listname.extend(element or group of elements)

**Example***:*

```
myList.extend([4,5,6])
for i in range(7, 9):
        myList.append(i)
print(myList)
```

**Output:**

[1, 2, 3, 'EduCBA', 'makes learning fun!', 4, 5, 6, 7, 8]

**3. insert()**

The insert() method can add an element at a given position in the list. Thus, unlike append(), it can add elements at any position but like append(), it can add only one element at a time. This method takes two arguments. The first argument specifies the position and the second argument specifies the element to be inserted.

**Syntax:** listname.insert(position, element)

**Example:**

```
myList = [1, 2, 3, "EduCBA", "makes learning fun!"]
myList.insert(3, 4)
myList.insert(4, 5)
myList.insert(5, 6)
print(myList)
```

**Output:** [1, 2, 3, 4, 5, 6, 'EduCBA', 'makes learning fun!']

**4. remove()**

The remove() method is used to remove an element from the list. In the case of multiple occurrences of the same element, only the first occurrence is removed.

**Syntax:** listname.remove(element)

**Example:**

```
myList = [1, 2, 3, "EduCBA", "makes learning fun!"]
myList.remove('makes learning fun!')
myList.insert(4, 'makes')
print(myList)
```

**Output:**

[1, 2, 3, 'EduCBA', 'makes']

**5. pop()**

The method pop() can remove an element from any position in the list. The parameter supplied to this method is the index of the element to be removed. If we didn't provide any parameter then it removes last element in the list automatically.

**Syntax:** listname.pop(index of the element to remove)

**Example:**

```
myList = [1, 2, 3, "EduCBA", "makes learning fun!"]
myList.pop(4)
myList.insert(4, 'makes')
myList.insert(5, 'learning')
myList.insert(6, 'so much fun!')
print(myList)
```

**Output:**

[1, 2, 3, 'EduCBA', 'makes', 'learning', 'so much fun!']

**6. Slice():**

The Slice operation is used to print a section of the list. The Slice operation returns a specific range of elements. It does not modify the original list.

**Syntax:**

- slice(stop)
- slice(start, stop, step)

**Parameters:**

**start:** Starting index where the slicing of object starts.
**stop:** Ending index where the slicing of object stops.
**step:** It is an optional argument that determines the increment between each index for slicing.

**Return Type**: Returns a sliced object containing elements in the given range only.

**Example:**

```
myList = [1, 2, 3, "EduCBA", "makes learning fun!"]
print(myList[slice(0,4)])  # prints from beginning to end index
print(myList[slice(2,len(myList))])  # prints from start index to end of list
print(myList[slice(2,4)]) # prints from start index to end index
```

**Output:**

[1, 2, 3, 'EduCBA']

[3, 'EduCBA', 'makes learning fun!']

[3, 'EduCBA']

**7. reverse():**

The reverse() operation is used to reverse the elements of the list. This method modifies the original list. To reverse a list without modifying the original one, we use the slice operation with negative indices. Specifying negative indices iterates the list from the rear end to the front end of the list.

**Syntax:** listname.reverse()

**Example:**

```
myList = [1, 2, 3, "EduCBA", "makes learning fun!"]
print(myList[::-1])  # does not modify the original list
myList.reverse()     # modifies the original list
print(myList)
```

**Output:**

```
['makes learning fun!', 'EduCBA', 3, 2, 1]
['makes learning fun!', 'EduCBA', 3, 2, 1]
```

**8. len():**

The len() method returns the length of the list, i.e. the number of elements in the list.

**Syntax:** len(listname)

**Example:**

```
myList = [1, 2, 3, "EduCBA", "makes learning fun!"]
print(len(myList))
```

**Output:**

5

**9. min() & max():**

The min() method returns the minimum value in the list. The max() method returns the maximum value in the list. Both the methods accept only homogeneous lists, i.e. list having elements of similar type.

**Syntax:** min(listname) & max(listname)

**Example:**

```
myList = [1, 2, 3]
print(min(myList))
```

**10. count():**

The function count() returns the number of occurrences of a given element in the list.

**Example:**

```
myList = [1, 2, 3]
```

```
print(myList.count(3))
```

**11. index()**

The index() method returns the position of the first occurrence of the given element. It takes two optional parameters – the begin index and the end index. These parameters define the start and end position of the search area on the list. When supplied, the element is searched only in the sub-list bound by the begin and end indices. When not supplied, the element is searched in the whole list.

**Syntax:** listname. index(element, starting_position, ending_position)

**Example:**

```
myList = [1, 2, 3,"EduCBA"]
print(myList.index('EduCBA'))          # searches in the whole list
print(myList.index('EduCBA', 0, 2))    # searches from 0th to 2nd position
```

**12. sort():**

The sort method sorts the list in ascending order. This operation can only be performed on homogeneous lists, i.e. lists having elements of similar type.

**Syntax:** listname.sort(key=..., reverse=...)

**Parameters:** By default, sort() doesn't require any extra parameters. However, it has two optional parameters:

**reverse** - If True, the sorted list is reversed (or sorted in Descending order). By default, the reverse value is false.

**key** - function that serves as a key for the sort comparison

**Example:**

```
yourList = [4, 2, 6, 5, 0, 1]
yourList.sort()     #sort the list in ascending order
print(yourList)
yourList.sort(reverse=True)  #sort the list in descending order
print(yourList)
```

**Output:**

```
[0, 1, 2, 4, 5, 6]
[6, 5, 4, 2, 1, 0]
```

**13. clear():**

This function erases all the elements from the list and empties it.

**Syntax:** listname.reverse()

**Example:**

```
yourList = [4, 2, 6, 5, 0, 1]
yourList.clear()
```

```
print(yourList)
```

**Output:** [ ]

**UNIT-III**

# Python Strings and Functions

**Python strings:** Concept, Slicing, Escape characters, String Special Operations, String formatting Operator, Triple Quotes, Raw String, Unicode Strings and Built-in String methods.
**Functions:** Defining a Function, Calling a Function, Types of Functions, Function Arguments, Anonymous functions, Global and Local Variables, Recursion.

## 1. Strings

- A string is a sequence of characters.
- A character is simply a symbol. Characters are internally stored and manipulated as a combination of 0s and 1s.
  This conversion of character to a number is called encoding, and the reverse process is decoding.
  - ASCII and Unicode are some of the popular encodings used.
  - In Python, a string is a sequence of Unicode characters. Unicode was introduced to include every character in all languages and bring uniformity in encoding.

### 1.1 create a string in Python

Strings can be created by enclosing characters inside a single quote or double-quotes. Even triple quotes can be used in Python but generally used to represent multiline strings and docstrings.

```
# defining strings in Python
# all of the following are equivalent

my_string = 'Hello'
my_string = "Hello"
my_string = '''Hello'''

# triple quotes string can extend multiple lines
my_string = """Hello, welcome to
                the world of Python"""
```

## 1.2 How to access characters in a string?

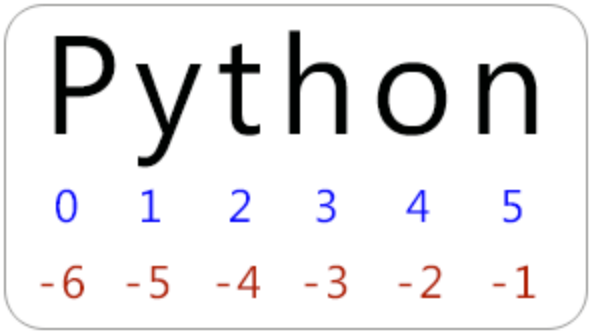
- We can access individual characters using indexing and a range of characters using slicing.
- Index starts from 0. Trying to access a character out of index range will raise an IndexError.
- The index must be an integer.
- Python also allows negative indexing for its sequences. The index of -1 refers to the last item, -2 to the second last item and so on.
- We can access a range of items in a string by using the slicing operator :(colon).

#Accessing string characters in Python

str = 'Python'

print('str[0] = ', str[0]) → #first character

print('str[-1] = ', str[-1]) → #last character




## 1.3 String slices:

A segment of a string is called a slice. The "slice" syntax is used to get sub-parts of a string. The slice S[start:end] is the elements beginning at the start (p) and extending up to end(q) but not including end(q).

The operator [p:q] returns the part of the string from the pth character to the qth character, including the first but excluding the last.

- If we 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:
- If the first index is greater than or equal to the second the result is an empty string, represented by two quotation marks.

Example – 1:

>> s = "Python"

- s[1:4] is 'yth' -- chars starting at index 1 and extending up to but not including index 4
- s[1:] is 'ython' -- omitting either index defaults to the start or end of the string
- s[:] is 'Python' -- Copy of the whole thing
- s[1:100] is 'ython' -- an index that is too big is truncated down to the string length

Python uses negative numbers to give easy access to the chars at the end of the string. Negative index numbers count back from the end of the string:

- s[-1] is 'n' -- last char (1st from the end)
- s[-4] is 't' -- 4th from the end
- s[:-3] is 'pyt' -- going up to but not including the last 3 chars
- s[-3:] is 'hon' -- starting with the 3rd char from the end and extending to the end of the string.

Example – 2:

>>> a = "Python String"

>>> print (a[0:5]) → Pytho

>>> print(a[6:11]) → Stri

>>> print(a[5:13]) → n String

**Specifying Stride while Slicing Strings:**

In slicing, we can also specify a third argument as the **stride** which refers to the number of characters to move forward after the first character is retrieved from string. The default of stride is 1.

Example – 3:

```
>>> str = "Welcome to the world of Python"
>>>print(str[2:10])        → lcome to        #default stride is 1
>>>print(str[2:10:2])      → loet            #skips every alternate character
>>>print(str[2:13:4])      → le              #skips every fourth character
>>>print(str[::3])         → WceohwloPh      #skips every third character
>>>print(str[::-1])        → nohtyP fo dlrow eht ot emocleW
   #considered entire string and reversed it through negative stride 1
```

## 1.4 How to change or delete a string?

Strings are immutable. This means that elements of a string cannot be changed once they have been assigned. We can simply reassign different strings to the same name.

```
>>> my_string = 'program'
>>> my_string[5] = 'e
```

TypeError: 'str' object does not support item assignment

We cannot delete or remove characters from a string. But deleting the string entirely is possible using the del keyword.

```
>>> del  my_string[1]
```

TypeError: 'str' object doesn't support item deletion

```
>>> del  my_string
>>> my_string
```

NameError: name 'my_string' is not defined

## 1.5 Escape Characters:

An escape character is a backslash \ followed by the character you want to insert. An escape character gets interpreted in a single quoted as well as double quoted strings

Following table is a list of escape or non-printable characters that can be represented with backslash notation.

| **Escape Sequence** | **Description** | **Example** | **Output** |
|---|---|---|---|
| \\ | Prints Backslash | print ("\\") | \ |
| \` | Prints single-quote | print ("\'") | ' |
| \" | Pirnts double quote | print ("\"") | " |
| \a | ASCII bell ringing alert sounds | print ("\a") | Bell sound |
| \b | Backspace ( BS) removes previous character | Print("ab" + "\b" + "c") | ac |
| \f | ASCII formfeed (FF ) | print ("hello\fworld") | hello world |
| \n | ASCII linefeed (LF) NewLine | print ("hello\nworld") | hello<br>world |
| \r | ASCII carriage return (CR). Moves all characters after ( CR ) the the beginning of the line while overriding same number of characters moved. | print ("123456\rXX_XX") | XX_XX6 |
| \t | ASCII horizontal tab (HTAB). | print ("\t* hello") | * hello |
| \v | ASCII vertical tab (VTAB). | print ("hello\vnworld") | hello<br>world |
| \uxxxx | Prints 16-bit hex value Unicode character | print (u"\u041b") | Λ |
| \Uxxxxxxxx | Prints 32-bit hex value Unicode character | print (u"\U000001a9") | Σ |
| \ooo | Prints character based on its octal value | print ("\043") | # |
| \xhh | Prints character based on its hex value | print ("\x23") | # |

## 1.6 String Special Operators

Assume string variable a holds 'Hello' and variable b holds 'Python', then –

| Operator | Description | Example |
|---|---|---|
| + | Concatenation - Adds values on either side of the operator | a + b will give HelloPython |
| * | Repetition - Creates new strings, concatenating multiple copies of the same string | a*2 will give -HelloHello |
| [] | Slice - Gives the character from the given index | a[1] will give e |
| [ : ] | Range Slice - Gives the characters from the given range | a[1:4] will give ell |
| In | Membership - Returns true if a character exists in the given string | H in a will give 1 |
| not in | Membership - Returns true if a character does not exist in the given string | M not in a will give 1 |

**Iterating Through a string:**

We can iterate through a string using a for loop. Here is an example to count the number of characters in a string.

```
# Iterating through a string
count = 0
for i  in 'Hello World':
        count += 1

print('Total characters  :', count)
```

**Output:**
Total characters :11

## Built-in functions:

Various built-in functions that work with sequence types can also work with strings as well. Some of the commonly used are enumerate() and len().

The enumerate() function returns an enumerate object. It contains the index and value of all the items in the string as pairs. This can be useful for iteration.

Similarly, len() returns the length (number of characters) of the string.

```
>>>str = 'cold'
>>>LE = list(enumerate(str))    # using enumerate()
>>>print('list(enumerate(str) = ', LE)
     list(enumerate(str) =  [(0, 'c'), (1, 'o'), (2, 'l'), (3, 'd')]

>>>print('String Length = ', len(str))      #use len() to character count
     String Length =  4
```

## 1.6 String Formatting:

### 1.6.1 String Formatting Operator %

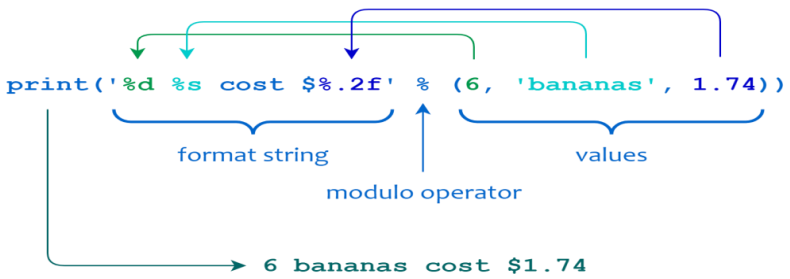
We can format strings like the printf() style used in C programming language. We use the % operator to accomplish this.

```
>>>print("My name is %s and weight is %d kg!" % ('Zara', 21))
    My name is Zara and weight is 21 kg!

>>> x = 12.3456789
>>> print('The value of x is %3.2f' %x)
    The value of x is 12.35

>>> print('The value of x is %3.4f' %x)
    The value of x is 12.3457
```

Example :




Here is the list of complete set of symbols which can be used along with %

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

## 1.6.2 String format() method:

The format() method is used to perform a string formatting operation. The string on which this method is called can contain literal text or replacement fields delimited by braces {}. Each replacement field contains either the numeric index of a positional argument or the name of a keyword argument.

Syntax:

str.format(*args, **kwargs) - Returns a copy of the string where each replacement field is replaced with the string value of the corresponding argument.

```
>>> '{} {}'.format('Python', 'Format')
    'Python Format'

>>> '{} {}'.format(10, 30)
    '10 30'

>>> '{1} {0}'.format('Python', 'Format')
     'Format Python'
```

**Example Program:**

```
# Python string format() method

# default (implicit) order
default_order = "{}, {} and {}".format('John','Bill','Sean')
print('\n--- Default Order ---')
print(default_order)

# order using positional argument
positional_order = "{1}, {0} and {2}".format('John','Bill','Sean')
print('\n--- Positional Order ---')
print(positional_order)

# order using keyword argument
keyword_order = "{s}, {b} and {j}".format(j='John',b='Bill',s='Sean')
print('\n--- Keyword Order ---')
print(keyword_order)
```

***When we run the above program, we get the following output:***

```
--- Default Order ---
John, Bill and Sean

--- Positional Order ---
Bill, John and Sean

--- Keyword Order ---
Sean, Bill and John
```

**Padding and aligning strings:**

A value can be padded to a specific length, The padding character can be spaces or a specified character. The format() method can have optional format specifications. They are separated from the field name using colon.

We can left-justify <,  right-justify > or center ^ a string in the given space.

```
>>> '{:>15}'.format('Python')           #Align Right
'         Python'

 >>>format('Python','>15')  #above can also be given with  built-in format() function
              '         Python'

>>> '{:15}'.format('Python')          #Align left
'Python         '

>>> '{:^16}'.format('Python')      #Align Center
'     Python     '

>>> '{:*<15}'.format('Python')      # use * as padding character
'Python*********'

>>> '{:.10}'.format('Python Programming')  #Truncating long string
'Python Pro'

>>> '{:5d}'.format(24)              #integer format with 5 positions
'   24'

>>> '{:05.2f}'.format(5.12345678123)    #using floating point format
'05.12'
```

We can also format integers as binary, hexadecimal, etc. and floats can be rounded or displayed in the exponent format.

```
# formatting integers as bimary
>>> "Binary representation of {0} is {0:b}".format(12)
'Binary representation of 12 is 1100'

# formatting floats as Exponent
>>> "Exponent representation: {0:e}".format(1566.345)
'Exponent representation: 1.566345e+03'

# round off
>>> "One third is: {0:.3f}".format(1/3)
'One third is: 0.333'

# string alignment
>>> "|{:<10}|{:^10}|{:>10}|".format('butter','bread','jam')
'|butter    |   bread   |       jam|'
```

## 1.7  Triple Quotes:

A multiline string in Python begins and ends with either three single quotes or three double quotes. Any quotes, tabs, or newlines in between the “triple quotes” are

considered part of the string. Python's indentation rules for blocks do not apply to lines inside a multiline string.

#create string of multiple lines

```
>>> S = """What's up Silber!

I'm Manish from RebellionRider.com.

Keep watching these tutorials.

And, you will be the master of "Python Programming" very soon...."""

>>> print(S)

What's up Silber!

I'm Manish from RebellionRider.com.

Keep watching these tutorials.

And, you will be the master of "Python Programming" very soon....
```

## 1.8 Raw String

Python raw string is created by prefixing a string literal with 'r' or 'R'. Python raw string treats backslash (\) as a literal character. This is useful when we want to have a string that contains backslash and don't want it to be treated as an escape character. We may wish to ignore the escape sequences inside a string. To do this we can place r or R in front of the string. This will imply that it is a raw string and any escape sequence inside it will be ignored.

```
>>>s = 'Hi\nHello'
>>>print(s)
   Hi
   Hello

>>>raw_s = r'Hi\nHello'
>>>print(raw_s)
     Hi\nHello

>>>print("This is \x61 \ngood example")          #\x61  equivalent character is a
This is  a  good example

>>> print(r"This is \x61 \ngood example")
This is \x61 \ngood example
```

## 1.9 Unicode Strings

Unicode specification aims to list every character used by human languages and give each character its own unique code. The Unicode specifications are continually revised and updated to add new languages and symbols.

- Python's **str** type is a collection of 8-bit characters. The English alphabet can be represented using these 8-bit characters, but symbols such as ±, ♠, Ω and ℑ cannot.
- Python's string type uses the Unicode Standard for representing characters, which lets Python programs work with all different languages of possible characters.
- Unicode is a standard for working with a wide range of characters. Each symbol has a codepoint (a number). A code point value is an integer in the range 0 to 0x10FFFF and these codepoints can be encoded (converted to a sequence of bytes) using a variety of encodings.
- UTF-8 is one such encoding. The low codepoints are encoded using a single byte, and higher codepoints are encoded as sequences of bytes.
- In python Unicode strings use the prefix u.
- Python's unicode type is a collection of codepoints.

#The following creates a Unicode string with 20 characters.
>>>ustring = u'A unicode \u018e string \xf1'
>>>print(ustring)
A unicode Ǝ string ñ

>>>S = ustring.encode('utf-8') → encodes the Unicode string using UTF-8.
This converts each codepoint to the appropriate byte or sequence of bytes. The result is a collection of bytes, which is returned as a str. The size of S is 22 bytes, because two of the characters have high codepoints and are encoded as a sequence of two bytes rather than a single byte.

>>>t = unicode(S, 'utf-8') → It does the opposite of encode().
It reconstructs the original codepoints by looking at the bytes of S and parsing byte sequences. The result is a Unicode string.

## 1.9 Built-in String Methods

Python supports many built-in methods to manipulate strings. A method is just like a function. The only difference between a function and method is that method is invoked or called on an object.

| Sr.No. | Methods with Description | Example |
|---|---|---|
| 1 | capitalize()<br><br>Capitalizes first letter of string | str="hello"<br><br>print(str.capitalize())<br><br>Output: Hello |

| | | |
|---|---|---|
| 2 | center(width, fillchar)<br>Returns a space-padded string with the original string centered to a total of width columns. | str=”hello”<br><br>print(str.center(10,’*’))<br><br>Output: ** Hello*** |
| 3 | count(str, beg= 0,end=len(string))<br>Counts how many times str occurs in string or in a substring of string if starting index beg and ending index end are given. | str=’he’<br>mes=’helloworldhellohello’<br>print(mes.count(str,0,len(mes))<br><br>Output: 3 |
| 4 | endswith(suffix, beg=0, end=len(string))<br>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. | M=’She is my best friend’<br>print(M.endswith(‘end’,0,len(M))<br><br>Output: True |
| 5 | startswith(str, beg=0,end=len(string))<br>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. | M=’The world is beautiful’<br>print(M.startswith(‘The’,0,len(M))<br><br>Output: True |
| 6 | find(str, beg=0, end=len(string))<br>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. | M=’She is my best friend’<br>print(M.find(‘my’,0,len(M))<br><br>Output: 7 |
| 7 | index(str, beg=0, end=len(string))<br>Same as find(), but raises an exception if str not found. | M=’She is my best friend’<br>print(M.index(‘best’,0,len(M))<br><br>Output: 10 |
| 8 | isalnum()<br>Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise. | Mes=”Jamesbond007”<br>print(Mes.isalnum())<br><br>Output: True |
| 9 | isalpha()<br>Returns true if string has at least 1 character and all characters are alphabetic and false otherwise. | Mes=”Jamesbond007”<br>print(Mes.isalpha())<br><br>Output: False |
| 10 | isdigit()<br>Returns true if string contains only digits and false otherwise. | Mes=”Jamesbond007”<br>print(Mes.isdigit())<br>Output: False |

| | | |
|---|---|---|
| 11 | islower()<br>Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise. | S=”welcome”<br>print(S.islower())<br><br>Output: True |
| 12 | isspace()<br>Returns true if string contains only whitespace characters and false otherwise. | mes=” “<br>print(mes.isspace())<br><br>Output: True |
| 13 | istitle()<br>Returns true if string is properly "titlecased" and false otherwise. | s="Hello world. How are you"<br>print(s.istitle())<br><br>Output: False |
| 14 | isupper()<br>Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise. | S=”HELLO”<br>print(S.isupper())<br><br>Output: True |
| 15 | join(seq)<br>Merges (concatenates) the string representations of elements in sequence seq into a string, with separator string. | L=['apple','banana','citrus','guava']<br>print('-'.join(L))<br><br>Output:<br>apple-banana-citrus-guava |
| 16 | ljust(width[, fillchar])<br>Returns a space-padded string with the original string left-justified to a total of width columns. | str=”Hello”<br><br>print(str.ljust(10,’*’))<br><br>Output: Hello***** |
| 17 | rjust(width[, fillchar])<br>Returns a space-padded string with the original string Right-justified to a total of width columns. | str=”Hello”<br><br>print(str.rjust(10,’*’))<br><br>Output: *****Hello |
| 18 | lower()<br>Converts all uppercase letters in string to lowercase. | str=”Hello”<br><br>print(str.lower())<br><br>Output: hello |
| 19 | lstrip()<br>Removes all leading whitespace in string. | str=” Hello”<br>print(str.lstrip())<br><br>Output: Hello |
| 20 | max(str)<br>Returns the max alphabetical character from the string str. | Str=”hello friendz”<br>Print(max(str))<br>Output: z |

| | | |
|---|---|---|
| 21 | min(str)<br>Returns the min alphabetical character from the string str. | Str=”hello friendz”<br>Print(min(str))<br>Output: d |
| 22 | replace(old, new [, max])<br>Replaces all occurrences of old in string with new or at most max occurrences if max given. | Str=”hello hello hello”<br>Print(Str.replace(‘he’,’fo’))<br><br>Output:<br>follo follo follo |
| 23 | rfind(str, beg=0,end=len(string))<br>Same as find(), but search backwards in string. | Str=”Is this your bag?”<br>Print(Str.rfind(‘is’,0,len(Str)))<br><br>Output: 5 |
| 24 | rindex( str, beg=0, end=len(string))<br>Same as index(), but search backwards in string. | Str=”Is this your bag?”<br>Print(Str.rindex(‘you’,0,len(Str)))<br><br>Output: 8 |
| 25 | rstrip()<br>Removes all trailing whitespace of string. | str=”Hello ”<br><br>print(str.rstrip())<br>Output: Hello |
| 26 | split(delim)<br>Splits string according to delimiter (space if not provided) and returns list of substrings. | Str=”one,two,three,four,five”<br>Print(Str.split(‘,’))<br>Output:<br>[‘one’,’two’,’three’,’four’,’five’] |
| 27 | strip([chars])<br>Performs both lstrip() and rstrip() on string. | str=” Hello ”<br><br>print(str.strip())<br>Output: Hello |
| 28 | swapcase()<br>Inverts case for all letters in string. | S='The World is Beautiful'<br>print(S.swapcase())<br>Output:<br>tHE wORLD IS bEAUTIFUL |
| 29 | title()<br>Returns "titlecased" version of string, that is, all words begin with uppercase and the rest are lowercase. | S='The World is Beautiful'<br>print(S.title())<br>Output:<br>The World Is Beautiful |
| 30 | enumerate(str)<br><br>Returns an enumerated object that lists the index and value of all characters in the string as pairs | S='Hello Python'<br>print(list(enumerate(S)))<br><br>Output:<br>[(0, 'H'), (1, 'e'), (2, 'l'), (3, 'l'), (4, 'o'), (5, ' '), (6, 'P'), (7, 'y'), (8, 't'), (9, 'h'), (10, 'o'), (11, 'n')] |

| | | |
|---|---|---|
| 31 | upper()<br>Converts lowercase letters in string to uppercase. | str=”Hello”<br><br>print(str.upper())<br>Output: HELLO |
| 32 | zfill (width)<br>Returns original string leftpadded with zeros to a total of width characters; intended for numbers, zfill() retains any sign given (less one zero). | Str=”1234”<br>Print(Str.zfill(10))<br><br>Output:<br>0000001234 |
| 33 | isidentifier()<br>Returns true if a string is valid Identifier and false otherwise. | s="num1"<br>print(s.isidentifier())<br>Output: True<br><br>s="Num-1"<br>print(s.isidentifier())<br>Output: False |
| 34 | len()<br>Returns the length of string | Str=”Hello”<br>Print(len(Str))<br>Output: 5 |

## 2. Functions

- A function is a block of organized, reusable code that performs a single, specific and well-defined task.
- Python enables the programmers to break up program into functions each of which can be independently of the others.
- Functions provide better modularity for application and a high degree of code reusing.
- There are many built-in functions like print(), input() and int() etc. but we can also create our own functions. These functions are called user-defined function.

### 2.1 Defining a Function

We can define functions to provide the required functionality. Here are simple rules to define a function in Python.

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

- Any input parameters or arguments should be placed within these parentheses.
- We can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or **docstring.**
- The code block within every function starts with a colon : and is indented.
- The statement return [expression] exits a function, optionally passing back an expression to the caller.
- A return statement with no arguments is the same as return None.

Syntax:

```
def   functionname( parameters ):
      "function_docstring"
       function_suite
       return [expression]
```

By default, parameters have a positional behavior and we need to inform them in the same order that they were defined.

Example:

The following function takes a string as input parameter and prints it on standard screen.

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

We can access the docstring by **__doc__** attribute like **FunctionName.__doc__**
Here is the example to show how we can access the docstring in Python functions.

```
def    Hello():

       """ Hello World """

       print ('Hi')

print ("The doctsring of the function Hello is: "+ Hello.__doc__)
```

**Output:**
The docstring of the function Hello is:  Hello World

## 2.2 Calling a Function:

Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code. Once the basic structure of a function is finalized, we can execute it by calling it from another function or directly from the Python prompt.

Following is the example to call Display function −

```
# Function definition is here
def   Display( str ):
        "This prints a passed string into this function"
        print (str)
        return;

# Now  we can call Display()  function
Display("First call to user defined function Display!")
Display("Again second call to the same function")
```

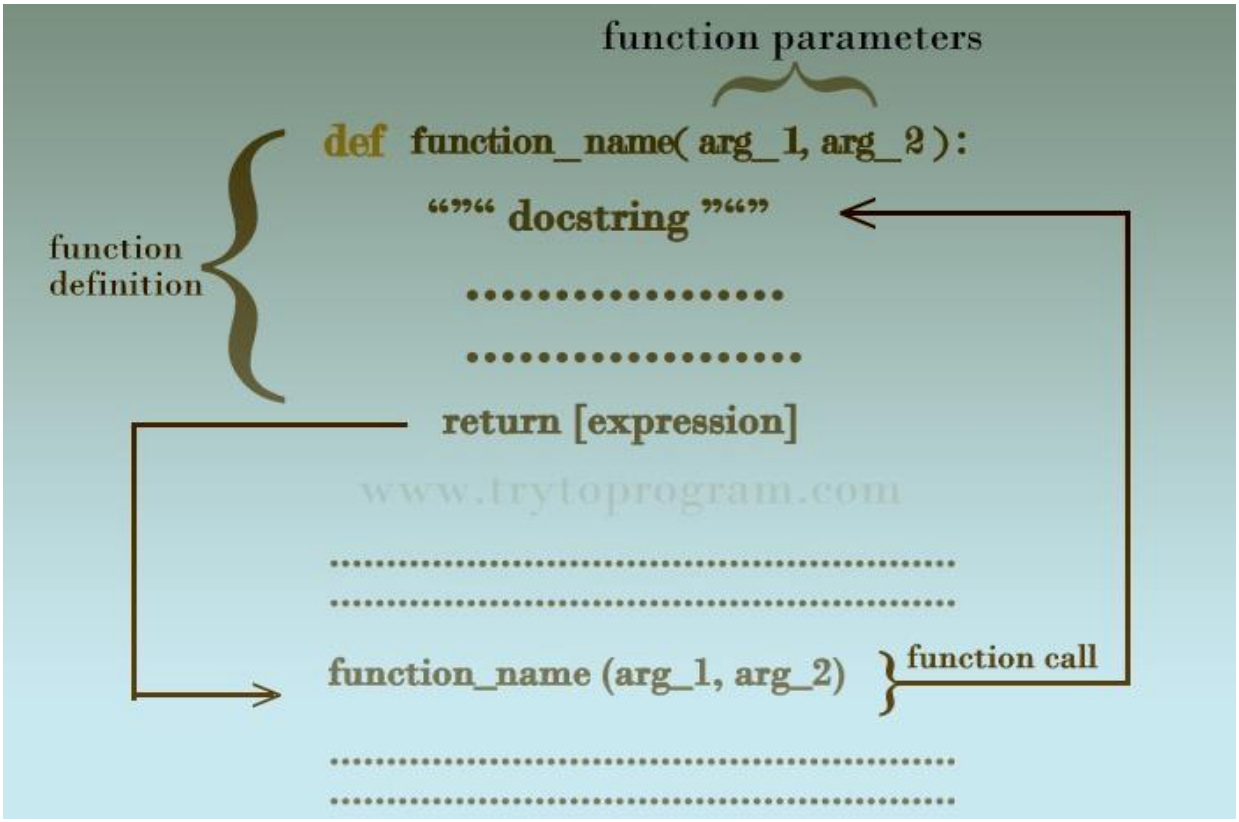
**Output:**

```
First call to user defined function Display!
Again second call to the sam e function
```

## Working of functions:

A Python function consists of **function definition** where the functionality of a function is defined. Function definition as seen in below diagram consists of a **function name, function arguments, docstring, code statements, and the return statement.**

Once a function is defined, we need to call the function in the main program to execute the function. For that, there is function call statement.




**Passing parameters by object reference:**

All parameters (arguments) in the Python functions are passed by object reference. It means we change a parameter refers to within a function, the change also reflects back in the calling function.

Example-1:

```
# Function definition is here
def  changeme( mylist ):
        "This changes a passed list into this function"
        mylist.append([1,2,3,4]);
        print("Values inside the function: ", mylist)
        return

# Now  we can call changeme()  function
mylist = [10,20,30];
changeme( mylist );
print("Values outside the function: ", mylist)
```

Here, we are maintaining reference of the passed object and appending values in the same object. So, this would produce the following result –

**Values inside the function: [10, 20, 30, [1, 2, 3, 4]]**

**Values outside the function: [10, 20, 30, [1, 2, 3, 4]]**

Example-2:

In this example argument is being passed by reference and the reference is being overwritten inside the called function.

```
# Function definition is here
def    changeme( mylist ):
       "This changes a passed list into this function"
       mylist = [1,2,3,4];   #This would assign new reference in mylist
       print( "Values inside the function: ", mylist)
       return

# Now we can call changeme()  function
mylist = [10,20,30];
changeme(mylist );
print("Values outside the function: ", mylist)
```

The parameter mylist is local to the function changeme(). Changing mylist within the function does not affect mylist. The function accomplishes nothing and finally this would produce the following result:

**Values inside the function: [1, 2, 3, 4]**
**Values outside the function: [10, 20, 30]**

## 2.3 Types of Functions

There are four types of functions in Python.

- Built-in Functions
  These are the built-in functions of Python with pre-defined functionalities. There are many built-in functions like print(), input() and int() etc.

- User-defined Functions

  These are the functions defined by the users as per the requirement at different stages. We use **def** keyword to define our own functionalities and give a name to these functions.

- User-defined function with return type has return statements to return values after calculation.
- User-defined function with non-return type has no return statement.

- Lambda functions (Anonymous functions)
  - These functions are called anonymous because they are not declared in the standard manner by using the **def** keyword.
  - We can use the **lambda** keyword to create small anonymous functions.
  - Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
  - An anonymous function cannot be a direct call to print because lambda requires an expression
  - Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.
- Recursive functions

  A function is called recursive, if the body of function calls the function itself until the condition for recursion is true. Thus, a Python recursive function has a termination condition (base condition).

## 2.4 Function Arguments

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

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

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

To call the function display(), we definitely need to pass one argument, otherwise it gives a syntax error as follows:

```
# Function definition is here
def   Display(str ):
        "This prints a passed string into this function"
        print str
        return;
# Now  we can call Display() function

S='Hello  How are You'
Display()          #this would raise  Type Error as it needs one argument
Display(S)         #takes exactly 1 argument
```

**Keyword arguments:**

With Keyword arguments, we can use the name of the parameter irrespective of its position while calling the function to supply the values. All the keyword arguments must match one of the arguments accepted by the function.

Example:

```
def    print_name(name1, name2):
        """ This function prints the name """
        print (name1 + " and " + name2 + " are friends")

#calling the function
print_name(name2 = 'John',name1 = 'Gary')
```

**Output:**

Gary and John are friends

**Default  arguments:**

Sometimes we may want to use parameters in a function that takes default values in case the user doesn't want to provide a value for them.

For this, we can use default arguments which assumes a default value if a value is not supplied as an argument while calling the function. In parameters list, we can give default values to one or more parameters.

An assignment operator '=' is used to give a default value to an argument.

Example:

```
def   sum(a=4, b=2):     #2 is supplied as default argument
      """ This function will print sum of two numbers
      print ("Sum is ",a+b)

sum(1,2)  #calling with arguments
sum( )    #calling without arguments, default values will be taken
```

**Output**

```
Sum is  3
Sum is  6
```

**Variable-length arguments:**

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

**Syntax for a function with non-keyword variable arguments is:**

```
def   functionnam e([form al_args,] * var_args_tuple ):
      "function_docstring"
       statements
       return [expression]
```

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

Following is a simple example −

```
# Function definition
def   printinfo( arg1, *vartuple ):
      "This prints a variable passed arguments"
      print(arg1)
      for  i  in  vartuple:
            print (i)
      return;

print("First  Information  :")
printinfo(10)
print("Second  Information  :")
printinfo(70,60,50)
```

**Output:**
First Information :
10
Second Information :
70
60
50

## 2.5 The Anonymous Functions (Lambda functions)

- Anonymous functions are not declared in the standard manner by using the **def** keyword. We use the **lambda** keyword to create small anonymous functions.

- Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
- Syntax:
  lambda [arg1 [,arg2,.....argn]]: expression

Example-1:

```
# Function definition is here
sum = lambda arg1, arg2: arg1 + arg2;

# Now you can call sum as a function
print "Value of total : ", sum ( 10, 20 )
print "Value of total : ", sum ( 20, 20 )
```

**Output:**
Value of total : 30
Value of total : 40

Example-2:

```
# Function definition is here
square = lambda x: x * x;

# Now you can call square as a function
print ("Value of square is : ", square(10) )
print ("Value of square is : ", sum ( 20, 20 )
```

**Output:**
Value of total : 30
Value of total : 40

## 2.6 Scope and lifetime of the variable

The scope of a variable determines its accessibility and availability in different portions of a program. Their availability depends on where they are defined. Similarly, life is a period in which the variable is stored in the memory.

Depending on the scope and the lifetime, there are two kinds of variables in Python.

- **Local Variables**
- **Global Variables**

Local Variables vs Global Variables:

Some of the points to list out the difference between global and local variable.

- Variables or parameters defined inside a function are called local variables as their scope is limited to the function only. On the contrary, Global variables are defined outside of the function.
- Local variables can't be used outside the function whereas a global variable can be used throughout the program anywhere as per requirement.
- The lifetime of a local variable ends with the termination or the execution of a function, whereas the lifetime of a global variable ends with the termination of the entire program.
- The variable defined inside a function can also be made global by using the global statement.

**Example:**

```
total = 0          # This is global variable.
# Function definition is here
def    sum (arg1, arg2 ):
       # Add both the param eters and return them ."
       total = arg1 + arg2;              # Here total is local variable.
       print ("Inside the function local total : ", total)
       return total;

# calling sum function
sum (10, 20 );
print("Outside the function global total : ", total)
```

**Output:**
Inside the function local total : 30
Outside the function global total : 0

Syntax to create **global** variable inside a function:

```
def   function_name(args):
      .............
      global   x   #Using global keyword to declare  global variable inside a function
      .............
```

## 2.7 Recursion:

Recursion is a method 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 satisfies the condition of recursion, we call this function a recursive function.

Termination condition: 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 end up in an infinite loop, if the base case is not met in the calls.

Example:

4! = 4 * 3!

3! = 3 * 2!

2! = 2 * 1

Replacing the calculated values gives us the following expression

4! = 4 * 3 * 2 * 1

A function is called recursive, if the body of function calls the function itself until the condition for recursion is true. Thus, a Python recursive function has a termination condition (base condition).

### Example : Python recursive function to find the factorial of 3

3! = 3 * 2! = 3 * (2 * 1!) = 3 * 2 * 1
This is how a factorial is calculated. Let's implement this same logic into a program.

```
#recursive function to calculate factorial
def   fact(n):
       """ Function to find factorial """
       if  n == 1:
           return 1
      else:
```

```
        return (n * fact(n-1))

#calling  fact function
print ("3! = ",fact(3))
```

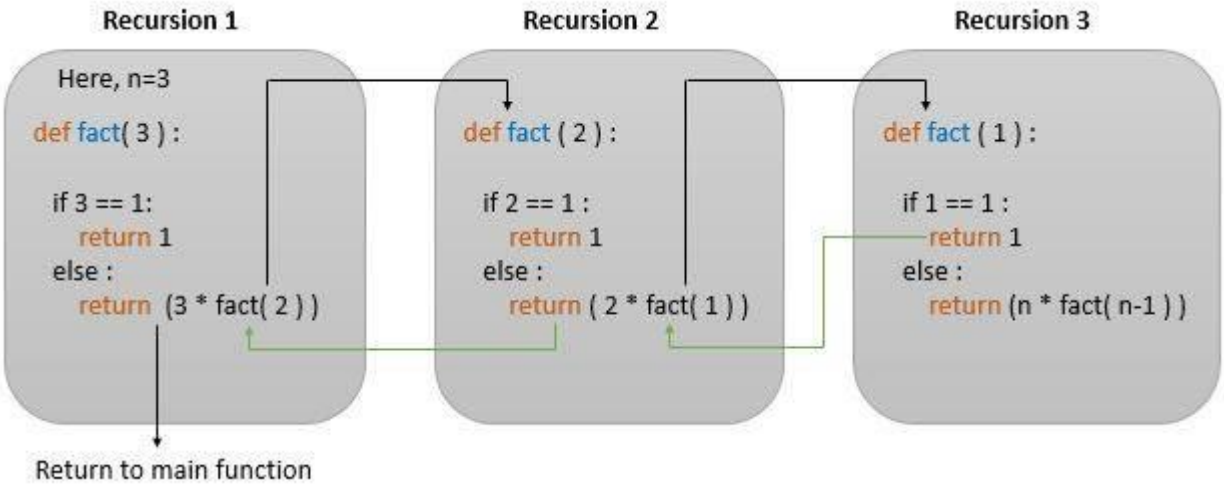
**Output**
3! = 6

This program can calculate factorial of any number supplied as the argument

Explanation of the program:

- First is a base condition in any recursive function. If the value of n is equal to 1, then the function will return 1 and exit. Till this base condition is met, the function will be iterated.

- In the program above, the value of the argument supplied is 3. Hence, the program operates in following way.




When the function is called with the value of n=3

**In recursion 1:** Function returns 3 * fact(2). This invokes the function again with the value of n = 2.

**In recursion 2**: Function checks if n = 1. Since it's False function return 3 * 2 * fact(1). This again invokes the function with the value of n = 1.
**In recursion 3**: Function checks if n = 1. This returns True making the function to exit and return 1.

Hence after 3 recursions, the final value returned is 3 * 2 * 1 = 6.

Advantages of Python Recursion

1. Reduces unnecessary calling of function, thus reduces length of program.
2. Very flexible in data structure like stacks, queues, linked list and quick sort.
3. Big and complex iterative solutions are easy and simple with Python recursion.
4. Algorithms can be defined recursively making it much easier to visualize and prove.

Disadvantages of Python Recursion

1. Slow.
2. Logical but difficult to trace and debug.
3. Requires extra storage space. For every recursive calls separate memory is allocated for the variables.

4. Recursive functions often throw a Stack Overflow Exception when processing or operations are too large.

**UNIT-IV**

# Python Tuples, sets and files

**Python Tuples** - Introduction, Creating & Deleting Tuples, Accessing values in a Tuple, Updating tuples, Delete Tuple Elements, basic Tuple Operations, Indexing, Slicing and Matrices, built- in tuple Functions.
Sets - Concept, Operations.
Files: Creating files, Operations on Files (open, close, read, write).

## 1. Tuples:

In Python programming, a tuple is similar to a list. The difference between the two is that we cannot change the elements of a tuple once it is assigned whereas in a list, elements can be changed.

**Advantages of Tuple over List**

Since, tuples are quite similiar to lists, both of them are used in similar situations as well. However, there are certain advantages of implementing a tuple over a list.

- We generally use tuple for heterogeneous (different) datatypes and list for homogeneous (similar) datatypes.
- Since tuple are immutable, iterating through tuple is faster than with list. So there is a slight performance boost.
- Tuples that contain immutable elements can be used as key for a dictionary. With list, this is not possible.
- If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected.

## 1.1 create a Tuple in Python

A tuple is created by placing all the items (elements) inside a parentheses (), separated by comma. The parentheses are optional but is a good practice to write it.

A tuple can have any number of items and they may be of different types (integer, float, list, string etc.

```
my_tuple = ()
print(my_tuple)

# tuple having integers
```

```
my_tuple = (1, 2, 3)
print(my_tuple)
Output: (1, 2, 3)

# tuple with mixed datatypes
my_tuple = (1, "Hello", 3.4)
print(my_tuple)
 Output: (1, "Hello", 3.4)
# nested tuple
my_tuple = ("mouse", [8, 4, 6], (1, 2, 3))
print(my_tuple)
Output:   ("mouse", [8, 4, 6], (1, 2, 3))

# tuple can be created without parenthese also called tuple packing
my_tuple = 3, 4.6, "dog"
print(my_tuple)
 Output: 3, 4.6, "dog"

# tuple unpacking is also possible
a, b, c = my_tuple
print(a)
print(b)
print(c)
 Output:
 3
 4.6
 dog
```

## 1.2 How to access elements in a tuple?

There are various ways in which we can access the elements of a tuple.

**1. Positive Indexing**

We can use the index operator [] to access an item in a tuple where the index starts from 0.

So, a tuple having 6 elements will have index from 0 to 5. Trying to access an element other that (6, 7,...) will raise an IndexError.

The index must be an integer, so we cannot use float or other types. This will result into TypeError.

Likewise, nested  tuple are accessed using nested indexing, as shown in the example below.

```
my_tuple = ('p','e','r','m','i','t')
# Output: 'p'
print(my_tuple[0])

# Output: 't'
print(my_tuple[5])

# IndexError: list index out of range
print(my_tuple[6])

# TypeError: list indices must be integers, not float
my_tuple[2.0]

# nested tuple
n_tuple = ("mouse", [8, 4, 6], (1, 2, 3))

# nested index
# Output: 's'
print(n_tuple[0][3])

# nested index
# Output: 4
print(n_tuple[1][1])
```

**2. Negative Indexing**

Python allows negative indexing for its sequences.
The index of -1 refers to the last item, -2 to the second last item and so on.

```
my_tuple = ('p','e','r','m','i','t')

# Output: 't'
print(my_tuple[-1])

# Output: 'p'
print(my_tuple[-6])
```

**3. Slicing**

We can access a range of items in a tuple by using the slicing operator - colon ":".

```
my_tuple = ('p','r','o','g','r','a','m','i','z')
```

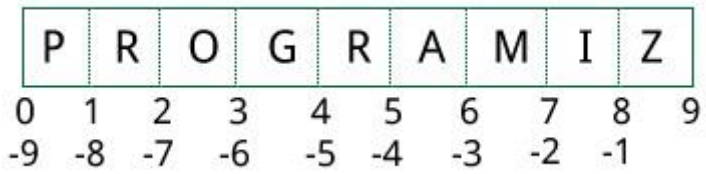
```
# elements 2nd to 4th
# Output: ('r', 'o', 'g')
print(my_tuple[1:4])

# elements beginning to 2nd
# Output: ('p', 'r')
print(my_tuple[:-7])

# elements 8th to end
# Output: ('i', 'z')
print(my_tuple[7:])

# elements beginning to end
# Output: ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
print(my_tuple[:])
```

Slicing can be best visualized by considering the index to be between the elements as shown below. So if we want to access a range, we need the index that will slice the portion from the tuple.




## 1.3 How to change elements in a tuple?

Unlike lists, tuples are immutable. This means that elements of a tuple cannot be changed once it has been assigned. But, if the element is itself a mutable datatype like list, its nested items can be changed. We can also assign a tuple to different values (reassignment).

```
my_tuple = (4, 2, 3, [6, 5])
# we cannot change an element
# you will get an error:
# TypeError: 'tuple' object does not support item assignment
my_tuple[1] = 9

# tuples can be reassigned
my_tuple = ('p','r','o','g','r','a','m','i','z')
print(my_tuple)
```

- We can use + operator to combine two tuples. This is also called concatenation.

We can also repeat the elements in a tuple for a given number of times using the * operator. Both + and * operations result into a new tuple.

```
# Concatenation
# Output: (1, 2, 3, 4, 5, 6)
print((1, 2, 3) + (4, 5, 6))

# Repeat
# Output: ('Repeat', 'Repeat', 'Repeat')
print(("Repeat",) * 3)
```

## 1.4 How to delete elements in a tuple?

As discussed above, we cannot change the elements in a tuple. That also means we cannot delete or remove items from a tuple. But deleting a tuple entirely is possible using the keyword **del**.

```
my_tuple = ('p','r','o','g','r','a','m','i','z')

# can't delete items, you will get an error:
# TypeError: 'tuple' object doesn't support item deletion
del my_tuple[3]

del my_tuple
```

## 1.5 Basic operations on tuple

### 1. concatenation

We can use + operator to combine two tuples. This is also called concatenation.

```
# Concatenation
# Output: (1, 2, 3, 4, 5, 6)
print((1, 2, 3) + (4, 5, 6))
```

### 2. Repetetion

We can also repeat the elements in a tuple for a given number of times using the * operator.

```
# Repeat
# Output: (1,2,3,1,2,3,1,2,3)

print((1,2,3) * 3)
```

Both + and * operations result into a new tuple.

### 3. Membership Test

We can test if an item exists in a tuple or not, using the keyword in.

```
my_tuple = ('a','p','p','l','e',)

# In operation
# Output: True
print('a' in my_tuple)

# Output: False
print('b' in my_tuple)

# Not in operation
# Output: True
print('g' not in my_tuple)
```

### 4. Iterating Through a Tuple

Using a for loop we can iterate though each item in a tuple.

```
# Output:
# Hello John
# Hello Kate
for name in ('John','Kate'):
    print("Hello", name)
```

### 5. Built-in Functions with Tuple

| Built-in Functions with Tuple | |
|---|---|
| Function | Description |
| all() | Return True if all elements of the tuple are true (or if the tuple is empty). |
| any() | Return True if any element of the tuple is true. If the tuple is empty, return False. |

| | |
|---|---|
| enumerate() | Return an enumerate object. It contains the index and value of all the items of tuple as pairs. |
| len() | Return the length (the number of items) in the tuple. |
| max() | Return the largest item in the tuple. |
| min() | Return the smallest item in the tuple |
| sorted() | Take elements in the tuple and return a new sorted list (does not sort the tuple itself). |
| sum() | Retrun the sum of all elements in the tuple. |
| tuple() | Convert an iterable (list, string, set, dictionary) to a tuple. |

## 1.6 Tuple Methods

Methods that add items or remove items are not available with tuple. Only the following two methods are available.

| Method | Description |
|---|---|
| count(x) | Return the number of items that is equal to x |
| index(x) | Return index of first item that is equal to x |

Some examples of Python tuple methods:

```
my_tuple = ('a','p','p','l','e',)
# Count
# Output: 2
print(my_tuple.count('p'))

# Index
# Output: 3
print(my_tuple.index('l'))
```

## 2.Sets

Mathematically a set is a collection of items not in any particular order. A Python set is similar to list with no duplicate entries. Set is mutable and un-ordered collection of elements.

- The elements in the set cannot be duplicates.
- The set as a whole is mutable. We can easily add or remove items in it.
- There is no index attached to any element in a python set. So they do not support any indexing or slicing operation.

## 2.1 Creating set

A set is created by using the set() function or placing all the elements within a pair of curly braces {}.

```
Days=set(["Mon","Tue","Wed","Thu","Fri","Sat","Sun"])
Months={"Jan","Feb","Mar"}
Dates={21,22,17}
print(Days)
print(Months)
print(Dates)
```

When the above code is executed, it produces the following result. Please note how the order of the elements has changed in the result.

```
set(['Wed', 'Sun', 'Fri', 'Tue', 'Mon', 'Thu', 'Sat'])
set(['Jan', 'Mar', 'Feb'])
set([17, 21, 22])
```

## 2.2 Accessing elements in set

We cannot access individual values in a set. We can only access all the elements together as shown above. But we can also get a list of individual elements by looping through the set.

```
Days=set(["Mon","Tue","Wed","Thu","Fri","Sat","Sun"])

for d in Days:
        print(d)
```

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

```
Wed
Sun
Fri
Tue
```

Mon
Thu
Sat

## 2.3 Adding elements to set

We can add elements to a set by using add() method. Again as discussed there is no specific index attached to the newly added element.

```
Days=set(["Mon","Tue","Wed","Thu","Fri","Sat"])
Days.add("Sun")
print(Days)
```

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

```
set(['Wed', 'Sun', 'Fri', 'Tue', 'Mon', 'Thu', 'Sat'])
```

## 2.4 Removing elements from set

We can remove elements from a set by using discard() method. Again as discussed there is no specific index attached to the newly added element.

```
Days=set(["Mon","Tue","Wed","Thu","Fri","Sat"])

Days.discard("Sun")
print(Days)
```

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

```
set(['Wed', 'Fri', 'Tue', 'Mon', 'Thu', 'Sat'])
```

## 2.5 Set operations

The sets in python are typically used for mathematical operations like union, intersection, difference and complement etc. We can create a set, access it's elements and carry out these mathematical operations as shown below.

### 1. Union of Sets

The union operation on two sets produces a new set containing all the distinct elements from both the sets. In the below example the element "Wed" is present in both the sets.

```
DaysA = set(["Mon","Tue","Wed"])
```

```
DaysB = set(["Wed","Thu","Fri","Sat","Sun"])
AllDays = DaysA|DaysB
print(AllDays)
```

When the above code is executed, it produces the following result. Please note the result has only one “wed”.

```
set(['Wed', 'Fri', 'Tue', 'Mon', 'Thu', 'Sat'])
```

## 2. Intersection of Sets

The intersection operation on two sets produces a new set containing only the common elements from both the sets. In the below example the element “Wed” is present in both the sets.

```
DaysA = set(["Mon","Tue","Wed"])
DaysB = set(["Wed","Thu","Fri","Sat","Sun"])
AllDays = DaysA & DaysB
print(AllDays)
```

When the above code is executed, it produces the following result. Please note the result has only one “wed”.

```
set(['Wed'])
```

## 3. Difference of Sets

The difference operation on two sets produces a new set containing only the elements from the first set and none from the second set. In the below example the element “Wed” is present in both the sets so it will not be found in the result set.

```
DaysA = set(["Mon","Tue","Wed"])
DaysB = set(["Wed","Thu","Fri","Sat","Sun"])
AllDays = DaysA - DaysB
print(AllDays)
```

When the above code is executed, it produces the following result. Please note the result has only one “wed”.

```
set(['Mon', 'Tue'])
```

## 4. Symmetric Difference of Sets

The Symmetric difference operation on two sets produces a new set containing the elements which are not common from the first set and from the second set. In the below example the element “Wed” is present in both the sets so it will not be found in the result set.

```
DaysA = set(["Mon","Tue","Wed"])
DaysB = set(["Wed","Thu","Fri","Sat","Sun"])
AllDays = DaysA ^ DaysB
print (AllDays)
```

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

```
set(['Mon', 'Tue','Thu','Fri','Sat','Sun'])
```

## 4. Comparision of Sets

We can check if a given set is a subset or superset of another set. The result is True or False depending on the elements present in the sets.

```
DaysA = set(["Mon","Tue","Wed"])
DaysB = set(["Mon","Tue","Wed","Thu","Fri","Sat","Sun"])
Subset = DaysA <= DaysB
Superset = DaysB >= DaysA
print(Subset)
print(Superset)
```

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

```
True
True
```

## 2.6 Built-in functions and methods on sets

| Operation | Equivalent | Result |
|---|---|---|
| len(s) | | cardinality of set s |
| x in s | | test x for membership in s |
| x not in s | | test x for non-membership in s |
| s.issubset(t) | s <= t | test whether every element in s is in t |
| s.issuperset(t) | s >= t | test whether every element in t is in s |
| s.union(t) | s \| t | new set with elements from both s and t |
| s.intersection(t) | s & t | new set with elements common to s and t |
| s.difference(t) | s - t | new set with elements in s but not in t |
| s.symmetric_difference(t) | s ^ t | new set with elements in either s or t but not both |
| s.copy() | | new set with a shallow copy of s |

| Operation | Equivalent | Result |
|---|---|---|
| s.update(t) | s \|= t | return set s with elements added from t |
| s.intersection_update(t) | s &= t | return set s keeping only elements also found in t |
| s.difference_update(t) | s -= t | return set s after removing elements found in t |
| s.symmetric_difference_update(t) | s ^= t | return set s with elements from s or t but not both |
| s.add(x) | | add element x to set s |
| s.remove(x) | | remove x from set s; raises KeyError if not present |
| s.discard(x) | | removes x from set s if present |
| s.pop() | | remove and return an arbitrary element from s; raises KeyError if empty |
| s.clear() | | remove all elements from set s |

## 3. Files in Python

Python provides basic functions and methods necessary to manipulate files by default. You can do most of the file manipulation using a file object.

### 3.1 open()

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 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).

Here is a list of the different modes of opening a file −

| Sr.No. | Modes & Description |
|---|---|
| 1 | R<br>Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode. |
| 2 | Rb<br>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. |
| 3 | r+<br>Opens a file for both reading and writing. The file pointer placed at the beginning of the file. |
| 4 | rb+<br>Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file. |
| 5 | W<br>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. |
| 6 | Wb<br>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. |
| 7 | w+<br>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 and writing. |
| 8 | wb+<br>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. |
| 9 | a<br>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. |

| | |
|---|---|
| 10 | ab<br>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. |
| 11 | a+<br>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. |
| 12 | ab+<br>Opens a file for both appending and reading in binary format. 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. |

## 3.2 file object Attributes

Once a file is opened and you have one file object, you can get various information related to that file.

Here is a list of all attributes related to file object −

| Sr.No. | Attribute & Description |
|---|---|
| 1 | file.closed<br>Returns true if file is closed, false otherwise. |
| 2 | file.mode<br>Returns access mode with which file was opened. |
| 3 | file.name<br>Returns name of the file. |

| | |
|---|---|
| 4 | file.softspace<br>Returns false if space explicitly required with print, true otherwise. |

Example:

```
# Open a file
f = open("foo.txt", "wb")
print ("Name of the file: ", f.name)
print("Closed or not : ", f.closed)
print("Opening mode : ", f.mode)
print("Softspace flag : ", f.softspace)
```

This produces the following result −

```
Name of the file:  foo.txt
Closed or not :  False
Opening mode :  wb
Softspace flag :  0
```

## 3.3 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

```
# Open a file
f = open("foo.txt", "wb")
print( "Name of the file: ", f.name)

# Close opend file
f.close()
```

This produces the following result −

```
Name of the file:  foo.txt
```

## 3.4 file opening with 'with' keyword

It is good practice to use the with keyword when dealing with file objects. The advantage is that the file is properly closed after its suite finishes, even if an exception is raised at some point.

```
>>> with open('workfile','r') as f:
...     read_data = f.read()
>>> f.closed
True
```

If you're not using the with keyword, then you should call f.close() to close the file and immediately free up any system resources used by it.

## 3.5 Reading and Writing Files

The file object provides a set of access methods to read and write.

**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.

Example

```
# Open a file
f = open("foo.txt", "wb")
f.write( "Python is a great language.\nYeah its great!!\n")

# Close opend file
f.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 following content.

Python is a great language.
Yeah its great!!

**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 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's take a file foo.txt, which we created above.

```
# Open a file
f = open("foo.txt", "r+")
str = f.read(10);
print ("Read String is : ", str)

# Close opend file
f.close()
```

This produces the following result −

Read String is :  Python is

**The readline() Method**

readline() reads a single line from the file; a newline character (\n) is left at the end of the string.

f=open('myfile.txt','r')

```
>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
''
```

For reading lines from a file, you can loop over the file object. This is memory efficient, fast, and leads to simple code:

```
>>> for  line  in  f:
...      print(line, end='')
...
This is the first line of the file.
Second line of the file
```

**The readlines() Method**

If you want to read all the lines of a file in a list you can use readlines().

```
f=open('myfile.txt','r')

lines= f.readlines()

for  x  in  lines:

      print(x, end=' ')
```

**The writelines() Method**

We can write list of lines to the opened file by using writelines() method.

```
f=open('myfile.txt','w')

lines= ['Hello ..','How are you..','Python language is simple']

f.writelines(lines)
```

## 3.6 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, it means use the beginning of the file as the reference position and 1 means use the current position as the reference position and if it is set to 2 then the end of the file would be taken as the reference position.

Example 1:

Let us take a file foo.txt, which we created already.

```
# Open a file
f= open("foo.txt", "r+")
str = f.read(10)
print (“Read String is : ", str)

# Check current position
position = f.tell()
print("Current file position : ", position)

# Reposition pointer at the beginning once again
position = f.seek(0, 0);
str = f.read(10)
print ("Again read String is : ", str)

# Close opened file
f.close()
```

This produces the following result −

```
Read String is :  Python is
Current file position :  10
Again read String is :  Python is
```

Example 2:

```
>>> f = open('workfile', 'r+')
>>> f.write('0123456789abcdef')
16
>>> f.seek(5)      # Go to the 6th byte in the file
5
>>> f.read(1)
'5'
>>> f.seek(-3, 2)  # Go to the 3rd byte before the end
13
>>> f.read(1)
'd'
```

**UNIT-V**

Searching Techniques – Linear Search and Binary Search.
Sorting Techniques – Bubble Sort, Selection Sort, Insertion Sort, Merge Sort and Heap Sort.
Exception Handling: Exception, Exception Handling, except clause, Try? finally clause, User Defined Exceptions.
Database: Introduction, Connections, Executing queries, Transactions, errorhandling.

## 1.Searching Techniques:

Search is a process of finding a value in a list of values. In other words, searching is the process of locating given value position in a list of values.

### 1. Linear Search Algorithm (Sequential Search Algorithm)

Linear search algorithm finds a given element in a list of elements with O(n) time complexity where n is total number of elements in the list. This search process starts comparing search element with the first element in the list. If both are matched then result is element found otherwise search element is compared with the next element in the list. Repeat the same until search element is compared with the last element in the list, if that last element also doesn't match, then the result is "Element not found in the list". That means, the search element is compared with element by element in the list.

Linear search is implemented using following steps...

- Step 1 - Read the search element from the user.
- Step 2 - Compare the search element with the first element in the list.
- Step 3 - If both are matched, then display "Given element is found!!!" and terminate the function
- Step 4 - If both are not matched, then compare search element with the next element in the list.
- Step 5 - Repeat steps 3 and 4 until search element is compared with last element in the list.
- Step 6 - If last element in the list also doesn't match, then display "Element is not found!!!" and terminate the function.

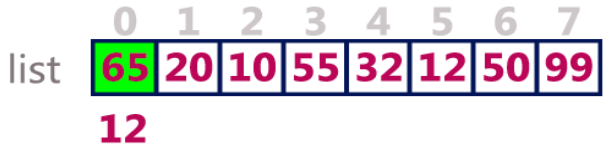
Example:

Consider the following list of elements and the element to be searched..


search element **12**

**Step 1:**

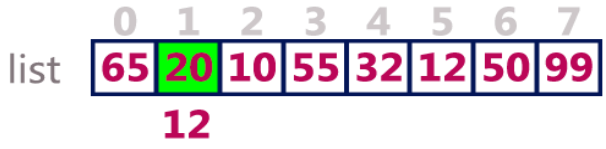
search element (12) is compared with first element (65)




Both are not matching. So move to next element

**Step 2:**

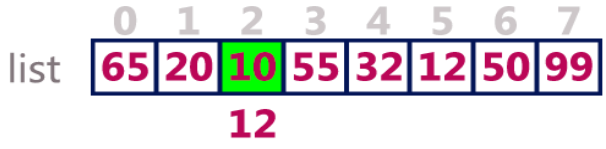
search element (12) is compared with next element (20)




Both are not matching. So move to next element

**Step 3:**

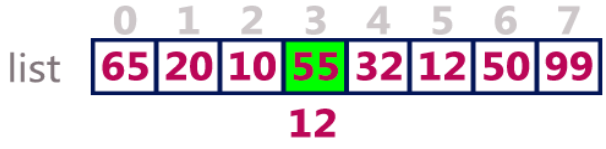
search element (12) is compared with next element (10)




Both are not matching. So move to next element

**Step 4:**

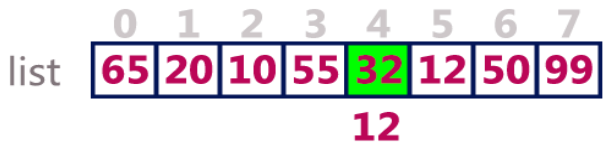
search element (12) is compared with next element (55)




Both are not matching. So move to next element

**Step 5:**

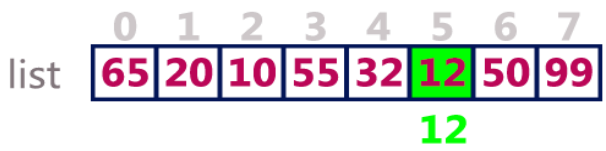
search element (12) is compared with next element (32)




Both are not matching. So move to next element

**Step 6:**

search element (12) is compared with next element (12)




Both are matching. So we stop comparing and display element found at index 5.

## 1.2 Binary Search Algorithm

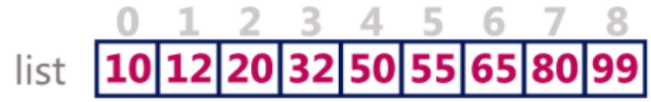
Binary search algorithm finds a given element in a list of elements with O(log n) time complexity where n is total number of elements in the list. The binary search algorithm can be used with only a sorted list of elements. That means the binary search is used only with a list of elements that are already arranged in an order. The binary search can not be used for a list of elements arranged in random order. This search process starts comparing the search element with the middle element in the list. If both are matched, then the result is "element found". Otherwise, we check whether the search element is smaller or larger than the middle element in the list. If the search element is smaller, then we repeat the same process for the left sublist of the middle element. If the search element is larger, then we repeat the same process for the right sublist of the middle element. We repeat this process until we find the search element in the list or until we left with a sublist of only one element. And if that element also doesn't match with the search element, then the result is "Element not found in the list".

Binary search is implemented using following steps...

- Step 1 - Read the search element from the user.
- Step 2 - Find the middle element in the sorted list.
- Step 3 - Compare the search element with the middle element in the sorted list.
- Step 4 - If both are matched, then display "Given element is found!!!" and terminate the function.
- Step 5 - If both are not matched, then check whether the search element is smaller or larger than the middle element.
- Step 6 - If the search element is smaller than middle element, repeat steps 2, 3, 4 and 5 for the left sublist of the middle element.
- Step 7 - If the search element is larger than middle element, repeat steps 2, 3, 4 and 5 for the right sublist of the middle element.
- Step 8 - Repeat the same process until we find the search element in the list or until sublist contains only one element.
- Step 9 - If that element also doesn't match with the search element, then display "Element is not found in the list!!!" and terminate the function.

Example

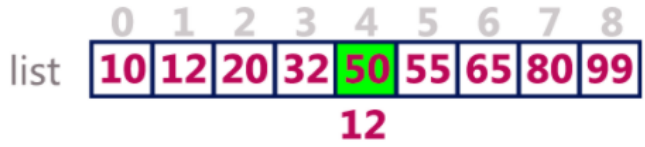
Consider the following list of elements and the element to be searched...




search element **12**

**Step 1:**

search element (12) is compared with middle element (50)




Both are not matching. And 12 is smaller than 50. So we search only in the left sublist (i.e. 10, 12, 20 & 32).


**Step 2:**

search element (12) is compared with middle element (12)


**Both are matching. So the result is "Element found at index 1"**

search element **80**

**Step 1:**

search element (80) is compared with middle element (50)


Both are not matching. And 80 is larger than 50. So we search only in the right sublist (i.e. 55, 65, 80 & 99).

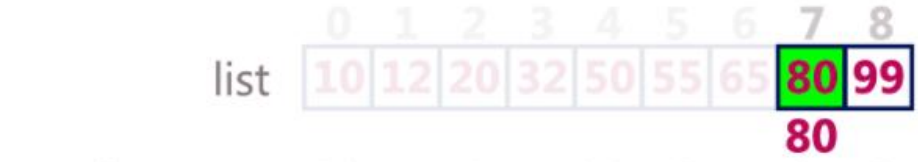

**Step 2:**

search element (80) is compared with middle element (65)


Both are not matching. And 80 is larger than 65. So we search only in the right sublist (i.e. 80 & 99).


**Step 3:**

search element (80) is compared with middle element (80)




**Both are not matching. So the result is "Element found at index 7"**

## 2.Sorting Techniques

### 2.1.Bubble Sorting:

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where n is the number of items.

How Bubble Sort Works?

We take an unsorted array for our example. Bubble sort takes $O(n^2)$ time complexity.

Bubble sort starts with very first two elements, comparing them to check which one is greater.


In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.


We find that 27 is smaller than 33 and these two values must be swapped.


The new array should look like this −


Next we compare 33 and 35. We find that both are in already sorted positions.

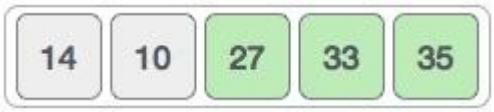

Then we move to the next two values, 35 and 10.


We know then that 10 is smaller 35. Hence they are not sorted.

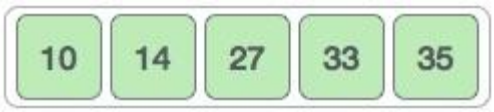

We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −


To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −


Notice that after each iteration, at least one value moves at the end.




And when there's no swap required, bubble sorts learns that an array is completely sorted.




## Algorithm

The bubble sort algorithm works as follows

**Step 1)** Get the total number of elements.

**Step 2)** Determine the number of outer passes (n - 1) to be done. Its length is list minus one

**Step 3)** Perform inner passes (n - 1) times for outer pass 1. Get the first element value and compare it with the second value. If the second value is less than the first value, then swap the positions

**Step 4)** Repeat step 3 passes until you reach the outer pass (n - 1). Get the next element in the list then repeat the process that was performed in step 3 until all the values have been placed in their correct ascending order.

**Step 5)** when all passes have been done,  Return the results of the sorted list.

## 2.2.Insertion Sort:

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'insert'ed in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, insertion sort.

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list in the same array. This algorithm is not suitable for large data sets as its average and worst case complexity are of O(n2), where n is the number of items.

How Insertion Sort Works?
We take an unsorted array for our example.


Insertion sort compares the first two elements.


It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.


Insertion sort moves ahead and compares 33 with 27.


And finds that 33 is not in the correct position.


It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27

is greater than 14. Hence, the sorted sub-list remains sorted after swapping.


By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.


These values are not in a sorted order.


So we swap them.


However, swapping makes 27 and 10 unsorted.


Hence, we swap them too.


Again we find 14 and 10 in an unsorted order.


We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.


This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.

**Algorithm:**

Simple steps by which we can achieve insertion sort.

Step 1 − If it is the first element, it is already sorted. return 1;
Step 2 − Pick next element
Step 3 − Compare with all elements in the sorted sub-list
Step 4 − Shift all the elements in the sorted sub-list that is greater than the value to be sorted
Step 5 − Insert the value
Step 6 − Repeat until list is sorted

## 2.3. Selection Sort:

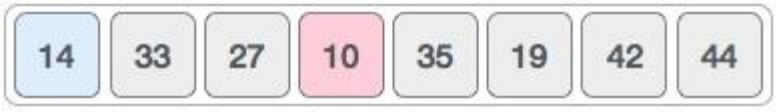
Selection sort is a simple sorting algorithm. The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

This algorithm is not suitable for large data sets as its average and worst case complexities are of $O(n^2)$, where n is the number of items.

How Selection Sort Works?

Consider the following depicted array as an example.

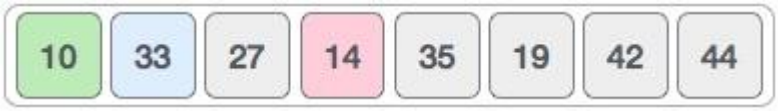

For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.




So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.


For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.


We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.




After two iterations, two least values are positioned at the beginning in a sorted manner.


The same process is applied to the rest of the items in the array.

Following is a pictorial depiction of the entire sorting process −


Now, let us learn some programming aspects of selection sort.

**Algorithm**

Step 1 − Set MIN to location 0

Step 2 − Search the minimum element in the list

Step 3 − Swap with value at location MIN

Step 4 − Increment MIN to point to next element

Step 5 − Repeat until list is sorted

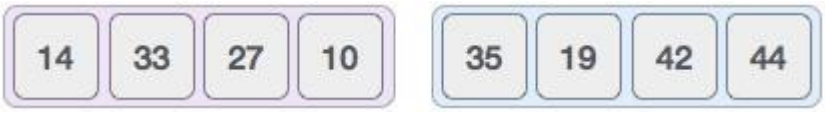
## 2.4. MergeSort:

Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being O(nlogn), it is one of the most respected algorithms.

How Merge Sort Works?

To understand merge sort, we take an unsorted array as the following –


We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.

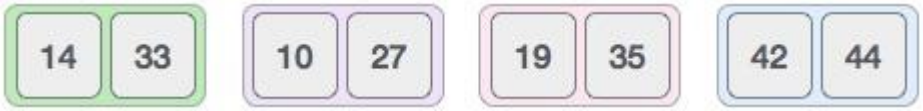



This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.


We further divide these arrays and we achieve atomic value which can no more be divided.


Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.

We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.



In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.


After the final merging, the list should look like this −


**Algorithm**

MergeSort(arr[ ], l, r)
If r > l

1. Find the middle point to divide the array into two halves:
   middle m = l + (r-l)/2
2. Call mergeSort for first half:
   Call mergeSort(arr, l, m)
3. Call mergeSort for second half:
   Call mergeSort(arr, m+1, r)
4. Merge the two halves sorted in step 2 and 3:
   Call merge(arr, l, m, r)

## 2.5. Quick Sorting:

Quick sort is a fast sorting algorithm used to sort a list of elements. The quick sort algorithm attempts to separate the list of elements into two parts and then sort each part recursively. That means it use divide and conque r strategy. In quick sort, the partition of the list is performed based on the element called pivot.
Here pivot element is one of the elements in the list. The list is divided into two partitions such that "all elements to the left of pivot are smaller than the pivot and all elements to the right of pivot are greater than or equal to the pivot".

**Step by Step Process**

In Quick sort algorithm, partitioning of the list is performed using following steps...

Step 1 - Consider the first element of the list as pivot (i.e., Element at first position in the list).

Step 2 - Define two variables i and j. Set i and j to first and last elements of the list respectively.

Step 3 - Increment i until list[i] > pivot then stop.

Step 4 - Decrement j until list[j] < pivot then stop.

Step 5 - If i < j then exchange list[i] and list[j].

Step 6 - Repeat steps 3,4 & 5 until i > j.

Step 7 - Exchange the pivot element with list[j] element.

Consider the following unsorted list of elements...


Define pivot, left & right. Set pivot = 0, left = 1 & right = 7. Here '7' indicates 'size-1'.


Compare List[left] with List[pivot]. If **List[left]** is greater than **List[pivot]** then stop left otherwise move left to the next.
Compare List[right] with List[pivot]. If List[right] is smaller than List[pivot] then stop right otherwise move right to the previous.
Repeat the same until **left>=right**.
If both left & right are stoped but left<right then swap List[left] with List[right] and countinue the process.
If left>=right then swap List[pivot] with List[right].


Compare List[left]<List[pivot] as it is true increment left by one and repeat the same, left will stop at 8.
Compare List[right]>List[pivot] as it is true decrement right by one and repeat the same, right will stop at 2.


Here left & right both are stoped and left is not greater than right so we need to swap List[left] and List[right]


Compare List[left]<List[pivot] as it is true increment left by one and repeat the same, left will stop at 6.
Compare List[right]>List[pivot] as it is true decrement right by one and repeat the same, right will stop at 4.


Here left & right both are stoped and left is greater than right so we need to swap List[pivot] and List[right]


Here we can observe that all the numbers to the left side of 5 are smaller and right side are greater. That means 5 is placed in its correct position.
Repeat the same process on the left sublist and right sublist to the number 5.


In the left sublist as there are no smaller number than the pivot left will keep on moving to the next and stops at last number. As the List[right] is smaller, right stops at same position. Now left and right both are equal so we swap pivot with right.


In the right sublist left is grester than the pivot, left will stop at same position.
As the List[right] is greater than List[pivot], right moves towords left and stops at pivot number position.
Now left > right so we swap pivot with right. (6 is swap by itself).


Repeat the same recursively on both left and right sublists until all the numbers are sorted.
The final sorted list will be as follows...

## 2.6 Heap Sort

Heap sorting is a comparison based sorting technique based on Binary Heap data structure.

A Binary Heap is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater(or smaller) than the values in its two children nodes. The former is called as max heap and the latter is called min-heap. The heap can be represented by a binary tree or array.

**Why array based representation for Binary Heap?**
Since a Binary Heap is a Complete Binary Tree, it can be easily represented as an array and the array-based representation is space-efficient. If the parent node is stored at index I, the left child can be calculated by 2 * I + 1 and right child by 2 * I + 2 (assuming the indexing starts at 0).

**Step by Step Process**

The Heap sort algorithm to arrange a list of elements in ascending order is performed using following steps...

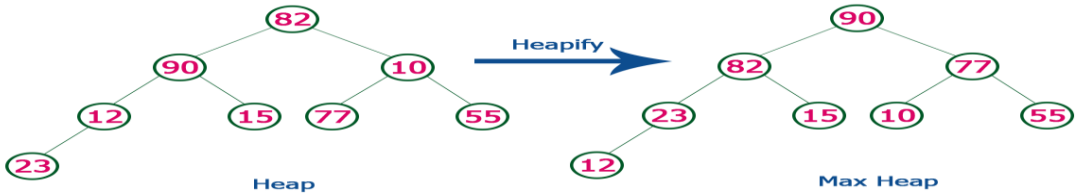
- Step 1 - Construct a Binary Tree with given list of Elements.
- Step 2 - Transform the Binary Tree (Heapify) into Max Heap.
- Step3- Delete the root element from Max Heap using Heapify method.
- Step 4 - Put the deleted element into the Sorted list.
- Step 5 - Repeat the same until Max Heap becomes empty.
- Step 6 - Display the sorted list.

Example

Consider the following list of unsorted numbers which are to be sort using Heap Sort

**82, 90, 10, 12, 15, 77, 55, 23**

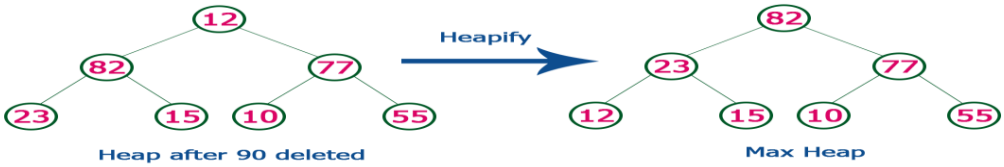
Step 1 - Construct a Heap with given list of unsorted numbers and convert to Max Heap




list of numbers after heap converted to Max Heap

**90, 82, 77, 23, 15, 10, 55, 12**

Step 2 - Delete root (**90**) from the Max Heap. To delete root node it needs to be swapped with last node (**12**). After delete tree needs to be heapify to make it Max Heap.




list of numbers after swapping 90 with 12.

**12, 82, 77, 23, 15, 10, 55, 90**

Step 3 - Delete root (**82**) from the Max Heap. To delete root node it needs to be swapped with last node (**55**). After delete tree needs to be heapify to make it Max Heap.


list of numbers after swapping 82 with 55.

**12, 55, 77, 23, 15, 10, 82, 90**

Step 4 - Delete root (**77**) from the Max Heap. To delete root node it needs to be swapped with last node (**10**). After delete tree needs to be heapify to make it Max Heap.


list of numbers after swapping 77 with 10.

**12, 55, 10, 23, 15, 77, 82, 90**

Step 5 - Delete root (**55**) from the Max Heap. To delete root node it needs to be swapped with last node (**15**). After delete tree needs to be heapify to make it Max Heap.


list of numbers after swapping 55 with 15.

**12, 15, 10, 23, 55, 77, 82, 90**

Step 6 - Delete root (**23**) from the Max Heap. To delete root node it needs to be swapped with last node (**12**). After delete tree needs to be heapify to make it Max Heap.


list of numbers after swapping 23 with 12.

**12, 15, 10, 23, 55, 77, 82, 90**

Step 7 - Delete root (**15**) from the Max Heap. To delete root node it needs to be swapped with last node (**10**). After delete tree needs to be heapify to make it Max Heap.


list of numbers after Deleting 15, 12 & 10 from the Max Heap.

**10, 12, 15, 23, 55, 77, 82, 90**

Whenever Max Heap becomes Empty, the list get sorted in Ascending order

## 3.Exceptions

Error in Python can be of two types :

1. Syntax errors  2. Exceptions.

Errors are the problems in a program due to which the program will stop the execution.

Exceptions are raised when some internal events occur which changes the normal flow of the program.

### **3.1** The difference between Syntax Error and Exceptions:

**Syntax Error:** As the name suggest this error is caused by wrong syntax in the code. It leads to the termination of the program.

**Exceptions:** Exceptions are raised when the program is syntactically correct but the code resulted in an error. This error does not stop the execution of the program, however, it changes the normal flow of the program.

Example code:

```
marks = 10000
# perform division with 0
a = marks / 0
print(a)
```

Output:

```
Traceback (most recent call last):
  File "/home/f3ad05420ab851d4bd106ffb04229907.py", line 4, in <module>
    a=marks/0
ZeroDivisionError: division by zero
```

In the above example, it raised the ZeroDivisionError as we are trying to divide a number by 0.

**Note:** Exception is the base class for all the exceptions in Python.

### **3.2 Try and Except in Exception Handling:**

#### Handling an exception

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.

```
# Python program to handle simple runtime error

a = [1, 2, 3]
try:
    print "Second element = %d" %(a[1])

    # Throws error since there are only 3 elements in array
    print "Fourth element = %d" %(a[3])

except  IndexError:
    print "An error occurred"
```

**Output:**

```
Second element = 2

An error occurred
```

```
# Program to handle multiple errors with one except statement
try :
        a = 3
        if a < 4 :

                # throws ZeroDivisionError for a = 3
                b = a/(a-3)

        # throws NameError if a >= 4
        Print( "Value of b = ", b)

except  (ZeroDivisionError, NameError):
        print ("\nError Occurred and Handled")
```

**Output:**

Error Occurred and Handled

### 3.3 else Clause

In python, you can also use else clause on the try-except block which must be present after all the except clauses. The code enters the else block only if the try clause does not raise an exception.

```
# Program to depict else clause with try-except

# Function which returns a/b
def   AbyB(a , b):
        try:
                c = ((a+b) / (a-b))
        except  ZeroDivisionError:
                 print "a/b result in 0"
        else:
                print c

# Driver program to test above function
AbyB(2.0, 3.0)
AbyB(3.0, 3.0)
```

The output for above program will be :

```
-5.0
a/b result in 0
```

## 3.4 finally Keyword in Python

Python provides a keyword finally, which is always executed after try and except blocks. The finally block always executes after normal termination of try block or after try block terminates due to some exception.

**Syntax:**

```
try:
        # Some Code....
except:
        # optional block
        # Handling of exception (if required)
else:
        # execute if no exception
finally:
       # Some code .....(always executed)
```

```
# Python program to demonstrate finally
try:
        k = 5//0 # raises divide by zero exception.
        print(k)

# handles zerodivision exception
except  ZeroDivisionError:
          print("Can't divide by zero")

finally:
        # this block is always executed regardless of exception generation.
        print('This is always executed')
```

**Output:**

```
Can't divide by zero
This is always executed
```

## 3.5 Raising Exception

We can raise exceptions in several ways by using the raise statement. The general syntax for the raise statement is as follows.

Syntax:

raise [Exception [, args]]

Here, Exception is the type of exception (for example, NameError) and argument is a value for the exception argument. The argument is optional; if not supplied, the exception argument is None.

The final argument, traceback, is also optional (and rarely used in practice), and if present, is the traceback object used for the exception.

Example:

An exception can be a string, a class or an object. Most of the exceptions that the Python core raises are classes, with an argument that is an instance of the class. Defining new exceptions is quite easy and can be done as follows −

```
def   functionName(level):
      if level < 1:
         raise "Invalid level!", level
         # The code below to this would not be executed
         # if we raise the exception
try:
      Business Logic here...
except  "Invalid level!":
      Exception handling here...
else:
      Rest of the code here...
```

## 3.6 List of Standard Exceptions −

| Sr.No. | Exception Name & Description |
|---|---|
| 1 | **Exception** : Base class for all exceptions |
| 2 | **StopIteration** : Raised when the next() method of an iterator does not point to any object. |

| | |
|---|---|
| 3 | **SystemExit :** Raised by the sys.exit() function. |
| 4 | **StandardError :** Base class for all built-in exceptions except StopIteration and SystemExit. |
| 5 | **ArithmeticError :** Base class for all errors that occur for numeric calculation. |
| 6 | **OverflowError :** Raised when a calculation exceeds maximum limit for a numeric type. |
| 7 | **FloatingPointError :** Raised when a floating point calculation fails. |
| 8 | **ZeroDivisionError :** Raised when division or modulo by zero takes place for all numeric types. |
| 9 | **ImportError :** Raised when an import statement fails. |
| 10 | **KeyboardInterrupt :** Raised when the user interrupts program execution, usually by pressing Ctrl+c. |
| 11 | **IndexError :** Raised when an index is not found in a sequence. |
| 12 | **KeyError :** Raised when the specified key is not found in the dictionary. |
| 13 | **NameError :** Raised when an identifier is not found in the local or global namespace. |
| 14 | **IndentationError :** Raised when indentation is not specified properly. |
| 15 | **TypeError :** Raised when an operation or function is attempted that is invalid for the specified data type. |
| 16 | **ValueError :** Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified. |
| 17 | **RuntimeError :** Raised when a generated error does not fall into any category. |

## 3.7 User-Defined Exceptions

Python also allows you to create your own exceptions by deriving classes from the 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 we defined above class, we can raise the exception as follows −

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

## 5.Data Base connection with python

Database is a collection of data in the form of tables.

Table is a collection of rows and columns.

- Row is a record which identifies unique object .
- Column is a field and also known as attribute (information about particular domain.
- Column needs specified datatype.

**DBMS:**

It is a software for storing and retrieving data by considering appropriate security measures. It allows the users to create their own data bases as per their requirement.

It consists of a group of programs which manipulate the data base and provide an interface between the database and user.

**DBMS operations:**

- Data Definition Language:
  Ex: create, drop
- Data Manipulation Language:
  Ex: insert, delete, update, select
- Transaction Control Language:
  Ex: commit, rollback

**Syntax for creating database:**

create database databaseName

**Syntax for creating table:**

create table tableName(col1 datatype,col2 datatype,……,coln datatype)

Ex: create table student(RollNo integer, Name varchar[20],Marks integer)

**Syntax for drop:**

drop table tableName

**Syntax for insert:**

insert into tableName(col1,col2,……,coln)

values (value1,value2,……..)

Example:

insert into student(Rollno, name, marks)

value(101,'john',99)

**Syntax for delete:**

delete from tableName → it will delete all records(rows) in the table

Ex:  delete from student where RollNo=101

**Syntax for update:**

update tableName  set  colName=value

Ex:  update student set marks=100

update student set marks=100 where rollNo=102

**Syntax for select:**

select command is used to get data from the table(read operation)

syntax:

select col1,col2,……,coln from tableName

Ex: 1)select  RollNo, Name  from student

2)select  *  from  student (* is used to select all columns)

**commit:**

Commit is used to store data permanently in the database. It is necessary to give command after manipulating the database.

**Rollback:**

It revert back the transaction (undo operation(ctrl+z))

**DBMS software:**

- oracle
- MySQL server
- Mongo DB
- MySql
- sqlite

In python, database connectors are available in the form of packages. A package is a collection of modules, A module is a collection of functions.

To connect python with database MySQL:

- we have to import mysql.connector
  mydb=mysql.connector(host='localHost',
  user='root',password='123$',database='sample')

  Above statement is used to connect to the database.

- **Cursor** : It is a temporary buffered area to do transactions
  cur=mydb.cursor()

- To create table into database
  cur.execute (create table student(RollNo integer,Name varchar(21),Marks integer)

- To insert one record
  s="insert into student(RollNo,Name) values(%s%s)"

  v=(101,'John')

  cur.execute(s,v)

  mydb.commit()

- To insert two or more records
  s = "insert into student(RollNo,Name) values(%s%s)"

  v=[(101,'John').(102,'Jack'),……..]

  cur.executemany(s,v)

  mydb.commit()

- To update the existing record
  S = "update student set Name='rama' when RollNo=101"
  cur.execute(S)

- To delete the existing record
  s="delete from student where RollNo=102"
  cur.execute(s)

- To select or view the records in table
  cur.execute("select * from student")

```
result = cur.fetchall()   #to get all records  from student table as
                                                          result list
#Iterating over result list
for  i  in  result:
    print(i)
```

- Write a Python program to perform various database operations
  ( create, insert, delete, update )

To connect to the database, we can use  mysql.connector module

**Program:**

```
#importing  mysql module
import   mysql.connector
#connecting to mysql database
mydb=mysql.connector.connect(hostname='localhost, user='root', passwd='abc$',
database='sample')
#creating cursor to work with database
cur=mycursor()
#drop table if already exists
cur.execute('drop table if exists customer')
#creating table in database
cur.execute('create table customer(name varchar(20), address varchar(50))')
#inserting rows into cutomer table
ins="insert into customer (name,address) values(%s,%s)"
val=[('john', 'Delhi'),('jack','Mumbai'),('Lal','Hyd'),('Ram','Vja')]
cur.executemany(ins,val)
mydb.commit()
print(cur.rowcount, ' records inserted')
#reading data from customer table
cur.execute("select * from customer")
```

```
res=cur.fetchall()
  for  i  in res:
            Print(i)
#deleting row from tables
s="delete from customer where address='Hyd'"
cur.execute(s)
print(cur.rowcount, "records deleted")
mydb.commit()
#updating rows  in table
s="update customer set name="Raj"  where address="Hyd"
cur.execute(s)
mydb.commit()
print(cur.rowcount, "records updated")
```

**Output:**

4 records inserted

('john', 'Delhi')

('jack','Mumbai')

('Lal','Hyd')

('Ram','Vja')

1 records deleted

1 records updated