Unit 1: **[18 MARKS]**

Beginning with Python, Data types, Operators, I/O and Control statements
Python, Features of Python, History of Python, Python Virtual Machine, Memory Management in Python, Garbage collection in python, Comparisons between C-Java-Python, writing first Python program, Python variable **Data types in Python:** Built-in data types, Numeric types, Explicit conversion of data types, Sequences Types: List, Tuple, range, set data type, frozen set, mapping types, Boolean types, Binary Types, Determining the data type of a variable **Operators:** Arithmetic operations, logical operators

Beginning with Python

Python is a high-level, interpreted, interactive and object-oriented scripting language. Python is designed to be highly readable. It uses English keywords frequently whereas the other languages use punctuations. It has fewer syntactical constructions than other languages.

- **Python is Interpreted:** Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.
- **Python is Interactive:** You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
- **Python is Object-Oriented:** Python supports Object-Oriented style or technique of programming that encapsulates code within objects.
- **Python is a Beginner's Language:** Python is a great language for the beginner level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

Why to use Python:

The following are the primary factors to use python in day-to-day life:

1. **Python is object-oriented:**
   Structure supports such concepts as polymorphism, operation overloading and multiple inheritances.
2. **Indentation:**
   Indentation is one of the greatest features in python
3. **It's free (open source):**
   Downloading python and installing python is free and easy
4. **It's Powerful:**
   - Dynamic typing
   - Built-in types and tools
   - Library utilities
   - Third party utilities (e.g. Numeric, NumPy, sciPy)

- Automatic memory management

5. **It's Portable:**
   - Python runs virtually every major platform used today
   - As long as you have a compatible python interpreter installed, python programs will run in exactly the same manner, irrespective of platform.
6. **It's easy to use and learn:**
   - No intermediate compile
   - Python Programs are compiled automatically to an intermediate form called byte code, which the interpreter then reads.
   - This gives python the development speed of an interpreter without the performance loss inherent in purely interpreted languages. Structure and syntax are pretty intuitive and easy to grasp.
7. **Interpreted Language:**
   Python is processed at runtime by python Interpreter
8. **Interactive Programming Language:**
   Users can interact with the python interpreter directly for writing the programs
9. **Straight forward syntax:**
   The formation of python syntax is simple and straight forward which also makes it popular

Features of Python

- **Easy-to-learn:** Python has few keywords, simple structure, and a clearly defined. Syntax. This allows a student to pick up the language quickly.
- **Easy-to-read**: Python code is more clearly defined and visible to the eyes.
- **Easy-to-maintain**: Python's source code is fairly easy-to-maintain.
- **A broad standard library**: Python's bulk of the library is very portable and cross platform compatible on UNIX, Windows, and Macintosh.
- **Interactive Mode**: Python has support for an interactive mode, which allows interactive testing and debugging of snippets of code.
- **Portable:** Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
- **Extendable:** You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.
- **Databases**: Python provides interfaces to all major commercial databases.
- **GUI Programming**: Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.
- **Scalable**: Python provides a better structure and support for large programs than shell scripting.

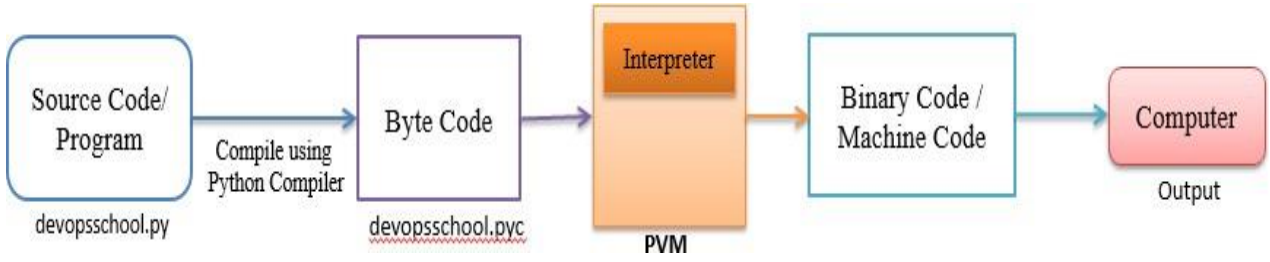
History of Python

Python was developed by Guido van Rossum in the late eighties and early nineties at the National Research Institute for Mathematics and Computer Science in the Netherlands.

- Python is derived from many other languages, including ABC, Modula-3, C, C++, Algol-68, SmallTalk, and UNIX shell and other scripting languages.
- Python is copyrighted. Like Perl, Python source code is now available under the GNU General Public License (GPL).
- Python is now maintained by a core development team at the institute, although Guido van Rossum still holds a vital role in directing its progress.
- Python 1.0 was released in November 1994. In 2000, Python 2.0 was released. Python 2.7.11 is the latest edition of Python 2.
- Meanwhile, Python 3.0 was released in 2008. Python 3 is not backward compatible with Python 2. The emphasis in Python 3 had been on the removal of duplicate programming constructs and modules so that "There should be one -- and preferably only one -- obvious way to do it." Python 3.5.1 is the latest version of Python 3.

Python Virtual Machine

Python Virtual Machine (PVM) is a program which provides programming environment. The role of PVM is to convert the byte code instructions into machine code so the computer can execute those machine code instructions and display the output.



So PVM is nothing but a software/interpreter that converts the byte code to machine code for given operating system. PVM is also called Python Interpreter and this is the reason Python is called an Interpreted language. We can't see the Byte Code of the program because this happens internally in memory.

Memory Management

Memory management is very important for software developers to work efficiently with any programming language. As we know, Python is a famous and widely used programming language. It is used almost in every technical domain. In contrast to a programming language, memory management is related to writing memory-efficient code. We cannot overlook the importance of memory management while implementing a large amount of data. Improper memory management leads to slowness on the application and the server-side components. It also becomes the reason of improper working. If the memory is not handled well, it will take much time while preprocessing the data.

In Python, memory is managed by the Python manager which determines where to put the application data in the memory. So, we must have the knowledge of Python memory manager to write efficient code and maintainable code.

Let's assume memory looks like an empty book and we want to write anything on the book's page. Then, we write data any data the manager find the free space in the book and provide it to the application. The procedure of providing memory to objects is called **allocation.**

Python Memory Allocation

Memory allocation is an essential part of the memory management for a developer. This process basically allots free space in the computer's virtual memory, and there are two types of virtual memory works while executing programs.

- Static Memory Allocation
- Dynamic Memory Allocation

Static Memory Allocation

Static memory allocation happens at the compile time. For example - In C/C++, we declare a static array with the fixed sizes. Memory is allocated at the time of compilation. However, we cannot use the memory again in the further program.

```
static int a=10
```

- **Stack Allocation**

  The Stack data structure is used to store the static memory. It is only needed inside the particular function or method call. The function is added in program's call stack whenever we call it. Variable assignment inside the function is temporarily stored in the function call stack; the function returns the value, and the call stack moves to the text task. The compiler handles all these processes, so we don't need to worry about it.

Call stack (stack data structure) holds the program's operational data such as subroutines or function call in the order they are to be called. These functions are popped up from the stack when we called.

Dynamic Memory Allocation

Unlike static memory allocation, Dynamic memory allocates the memory at the runtime to the program. For example - In C/C++, there is a predefined size of the integer of float data type but there is no predefine size of the data types. Memory is allocated to the objects at the run time. We use the **Heap** for implement dynamic memory management. We can use the memory throughout the program.

```
int *a
p = new int
```

As we know, everything in Python is an object means dynamic memory allocation inspires the Python memory management. Python memory manager automatically vanishes when the object is no longer in use.

Garbage Collection

Python removes those objects that are no longer in use or can say that it frees up the memory space. This process of vanish the unnecessary object's memory space is called the Garbage Collector. The Python garbage collector initiates its execution with the program and is activated if the reference count falls to zero.

When we assign the new name or placed it in containers such as a dictionary or tuple, the reference count increases its value. If we reassign the reference to an object, the reference counts decreases its value if. It also decreases its value when the object's reference goes out of scope or an object is deleted.

As we know, Python uses the dynamic memory allocation which is managed by the Heap data structure. Memory Heap holds the objects and other data structures that will be used in the program. Python memory manager manages the allocation or de-allocation of the heap memory space through the API functions.

Comparisons between C++-Java-Python

C++	JAVA	PYTHON
Compiled Programming language	Compiled Programming Language	Interpreted Programming Language
Supports Operator overloading	Does not support Operator Overloading	Supports Operator overloading
Provide both single and multiple inheritance	Provide partial multiple inheritance using interfaces	Provide both single and multiple inheritance
Platform dependent	Platform Independent	Platform Independent
Does Not support threads	Has in build multithreading support	Supports multithreading
Has limited number of library support	Has library support for many concepts like UI	Has a huge set of libraries that make it fit for AI, data science,
Code length is a bit lesser, 1.5 times less that java.	Java has quite huge code.	Smaller code length, 3-4 times less than java.
Functions and variables are used outside the class	Every bit of code is inside a class.	Functions and variables can be declared and used outside the class also.
C++ program is a fast compiling programming	Java Program compiler a bit slower than C++	Due to the use of interpreter execution is

C++	JAVA	PYTHON
language.		slower.
Strictly uses syntax norms	Strictly uses syntax norms	Use of ; is not compulsory.
like ; and {}.	like punctuations , ; .	

How to create and run program,

Follow the following steps to run Python on your computer.

1. Go to start menu -> click python 2.7 -> click IDLE (python GUI)
2. Go to: File -> New.
3. save the file with p1.py
4. Write Python code in the file and save it.
5. Go to Run -> Run current script or simply click F5 to run it.

Note: all programs are automatically save in C:\Python2.7

Writing first Python program:

Python Program to Print Hello World

```
print("Hello World")
```

output

Hello World

Comments in Python

Single Line Comments

A hash sign (#) that is not inside a string literal begins a comment. All characters after the # and up to the end of the physical line are part of the comment and the Python interpreter ignores them.

Comments starts with a #, and Python will ignore them:

Example

```
#This is a comment
print("Hello, World!")
```

Comments can be placed at the end of a line, and Python will ignore the rest of the line:

Example

```
print("Hello, World!") #This is a comment
```

Multi Line Comments

Python does not really have a syntax for multi line comments. To add a multiline comment you could insert a # for each line:

Example

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

Or, not quite as intended, you can use a multiline string.

Since Python will ignore string literals that are not assigned to a variable, you can add a multiline string (triple quotes) in your code, and place your comment inside it:

Example

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

Reserved Words

The following list shows the Python keywords. These are reserved words and you cannot use them as constant or variable or any other identifier names. All the Python keywords contain lowercase letters only.

and	Exec	not
assert	Finally	or
break	For	pass
class	From	print
continue	Global	raise
def	If	return
del	Import	try
elif	In	while
else	Is	with
except	Lambda	yield

Python variable:

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals or characters in these variables.

Assigning Values to Variables

Python variables do not need explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables.

The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable. For example –

```
counter = 100          # An integer assignment
miles   = 1000.0       # A floating point
name    = "John"       # A string
```

Data types in Python:

Built-in data types

Built-in Data types are those data types that are pre-defined by the programming language. Most languages have native data types that are created for ease of execution of data. Such Data types are called Built-in Data Types. These data types can be used directly in the program without the hassle of creating them.

Python has the following data types built-in by default, in these categories:

Text Type:	Str
Numeric Types:	int, float, complex
Sequence Types:	list, tuple, range
Mapping Type:	Dict
Set Types:	set, frozenset
Boolean Type:	Bool
Binary Types:	bytes, bytearray, memoryview

Strings

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

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

```
str = 'Hello World!'
```

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

This will produce the following result –

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

String Methods

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

Method	Description
capitalize()	Converts the first character to upper case
casefold()	Converts string into lower case
center()	Returns a centered string
count()	Returns the number of times a specified value occurs in a string
encode()	Returns an encoded version of the string
endswith()	Returns true if the string ends with the specified value

expandtabs()	Sets the tab size of the string
find()	Searches the string for a specified value and returns the position of where it was found
format()	Formats specified values in a string
format_map()	Formats specified values in a string
index()	Searches the string for a specified value and returns the position of where it was found
isalnum()	Returns True if all characters in the string are alphanumeric
isalpha()	Returns True if all characters in the string are in the alphabet
isascii()	Returns True if all characters in the string are ascii characters
isdecimal()	Returns True if all characters in the string are decimals
isdigit()	Returns True if all characters in the string are digits
isidentifier()	Returns True if the string is an identifier

islower()	Returns True if all characters in the string are lower case
isnumeric()	Returns True if all characters in the string are numeric
isprintable()	Returns True if all characters in the string are printable
isspace()	Returns True if all characters in the string are whitespaces
istitle()	Returns True if the string follows the rules of a title
isupper()	Returns True if all characters in the string are upper case
join()	Converts the elements of an iterable into a string
ljust()	Returns a left justified version of the string
lower()	Converts a string into lower case
lstrip()	Returns a left trim version of the string
maketrans()	Returns a translation table to be used in translations

partition()	Returns a tuple where the string is parted into three parts
replace()	Returns a string where a specified value is replaced with a specified value
rfind()	Searches the string for a specified value and returns the last position of where it was found
rindex()	Searches the string for a specified value and returns the last position of where it was found
rjust()	Returns a right justified version of the string
rpartition()	Returns a tuple where the string is parted into three parts
rsplit()	Splits the string at the specified separator, and returns a list
rstrip()	Returns a right trim version of the string
split()	Splits the string at the specified separator, and returns a list
splitlines()	Splits the string at line breaks and returns a list

startswith()	Returns true if the string starts with the specified value
strip()	Returns a trimmed version of the string
swapcase()	Swaps cases, lower case becomes upper case and vice versa
title()	Converts the first character of each word to upper case
translate()	Returns a translated string
upper()	Converts a string into upper case
zfill()	Fills the string with a specified number of 0 values at the beginning

Numeric types

Numbers

Python includes three numeric types to represent numbers: **integers, float**, and **complex number**.

- Number data types store numeric values. Number objects are created when you assign a value to them. For example –
- var1 = 1
- var2 = 10

Python supports four different numerical types –

These are number types. They are created when a number is assigned to a variable.

- int holds signed integers of non-limited length.
- float holds floating precision numbers and they are accurate upto 15 decimal places.
- complex – A complex number contains the real and imaginary part.

Examples

Here are some examples of numbers –

int	long	float	complex
10	51924361L	0.0	3.14j
100	-0x19323L	15.20	45.j
-786	0122L	-21.9	9.322e-36j
080	0xDEFABCECBDAECBFBAEl	32.3+e18	.876j
-0490	535633629843L	-90.	-.6545+0J
-0x260	-052318172735L	-32.54e100	3e+26J
0x69	-4721885298529L	70.2-E12	4.53e-7j

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

Data Type Conversion

A cast, or explicit type conversion, is special programming instuction which specifies what data type to treat a variable as (or an intermediate calculation result)

in a given expression. Casting will ignore extra information (but never adds information to the type being casted)

Sometimes, you may need to perform conversions between the built-in types. To convert between types, you simply use the type name as a function.

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

Sr.No.	Function & Description
1	**int(x [,base])** Converts x to an integer. base specifies the base if x is a string.
2	**long(x [,base])** Converts x to a long integer. base specifies the base if x is a string.
3	**float(x)** Converts x to a floating-point number.
4	**complex(real [,imag])** Creates a complex number.
5	**str(x)** Converts object x to a string representation.
6	**repr(x)** Converts object x to an expression string.
7	**eval(str)** Evaluates a string and returns an object.
8	**tuple(s)** Converts s to a tuple.
9	**list(s)**

	Converts s to a list.
10	**set(s)** Converts s to a set.
11	**dict(d)** Creates a dictionary. d must be a sequence of (key,value) tuples.
12	**frozenset(s)** Converts s to a frozen set.
13	**chr(x)** Converts an integer to a character.
14	**unichr(x)** Converts an integer to a Unicode character.
15	**ord(x)** Converts a single character to its integer value.
16	**hex(x)** Converts an integer to a hexadecimal string.
17	**oct(x)** Converts an integer to an octal string.

You can convert from one type to another with the int(), float(), and complex() methods:

Example

Convert from one type to another:

```
x = 1    # int
y = 2.8  # float
z = 1j   # complex
```

```
#convert from int to float:
a = float(x)

#convert from float to int:
b = int(y)

#convert from int to complex:
c = complex(x)

print(a)
print(b)
print(c)

print(type(a))
print(type(b))
print(type(c))
```

OUTPUT

```
1.0
2
(1+0j)
<class 'float'>
<class 'int'>
<class 'complex'>
```

Sequences Types

Lists

Lists are the most versatile of Python's compound data types. A list is a data structure in Python that is a mutable, or changeable, ordered sequence of elements. Each element or value that is inside of a list is called an item. Just as strings are defined as characters between quotes, lists are defined by having values between square brackets [ ].

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

The values stored in a list can be accessed using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1. The plus (+) sign is the list concatenation operator, and the asterisk (*) is the repetition operator. For example –

```
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
```

```
tinylist = [123, 'john']

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

This produce the following result –

```
['abcd', 786, 2.23, 'john', 70.2]
OUTPUT
abcd
[786, 2.23]
[2.23, 'john', 70.2]
[123, 'john', 123, 'john']
['abcd', 786, 2.23, 'john', 70.2, 123, 'john']
```

List/Array Methods

Python has a set of built-in methods that you can use on lists/arrays.

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list

index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the first item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

Tuples

A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses.

A Tuple is a collection of Python objects separated by commas. In someways a tuple is similar to a list in terms of indexing, nested objects and repetition but a tuple is immutable unlike lists which are mutable.

The main differences between lists and tuples are: Lists are enclosed in brackets ([ ]) and their elements and size can be changed, while tuples are enclosed in parentheses (( )) and cannot be updated. Tuples can be thought of as **read-only** lists. For example –

```
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )
tinytuple = (123, 'john')

print tuple              # Prints the complete tuple
print tuple[0]           # Prints first element of the tuple
print tuple[1:3]         # Prints elements of the tuple starting from 2nd till 3rd
print tuple[2:]          # Prints elements of the tuple starting from 3rd element
```

```
print tinytuple * 2       # Prints the contents of the tuple twice
print tuple + tinytuple   # Prints concatenated tuples
```

This produce the following result –

```
('abcd', 786, 2.23, 'john', 70.2)
abcd
(786, 2.23)
(2.23, 'john', 70.2)
(123, 'john', 123, 'john')
('abcd', 786, 2.23, 'john', 70.2, 123, 'john')
```

The following code is invalid with tuple, because we attempted to update a tuple, which is not allowed. Similar case is possible with lists –

```
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
list = [ 'abcd', 786 , 2.23,  'john',  70.2  ]
tuple[2] = 1000    # Invalid syntax with tuple
list[2] = 1000     # Valid syntax with list
```

Tuple Methods

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

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

Range

The range() is an in-built function in Python. It returns a sequence of numbers starting from zero and increment by 1 by default and stops before the given number.

Using a for loop with range() , we can repeat an action a specific number of times. For example, let's see how to use the range() function of Python 3 to produce the first six numbers.

Syntax

range(start, stop, step)

Parameter Values

Parameter	Description
start	Optional. An integer number specifying at which position to start. Default is 0
stop	Required. An integer number specifying at which position to stop (not included).
step	Optional. An integer number specifying the incrementation. Default is 1

Example 1

```
x = range(6)
for n in x:
  print(n)
```

OUTPUT

```
0
1
2
3
4
5
```

Example 2

```
x = range(3, 20, 2)

for n in x:

  print(n)
```

OUTPUT
3
5
7
9
11
13
15
17
19

Set

Sets are used to store multiple items in a single variable.

Set is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Tuple, and Dictionary, all with different qualities and usage.

A set is a collection which is *unordered*, *unchangeable**, and *unindexed*.

Sets are written with curly brackets.

Example

Create a Set:

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

OUTPUT

{'banana', 'apple', 'cherry'}

Set Items
Set items are unordered, unchangeable, and do not allow duplicate values.

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

Unchangeable
Set items are unchangeable, meaning that we cannot change the items after the set has been created. Once a set is created, you cannot change its items, but you can remove items and add new items.

Duplicates Not Allowed

Sets cannot have two items with the same value.

Example
Duplicate values will be ignored:

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

print(thisset)
```

OUTPUT
{'banana', 'cherry', 'apple'}

Set Methods

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

Method	Description
add()	Adds an element to the set
clear()	Removes all the elements from the set
copy()	Returns a copy of the set
difference()	Returns a set containing the difference between two or more sets
difference_update()	Removes the items in this set that are also included in another, specified set

discard()	Remove the specified item
intersection()	Returns a set, that is the intersection of two or more sets
intersection_update()	Removes the items in this set that are not present in other, specified set(s)
isdisjoint()	Returns whether two sets have a intersection or not
issubset()	Returns whether another set contains this set or not
issuperset()	Returns whether this set contains another set or not
pop()	Removes an element from the set
remove()	Removes the specified element
symmetric_difference()	Returns a set with the symmetric differences of two sets
symmetric_difference_update()	inserts the symmetric differences from this set and another

union()	Return a set containing the union of sets
update()	Update the set with another set, or any other iterable

Frozenset

The frozenset() is an inbuilt function in Python which takes an iterable object as input and makes them immutable. Simply it freezes the iterable objects and makes them unchangeable. ... This function takes input as any iterable object and converts them into an immutable object.
The frozenset() function returns an unchangeable frozenset object (which is like a set object, only unchangeable). The **frozen sets** are the immutable form of the normal sets. It means we cannot remove or add any item into the frozen set.

Syntax
frozenset(iterable)

Parameter Values

Parameter	Description
iterable	An iterable object, like list, set, tuple etc.

Example 1

```
mylist = ['apple', 'banana', 'cherry']
x = frozenset(mylist)
  print(x)
```

OUTPUT

```
frozenset({'cherry', 'banana', 'apple'})
```

Example 2
Try to change the value of a frozenset item. This will cause an error:

```
mylist = ['apple', 'banana', 'cherry']
x = frozenset(mylist)
x[1] = "strawberry"
```

OUTPUT

```
Traceback (most recent call last):
  File "demo_ref_frozenset2.py", line 3, in <module>
    x[1] = "strawberry"
TypeError: 'frozenset' object does not support item assignment
```

Mapping types

There is one type of mapping data type in python that is dictionary.

Python's map() is a built-in function that allows you to process and transform all the items in an iterable without using an explicit for loop, a technique commonly known as mapping. map() is useful when you need to apply a transformation function to each item in an iterable and transform them into a new iterable.

Dictionary

The mapping objects are used to map hash table values to arbitrary objects. In python there is mapping type called dictionary. It is mutable.

The keys of the dictionary are arbitrary. As the value, we can use different kind of elements like lists, integers or any other mutable type objects.

Python's dictionaries are kind of hash table type. They work like associative arrays or hashes found in Perl and consist of key-value pairs. A dictionary key can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object.

Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]). For example –

```
dict = {}
dict['one'] = "This is one"
dict[2]     = "This is two"

tinydict = {'name': 'john','code':6734, 'dept': 'sales'}


print dict['one']       # Prints value for 'one' key
print dict[2]           # Prints value for 2 key
print tinydict          # Prints complete dictionary
print tinydict.keys()   # Prints all the keys
print tinydict.values() # Prints all the values
```

This produce the following result –

```
This is one
This is two
{'dept': 'sales', 'code': 6734, 'name': 'john'}
['dept', 'code', 'name']
```

['sales', 6734, 'john']

Dictionaries have no concept of order among elements. It is incorrect to say that the elements are "out of order"; they are simply unordered.

Dictionary Methods

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

Method	Description
clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and value
get()	Returns the value of the specified key
items()	Returns a list containing a tuple for each key value pair
keys()	Returns a list containing the dictionary's keys
pop()	Removes the element with the specified key
popitem()	Removes the last inserted key-value pair

setdefault()	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
update()	Updates the dictionary with the specified key-value pairs
values()	Returns a list of all the values in the dictionary

Boolean types

Python boolean type is one of the built-in data types provided by Python, which represents one of the two values i.e. True or False. Generally, it is used to represent the truth values of the expressions. For example, 1== 0 is True whereas 2<1 is False. bool values are the two constant objects False and True. They are used to represent truth values. In numeric contexts, they behave like the integers 0 and 1, respectively.

Example

```
x = True
y = False

print(x)    #True
print(y)    #False

print(bool(1)) #True
print(bool(0))  #False
```

Binary Type

A data type defines the type of a variable. Since everything is an object in Python, data types are actually classes; and the variables are instances of the classes.

bytes, bytearray, memoryview

bytes and **bytearray** are used for manipulating binary data. The **memoryview** uses the buffer protocol to access the memory of other binary objects without needing to make a copy.

Bytes objects are **immutable** sequences of single bytes. We should use them only when working with ASCII compatible data.

The syntax for bytes literals is same as string literals, except that a 'b' prefix is added.

bytearray objects are always created by calling the constructor bytearray(). These are **mutable** objects.

Example of bytes, memoryview types

```
x = b'char_data'
x = b"char_data"

y = bytearray(5)

z = memoryview(bytes(5))

print(x)    #  b'char_data'
print(y)    # bytearray(b'\x00\x00\x00\x00\x00')
print(z)    # <memory at 0x014CE328>
```

Operators:

1. Arithmetic operations
2. Logical operators.

Arithmetic Operators

Arithmetic operators are used with numeric values to perform common mathematical operations:

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y

/	Division	x / y
%	Modulus	x % y
**	Exponentiation	x ** y
//	Floor division	x // y

Logical Operators

Logical operators are used to combine conditional statements:

Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

Unit 2: **[17 MARKS]**

Control statements, Looping statements, Arrays, Functions, Modules:
Control statements: Conditional Statement. If Statement, If Else statement, Comprehension statement (multiple conditions).
Looping: for loop, while loop
Arrays: Advantages of Array, Creating an Array, looping array, importing the array module, Slicing and Processing the arrays
Functions: Defining-calling and returning(single and multiple) results from a function, Pass by Object Reference, Positional arguments, Keyword arguments, Default arguments, Variable length arguments, Inbuilt Functions and Methods List, Python Lambda
Modules: Creating our own modules in python.

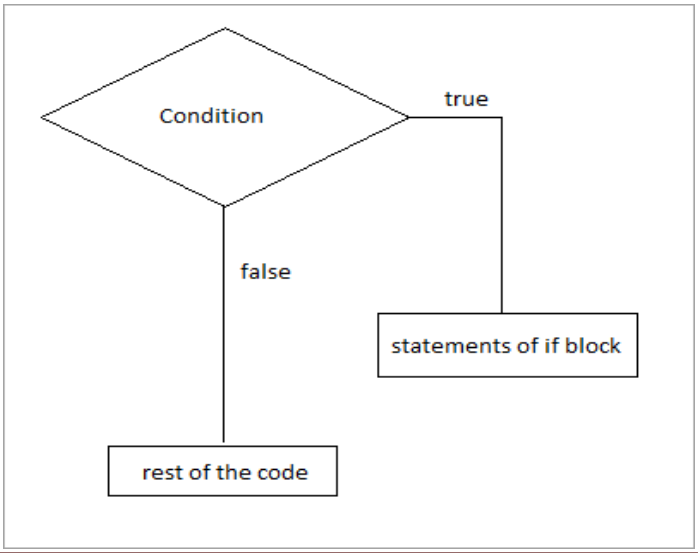
Control Statements:

Conditional Statement:
Conditional Statement in Python performs different computations or actions depending on whether a specific Boolean constraint evaluates to true or false. Conditional statements are handled by IF statements in Python.

Type of condition statement in Python:

- If statement.
- If Else statement.
- Elif statement.
- Nested if statement.
- Nested if else statement.

If Statement: is used for decision-making operations. It contains a body of code which runs only when the condition given in the if statement is true. If the condition is false, then the optional else statement runs which contains some code for the else condition.



Syntax:

```
if (expression== True)
      Statement(body of statements)
```

Python supports the usual logical conditions from mathematics:

- Equals: a == b
- Not Equals: a != b

- Less than: a < b
- Less than or equal to: a <= b
- Greater than: a > b
- Greater than or equal to: a >= b

These conditions can be used in several ways, most commonly in "if statements" and loops.

An "if statement" is written by using the if keyword.

Example:

```
a = 30
b = 20
if a > b:
    print("a is greater than b")
```

OUTPUT
a is greater than b

In this example we use two variables, a and b, which are used as part of the if statement to test whether a is greater than b. As a is 30, and b is 20, we know that 30 is greater than 20, and so we print to screen that "a is greater than b".

Comprehension Statement (Multiple conditions).

Else Statement:

The statement itself says if a given condition is true then execute the statements present inside the “if block” and if the condition is false then execute the “else” block.

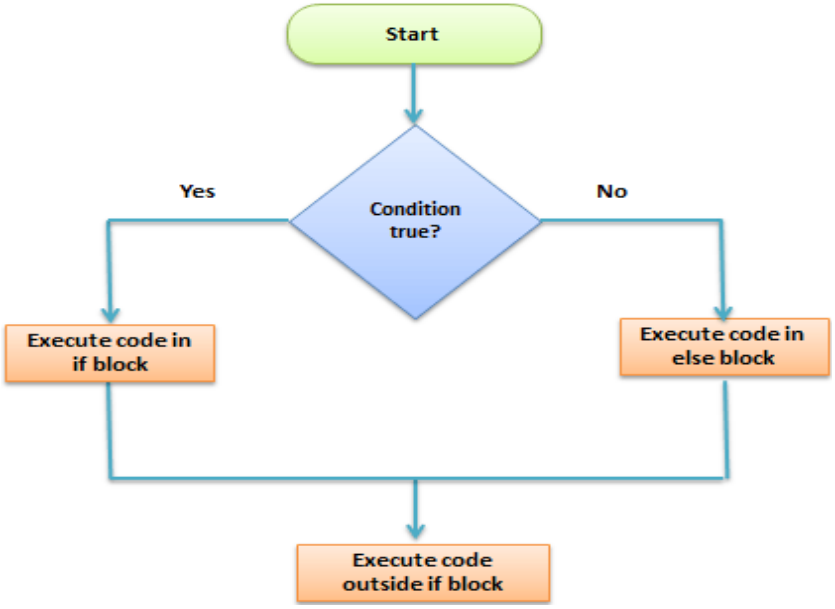
The “else” block will execute only when the condition becomes false. It is the block where you will perform some actions when the condition is not true.

if-else statement evaluates the Boolean expression. If the condition is TRUE then, the code present in the “ if “ block will be executed otherwise the code of the “else“ block will be executed
Syntax:

```
If (EXPRESSION == TRUE):
        Statement (Body of the block)
else:
        Statement (Body of the block)
```

Here, the condition will be evaluated to a Boolean expression (true or false). If the condition is true then the statements or program present inside the “if” block will be

executed and if the condition is false then the statements or program present inside the “else” block will be executed.



Let‟s see an example of Python if else Statement:

Example:

```
a = 10
b = 20
if a > b:
    print("a is greater than b")
else:
    print("b is greater than a")
```

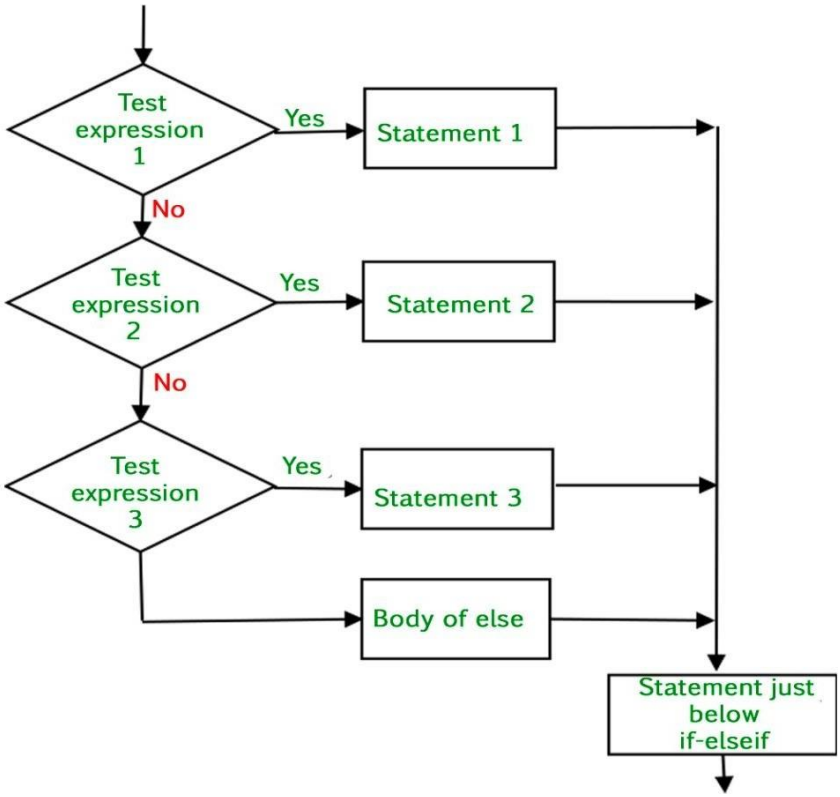
OUTPUT

```
b is greater than a
```

In this example b is greater than a, so the first condition is not true, also, so we go to the else condition and print to screen that "b is greater than a".

Elif Statement:

The elif keyword is pythons way of saying "if the previous conditions were not true, then try this condition".



in Python, we have one more conditional statement called “elif” statements. “elif” statement is used to check multiple conditions only if the given condition is false. It‟s similar to an “if-else” statement and the only difference is that in “else” we will not check the condition but in “elif” we will check the condition.

“elif” statements are similar to “if-else” statements but “elif” statements evaluate multiple conditions.

Syntax:

```
if (condition):
        Set of statement to execute if condition is true
elif (condition):
        Set of statements to be executed when if condition is false and elif condition is true
else:
        Set of statement to be executed when both if and elif conditions are false
```

Example:

```
a = 33
b = 33
if a > b:
        print("b is greater than a")
elif a == b:
```

```
        print("a and b are equal")
else:
        print("b is greater than a")
```

OUTPUT
a and b are equal

Short Hand If Statement:

If you have only one statement to execute, you can put it on the same line as the if statement.

Example:

One hand or single line statement

```
a=30

b=20

if a > b: print("a is greater than b")
```

OUTPUT
a is greater than b

Short Hand If ... Else Statement:

If you have only one statement to execute, one for if, and one for else, you can put it all on the same line:

Example:

One hand or single line if- else statement

```
a = 20
b = 30
print("A is greater ") if a > b else print("B is greater ")
```

OUTPUT
B is greater

And

The and keyword is a logical operator, and is used to combine conditional statements:

Example:

Test if a is greater than b, AND if c is greater than a

```
a = 60
b = 30
c = 70
if a > b and c > a:
  print("Both conditions are True")
```

OUTPUT
Both conditions are True

Or

The or keyword is a logical operator, and is used to combine conditional statements:

Example:

Test if a is greater than b, OR if a is greater than c

```
a = 60
b = 10
c = 70
if a > b or a > c:
  print("At least one of the conditions is True")
```

OUTPUT
At least one of the conditions is True

Nested If Statement:

You can have if statements inside if statements, this is called *nested* if statements.

A nested if statement is an if statement placed inside another if statement. Nested if statements are often used when you must test a combination of conditions before deciding on the proper action.

```
x = 40

if x > 10:
        print("Above ten,")
if x > 20:
        print("and also above 20!")
else:
        print("but not above 30.")
```

OUTPUT
Above ten,
and also above 20!

Nested If Else Statement:

A nested if statement is an if statement placed inside another if statement. Nested if statements are often used when you must test a combination of conditions before deciding on the proper action.

Syntax

The syntax of the nested if...elif...else construct may be –

```
if expression1:
       statement(s)
       if expression2:
            statement(s)
       elif expression3:
            statement(s)
       elif expression4:
            statement(s)
       else:
            statement(s)
else:
       statement(s)
```

Example

```
var = 100
if var < 200:
    print ("Expression value is less than 200")
    if var == 150:
        print ("Which is 150")
    elif var == 100:
        print ("Which is 100")
    elif var == 50:
        print ("Which is 50")
    elif var < 50:
        print ("Expression value is less than 50")
else:
    print ("Could not find true expression")

print ("Good bye!")
```

OUTPUT

```
Expression value is less than 200
Which is 100
Good bye!
```

Looping

Statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. There may be a situation when you need to execute a block of code several number of times.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times. The following diagram illustrates a loop statement –

Python has two primitive loop commands:

- for loops
- while loops

For loop

Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.

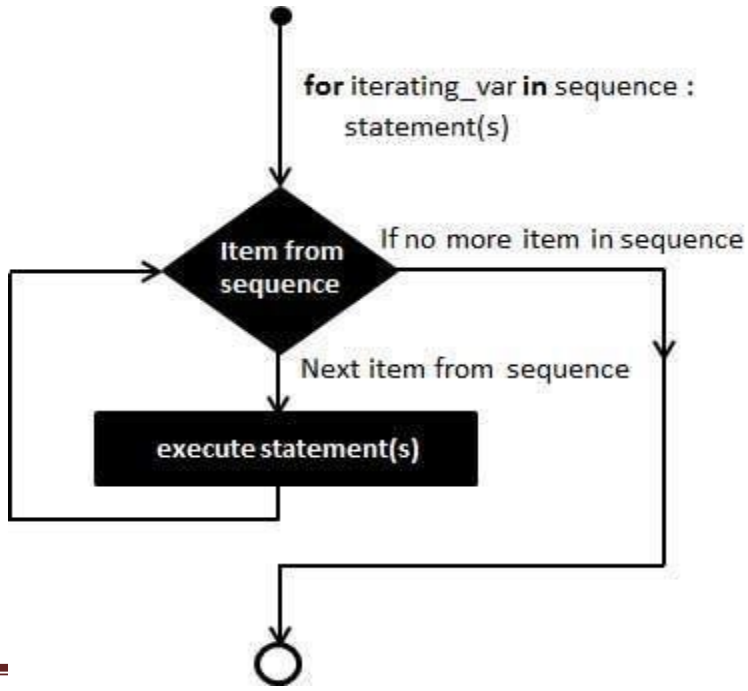
It has the ability to iterate over the items of any sequence, such as a list or a string.

Syntax:

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

If a sequence contains an expression list, it is evaluated first. Then, the first item in the sequence is assigned to the iterating variable *iterating_var*. Next, the statements block is executed. Each item in the list is assigned to *iterating_var*, and the statement(s) block is executed until the entire sequence is exhausted.

Flow Diagram



A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).

This is less like the for keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages.

With the for loop we can execute a set of statements, once for each item in a list, tuple, set etc.

Example

print each fruit in a fruit list:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    print(x)
```

OUTPUT
apple
banana
cherry

Looping Through a String

Even strings are iterable objects, they contain a sequence of characters:

Example:

Loop through the letters in the word “banana”.

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

OUTPUT
b
a
n
a
n
a

The **break** Statement

With the break statement we can stop the loop before it has looped through all the items:

Example:

Exit the loop when x is “banana”.

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
      print(x)
      if x == "banana":
        break
```

OUTPUT
apple
banana

The **continue** Statement

With the continue statement we can stop the current iteration of the loop, and continue with the next:

Example:

Do not print banana:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    if x == "banana":
       continue
    print(x)
```

OUTPUT
apple
cherry

The **range()** Function

To loop through a set of code a specified number of times, we can use the range() function,

The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

Example

Using the range() function:

```
for x in range(6):
    print(x)
```

OUTPUT
0
1
2
3
4
5

The range() function defaults to 0 as a starting value, however it is possible to specify the starting value by adding a parameter: range(2, 6), which means values from 2 to 6 (but not including 6):

Example

Using the range() function start parameter:

```
for x in range(2, 6):
     print(x)
```

OUTPUT
2
3
4
5

The range() function defaults to increment the sequence by 1, however it is possible to specify the increment value by adding a third parameter: range(2, 30, **3**):

Example

Using the range() function start parameter:

Increment the sequence with 3 (default is 1):

```
for x in range(2, 30, 3):
    print(x)
```

OUTPUT
2

```
5
8
11
14
17
20
23
26
29
```

Nested Loops

A nested loop is a loop inside a loop.

The "inner loop" will be executed one time for each iteration of the "outer loop":

Example

Print each adjective for every fruit:

```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]

for x in adj:
    for y in fruits:
        print(x, y)
```

OUTPUT

```
red apple
red banana
red cherry
big apple
big banana
big cherry
tasty apple
tasty banana
tasty cherry
```

While Loop

Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.

A **while** loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true.

Syntax

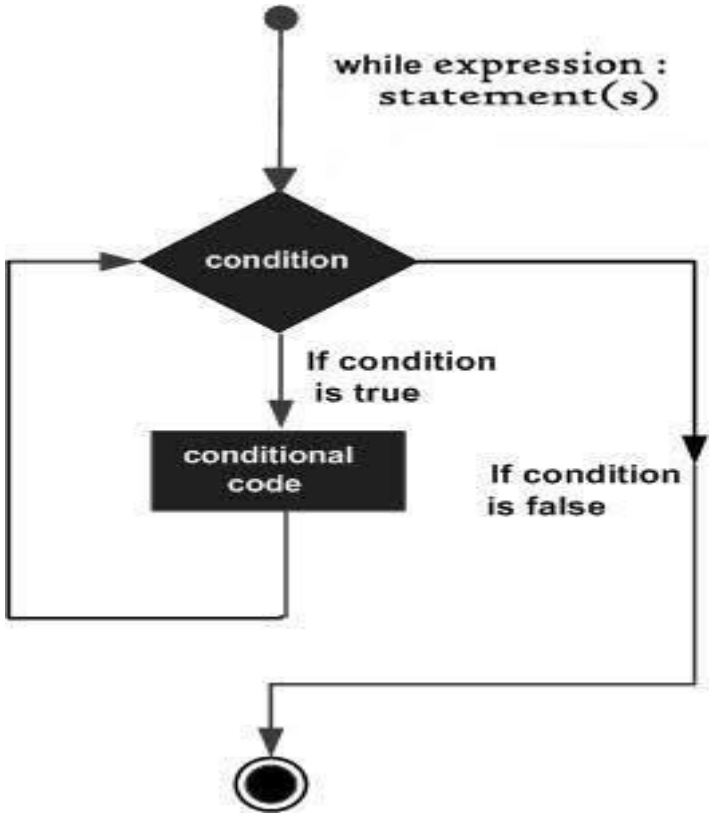
```
while expression:
    statement(s)
```

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any non-zero value. The loop iterates while the condition is true.

When the condition becomes false, program control passes to the line immediately following the loop.

In Python, all the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code. Python uses indentation as its method of grouping statements.

Flow Diagram



Here, key point of the while loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

With the while loop we can execute a set of statements as long as a condition is true.

Example

Print i as long as i is less than 6:

```
i = 1
while i < 6:
```

```
    print(i)
  i += 1
```

OUTPUT
1
2
3
4
5

The break Statement

With the break statement we can stop the loop even if the while condition is true:

Example

Exit the loop when i is 3:

```
i = 1
while i < 6:
   print(i)
   if i == 3:
        break
   i =i + 1
```

OUTPUT
1
2
3

The continue Statement

With the continue statement we can stop the current iteration, and continue with the next:

Example

Continue to the next iteration if i is 3:

```
i = 0
while i < 6:
   i += 1
   if i == 3:
       continue
  print(i)
```

OUTPUT

```
1
2
4
5
6
```

The else Statement

With the else statement we can run a block of code once when the condition no longer is true:

Example

Print a message once the condition is false:

```
i = 1
while i < 6:
    print(i)
    i += 1
else:
  print("i is no longer less than 6")
```

OUTPUT

```
1
2
3
4
5
i is no longer less than 6
```

Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

Sr.No.	Control Statement & Description
1	break statement Terminates the loop statement and transfers execution to the statement immediately following the loop.

2	continue statement Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
3	pass statement The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.

Nested Loop

Python programming language allows to use one loop inside another loop. Following section shows few examples to illustrate the concept.

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)
```

A final note on loop nesting is that 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 for loop to find the prime numbers from 2 to 100 −

```
i = 2
while(i < 100):
   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
```

OUTPUT
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
31 is prime
37 is prime
41 is prime
43 is prime
47 is prime
53 is prime
59 is prime
61 is prime
67 is prime
71 is prime
73 is prime
79 is prime
83 is prime
89 is prime
97 is prime

Arrays or List

Introduction:

An array is defined as a collection of items that are stored at contiguous memory locations. It is a container which can hold a fixed number of items, and these items should be of the same type. An array is popular in most programming languages like C/C++, JavaScript, Python etc.

Array is an idea of storing multiple items of the same type together and it makes easier to calculate the position of each element by simply adding an offset to the base value. A combination of the arrays could save a lot of time by reducing the overall size of the code. It is used to store multiple values in single variable.

The most basic data structure in Python is the **sequence**. Each element of a sequence is assigned a number - its position or index. The first index is zero, the second index is one, and so forth.

Python has six built-in types of sequences, but the most common ones are lists and tuples, which we would see in this tutorial.

There are certain things you can do with all sequence types. These operations include indexing, slicing, adding, multiplying, and checking for membership. In addition, Python has built-in functions for finding the length of a sequence and for finding its largest and smallest elements.

Advantages of Array/ List

There are advantages and disadvantages of using both types. It is often convenient to use a list where you do not need to know the data type or length. However, arrays are a more direct way of accessing data in memory so they are much faster and more efficient.

Lists

The list is a most versatile data type available in Python which can be written as a list of comma-separated values (items) between square brackets. Important thing about a list is that items in a list need not be of the same type.

Creating a list is as simple as putting different comma-separated values between square brackets. For example –

```
list1 = ['physics', 'chemistry', 1997, 2000]
list2 = [1, 2, 3, 4, 5 ]
list3 = ["a", "b", "c", "d"]
```

Similar to string indices, list indices start at 0, and lists can be sliced, concatenated and so on.

Access the Elements of an Array

You refer to an array element by referring to the *index number*.

Example

Get the value of the first array item:

```
list1 = ['physics', 'chemistry', 1997, 2000];

x = len(list1)

print(x)
```

OUTPUT
4

Looping List Elements

You can use the for in loop to loop through all the elements of an array.

Example

Print each item in the list1 array:

```
list1 = ['physics', 'chemistry', 1997, 2000]
for x in list1:
    print(x)
```

OUTPUT

physics

chemistry

1997

2000

Adding List Elements

You can use the append () method to add an element to an array.

Example

Add one more element to the list1 array:

```
list1 = ['physics', 'chemistry', 1997, 2000]

list1.append("100")

for x in list1:

   print(x)physics
```

OUTPUT

chemistry

1997

2000

100

Built-in List Functions & Methods

Python has a set of built-in methods that you can use on lists/arrays.

Method	Description	Example
append()	Adds an element at the end of the list Syntax list.append(value)	fruits = ['apple', 'banana', 'cherry'] fruits.append("orange") print(fruits) OUTPUT ['apple', 'banana', 'cherry', 'orange']
clear()	Removes all the elements from the list Syntax list.clear()	fruits=['apple','banana','cherry','orange'] fruits.clear() OUTPUT []
copy()	Returns a copy of the list Syntax list.copy()	fruits = ['apple', 'banana', 'cherry', 'orange'] x = fruits.copy() OUTPUT ['apple', 'banana', 'cherry']
count()	Returns the number of elements with the specified value Syntax list.count(value)	fruits = ["apple", "banana", "cherry","cherry"] x = fruits.count("cherry") print(x)

		OUTPUT 2
extend()	Add the elements of a list (or any iterable), to the end of the current list Syntax list.extend(iterable)	fruits = ['apple', 'banana', 'cherry'] cars = ['Ford', 'BMW', 'Volvo'] fruits.extend(cars) OUTPUT ['apple', 'banana', 'cherry', 'Ford', 'BMW', 'Volvo']
index()	Returns the index of the first element with the specified value Syntax list.index(elmnt)	fruits = ['apple', 'banana', 'cherry'] x = fruits.index("cherry") print(x) OUTPUT 2
insert()	Adds an element at the specified position Syntax list.insert(pos, elmnt)	fruits = ['apple', 'banana', 'cherry'] fruits.insert(1, "orange") OUTPUT ['apple', 'orange', 'banana', 'cherry']
pop()	Removes the element at the	fruits = ['apple', 'banana', 'cherry']

	specified position Syntax list.pop(pos)	fruits.pop(1) OUTPUT ['apple', 'cherry']
remove()	Removes the first item with the specified value Syntax list.remove(elmnt)	fruits = ['apple', 'banana', 'cherry'] fruits.remove("banana") OUTPUT ['apple', 'cherry']
reverse()	Reverses the order of the list Syntax list.reverse()	fruits = ['apple', 'banana', 'cherry'] fruits.reverse() OUTPUT ['cherry', 'banana', 'apple']
sort()	Sorts the list Syntax list.sort(reverse=True|False, key=myFunc)	cars = ['Ford', 'BMW', 'Volvo'] cars.sort() OUTPUT ['BMW', 'Ford', 'Volvo']

Sr.No.	**Function with Description**

1	cmp(list1, list2) Compares elements of both lists.
2	len(list) Gives the total length of the list.
3	max(list) Returns item from the list with max value.
4	min(list) Returns item from the list with min value.
5	list(seq) Converts a tuple into list.

Importing the array module

The array module defines an object type that can compactly represent an array of some basic values as characters, integers, floating-point numbers. Arrays are sequence types and behave similarly as lists, except that the type of objects stored in them is constrained

Array in Python can be created by importing array module.

array(data_type, value_list) is used to create an array with data type and value list specified in its arguments.

The array module is an extremely useful module for creating and maintaining arrays. These arrays are similar to the arrays in the C language. This article explains how to create arrays and several other useful methods to make working with arrays easier. This is a Python built-in module and comes ready to use in the Python Standard Library.

To use the array module, we need to first import it. To do this, we can use keyword import.

Example of importing the array module

```
import array

print(dir(array))
```

OUTPUT

```
>>> import array

>>> print(dir(array))

['ArrayType', '__doc__', '__loader__', '__name__', '__package__', '_array_reconstructor',
'array', 'typecodes']
```

In the above code example, we imported the array module and returned the list of all useful objects in the module.

Type codes

Type code	C Type	Python Type	Minimum Size (Bytes)
b	signed char	int	1
B	unsigned char	int	1
u	Py_UNICODE	Unicode Character	2
h	signed short	int	2
H	unsigned short	int	2
i	signed int	int	2
I	unsigned int	int	2

l	signed long	int	4
L	unsigned long	int	4
q	signed long long	int	8
Q	unsigned long long	int	8
f	Float	float	4
d	Double	float	8

Creating an Array

We can get a string representing all the type codes by using the attribute typecodes.

Example of using typecodes in Python

```
import array
print(array.typecodes)
```

OUTPUT

```
bBuhHiIlLqQfd
```

class array

class array.array(typecode[, initializer])

- A new array whose items are restricted by typecode, and initialized from the optional initializer value, which must be a list, a bytes-like object, or iterable over elements of the appropriate type.
- If given a list or string, the initializer is passed to the new array‟s fromlist(), frombytes(), or fromunicode() method (see below) to add initial items to the array. Otherwise, the iterable initializer is passed to the extend() method.

Syntax of the class array in Python

class array.array(typecode[, initializer])

Let us create an array using the above class.

Example of creating an array in Python

```
import array

numbers = array.array('i', [1,3,5,7])

print(numbers)
```

OUTPUT

array(„i", [1, 3, 5, 7])

There are several attributes and methods that we can use on the array object that we created.

Slicing an array

Slicing in python means taking elements from one given index to another given index. We pass slice instead of index like this: [start:end] . We can also define the step, like this: [start:end:step] .

you can use slice [start:stop:step] to select a part of a sequence object such as a list, string, or tuple to get a value or assign another value. It is also possible to select a subarray by slicing for the NumPy array numpy. ndarray and extract a value or assign another value.

Common examples of array slicing are extracting a substring from a string of characters, the "ell" in "hello", extracting a row or column from a two-dimensional array, or extracting a vector from a matrix. Depending on the programming language, an array slice can be made out of non-consecutive elements.

Slicing in Python is a feature that enables accessing parts of sequences like strings, tuples, and lists. You can also use them to modify or delete the items of mutable sequences such as lists. Slices can also be applied on third-party objects like NumPy arrays, as well as Pandas series and data frames.

Syntax

array[start: end: step]

Parameter Values

Parameter	Description
start	Optional. An integer number specifying at which position to start the slicing. Default is 0
end	An integer number specifying at which position to end the slicing
step	Optional. An integer number specifying the step of the slicing. Default is 1

Example 1: of slicing an array

```
a = ("a", "b", "c", "d", "e", "f", "g", "h")

x = slice(2)

print(a[x])
```

OUTPUT

('a', 'b', 'c', 'd', 'e')

Example 2:

```
a = ("a", "b", "c", "d", "e", "f", "g", "h")

x = slice(3, 5)

print(a[x])
```

OUTPUT

('d', 'e')

Example 3:

```
import array

numbers1 = array.array('i', [1, 3, 5, 4, 8, 12, 1, 7, 2, 6])

print(numbers1[1: 8: 2])
```

OUTPUT

```
array(„i", [3, 4, 12, 7])
```

Functions:

A function is a block of code which only runs when it is called. You can pass data, known as parameters, into a function. A function can return data as a result.

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

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

Defining a Function

- You 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. You 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 you need to inform them in the same order that they were defined.

Creating a Function

In Python a function is defined using the def keyword:

Example

```
def my_function():
  print("Hello from a function")
```

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, you can execute it by calling it from another function or directly from the Python prompt.

To call a function, use the function name followed by parenthesis:

Example

```
def my_function():
  print("Hello from a function")
my_function()
```

OUTPUT

Hello from a function

Returning(single and multiple) results from a function

you can return multiple values by simply return them separated by commas. As an example, define a function that returns a string and a number as follows: Just write each value after the return , separated by commas .

Python functions can return multiple variables. These variables can be stored in variables directly. A function is not required to return a variable, it can return zero, one, two or more variables.

Returning multiple values from a function is quite cumbersome in C and other languages, but it is very easy in Python.

Single variable

you can return variables from a function. In the most simple case you can return a single variable:

Example :

```
def complexfunction(a,b):

    sum = a +b

    print(sum)

    return  sum

complexfunction(2,3)
```

OUTPUT

5

Call the function with complexfunction(2,3) and its output can be used or saved.

Multiple return

Create a function getPerson(). As you already know a function can return a single variable, but it can also return multiple variables.

We‟ll store all of these variables directly from the function call.

```
def getPerson():

    name = "SHAILESH PATEL"

    age = 35
```

```
    country = "INDIA"

    return name,age,country

name,age,country = getPerson()

print(name)

print(age)

print(country)
```

OUTPUT

SHAILES

H PATEL

35

INDIA

Pass by Object Reference

All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function.

For example 1:

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

OUTPUT

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

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

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

Example 2:

There is one more example where argument is being passed by reference and the reference is being overwritten inside the called function.

```
def changeme( mylist ):

   "This changes a passed list into this function"

   mylist = [1,2,3,4]; # This would assig new reference in mylist

   print ("Values inside the function: ", mylist)

   return

# Now you can call changeme function

mylist = [10,20,30];

changeme( mylist );

print ("Values outside the function: ", mylist)
```

OUTPUT

```
Values inside the function: [1, 2, 3, 4]

Values outside the function: [10, 20, 30]
```

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 is above.

Function Arguments

The terms parameter and argument can be used for the same thing: information that are passed into a function.

From a function's perspective:

A parameter is the variable listed inside the parentheses in the function definition.

An argument is the value that is sent to the function when it is called.

```
def function_name(param1, param2...):
```

```
    do_something
    return return_value
```

When calling the defined function, write as follows:

```
function_name(arg1, arg2...)
```

Example:

```
def add(a, b):
    x = a + b
    return x
x = add(3, 4)
print(x)
```

OUTPUT

```
7
```

Parameters and return values by return can be omitted if they are not necessary.

Example:

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

OUTPUT

```
Hello
```

Types of Function arguments

- Positional arguments
- Keyword arguments
- Default arguments
- Variable-length arguments

Positional arguments

Keyword arguments

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

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

You can also send arguments with the *key* = *value* syntax.

This way the order of the arguments does not matter.

Example:

```
def my_function(child3, child2, child1):

  print("The youngest child is " + child3)

my_function(child1 = "Manya", child2 = "Lavisha", child3 = "Drushti")
```

OUTPUT

The youngest child is Drushti

Default arguments

Default arguments in Python functions are those arguments that take default values if no explicit values are passed to these arguments from the function call. Let's define a function with one default argument.

If we call the function without argument, it uses the default value:

Example

```
def my_function(country = "Delhi"):

  print("I am from " + country)

my_function("Aburoad")

my_function("Baroda")
```

```
my_function()

my_function("Ahmedabad")
```

OUTPUT

```
I am from Aburoad

I am from Baroda

I am from Delhi

I am from Ahmedabad
```

Variable-length argument

A variable-length argument is a feature that allows a function to receive any number of arguments. There are situations where a function handles a variable number of arguments according to requirements, such as: Sum of given numbers. Minimum of given numbers and many more.

Types of Variable length arguments

- Non - Keyworded Arguments (*args)
- Keyworded Arguments (**kwargs)

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

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

```
def functionname([formal_args,] *var_args_tuple ):

  "function_docstring"

   Function_suite

   return [expression]
```

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

Example

```
# Function definition is  here

def printinfo( arg1, *vartuple ):

#"This prints a variable passed arguments"

   print ("Output is: ")

   print (arg1)

   for var in vartuple:

      print(var)

   return

# Now you can call printinfo function

printinfo( 10 )

printinfo( 70, 60, 50 )Output
```

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

```
Output is:

10

Output is:

70

60

50
```

Variable-length arguments, varargs for short, are arguments that can take an unspecified amount of input. When these are used, the programmer does not need to wrap the data in a list or an alternative sequence.

In Python, varargs are defined using the *args syntax. Let's reimplement our my_min() function with *args:

```
def my_min(*args):

    result = args[0]

    for num in args:
```

```
        if num < result:
            result = num
    return result
my_min(4, 5, 6, 7, 2)
```

OUTPUT

2

With *args, we can accept multiple arguments in sequence as is done in my_min(). These arguments are processed by their position. What if we wanted to take multiple arguments, but reference them by their name? We'll take a look at how to do this in the next section.

Using Many Named Arguments with **kwargs

Python can accept multiple keyword arguments, better known as **kwargs. It behaves similarly to *args, but stores the arguments in a dictionary instead of tuples:

```
def kwarg_type_test(**kwargs):
    print(kwargs)
kwarg_type_test(a="hi")
kwarg_type_test(roses="red", violets="blue")
```

OUTPUT

{'a': 'hi'}

{'roses': 'red', 'violets': 'blue'}

By using a dictionary, **kwargs can preserve the names of the arguments, but it would not be able to keep their position.

Since **kwargs is a dictionary, you can iterate over them like any other using the .items() method:

```
def kwargs_iterate(**kwargs):
    for i, k in kwargs.items():
        print(i, '=', k)
```

```
kwargs_iterate(hello='world')
```

OUTPUT

```
hello = world
```

Keyword arguments are useful when you aren't sure if an argument is going to be available. For example, if we had a function to save a blog post to a database, we would save the information like the content and the author. A blog post may have tags and categories, though those aren't always set.

Combining Varargs and Keyword Arguments

Quite often we want to use both *args and **kwargs together, especially when writing Python libraries or reusable code. Lucky for us, *args and **kwargs play nicely together, and we can use them in the following way:

```
def combined_varargs(*args, **kwargs):
    print(args)
    print(kwargs)
combined_varargs(1, 2, 3, a="hi")
```

OUTPUT

```
(1, 2, 3)
{'a': 'hi'}
```

When mixing the positional and named arguments, positional arguments must come before named arguments. Furthermore, arguments of a fixed length come before arguments with variable length. Therefore, we get an order like this:

Known positional arguments

*args

Known named arguments

**kwargs

Inbuilt Functions

A function which is already defined in a program or programming framework with a set of statements, which together performs a task and it is called Build-in function.

So users need not create this type of function and can use directly in their program or application.

Many build in functions are available in most of the programs or programming frameworks and these differ from each other; each one is having its unique functionality.

Python has a set of built-in functions.

Function	Description
abs()	Returns the absolute value of a number
all()	Returns True if all items in an iterable object are true
any()	Returns True if any item in an iterable object is true
ascii()	Returns a readable version of an object. Replaces none-ascii characters with escape character
bin()	Returns the binary version of a number
bool()	Returns the boolean value of the specified object
bytearray()	Returns an array of bytes
bytes()	Returns a bytes object

callable()	Returns True if the specified object is callable, otherwise False
chr()	Returns a character from the specified Unicode code.
classmethod()	Converts a method into a class method
compile()	Returns the specified source as an object, ready to be executed
complex()	Returns a complex number
delattr()	Deletes the specified attribute (property or method) from the specified object
dict()	Returns a dictionary (Array)
dir()	Returns a list of the specified object's properties and methods
divmod()	Returns the quotient and the remainder when argument1 is divided by argument2
enumerate()	Takes a collection (e.g. a tuple) and returns it as an enumerate object
eval()	Evaluates and executes an expression

exec()	Executes the specified code (or object)
filter()	Use a filter function to exclude items in an iterable object
float()	Returns a floating point number
format()	Formats a specified value
frozenset()	Returns a frozenset object
getattr()	Returns the value of the specified attribute (property or method)
globals()	Returns the current global symbol table as a dictionary
hasattr()	Returns True if the specified object has the specified attribute (property/method)
hash()	Returns the hash value of a specified object
help()	Executes the built-in help system
hex()	Converts a number into a hexadecimal value

id()	Returns the id of an object
input()	Allowing user input
int()	Returns an integer number
isinstance()	Returns True if a specified object is an instance of a specified object
issubclass()	Returns True if a specified class is a subclass of a specified object
iter()	Returns an iterator object
len()	Returns the length of an object
list()	Returns a list
locals()	Returns an updated dictionary of the current local symbol table
map()	Returns the specified iterator with the specified function applied to each item
max()	Returns the largest item in an iterable

memoryview()	Returns a memory view object
min()	Returns the smallest item in an iterable
next()	Returns the next item in an iterable
object()	Returns a new object
oct()	Converts a number into an octal
open()	Opens a file and returns a file object
ord()	Convert an integer representing the Unicode of the specified character
pow()	Returns the value of x to the power of y
print()	Prints to the standard output device
property()	Gets, sets, deletes a property
range()	Returns a sequence of numbers, starting from 0 and increments by 1 (by default)

repr()	Returns a readable version of an object
reversed()	Returns a reversed iterator
round()	Rounds a numbers
set()	Returns a new set object
setattr()	Sets an attribute (property/method) of an object
slice()	Returns a slice object
sorted()	Returns a sorted list
staticmethod()	Converts a method into a static method
str()	Returns a string object
sum()	Sums the items of an iterator
super()	Returns an object that represents the parent class

tuple()	Returns a tuple
type()	Returns the type of an object
vars()	Returns the __dict__property of an object
zip()	Returns an iterator, from two or more iterators

Lambda Function

A lambda function is a single-line function declared with no name, which can have any number of arguments, but it can only have one expression. Such a function is capable of behaving similarly to a regular function declared using the Python's def keyword.

How to create lambda function

To create a lambda function first write keyword lambda followed by one of more arguments separated by comma (,), followed by colon a (:), followed by a single line expression. Here we are using two arguments x and y , expression after colon is the body of the lambda function.

Creating a function with lambda is slightly faster than creating it with def . The difference is due to def creating a name entry in the locals table. The resulting function has the same execution speed.

A lambda function is a small anonymous function.

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

Syntax

lambda arguments : expression

The expression is executed and the result is returned:

Example

Add 10 to argument a, and return the result:

```
x = lambda a : a + 10

print(x(5))
```

OUTPUT

15

Lambda functions can take any number of arguments:

Example

Multiply argument a with argument b and return the result:

```
x = lambda a, b : a * b

print(x(5, 6))
```

OUTPUT

30

Example

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

```
x = lambda a, b, c : a + b + c

print(x(5, 6, 2))
```

OUTPUT

13

Why Use Lambda Functions?

The power of lambda is better shown when you use them as an anonymous function inside another function.

Say you have a function definition that takes one argument, and that argument will be multiplied with an unknown number:

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

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

Example

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

mydoubler = myfunc(2)

print(mydoubler(11))
```

OUTPUT

22

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

Example

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

mytripler = myfunc(3)

print(mytripler(11))
```

OUTPUT

33

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

Example

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

mydoubler = myfunc(2)
```

```
mytripler = myfunc(3)

print(mydoubler(11))

print(mytripler(11))
```

OUTPUT

22

33

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

Modules

Library: - Library is a collection of modules (and packages) that together cater to a specific type of applications or requirements.

Modules: - The act of partitioning a program into individual components (known as modules) is called modularity.

A module is a separate unit in itself.

A python module is a file (.py file) containing variables, class definitions, statement and functions related to a particular task.

The python module that come preloaded with python are called standard library modules.

Advantage of module: -

- It reduces its complexity to some degree.
- It creates a number of well defined, documented boundaries within the program.
- Its contents can be reused in other programs, without having to rewrite or recreate them.

Importing Modules in a Python Program

Python provides import statement to import modules in a program. The import statement can be used in two forms.

(1) To import entire module: the import <module> command

(2) To import selected objects from a module: the from <module> import <object> command

Importing Entire Module: -

The imports statement can be used to import entire module and even for importing selected items.

To import entire module syntax is -

import <module 1 >, <module 2>....

For example:

import time

import decimals, fractions

Dot notation :- After importing a module, to access one of the functions, you have to specify the name of the module and the name of the function, separated by a dot (also known as a period) this format is called dot notation.

Syntax: -

<module-name>.<function-name> ()

This way of referring to a module's object is called dot notation.

For example:

import math

math.sqrt(16)

• You can give alias name to imported module as

Syntax :

import <module> as <alias name>

For Example :-

import math as a

a.sqrt (16)

Importing Select Objects from a Module: -

If you want to import some selected items, not all from a module, then you can use following syntax:-

from <module name >import<object name>

For example:

from math import sqrt

To Import Multiple Objects:-

If you want to import multiple objects from the module then you can use following syntax :-

from <module name >import<object name>,<object name>,<object name>....

For example: -

from math import sqrt, pi, pow

USING PYTHON STANDARD LIBRARY'S FUNCTIONS AND MODULES: -

Python's standard library offers many built in functions and modules for specialized type of functionality.

For example:-

len(),str(),type()

math module, random module , etc.

Consider a module to be the same as a code library.

A file containing a set of functions you want to include in your application.

Create a Module

To create a module just save the code you want in a file with the file extension .py:

Example

Save this code in a file named mymodule.py

def greeting(name):

```
  print("Hello, " + name)
```

Use a Module

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

Example

Import the module named mymodule, and call the greeting function:

```
import mymodule

mymodule.greeting("Jonathan")
```

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

Variables in Module

The module can contain functions, as already described, but also variables of all types (arrays, dictionaries, objects etc):

Example 2 programs are combined first create mymodule.py then create `demo_module2.py then run this file demo_module2.py`

Save this code in the file mymodule.py

```
person1 = {

  "name": "John",

  "age":   36,

  "country": "Norway"

}
```

Import the module named mymodule, and access the person1 dictionary: `save file name this demo_module2.py and run`

```
import mymodule

a = mymodule.person1["age"]

b=mymodule.person1[“name”]
```

```
print(a)

print(b)
```

OUTPUT

36

John

Naming a Module

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

Re-naming a Module

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

Example

Create an alias for mymodule called mx:

```
import mymodule as mx

a = mx.person1["age"]

print(a)
```

Built-in Modules

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

Example

Import and use the platform module:

```
import platform

x = platform.system()

print(x)
```

OUTPUT

Windows

Using the dir() Function

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

Example

List all the defined names belonging to the platform module:

```
import platform

x = dir(platform)

print(x)
```

OUTPUT

```
['DEV_NULL', '__builtins__', '__cached__', '__copyright__', '__doc__', '__file__', '__initializing__', '__loader__', '__name__', '__package__', '__version__', '_bcd2str', '_default_architecture', '_dist_try_harder', '_follow_symlinks', '_ironpython26_sys_version_parser', '_ironpython_sys_version_parser', '_java_getprop', '_libc_search', '_lsb_release_version', '_mac_ver_gestalt', '_mac_ver_lookup', '_mac_ver_xml', '_node', '_norm_version', '_parse_release_file', '_platform', '_platform_cache', '_pypy_sys_version_parser', '_release_filename', '_release_version', '_supported_dists', '_sys_version', '_sys_version_cache', '_sys_version_parser', '_syscmd_file', '_syscmd_uname', '_syscmd_ver', '_uname_cache', '_ver_output', '_win32_getvalue', 'architecture', 'collections', 'dist', 'java_ver', 'libc_ver', 'linux_distribution', 'mac_ver', 'machine', 'node', 'os', 'platform', 'popen', 'processor', 'python_branch', 'python_build', 'python_compiler', 'python_implementation', 'python_revision', 'python_version', 'python_version_tuple', 're', 'release', 'subprocess', 'sys', 'system', 'system_alias', 'uname', 'uname_result', 'version', 'win32_ver']
```

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

Unit 3: **[18 MARKS]**

List, Tuples, File Handling, Classes, Inheritance and Polymorphism
List and Tuples: Exploring List, Loop Lists, List Comprehension, Join Two Lists, Updating the elements of the list, Tuples, Creating and accessing Tuple elements, Basic operations on Tuples.
File: Create, reading, writing, deleting.
Classes: Creation of class and object, The__init___() function, Self parameter, Modifying the property of a class.
Inheritance & Encapsulation, Polymorphism.

Exploring List

A list in Python is used to store the sequence of various types of data. Python lists are mutable type its mean we can modify its element after it created. However, Python consists of six data-types that are capable to store the sequences, but the most common and reliable type is the list.

A list can be defined as a collection of values or items of different types. The items in the list are separated with the comma (,) and enclosed with the square brackets [].

A list can be define as below

L1 = ["John", 102, "USA"]

L2 = [1, 2, 3, 4, 5, 6]

If we try to print the type of L1, L2, and L3 using type() function then it will come out to be a list.

```
print(type(L1))

print(type(L2))
```

Output:

```
<class 'list'>

<class 'list'>
```

Characteristics of Lists

- The lists are ordered.
- The element of the list can access by index.
- The lists are the mutable type.
- The lists are mutable types.
- A list can store the number of various elements.

- Let's check the first statement that lists are the ordered.

Example

```
mylist=[]     # empty list

print(mylist)

names=["Manya", "Lavisha", "Adiyta", "Drushti"]      # string list
print(names)

item=[1, "Manya", "Computer", 75.50, True]      # list with heterogeneous data
print(item)
```

OUTPUT

```
[]
['Manya', 'Lavisha', 'Adiyta', 'Drushti']
[1, 'Manya', 'Computer', 75.5, True]
```

A list can contain unlimited data depending upon the limitation of your computer's memory.

Example

```
names=["Shailesh Patel", "Manya", "Anik",

"Lavisha"]print(names[0]) # returns "Jeff"

print(names[1]) # returns "Bill"

print(names[2]) # returns "Steve"

print(names[3]) # returns "Mohan"

print(names[4]) # throws IndexError: list index out of range.
```

OUTPUT

Shailesh

Patel

Manya

Anik

Lavisha

Traceback (most recent call last):

```
  File "C:/Python27/listex.py", line 6, in <module>
    print(names[4])
IndexError: list index out of range
```

A list can contain multiple inner lists as items that can be accessed using indexes.

Example

```
nums=[1, 2, 3, [4, 5, 6, [7, 8, [9]]], 10]
print(nums[0]) # returns 1
print(nums[1]) # returns 2
print(nums[3]) # returns [4, 5, 6, [7, 8, [9]]]
print(nums[4]) # returns 10
print(nums[3][0]) # returns 4
print(nums[3][3]) # returns [7, 8, [9]]
print(nums[3][3][0]) # returns 7
print(nums[3][3][2]) # returns [9]
```

OUTPUT

```
1
2
[4, 5, 6, [7, 8, [9]]]
10
4
[7, 8, [9]]
7
[9]
```

List Class

All the list objects are the objects of the list class in Python. Use the list() constructor to convert from other sequence types such as tuple, set, dictionary, string to list.

```
nums=[1,2,3,4]
print(type(nums))

mylist=list('Hello')
print(mylist)
```

```
nums=list({1:'one',2:'two'})
print(nums)

nums=list((10, 20, 30))
print(nums)

nums=list({100, 200, 300})
print(nums)
```

OUTPUT

```
<class  'list'>
['H', 'e', 'l', 'l', 'o']
[1, 2]
[10, 20, 30]
[100, 200, 300]
```

Loop Lists

A list items can be iterate using the for loop.

```
names=["Manya",  "Shailesh Patel",  "Ram",

"Aditya"]for name in names:

    print(name)
```

OUTPUT

Manya

Shailesh

Patel

Ram

Aditya

List Comprehension

List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.

Syntax

```
newlist = [expression for item in iterable if condition == True]
```

The return value is a new list, leaving the old list unchanged.

Example:

Based on a list of fruits, you want a new list, containing only the fruits with the letter "a" in the name.

Without list comprehension you will have to write a for statement with a conditional test inside:

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = []

for x in fruits:
  if "a" in x:
    newlist.append(x)

print(newlist)
```

With list comprehension you can do all that with only one line of code:

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]

newlist = [x for x in fruits if "a" in x]

print(newlist)
```

OUTPUT
['apple', 'banana', 'mango']

Condition
The condition is like a filter that only accepts the items that valuate to True.

Example
Only accept items that are not "apple":

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]

newlist = [x for x in fruits if x != "apple"]
print(newlist)
```

OUTPUT
['banana', 'cherry', 'kiwi', 'mango']

The condition if x != "apple" will return True for all elements other than "apple", making the new list contain all fruits except "apple".

the list comprehension syntax contains three parts: an expression, one or more for loop, and optionally, one or more if conditions. The list comprehension must be in the square brackets []. The result of the first expression will be stored in the new list.

The for loop is used to iterate over the iterable object that optionally includes the if condition.

Suppose we want to find even numbers from 0 to 20 then we can do it using a for loop, as shown below:

Example: Create List of Even Numbers without List Comprehension

```
even_nums = []

for x in range(21):
    if x%2 == 0:
        even_nums.append(x)
print(even_nums)
```

Or

Example: Create List of Even Numbers with List Comprehension

```
even_nums = [x for x in range(21) if x%2 == 0]
print(even_nums)
```

OUTPUT
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

In the above example, [x for x in range(21) if x%2 == 0] returns a new list using the list comprehension. First, it executes the for loop for x in range(21) if x%2 == 0. The element x would be returned if the specified condition if x%2 == 0 evaluates to True. If the condition evaluates to True, then the expression before for loop would be executed and stored in the new list. Here, expression x simply stores the value of x into a new list.

List comprehension works with string lists also. The following creates a new list of strings that contains 'a'.

Example: List Comprehension with String List

```
names = ['Steve', 'Bill', 'Ram', 'Mohan', 'Abdul']
names2 = [s for s in names if 'a' in s]
print(names2)
```

OUTPUT
['Ram', 'Mohan']

Above, the expression if 'a' in s returns True if an element contains a character 'a'. So, the new list will include names that contain 'a'.

The following example uses a list comprehension to build a list of squares of the numbers between 1 and 10.

Example: List Comprehension

```
squares = [x*x for x in range(11)]
print(squares)
```

OUTPUT
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

Above, a for loop for x in range(11) is executed without any if condition. The expression before for loop x*x stores the square of the element in the new list.

List Comprehension using Nested Loops

It is possible to use nested loops in a list comprehension expression. In the following example, all combinations of items from two lists in the form of a Tuple are added in a third list object.

Example: List Comprehension

```
nums1 = [1, 2, 3]
nums2 = [4, 5, 6]
nums=[(x,y) for x in nums1 for y in nums2]
print(nums)
```

OUTPUT
[(1, 4), (1, 5), (1, 6), (2, 4), (2, 5), (2, 6), (3, 4), (3, 5), (3, 6)]

List Comprehension with Multiple if Conditions

We can use nested if conditions with a list comprehension.

Example: List Comprehension

```
nums = [x for x in range(21) if x%2==0 if x%5==0]
print(nums)
```

OUTPUT
[0, 10, 20]

List Comprehension with if-else Condition
The following example demonstrates the if..else loop with a list comprehension.

Example: List Comprehension

```
odd_even_list = ["Even" if i%2==0 else "Odd" for i in range(5)]
print(odd_even_list)
odd_even_list = [str(i) + '=Even' if i%2==0 else str(i) + "=Odd" for i in range(5)]
print(odd_even_list)
```

OUTPUT
['Even', 'Odd', 'Even', 'Odd', 'Even']
['0=Even', '1=Odd', '2=Even', '3=Odd', '4=Even']

Add List Items

Append Items
To add an item to the end of the list, use the append() method:

Example
Using the append() method to append an item:

```
mylist = ["apple", "banana", "cherry"]
mylist.append("orange")
print(mylist)
```

OUTPUT
['apple', 'banana', 'cherry', 'orange']

Insert Items
To insert a list item at a specified index, use the insert() method.

The insert() method inserts an item at the specified index:

Example
Insert an item as the second position:

```
mylist = ["apple", "banana", "cherry"]
mylist.insert(1, "orange")
print(mylist)
```

OUTPUT
['apple', 'orange', 'banana', 'cherry']

Note: As a result of the examples above, the lists will now contain 4 items.

Extend List
To append elements from another list to the current list, use the extend() method.

Example
Add the elements of tropical to thislist:

```
mylist = ["apple", "banana", "cherry"]
tropical = ["mango", "pineapple", "papaya"]
mylist.extend(tropical)
print(mylist)
```

OUTPUT
['apple', 'banana', 'cherry', 'mango', 'pineapple', 'papaya']

The elements will be added to the end of the list.

Add Any Iterable
The extend() method does not have to append lists, you can add any iterable object (tuples, sets, dictionaries etc.).

Example
Add elements of a tuple to a list:

```
mylist = ["apple", "banana", "cherry"]
mytuple = ("kiwi", "orange")
mylist.extend(mytuple)
print(mylist)
```

OUTPUT
['apple', 'banana', 'cherry', 'kiwi', 'orange']

Sort Lists

Sort List Alphanumerically
List objects have a sort() method that will sort the list alphanumerically, ascending, by default.

Example
Sort the list alphabetically:

```
mylist = ["orange", "mango", "kiwi", "pineapple", "banana"]
mylist.sort()
print(mylist)
```

OUTPUT
['banana', 'kiwi', 'mango', 'orange', 'pineapple']

Example
Sort the list numerically:

```
mylist = [100, 50, 65, 82, 23]
mylist.sort()
print(mylist)
```

OUTPUT
[23, 50, 65, 82, 100]

Sort Descending
To sort descending, use the keyword argument reverse = True:

Example
Sort the list descending:

```
mylist = ["orange", "mango", "kiwi", "pineapple", "banana"]
mylist.sort(reverse = True)
print(mylist)
```

OUTPUT
['pineapple', 'orange', 'mango', 'kiwi', 'banana']

Example
Sort the list descending:

```
mylist = [100, 50, 65, 82, 23]
mylist.sort(reverse = True)
print(mylist)
```
OUTPUT
[100, 82, 65, 50, 23]

Customize Sort Function
You can also customize your own function by using the keyword argument key = function.

The function will return a number that will be used to sort the list (the lowest number first):

Example
Sort the list based on how close the number is to 50:

```
def myfunc(n):
  return abs(n - 50)

mylist = [100, 50, 65, 82, 23]
mylist.sort(key = myfunc)
print(mylist)
```

OUTPUT
[50, 65, 23, 82, 100]

Case Insensitive Sort

By default the sort() method is case sensitive, resulting in all capital letters being sorted before lower case letters:

Example
Case sensitive sorting can give an unexpected result:

```
mylist = ["banana", "Orange", "Kiwi", "cherry"]
mylist.sort()
print(mylist)
```

OUTPUT
['Kiwi', 'Orange', 'banana', 'cherry']

Luckily we can use built-in functions as key functions when sorting a list.

So if you want a case-insensitive sort function, use str.lower as a key function:

Example
Perform a case-insensitive sort of the list:

```
mylist = ["banana", "Orange", "Kiwi", "cherry"]
mylist.sort(key = str.lower)
print(mylist)
```

OUTPUT
['banana', 'cherry', 'Kiwi', 'Orange']

Reverse Order

What if you want to reverse the order of a list, regardless of the alphabet?

The reverse() method reverses the current sorting order of the elements.

Example
Reverse the order of the list items:

```
mylist = ["banana", "Orange", "Kiwi", "cherry"]
mylist.reverse()
print(mylist)
```

OUTPUT
['cherry', 'Kiwi', 'Orange', 'banana']

Copy a List

You cannot copy a list simply by typing list2 = list1, because: list2 will only be a reference to list1, and changes made in list1 will automatically also be made in list2.

There are ways to make a copy, one way is to use the built-in List method copy().

Example
Make a copy of a list with the copy() method:

```
thislist = ["apple", "banana", "cherry"]
mylist = thislist.copy()
print(mylist)
```

Another way to make a copy is to use the built-in method list().

Example
Make a copy of a list with the list() method:

```
thislist = ["apple", "banana", "cherry"]
mylist = list(thislist)
print(mylist)
```

OUTPUT
['apple', 'banana', 'cherry']

Join Two Lists

There are several ways to join, or concatenate, two or more lists in Python.

One of the easiest ways are by using the + operator.

Example
Join two list:

```
list1 = ["a", "b", "c"]
list2 = [1, 2, 3]

list3 = list1 + list2
print(list3)
```

OUTPUT
['a', 'b', 'c', 1, 2, 3]

Another way to join two lists is by appending all the items from list2 into list1, one by one:

Example
Append list2 into list1:

```
list1 = ["a", "b" , "c"]
list2 = [1, 2, 3]
```

```
for x in list2:
  list1.append(x)

print(list1)
```

OUTPUT
['a', 'b', 'c', 1, 2, 3]

Or you can use the extend() method, which purpose is to add elements from one list to another list:

Example
Use the extend() method to add list2 at the end of list1:

```
list1 = ["a", "b" , "c"]
list2 = [1, 2, 3]
list1.extend(list2)
print(list1)
```

OUTPUT
['a', 'b', 'c', 1, 2, 3]

Updating the elements of the list

The list is mutable. You can add new items in the list using the append () or insert () methods, and update items using indexes.
Example:

```
names=["Manya", "Shailesh Patel", "Lavisha", "Aditya"]

names[0]="Newton"          # update 1st item at index 0

names[1]="Ram"             # update 2nd item at index 1

names.append("Abdul")    # adds new item at the end

print(names)
```

OUTPUT

['Newton', 'Ram', 'Lavisha', 'Aditya', 'Abdul']

List Operators

Like the string, the list is also a sequence. Hence, the operators used with strings are also available for use with the list (and tuple also).

Operator	Example
The + operator returns a list containing all the elements of	>>> L1=[1,2,3]

Operator	Example

the first and the second list.	>>> L2=[4,5,6]
	>>> L1+L2
	[1, 2, 3, 4, 5, 6]

The * operator concatenates multiple copies of the same list.	>>> L1=[1,2,3]
	>>> L1*3
	[1, 2, 3, 1, 2, 3, 1, 2, 3]

The slice operator [] returns the item at the given index. A negative index counts the position from the right side.	>>> L1=[1, 2, 3]
	>>> L1[0]
	1
	>>> L1[-3]
	1
	>>> L1[1]
	2
	>>> L1[-2]
	2
	>>> L1[2]
	3
	>>> L1[-1]
	3

The range slice operator [FromIndex : Untill Index - 1] fetches items in the range specified by the two index operands separated by : symbol.
If the first operand is omitted, the range starts from the index 0. If the second operand is omitted, the range goes up to the end of the list.
	>>> L1=[1, 2, 3, 4, 5, 6]
	>>> L1[1:]
	[2, 3, 4, 5, 6]
	>>> L1[:3]
	[1, 2, 3]
	>>> L1[1:4]
	[2, 3, 4]
	>>> L1[3:]
	[4, 5, 6]
	>>> L1[:3]
	[1, 2, 3]
	>>> L1[-5:-3]
	[2, 3]

The in operator returns true if an item exists in the given list.	>>> L1=[1, 2, 3, 4, 5, 6]
	>>> 4 in L1
	True
	>>> 10 in L1
	False

The not in operator returns true if an item does not exist in the given list.	>>> L1=[1, 2, 3, 4, 5, 6]
	>>> 5 not in L1
	False
	>>> 10 not in L1
	True

Tuple

A Tuple is a collection of Python objects separated by commas. In some way a Tuple is similar to a list in terms of indexing, nested objects and repetition but a Tuple is immutable unlike lists which are mutable.

Tuples are used to store multiple items in a single variable.

Tuple is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Set, and Dictionary, all with different qualities and usage.

A tuple is a collection which is ordered and unchangeable. Tuples are written with round brackets.

Create a Tuple:

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

print(mytuple)
```

OUTPUT

```
('apple', 'banana', 'cherry')
```

Tuple Items

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

Ordered

When we say that tuples are ordered, it means that the items have a defined order, and that order will not change.

Unchangeable

Tuples are unchangeable, meaning that we cannot change, add or remove items after the tuple has been created.

Allow Duplicates

Since tuples are indexed, they can have items with the same value:

Example

Tuples allow duplicate values:

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

print(mytuple)
```

OUTPUT

('apple', 'banana', 'cherry', 'apple', 'cherry')

Tuple Length

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

Example

Print the number of items in the tuple:

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

print(len(mytuple))
```

OUTPUT

3

Create Tuple with One Item

To create a tuple with only one item, you have to add a comma after the item, otherwise Python will not recognize it as a tuple.

Example

One item tuple, remember the comma:

```
mytuple = ("apple",)

print(type(mytuple))

#NOT a tuple

mytuple = ("apple")
```

OUTPUT

<type 'tuple'>

Tuple Items - Data Types
Tuple items can be of any data type:

Example
String, int and boolean data types:

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

A tuple can contain different data types:

Example
A tuple with strings, integers and boolean values:

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

OUTPUT
<class 'tuple'>

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

Example
What is the data type of a tuple?

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

The tuple() Constructor

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

Example
Using the tuple() method to make a tuple:

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

Access Tuple Items
You can access tuple items by referring to the index number, inside square brackets:

Example
Print the second item in the tuple:

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

OUTPUT
Banana

Note: The first item has index 0.

Negative Indexing
Negative indexing means start from the end.

-1 refers to the last item, -2 refers to the second last item etc.

Example
Print the last item of the tuple:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[-1])
```

OUTPUT
cherry

Range of Indexes
You can specify a range of indexes by specifying where to start and where to end the range. When specifying a range, the return value will be a new tuple with the specified items.

Example
Return the third, fourth, and fifth item:

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[2:5])
```

OUTPUT
('cherry', 'orange', 'kiwi')

Note: The search will start at index 2 (included) and end at index 5 (not included).

Remember that the first item has index 0.

By leaving out the start value, the range will start at the first item:

Example
This example returns the items from the beginning to, but NOT included, "kiwi":

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[:4])
```

OUTPUT
('apple', 'banana', 'cherry', 'orange')

By leaving out the end value, the range will go on to the end of the list:

Example
This example returns the items from "cherry" and to the end:

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[2:])
```

OUTPUT
('cherry', 'orange', 'kiwi', 'melon', 'mango')

Range of Negative Indexes
Specify negative indexes if you want to start the search from the end of the tuple:

Example
This example returns the items from index -4 (included) to index -1 (excluded)

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[-4:-1])
```

OUTPUT
('orange', 'kiwi', 'melon')

Check if Item Exists
To determine if a specified item is present in a tuple use the in keyword:

Example
Check if "apple" is present in the tuple:

```
thistuple = ("apple", "banana", "cherry")
if "apple" in thistuple:
  print("Yes, 'apple' is in the fruits tuple")
```

OUTPUT
Yes, 'apple' is in the fruits tuple

Update Tuples
Tuples are unchangeable, meaning that you cannot change, add, or remove items once the tuple is created but there are some workarounds.

Change Tuple Values
Once a tuple is created, you cannot change its values. Tuples are unchangeable, or immutable as it also is called.

But there is a workaround. You can convert the tuple into a list, change the list, and convert the list back into a tuple.

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

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

print(x)
```

OUTPUT

('apple', 'kiwi', 'cherry')

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

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

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

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

OUTPUT
('apple', 'banana', 'cherry', 'orange')

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

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

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

OUTPUT
('apple', 'banana', 'cherry', 'orange')

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

Remove Items
Note: You cannot remove items in a tuple.

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

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

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

OUTPUT
('banana', 'cherry')

Or you can delete the tuple completely:

Example
The del keyword can delete the tuple completely:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple)
del thistuple
print(thistuple)
```

OUTPUT
('apple', 'banana', 'cherry')

```
Traceback (most recent call last):
  File "C:/Python27/tuple1.py", line 4, in <module>
    print(thistuple)
NameError: name 'thistuple' is not defined
```

Built-in Tuple Functions

Python includes the following tuple functions –

Sr.No.	Function with Description
1	cmp(tuple1, tuple2) Compares elements of both tuples.

2	len(tuple) Gives the total length of the tuple.
3	max(tuple) Returns item from the tuple with max value.
4	min(tuple) Returns item from the tuple with min value.
5	tuple(seq) Converts a list into tuple.

Comparison of two tuples.

Syntax

cmp(tuple1, tuple2)

Parameters
tuple1 – This is the first tuple to be compared

tuple2 – This is the second tuple to be compared

Return Value
If elements are of the same type, perform the compare and return the result. If elements are different types, check to see if they are numbers.

If numbers, perform numeric coercion if necessary and compare.

If either element is a number, then the other element is "larger" (numbers are "smallest").

Otherwise, types are sorted alphabetically by name.

If we reached the end of one of the tuples, the longer tuple is "larger." If we exhaust both tuples and share the same data, the result is a tie, meaning that 0 is returned.

Example

```
tuple1, tuple2 = (123, 'xyz'), (123, 'xyz')
print cmp(tuple1, tuple2)
print cmp(tuple2, tuple1)
```

```
tuple3 = tuple2 + (786,);
print(tuple3)
print cmp(tuple2, tuple3)
```

OUTPUT
0
0
(123, 'xyz', 786)
-1

Above example 0 shows that comparison is match or -1 shows that comparison is not match.

Length of tuple.

Python tuple method len() returns the number of elements in the tuple.

Syntax

```
len(tuple)
```

Parameters
tuple − This is a tuple for which number of elements to be counted.

Return Value
This method returns the number of elements in the tuple.

Example

```
tuple1, tuple2 = (123, 'xyz', 'zara'), (456, 'abc')
print "First tuple length : ", len(tuple1)
print "Second tuple length : ", len(tuple2)
```

OUTPUT
First tuple length : 3
Second tuple length : 2

Maximum tuple

Python tuple method max() returns the elements from the tuple with maximum value.

Syntax
Following is the syntax for max() method −

```
max(tuple)
```

Parameters
tuple − This is a tuple from which max valued element to be returned.

Return Value
This method returns the elements from the tuple with maximum value.

Example

```
tuple1 = (123, 'xyz', 'zara', 'abc')
tuple2 = (456, 700, 200)
print "Max value element : ", max(tuple1)
print "Max value element : ", max(tuple2)
```

OUTPUT
Max value element : zara
Max value element : 700

Minimum tuple

Python tuple method min() returns the elements from the tuple with minimum value.

Syntax

```
min(tuple)
```

Parameters
tuple − This is a tuple from which min valued element to be returned.

Return Value
This method returns the elements from the tuple with minimum value.

Example

```
tuple1 = (123, 'xyz', 'zara', 'abc')
tuple2 = (456, 700, 200)
print "min value element : ", min(tuple1)
print "min value element : ", min(tuple2)
```

OUTPUT
min value element : 123
min value element : 200

Sequence of tuple

Python tuple method tuple() converts a list of items into tuples

Syntax

```
tuple( seq )
```

Parameters

seq – This is a sequence to be converted into tuple.

Return Value
This method returns the tuple.

Example

```
aList = [123, 'xyz', 'zara', 'abc']
aTuple = tuple(aList)
print ("Tuple elements : ", aTuple)
```

OUTPUT
('Tuple elements : ', (123, 'xyz', 'zara', 'abc'))

Basic Tuples Operations

Tuples respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new tuple, not a string.

In fact, tuples respond to all of the general sequence operations we used on strings in the prior chapter –

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

Length of Tuple

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

Example

Print the number of items in the tuple:

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

print(len(thistuple))
```

OUTPUT

3

Concatenation Two Tuples

To join two or more tuples you can use the + operator:

Example

Join two tuples:

```
tuple1 = ("a", "b" , "c")

tuple2 = (1, 2, 3)

tuple3 = tuple1 + tuple2

print(tuple3)
```

OUTPUT

('a', 'b', 'c', 1, 2, 3)

Repetition of Tuple

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

Example

Multiply the fruits tuple by 2:

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

mytuple = fruits * 2

print(mytuple)
```

OUTPUT

('apple', 'banana', 'cherry', 'apple', 'banana', 'cherry')

Iteration of Tuple

```
for x in (1, 2, 3):
  print (x)
```

OUTPUT

1

2

3

Indexing, Slicing, and Matrixes

Because tuples are sequences, indexing and slicing work the same way for tuples as they do for strings. Assuming following input –

L = ('spam', 'Spam', 'SPAM!')

Python Expression	Results	Description
L[2]	'SPAM!'	Offsets start at zero
L[-2]	'Spam'	Negative: count from the right
L[1:]	['Spam', 'SPAM!']	Slicing fetches sections

File

A file is some information or data which stays in the computer storage devices. Python gives you easy ways to manipulate these files. Generally we divide files in two categories, text file and binary file. Text files are simple text where as the binary files contain binary data which is only readable by computer.

Sometimes, it is not enough to only display the data on the console. The data to be displayed may be very large, and only a limited amount of data can be displayed on the console since the memory is volatile, it is impossible to recover the programmatically generated data again and again.

The file handling plays an important role when the data needs to be stored permanently into the file. A file is a named location on disk to store related information. We can access the stored information (non-volatile) after the program termination.

The file-handling implementation is slightly lengthy or complicated in the other programming language, but it is easier and shorter in Python.

File handling is an important part of any web application.

Python has several functions for creating, reading, updating, and deleting files.

The key function for working with files in Python is the open() function.

The open() function takes two parameters; filename, and mode.

1. Create
2. Reading
3. Writing
4. Deleting

There are four different methods (modes) for opening a file:

Creating a File

To open a file we use open() function. It requires two arguments, first the file path or file name, second which mode it should open.

Modes are like

“r” -> open read only, you can read the file but cannot edit / delete anything inside
“w” -> open with write power, means if the file exists then delete all content and open it to write
“a” -> open in append mode

In addition you can specify if the file should be handled as binary or text mode

"t" - Text - Default value. Text mode

"b" - Binary - Binary mode (e.g. images)

The default mode is read only, i.e. if you do not provide any mode it will open the file as read only. Let us open a file

```
myfile = open("bca.txt")
```

```
myfile = open("bca.txt", “r”)
```

Because "r" for read, and "t" for text are the default values, you do not need to specify them.

Note: Make sure the file exists, or else you will get an error.

Closing a file
After opening a file one should always close the opened file. We use method close() for this.

myfile.close()

Always make sure you explicitly close each open file, once its job is done and you have no reason to keep it open. Because

- There is an upper limit to the number of files a program can open. If you exceed that limit, there is no reliable way of recovery, so the program could crash.
- Each open file consumes some main-memory for the data-structures associated with it, like file descriptor/handle or file locks etc. So you could essentially end-up wasting lots of memory if you have more files open that are not useful or usable.
- Open files always stand a chance of corruption and data loss.

Reading a File

bca.txt

Hello! Welcome to bca.txt
This file is for testing purposes.
Good Luck!

Readfile.py

```
f = open("bca.txt", "r")
print(f.read())
```

OUTPUT
Hello! Welcome to bca.txt
This file is for testing purposes.
Good Luck!

Read Only Parts of the File

By default the read() method returns the whole text, but you can also specify how many characters you want to return:

Example
Return the 5 first characters of the file:

```
f = open("bca.txt", "r")
print(f.read(5))
```

OUTPUT
Hello

Read Lines

You can return one line by using the readline() method:

Example
Read one line of the file:

```
f = open("bca.txt", "r")
print(f.readline())
```

OUTPUT
Hello! Welcome to bca.txt

Example

```
f = open("bca.txt", "r")
print(f.readline())
print(f.readline())
```

OUTPUT
Hello! Welcome to demofile.txt

This file is for testing purposes.

Example

```
f = open("bca.txt", "r")
for x in f:
  print(x)
```

OUTPUT
Hello! Welcome to demofile.txt

This file is for testing purposes.

Good Luck!

Close Files

It is a good practice to always close the file when you are done with it.

Example
Close the file when you are finish with it:

```
f = open("bca.txt", "r")
print(f.readline())
f.close()
```

Write to an Existing File

To write to an existing file, you must add a parameter to the open() function:

"a" - Append - will append to the end of the file

"w" - Write - will overwrite any existing content

Example

bca.txt

```
Hello! Welcome to bca.txt
This file is for testing purposes.
Good Luck!
```

Readfile.py

```
f = open("bca.txt", "a")
f.write("Now the file has more content!")
f.close()

#open and read the file after the appending:
f = open("bca.txt", "r")
print(f.read())
```

OUTPUT

```
Hello! Welcome to bca.txt
This file is for testing purposes.
Good Luck!

Now the file has more content!
```

Example

```
f = open("bca2.txt", "w")
f.write("Welcome to CCMS COLLEGE VADU!!")
f.close()
```

OUTPUT

Output shows in this file Bca2.txt
Welcome to CCMS COLLEGE VADU!

Deleting File
To delete a file, you must import the OS module, and run its os.remove() function.

Example
Remove the file "bca123.txt":

```
import os
os.remove("bca123.txt")
print("file is deleted")
```

Example
Check if File exist:
To avoid getting an error, you might want to check if the file exists before you try to delete it:

```
import os
if os.path.exists("bca123.txt"):
  os.remove("bca123.txt")
else:
  print("The file does not exist")
```

OUTPUT
Deleted bca123.txt file check drive.

Delete Folder
To delete an entire folder, use the os.rmdir() method:

Example
Remove the folder "myfolder":

```
import os
os.rmdir("bca")
print("folder is deleted")
```

OUTPUT
folder is deleted

An object-oriented programming (OOP) is a programming language model which is organized around “objects" rather than "actions" and data rather than logic.
Before the introduction of the Object Oriented Programming paradigm, a program was viewed as a logical procedure that takes input data, processes it, and produces output. But in case of OOP a problem is viewed in terms of objects rather than procedure for doing it.

The basic concepts related to OOP are as follows:

1. Objects
2. Classes
3. Encapsulation
4. Abstraction
5. Data Hiding
6. Polymorphism
7. Inheritance

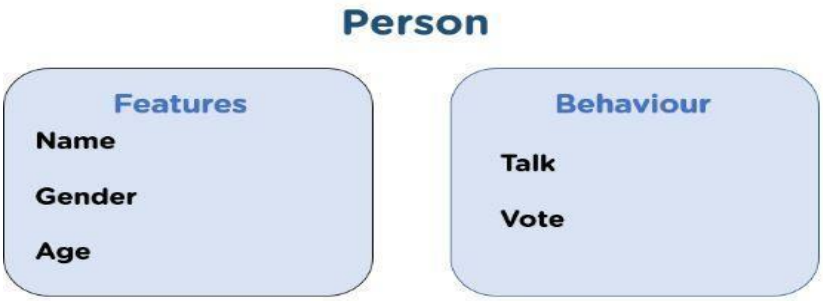
Classes

Python is an “object-oriented programming language.” This means that almost all the code is implemented using a special construct called classes. Programmers use classes to keep related things together. This is done using the keyword “class,” which is a grouping of object-oriented constructs.

What is a class?
A class is a code template for creating objects. Objects have member variables and have behavior associated with them. In python a class is created by the keyword class.

The class can be defined as a collection of objects that define the common attributes and behaviors of all the objects. To sum up, it can be called a blueprint of similar objects.

For better understanding, let‟s go through an example. In this case, we are considering a Person as a class. Now, if a person is a class, every person has certain features, such as name, gender, and age. Every person has certain behaviors, they can talk, walk, run, or vote.



Class is defined under a “class” keyword.

Example:

Python is an object oriented programming language. Almost everything in Python is an object, with its properties and methods. A Class is like an object constructor, or a "blueprint" for creating objects.

An object is created using the constructor of the class. This object will then be called the instance of the class. In Python we create instances in the following manner

Instance = class(arguments)

A class is group of objects with same attributes and common behaviours. It is basically a blueprint to create objects. An object is a basic key concept of OOP but classes provide an ability to generalize similar type of objects. Both data and functions operating on the data are bundled as a unit in a class for the same category of objects. Here to explain the term 'same category of object', let us take the example of mobile phone. A Windows phone, Android phone and i-phone, all fall into the category of mobile phones. All of these are instances of a class, say Mobile_phone and are called objects. Similarly we can have another example where students named Rani and Ravish are objects. They have properties like name, date of birth, address, class, marks etc. and the behaviour can be giving examinations. Anybody pursuing any course, giving any type of examination will come into the category of students. So a student is said to be a class as they share common properties and behaviours. Although a student can be a school student, a college student or a university student or a student pursuing a music course and so on, yet all of these have some properties and behaviours in common which will form a class. An analogy is that you can have variables of type int which translates to saying that variables that store integers are variables which are instances (objects) of the int class. A real instance of a class is called an object and creating the new

object is called instantiation. Objects can also have functionality by using functions that belong to a class. Such functions are called methods of the class. This terminology is important because it helps us to differentiate between functions and variables which are independent and those which belong to a class or object. Collectively, the fields and methods can be referred to as the attributes of that class. Let us take the example of the class Mobile_phone which is represented in the block diagram below:

A class is defined before the creation of objects of its type. The objects are then created as instances of this class as shown in the figure below.
Fig 1: Class Mobile_phone

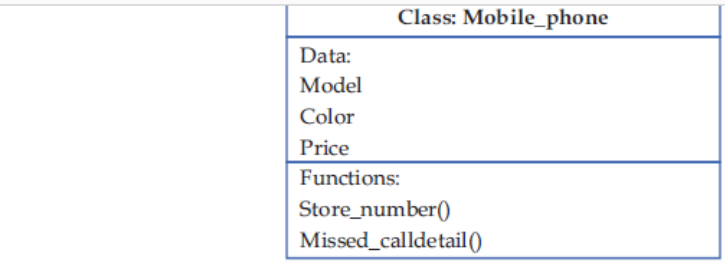


Fig 1: Class Mobile_phone

A class is defined before the creation of objects of its type. The objects are then created as instances of this class as shown in the figure below.

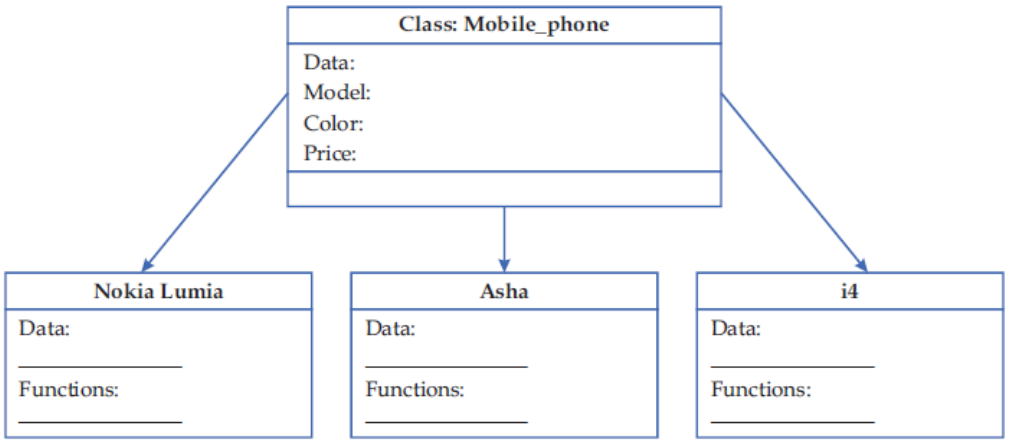


Fig 2: Class and Objects

In the above example, Nokia Lumia, Asha and i4 are all instances of the class Mobile_phone. All these instances are similar in the sense that all have basic features that a mobile phone should have. So all of these are objects of the class Mobile_phone

What is an object?

An object can be anything that we notice around us. It can be a person (described by name, address, date of Birth etc, his typing speed), a cup (described by size, color, price etc.) , a car (described by model , color , engine etc., its mileage, speed) and so on. In fact it can be an identifiable entity. The whole idea behind an object oriented model is to make programming closer to the real world thereby making it a very natural way of programming. The core of pure object-oriented programming is to combine into a single unit both
data and functions or methods that operate on that data.
Simula was the first object-oriented programming language. Java, Python, C++, Visual Basic, .NET and

An object is simply a collection of data (variables) and methods (functions) that act on those data.
Similarly, a class is a blueprint for that object. We can think of a class as a sketch (prototype) of a house.

Every object is characterized by:

Identity: This is the name that identifies an object. For example a Student is the name given to anybody who is pursuing a course. Or an i-phone is a mobile phone that has been launched by Apple Inc.
Properties: These are the features or attributes of the object. For example a student will have his name age, class, date of birth etc. as his attributes or properties. A mobile phone has model, color, price as its properties.
Behaviors: The behavior of an object signifies what all functions an object can perform. For example a student can pass or fail the examination. A mobile phone can click and store photographs (behave like a camera).

So an object clearly defines an entity in terms of its properties and behavior. Consider an example of an object - Windows mobile phone. This phone has certain properties and certain functions which are different from any other mobile phone- say an Android phone. Both are mobile phones and so possess common features that every mobile phone should have but yet they have their own properties and

behaviors. The data of a particular object can be accessed by functions associated with that object only. The functions of one object cannot access the data of another object.

Create a Class
To create a class, use the keyword class:

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

```
class MyClass:
  x = 5
print(MyClass)
```

OUTPUT
__main__.MyClass

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

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

```
class MyClass:
  x = 5

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

OUTPUT
5

The __init__() Function

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

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

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

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

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

p1 = Person("John", 36)

print(p1.name)
print(p1.age)
```

OUTPUT
John
36

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

Object Methods

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

Let us create a method in the Person class:

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

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

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

p1 = Person("Shailesh Patel
Modi", 36)p1.myfunc()
```

OUTPUT
Hello my name is Shailesh Patel Modi

Self Parameter

The self parameter is a reference to the current instance of the class, and is used to access a variable that belongs to the class.

It does not have to be named self, you can call it whatever you like, but it has to be the first parameter of any function in the class:

Example
Use the words myobject and abc instead of self:

```
class Person:
  def __init_(myobject, name, age):
    myobject.name = name
    myobject.age = age

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

p1= Person("Shailesh Patel Modi",
36)p1.myfunc()
```

OUTPUT
Hello my name is Shailesh Patel Modi

Modify Object Properties

You can modify properties on objects like this:

Example
Set the age of p1 to 40: p1.age = 40

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

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

p1 = Person("John", 36)

p1.age = 40

print(p1.age)
```

OUTPUT
40

Delete Object Properties

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

Example
Delete the age property from the p1 object:

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

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

p1 = Person("John", 36)

del p1.age

print(p1.age)
```

OUTPUT

```
Traceback (most recent call last):
  File "C:/Python27/object.py", line 13, in <module>
    print(p1.age)
AttributeError: Person instance has no attribute 'age'
```

Delete Objects

an delete objects by using the del keyword:

Example

Delete the p1 object:

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

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

p1 = Person("John", 36)

del p1

print(p1)
```

OUTPUT

```
Traceback (most recent call last):
  File "C:/Python27/object.py", line 13, in <module>
    print(p1)
NameError: name 'p1' is not defined
```

Inheritance

Inheritance allows us to define a class that inherits all the methods and properties from another class.

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

Parent class is the class being inherited from, also called base class.

Child class is the class that inherits from another class, also called derived class.

Inheritance is a concept in object oriented programming where existing classes can be modified by a new class. The existing class is called the base class and the new class is called the derived class.

Python Inheritance. Inheritance is an important aspect of the object-oriented paradigm. Inheritance provides code reusability to the program because we can use an existing class to create a new class instead of creating it from scratch.

Benefits of Inheritance.

- Code reusability- we do not have to write the same code again and again, we can just inherit the properties we need in a child class.
- It represents a real world relationship between parent class and child class.
- It is transitive in nature. If a child class inherits properties from a parent class, then all other sub-classes of the child class will also inherit the properties of the parent class.

The syntax of class inheritance is as follows:

```
class BaseClass:
   #body of BaseClass
class DerivedClass(BaseClass):
   #body of DerivedClass
```

Below is a simple example of inheritance.

```
class Parent():
      def              first(self):
         print('first function')

class Child(Parent):
      def second(self):
        print('second function')
```

```
ob = Child()
ob.first()
ob.second()
```

OUTPUT
first function
second function

In the above program, you can access the parent class function using the child class object.

Create a Parent Class
Any class can be a parent class, so the syntax is the same as creating any other class:

Example

Create a class named Person, with firstname and lastname properties, and a printname method:

```
class Person:
    def __init_(self, fname, lname):
        self.firstname = fname
      self.lastname = lname

  def printname(self):
      print(self.firstname, self.lastname)

#Use the Person class to create an object, and then execute the printname method:

x = Person("John", "Doe")
x.printname()
```

OUTPUT
('John', 'Doe')

Create a Child Class

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

Example
Create a class named Student, which will inherit the properties and methods from the Person class:

```
class Student(Person):
  pass
```

Note: Use the pass keyword when you do not want to add any other properties or methods to the class.

Now the Student class has the same properties and methods as the Person class.

Example
Use the Student class to create an object, and then execute the printname method:

```
class Person:
  def __init_(self, fname, lname):
      self.firstname = fname
      self.lastname = lname

  def printname(self):
      print(self.firstname, self.lastname)

class Student(Person):
  pass

x = Student("Mike", "Olsen")
x.printname()
```

OUTPUT
('Mike', 'Olsen')

Add the __init__() Function

So far we have created a child class that inherits the properties and methods from its parent.

We want to add the __init_() function to the child class (instead of the pass keyword).

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

Example
Add the __init__() function to the Student class:

```
class Student(Person):
  def __init__(self, fname, lname):
    #add properties etc.
```

When you add the __init_() function, the child class will no longer inherit the parent's __init_() function.

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

To keep the inheritance of the parent's __init__() function, add a call to the parent's __init__() function:

Example

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

class Student(Person):
  def __init_(self, fname, lname):
     Person._init_(self, fname, lname)

x = Student("Mike", "Olsen")
x.printname()
```

OUTPUT
('Mike', 'Olsen')

Now we have successfully added the __init_() function, and kept the inheritance of the parent class, and we are ready to add functionality in the __init_() function.

Use the super() Function

Python also has a super() function that will make the child class inherit all the methods and properties from its parent:

Example

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

class Student(Person):
  def __init_(self, fname, lname):
    super()._init_(fname, lname)
```

```
x = Student("Mike", "Olsen")
x.printname()
```

OUTPUT
Mike Olsen

By using the super() function, you do not have to use the name of the parent element, it will automatically inherit the methods and properties from its parent.

Add Properties
Example
Add a property called graduationyear to the Student class:

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

class Student(Person):
  def __init__(self, fname, lname):
    super()._init_(fname, lname)
    self.graduationyear = 2019

x = Student("Mike", "Olsen")
print(x.graduationyear)
```

OUTPUT
2019

In the example below, the year 2019 should be a variable, and passed into the Student class when creating student objects. To do so, add another parameter in the __init__() function:

Example
Add a year parameter, and pass the correct year when creating objects:

```
class Person:
  def __init_(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

class Student(Person):
```

```
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

x = Student("Mike", "Olsen", 2019)
print(x.graduationyear)
```

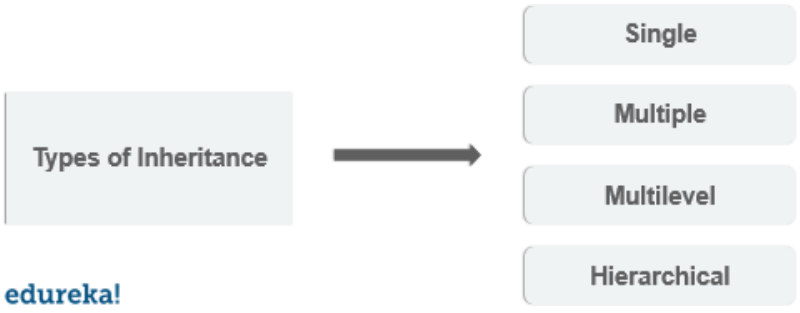
OUTPUT
2019
Add Methods

Example
Add a method called welcome to the Student class:

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

class Student(Person):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

 def welcome(self):
  print("Welcome", self.firstname, self.lastname, "to the class of", self.graduationyear)

x = Student("Mike", "Olsen", 2019)
x.welcome()
```

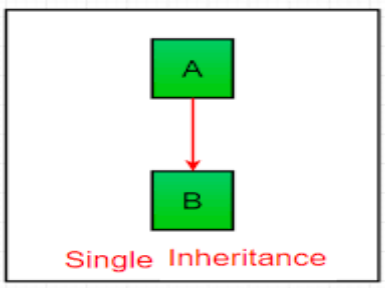
OUTPUT
Welcome Mike Olsen to the class of 2019

Types of Inheritance:



Single Inheritance

When a child class inherits only a single parent class.



In python single inheritance, a derived class is derived only from a single parent class and allows the class to derive behavior and properties from a single base class. This enables code reusability of a parent class, and adding new features to a class makes code more readable, elegant and less redundant.

And thus, single inheritance is much safer than multiple inheritances if it‟s done in the right way and enables derived class to call parent class method and also to override parent classes‟ existing methods.

```
class Parent:
    def func1(self):
        print("this is parent function ")
class Child(Parent):
    def func2(self):
        print("this is child function ")
ob = Child()
ob.func1()
ob.func2()
```

OUTPUT
this is parent function
this is child function

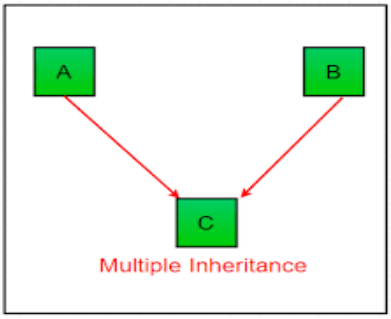
Multiple Inheritance

Multiple inheritance is a feature of some object-oriented computer programming languages in which an object or class can inherit characteristics and features from more than one parent object or the parent class .

When a child class inherits from more than one parent class.

It allows a class to inherit the functionality of more than one base class; thus allowing for modeling of complex relationships.

You categorize classes in many different ways. Multiple inheritance is a way of showing our natural tendency to organize the world. During analysis, for example, we use multiple inheritance to capture the way users classify objects.



By having multiple superclasses, your subclass has more opportunities to reuse the inherited attributes and operations of the superclasses.

Application development time is less and application takes less memory.

Syntax:

Class Base1: Body of the class

Class Base2: Body of the class

Class Derived(Base1, Base2): Body of the class

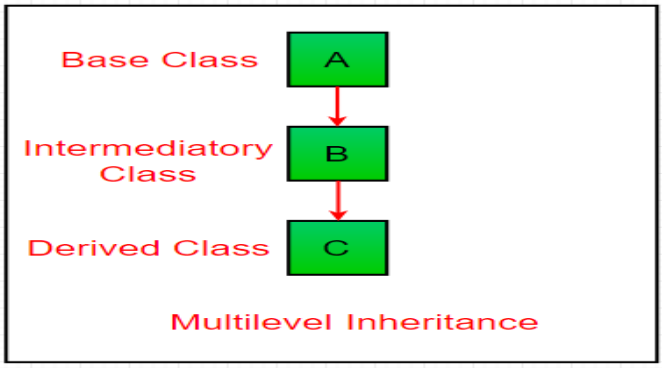
Example:

```
class Parent:
   def func1(self):
        print("this is function 1")
class Parent2:
   def func2(self):
        print("this is function 2")
class Child(Parent , Parent2):
    def func3(self):
        print("this is function 3")

ob = Child()
ob.func1()
ob.func2()
ob.func3()
```

OUTPUT
this is function 1
this is function 2
this is function 3

Multilevel Inheritance

When a child class becomes a parent class for another child class.



Multi-Level inheritance is possible in python like other object-oriented languages. Multi-level inheritance is archived when a derived class inherits another derived class. There is no limit on the number of levels up to which, the multi-levelinheritance is archived in python.

Multilevel inheritance is one type of inheritance being used to inherit both base class and derived class features to the newly derived class when we inherit a derived class from a base class and another derived class from the previous derived class up to any extent of depth of classes in python is called multilevel inheritance. While inheriting the methods and members of both base class and derived class will be inherited to the newly derived class. By using inheritance, we can achieve the reusability of code, relationships between the classes. If we have three classes A, B, and C where A is the superclass (Parent), B is the subclass (child), and C is the subclass of B (Grandchild).

Syntax:

```
class class1:
   <class-suite>
class class2(class1):
   <class suite>
class class3(class2):
   <class suite>
```

Example:

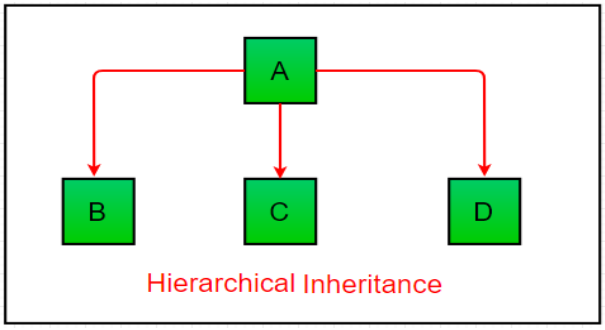
```
class Parent:
     def func1(self):
        print("this is function 1")
class Child(Parent):
     def func2(self):
        print("this is function 2")
class Child2(Child):
```

```
    def func3(self):
        print("this is function 3")
ob = Child2()
ob.func1()
ob.func2()
ob.func3()
```

OUTPUT
this is function 1
this is function 2
this is function 3

Hierarchical Inheritance

Hierarchical inheritance involves multiple inheritance from the same base or parent class.



When more than one derived classes are created from a single base this type of inheritance is called hierarchical inheritance. In this program, we have a parent (base) class and two child (derived) classes.

Example:

```
class Parent:
    def func1(self):
        print("this is function 1")
class Child(Parent):
    def func2(self):
        print("this is function 2")
class Child2(Parent):
    def func3(self):
        print("this is function 3")

ob = Child()
ob1 = Child2()
ob.func1()
ob.func2()
```

OUTPUT
this is function 1
this is function 2

Hybrid Inheritance

Hybrid inheritance involves multiple inheritance taking place in a single program.

```
class Parent:
    def func1(self):
        print("this is function one")

class Child(Parent):
    def func2(self):
        print("this is function 2")

class Child1(Parent):
    def func3(self):
        print(" this is function 3")

class Child3(Parent , Child1):
    def func4(self):
        print(" this is function 4")

ob = Child3()
ob.func1()
```

OUTPUT
this is function one

Encapsulation

Encapsulation is one of the four fundamental concepts in object-oriented programming including abstraction, encapsulation, inheritance, and polymorphism.

Encapsulation in Python describes the concept of bundling data and methods within a single unit. So, for example, when you create a class, it means you are implementing encapsulation. A class is an example of encapsulation as it binds all the data members (instance variables) and methods into a single unit.

Encapsulation is the packing of data and functions that work on that data within a single object.

By doing so, you can hide the internal state of the object from the outside. This is known as information hiding.

A class is an example of encapsulation. A class bundles data and methods into a single unit. And a class provides the access to its attributes via methods.

The idea of information hiding is that if you have an attribute that isn‟t visible to the outside, you can control the access to its value to make sure your object is always has a valid state.

Encapsulation is the most basic concept of OOP. It is the combining of data and the functions associated with that data in a single unit. In most of the languages including python, this unit is called a class.

In Fig -1 showing class Mobile_phone, given under the subtopic Classes, we see that the name of the class, its properties or attributes and behaviours are all enclosed under one independent unit. This is encapsulation, implemented through the unit named class. In simple terms we can say that encapsulation is implemented through classes. In fact the data members of a class can be accessed through its member functions only. It keeps the data safe from any external interference and misuse. The only way to access the data is through the functions of the class. In the example of the class Mobile_phone, the class encapsulates the data (model, color, price) and the associated functions into a single independent unit.

Advantages of Encapsulation

- **Security**: The main advantage of using encapsulation is the security of the data. Encapsulation protects an object from unauthorized access. It allows private and protected access levels to prevent accidental data modification.
- **Data Hiding**: The user would not be knowing what is going on behind the scene. They would only be knowing that to modify a data member, call the setter method. To read a data member, call the getter method. What these setter and getter methods are doing is hidden from them.
- **Simplicity**: It simplifies the maintenance of the application by keeping classes separated and preventing them from tightly coupling with each other.
- **Aesthetics**: Bundling data and methods within a class makes code more readable and maintainable.

Polymorphism

The word polymorphism means having many forms. In programming, polymorphism means the same function name (but different signatures) being used for different types.
Polymorphism in python defines methods in the child class that have the same name as the methods in the parent class. In inheritance, the child class inherits the methods from the parent class. Also, it is possible to modify a method in a child class that it has inherited from the parent class.

The word Polymorphism is formed from two words - poly and morph where poly means many and morph means forms. So polymorphism is the ability to use an operator or function in various forms. That is a single function or an operator behaves differently depending upon the data provided to them.

Polymorphism can be achieved in two ways:

1. Operator Overloading

you have worked with '+ 'operator. You must have noticed that the '+' operator behaves differently with different data types. With integers it adds the two numbers and with strings it concatenates or joins two strings.

For example:

Print 8+9 will give 17 and
Print "Python" + "programming" will give the output as Pythonprogramming.
This feature where an operator can be used in different forms is known as Operator Overloading and is one of the methods to implement polymorphism.

2. Function Overloading

Polymorphism in case of functions is a bit different. A named function can also vary depending on the parameters it is given. For example, we define multiple functions with same name but different argument list as shown below:

```
def test(): #function 1
print "hello"
def test(a, b): #function 2
return a+b
def test(a, b, c): #function 3
return a+b+c
```

In the example above, three functions by the same name have been defined but with different number of arguments. Now if we give a function call with no argument, say test(), function 1 will be called.

The statement test(10,20) will lead to the execution of function 2 and if the statement test(10,20,30) is given Function 3 will be called. In either case, all the functions would be known in the program by the same name. This is another way to implement polymorphism and is known as Function Overloading. As we see in the examples above, the function called will depend on the argument list - data types and number of arguments. These two i.e. data types and the number of arguments together form the function signature. Please note that the return types of the function are no where responsible for function overloading and that is why they are not part of function signature. Here it must be taken into consideration that Python does not support function overloading as shown above although languages like Java and C/C++ do. If you run the code of three test functions, the second test() definition will overwrite the first one. Subsequently the third test() definition will overwrite the second one. That means if you give the function call test(20,20) , it will flash an error stating, "Type Error: add() takes exactly 3 arguments (2 given)". This is because, Python understands the latest definition of the function test() which takes three arguments.

Examples of in-built polymorphic functions :
Example:

```
print(len("geeks"))         # len() being used for a string
print(len([10, 20, 30]))    # len() being used for a list
```

OUTPUT
5
3

Examples of user-defined polymorphic functions :

```
def add(x, y, z = 0):
    return x + y+z


print(add(2, 3))
print(add(2, 3, 4))
```

OUTPUT

5

9

Unit 4: **[17 MARKS]**

Exception Handling, Standard Library and Python Database connectivity

Exceptions: Exception handling, Types of exceptions, String Formatting, File wildcards,
Command line arguments, Python RegEx,
Python and MySQL: Installing MySQL Driver, Verifying the Connector Installation, Using MySQL from Python, Retrieving all rows from a table, Inserting rows into a table, Deleting rows from table, Updating rows in a table, Creating database tables through Python.

Exceptions:

When an error occurs, or exception as we call it, Python will normally stop and generate an error message. These exceptions can be handled using the try statement.

When we plan our code/program, we always work for situations that are normally expected, and our program works very well in those situations. But, we all understand that programs have to deal with errors. Here errors are not syntax errors instead they are the unexpected condition(s) that are not part of normal operations planned during coding.

Partial list of such kinds of errors are:

- Out of Memory
- Invalid filename
- Attempting to write into read only file
- Getting an incorrect input from user
- Division by zero
- Accessing an out of bound list element
- Trying to read beyond end of file
- Sending illegal arguments to a method

If any of such situations is encountered, a good program will either have the code check for them or perform some suitable action to remedy them, or at least stop processing in a well defined way after giving appropriate message(s). So what we are saying is if an error happened, there must be code written in program, to recover from the error. In case if it is not possible to handle the error then it must be reported in user friendly way to the user. Errors are exceptional, unusual and unexpected situations and they are never part of the normal flow of a program. We need a process to identify and handle them to write a good program. Exceptions

handling is the process of responding in such situations. Most of the modern programming languages provide support with handling exceptions.

They offer a dedicated exception handling mechanism, which simplifies the way in which an exception situation is reported and handled. Before moving ahead with exception handling, let's understand, some of the terms associated with it - when an exception occurs in the program, we say that exception was raised or thrown. Next, we deal with 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.

```
marks = 10000

# perform division with 0
a = marks / 0
print(a)
```

In the above example raised the ZeroDivisionError as we are trying to divide a number by 0.

Try and Except Statement – Catching Exceptions

Try and except statements are used to catch and handle exceptions in Python. Statements that can raise exceptions are kept inside the try clause and the statements that handle the exception are written inside except clause.

Catching Specific Exception

A try statement can have more than one except clause, to specify handlers for different exceptions. Please note that at most one handler will be executed. For example, we can add IndexError in the above code.

Syntax for adding specific exceptions are

```
try:
    # statement(s)
except IndexError:
   # statement(s)
except ValueError:
   # statement(s)
```

Example

The try block will generate an exception, because x is not defined:

```
try:
  print(x)
except:
  print("An exception occurred")
```

OUTPUT
An exception occurred

Since the try block raises an error, the except block will be executed.

Without the try block, the program will crash and raise an error.

Many Exceptions

You can define as many exception blocks as you want, e.g. if you want to execute a special block of code for a special kind of error:

Example

Print one message if the try block raises a NameError and another for other errors:

```
try:

  print(x)

except NameError:

  print("Variable x is not defined")

except:

  print("Something else went wrong")
```

OUTPUT

Variable x is not defined

Else
You can use the else keyword to define a block of code to be executed if no errors were raised:

Example
In this example, the try block does not generate any error:

```
try:
  print("Hello")
except:
```

```
  print("Something went wrong")
else:
  print("Nothing went wrong")
```

OUTPUT
Hello
Nothing went wrong

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.

Finally Keyword
Python provides a keyword finally, which is always executed after the try and except blocks. The final 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)
```

The finally block, if specified, will be executed regardless if the try block raises an error or not.

Example

```
try:
  print(x)
except:
  print("Something went wrong")
finally:
  print("The 'try except' is finished")
```

OUTPUT
Something went wrong
The 'try except' is finished

Raising Exception

The raise statement allows the programmer to force a specific exception to occur. The sole argument in raise indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from Exception).

Built-in Exceptions

The table below shows built-in exceptions that are usually raised in Python:

Exception	Description
ArithmeticError	Raised when an error occurs in numeric calculations

AssertionError	Raised when an assert statement fails
AttributeError	Raised when attribute reference or assignment fails
Exception	Base class for all exceptions
EOFError	Raised when the input() method hits an "end of file" condition (EOF)
FloatingPointError	Raised when a floating point calculation fails
GeneratorExit	Raised when a generator is closed (with the close() method)
ImportError	Raised when an imported module does not exist
IndentationError	Raised when indendation is not correct
IndexError	Raised when an index of a sequence does not exist
KeyError	Raised when a key does not exist in a dictionary
KeyboardInterrupt	Raised when the user presses Ctrl+c, Ctrl+z or Delete
LookupError	Raised when errors raised cant be found
MemoryError	Raised when a program runs out of memory
NameError	Raised when a variable does not exist
NotImplementedError	Raised when an abstract method requires an inherited class to override the method
OSError	Raised when a system related operation causes an error

OverflowError	Raised when the result of a numeric calculation is too large
ReferenceError	Raised when a weak reference object does not exist
RuntimeError	Raised when an error occurs that do not belong to any specific expections
StopIteration	Raised when the next() method of an iterator has no further values
SyntaxError	Raised when a syntax error occurs
TabError	Raised when indentation consists of tabs or spaces
SystemError	Raised when a system error occurs
SystemExit	Raised when the sys.exit() function is called
TypeError	Raised when two different types are combined
UnboundLocalError	Raised when a local variable is referenced before assignment
UnicodeError	Raised when a unicode problem occurs
UnicodeEncodeError	Raised when a unicode encoding problem occurs
UnicodeDecodeError	Raised when a unicode decoding problem occurs

UnicodeTranslateError	Raised when a unicode translation problem occurs
ValueError	Raised when there is a wrong value in a specified data type
ZeroDivisionError	Raised when the second operator in a division is zero

String formatting

Definition and Usage

String formatting is also known as String interpolation. It is the process of inserting a custom string or variable in predefined text. custom_string = "String formatting" print(f"{custom_string} is a powerful technique") String formatting is a powerful technique.

The format() method formats the specified value(s) and insert them inside the string's placeholder.

The placeholder is defined using curly brackets: {}. Read more about the placeholders in the Placeholder section below.

The format() method returns the formatted string.

Syntax

string.format(value1, value2...)

Parameter Values

Parameter	Description

value1, value2...	Required. One or more values that should be formatted and inserted in the string. The values are either a list of values separated by commas, a key=value list, or a combination of both. The values can be of any data type.

The Placeholders

The placeholders can be identified using named indexes {price}, numbered indexes {0}, or even empty placeholders {}.

Example

```
#named indexes:

txt1 = "My name is {fname}, I'm {age}".format(fname = "John", age = 36)

#numbered indexes:

txt2 = "My name is {0}, I'm {1}".format("John",36)

#empty placeholders:

txt3 = "My name is {}, I'm {}".format("John",36)

print(txt1)

print(txt2)

print(txt3)
```

OUTPUT

```
My name is John, I'm 36
My name is John, I'm 36
My name is John, I'm 36
```

Formatting Types

Inside the placeholders you can add a formatting type to format the result:

:<	Left aligns the result (within the available space)
:>	Right aligns the result (within the available space)
:^	Center aligns the result (within the available space)
:=	Places the sign to the left most position
:+	Use a plus sign to indicate if the result is positive or negative
:-	Use a minus sign for negative values only
:	Use a space to insert an extra space before positive numbers (and a minus sign before negative numbers)
:,	Use a comma as a thousand separator
:_	Use a underscore as a thousand separator
:b	Binary format
:c	Converts the value into the corresponding unicode character

:d	Decimal format
:e	Scientific format, with a lower case e
:E	Scientific format, with an upper case E
:f	Fix point number format
:F	Fix point number format, in uppercase format (show inf and nan as INF and NAN)
:g	General format
:G	General format (using a upper case E for scientific notations)
:o	Octal format
:x	Hex format, lower case
:X	Hex format, upper case
:n	Number format

```
:% Percentage format
```

Wildcards

An asterisk (*) matches zero or more characters in a segment of a name. For example, dir/*.

```
import glob
for name in glob.glob('dir/*'):
    print name
```

The pattern matches every pathname (file or directory) in the directory dir, without recursing further into subdirectories.

```
$ python glob_asterisk.py

dir/file.txt
dir/file1.txt
dir/file2.txt
dir/filea.txt
dir/fileb.txt
dir/subdir
```

To list files in a subdirectory, you must include the subdirectory in the pattern:

```
import glob

print 'Named explicitly:'
for name in glob.glob('dir/subdir/*'):
    print '\t', name

print 'Named with  wildcard:'
for name in glob.glob('dir/*/*'):
    print '\t', name
```

The first case above lists the subdirectory name explicitly, while the second case depends on a wildcard to find the directory.

```
$ python glob_subdir.py

Named explicitly:
```

```
    dir/subdir/subfile.txt
Named with wildcard:
    dir/subdir/subfile.txt
```

The results, in this case, are the same. If there was another subdirectory, the wildcard would match both subdirectories and include the filenames from both.

Single Character Wildcard

The other wildcard character supported is the question mark (?). It matches any single character in that position in the name. For example,

```
import glob

for name in glob.glob('dir/file?.txt'):
    print name
```

Matches all of the filenames which begin with “file”, have one more character of any type, then end with ”.txt”.

```
$ python glob_question.py

dir/file1.txt
dir/file2.txt
dir/filea.txt
dir/fileb.txt
```

Character Ranges

When you need to match a specific character, use a character range instead of a question mark. For example, to find all of the files which have a digit in the name before the extension:

```
import glob
for name in glob.glob('dir/*[0-9].*'):
    print name
```

The character range [0-9] matches any single digit. The range is ordered based on the character code for each letter/digit, and the dash indicates an unbroken range of sequential characters. The same range value could be written [0123456789].

```
$ python glob_charrange.py

dir/file1.txt
dir/file2.txt
```

Command-line arguments.

Python provides a getopt module that helps you parse command-line options and arguments.

$ python test.py arg1 arg2 arg3

The Python sys module provides access to any command-line arguments via the sys.argv. This serves two purposes –

sys.argv is the list of command-line arguments.

len(sys.argv) is the number of command-line arguments.

Here sys.argv[0] is the program ie. script name.

Example

Consider the following script test.py –

```
#!/usr/bin/python

import sys

print 'Number of arguments:', len(sys.argv), 'arguments.'

print 'Argument List:', str(sys.argv)
```

Now run above script as follows –

$ python test.py arg1 arg2 arg3

OUTPUT

```
Number of arguments: 4 arguments.

Argument List: ['test.py', 'arg1', 'arg2', 'arg3']
```

NOTE – As mentioned above, first argument is always script name and it is also being counted in number of arguments.

Arbitrary Arguments, *args

If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition.

This way the function will receive a tuple of arguments, and can access the items accordingly:

Example

If the number of arguments is unknown, add a * before the parameter name:

```
def my_function(*kids):
  print("The youngest child is " + kids[2])

my_function("Emil", "Tobias", "Linus")
```

OUTPUT

The youngest child is Linus

Python RegEx

A regular expression is a special sequence of characters that helps you match or find other strings or sets of strings, using a specialized syntax held in a pattern.

A RegEx, or Regular Expression, is a sequence of characters that forms a search pattern.

RegEx can be used to check if a string contains the specified search pattern.

RegEx Module

Python has a built-in package called re, which can be used to work with Regular Expressions.

Import the re module:

```
import re
```

RegEx in Python

When you have imported the re module, you can start using regular expressions:

Example

Search the string to see if it starts with "The" and ends with "Spain":

```
import re
```

```
#Check if the string starts with "The" and ends with "Spain":

txt = "The rain in Spain"

x = re.search("^The.*Spain$", txt)

if x:

  print("YES! We have a match!")

else:

  print("No match")
```

OUTPUT

YES! We have a match!

RegEx Functions

The re module offers a set of functions that allows us to search a string for a match:

Function	Description
findall	Returns a list containing all matches
search	Returns a Match object if there is a match anywhere in the string
split	Returns a list where the string has been split at each match
sub	Replaces one or many matches with a string

Metacharacters

Metacharacters are characters with a special meaning:

Character	Description	Example
[]	A set of characters	"[a-m]"
\	Signals a special sequence (can also be used to escape special characters)	"\d"
.	Any character (except newline character)	"he..o"
^	Starts with	"^hello"
$	Ends with	"planet$"
*	Zero or more occurrences	"he.*o"
+	One or more occurrences	"he.+o"
?	Zero or one occurrences	"he.?o"
{}	Exactly the specified number of occurrences	"he.{2}o"

|	Either or	"falls|stays"
()	Capture and group	

Special Sequences

A special sequence is a \ followed by one of the characters in the list below, and has a special meaning:

Character	Description	Example
\A	Returns a match if the specified characters are at the beginning of the string	"\AThe"
\b	Returns a match where the specified characters are at the beginning or at the end of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string")	r"\bain" r"ain\b"
\B	Returns a match where the specified characters are present, but NOT at the beginning (or at the end) of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string")	r"\Bain" r"ain\B"
\d	Returns a match where the string contains digits (numbers from 0-9)	"\d"

\D	Returns a match where the string DOES NOT contain digits	"\D"
\s	Returns a match where the string contains a white space character	"\s"
\S	Returns a match where the string DOES NOT contain a white space character	"\S"
\w	Returns a match where the string contains any word characters (characters from a to Z, digits from 0-9, and the underscore _ character)	"\w"
\W	Returns a match where the string DOES NOT contain any word characters	"\W"
\Z	Returns a match if the specified characters are at the end of the string	"Spain\Z"

Sets

A set is a set of characters inside a pair of square brackets [] with a special meaning:

Set	Description
[arn]	Returns a match where one of the specified characters (a, r, or n) are present

[a-n]	Returns a match for any lower case character, alphabetically between a and n
[^arn]	Returns a match for any character EXCEPT a, r, and n
[0123]	Returns a match where any of the specified digits (0, 1, 2, or 3) are present
[0-9]	Returns a match for any digit between 0 and 9
[0-5][0-9]	Returns a match for any two-digit numbers from 00 and 59
[a-zA-Z]	Returns a match for any character alphabetically between a and z, lower case OR upper case
[+]	In sets, +, *, ., |, (), $,{} has no special meaning, so [+] means: return a match for any + character in the string

The findall() Function

The findall() function returns a list containing all matches.

Example

Print a list of all matches:

```
import re

#Return a list containing every occurrence of "ai":

txt = "The rain in Spain"
```

```
x = re.findall("ai", txt)

print(x)
```

OUTPUT

```
['ai', 'ai']
```

The list contains the matches in the order they are found.

If no matches are found, an empty list is returned:

Example

Return an empty list if no match was found:

```
import re

txt = "The rain in Spain"

#Check if "Portugal" is in the string:

x = re.findall("Portugal", txt)

print(x)

if (x):

  print("Yes, there is at least one match!")

else:

  print("No match")
```

OUTPUT

```
[]

No match
```

The search() Function

The search() function searches the string for a match, and returns a Match object if there is a match.

If there is more than one match, only the first occurrence of the match will be returned:

Example

Search for the first white-space character in the string:

```
import re

txt = "The rain in Spain"

x = re.search("\s", txt)

print("The first white-space character is located in position:", x.start())
```

OUTPUT

The first white-space character is located in position: 3

The split() Function

The split() function returns a list where the string has been split at each match:

Example

Split at each white-space character:

```
import re

#Split the string at every white-space character:

txt = "The rain in Spain"

x = re.split("\s", txt)

print(x)
```

OUTPUT

['The', 'rain', 'in', 'Spain']

The sub() Function

The sub() function replaces the matches with the text of your choice:

Example

Replace every white-space character with the number 9:

```
import re


#Replace all white-space characters with the digit "9":

txt = "The rain in Spain"

x = re.sub("\s", "9", txt)

print(x)
```

OUTPUT

The9rain9in9Spain

Match Object

A Match Object is an object containing information about the search and the result.

Note: If there is no match, the value None will be returned, instead of the Match Object.

Example

Do a search that will return a Match Object:

```
import re

#The search() function returns a Match object:

txt = "The rain in Spain"

x = re.search("ai", txt)
```

```
print(x)
```

OUTPUT

```
<_sre.SRE_Match object; span=(5, 7), match='ai'>
```

Python and MySQL:

Installing MySQL Driver

- Python needs a MySQL driver to access the MySQL database.
- In this tutorial we will use the driver "MySQL Connector".
- We recommend that you use PIP to install "MySQL Connector".
- PIP is most likely already installed in your Python environment.
- Navigate your command line to the location of PIP, and type the following:

Download and install "MySQL Connector":

```
C:\Users\Your Name\AppData\Local\Programs\Python\Python36-32\Scripts>python -m pip install mysql-connector-python
```

Now you have downloaded and installed a MySQL driver.

Verifying the Connector Installation

Test MySQL Connector

To test if the installation was successful, or if you already have "MySQL Connector" installed, create a Python page with the following content:

demo_mysql_test.py:

```
import mysql.connector
```

OUTPUT

```
C:\Users\My Name>python demo_mysql_test.py
```

If the above code was executed with no errors, "MySQL Connector" is installed and ready to be used.

Create Connection

Start by creating a connection to the database.

Use the username and password from your MySQL database:

demo_mysql_connection.py:

```
import  mysql.connector

mydb = mysql.connector.connect(

  host="localhost",

  user="myusername",

  password="mypassword"

)

print(mydb)
```

OUTPUT

```
C:\Users\My Name>python demo_mysql_connection.py

<mysql.connector.connection.MySQLConnection object ar 0x016645F0>
```

Now you can start querying the database using SQL statements.

Creating a Database

To create a database in MySQL, use the "CREATE DATABASE" statement:

Example

create a database named "mydatabase":

```
import mysql.connector

mydb  =  mysql.connector.connect(

  host="localhost",
```

```
  user="myusername",
  password="mypassword"
)
mycursor  =  mydb.cursor()
mycursor.execute("CREATE DATABASE mydatabase")
#If this page is executed with no error, you have successfully created a database.
```

OUTPUT

```
C:\Users\My Name>python demo_mysql_create_db.py
```

Check if Database Exists

You can check if a database exist by listing all databases in your system by using the "SHOW DATABASES" statement:

Example

Return a list of your system's databases:

```
import mysql.connector
mydb = mysql.connector.connect(
  host="localhost",
  user="myusername",
  password="mypassword"
)
mycursor = mydb.cursor()
mycursor.execute("SHOW DATABASES")
for x in mycursor:
  print(x)
```

OUTPUT

```
C:\Users\My Name>python demo_mysql_show_databases.py

('information_scheme',)

('mydatabase',)

('performance_schema',)

('sys',)
```

Creating a Table

To create a table in MySQL, use the "CREATE TABLE" statement.

Make sure you define the name of the database when you create the connection

Example

Create a table named "customers":

```
import mysql.connector

mydb = mysql.connector.connect(

  host="localhost",

  user="myusername",

  password="mypassword",

  database="mydatabase"

)

mycursor = mydb.cursor()

mycursor.execute("CREATE TABLE customers (name VARCHAR(255), address VARCHAR(255))")
```

#If this page is executed with no error, you have successfully created a table named "customers".

OUTPUT

```
C:\Users\My Name>python demo_mysql_create_table.py
```

Example

Assume we have created a table in MySQL with name cricketers_data as −

```
CREATE TABLE cricketers_data(
   First_Name VARCHAR(255),
   Last_Name VARCHAR(255),
   Date_Of_Birth date,
   Place_Of_Birth VARCHAR(255),
   Country VARCHAR(255)
);
```

And if we have inserted 5 records in to it using INSERT statements as −

```
insert into cricketers_data values(
   'Shikhar', 'Dhawan', DATE('1981-12-05'), 'Delhi', 'India');
insert into cricketers_data values(
   'Jonathan', 'Trott', DATE('1981-04-22'), 'CapeTown', 'SouthAfrica');
insert into cricketers_data values(
   'Kumara', 'Sangakkara', DATE('1977-10-27'), 'Matale', 'Srilanka');
insert into cricketers_data values(
   'Virat', 'Kohli', DATE('1988-11-05'), 'Delhi', 'India');
insert into cricketers_data values(
   'Rohit', 'Sharma', DATE('1987-04-30'), 'Nagpur', 'India');
```

Following query retrieves the FIRST_NAME and Country values from the table.

```
mysql> select FIRST_NAME, Country from cricketers_data;

+------------+-------------+
| FIRST_NAME | Country     |
+------------+-------------+
| Shikhar    | India       |
| Jonathan   | SouthAfrica |
| Kumara     | Srilanka    |
| Virat      | India       |
| Rohit      | India       |
+------------+-------------+
5 rows in set (0.00 sec)
```

You can also retrieve all the values of each record using * instated of the name of the columns as −

```
mysql> SELECT * from cricketers_data;
```

```
+------------+-----------+---------------+---------------+-------------+
| First_Name | Last_Name | Date_Of_Birth | Place_Of_Birth | Country     |
+------------+-----------+---------------+---------------+-------------+
| Shikhar    | Dhawan    | 1981-12-05    | Delhi          | India       |
| Jonathan   | Trott     | 1981-04-22    | CapeTown       | SouthAfrica |
| Kumara     | Sangakkara | 1977-10-27    | Matale         | Srilanka    |
| Virat      | Kohli     | 1988-11-05    | Delhi          | India       |
| Rohit      | Sharma    | 1987-04-30    | Nagpur         | India       |
+------------+-----------+---------------+---------------+-------------+
5 rows in set (0.00 sec)
```

Reading data from a MYSQL table using Python

READ Operation on any database means to fetch some useful information from the database. You can fetch data from MYSQL using the fetch() method provided by the mysql-connector-python.

The cursor.MySQLCursor class provides three methods namely fetchall(), fetchmany() and, fetchone() where,

- The fetchall() method retrieves all the rows in the result set of a query and returns them as list of tuples. (If we execute this after retrieving few rows it returns the remaining ones).
- The fetchone() method fetches the next row in the result of a query and returns it as a tuple.
- The fetchmany() method is similar to the fetchone() but, it retrieves the next set of rows in the result set of a query, instead of a single row.

Note – A result set is an object that is returned when a cursor object is used to query a table.

rowcount – This is a read-only attribute and returns the number of rows that were affected by an execute() method.

Example

Following example fetches all the rows of the EMPLOYEE table using the SELECT query and from the obtained result set initially, we are retrieving the first row using the fetchone() method and then fetching the remaining rows using the fetchall() method.

```
import mysql.connector

#establishing the connection
conn = mysql.connector.connect(
   user='root', password='password', host='127.0.0.1', database='mydb')

#Creating a cursor object using the cursor() method
cursor = conn.cursor()
```

```
#Retrieving single row
sql = '''SELECT * from EMPLOYEE'''

#Executing the query
cursor.execute(sql)

#Fetching 1st row from the table
result = cursor.fetchone();
print(result)

#Fetching 1st row from the table
result = cursor.fetchall();
print(result)

#Closing the connection
conn.close()
```

Output

```
('Krishna', 'Sharma', 19, 'M', 2000.0)
[('Raj', 'Kandukuri', 20, 'M', 7000.0), ('Ramya', 'Ramapriya', 25, 'M', 5000.0)]
```

To delete records from a SQLite table, you need to use the DELETE FROM statement. To remove specific records, you need to use WHERE clause along with it.

To update specific rows, you need to use the WHERE clause along with it.

Syntax

Following is t he syntax of the DELETE query in SQLite –

```
DELETE FROM table_name [WHERE Clause]
```

```
UPDATE EMPLOYEE SET AGE = AGE + 1 WHERE SEX = 'M';
Query OK, 3 rows affected (0.06 sec)
Rows matched: 3 Changed: 3 Warnings: 0
```

If you retrieve the contents of the table, you can see the updated values as –

```
mysql> select * from EMPLOYEE;
+------------+-------------+-------+-------+---------+
| FIRST_NAME | LAST_NAME | AGE  | SEX  | INCOME |
+------------+-------------+-------+-------+---------+
| Krishna    | Sharma    | 20   | M    | 2000   |
| Raj        | Kandukuri | 21   | M    | 7000   |
| Ramya      | Ramapriya | 25   | F    | 5000   |
| Mac        | Mohan     | 27   | M    | 2000   |
+------------+-------------+-------+-------+---------+
4 rows in set (0.00 sec)
```

1. Write a Python program to display 'Hello World" Message on Screen.

```
print("Hello World")
```

OUTPUT
Hello World

2. Write a Python program to swap two variables.

```
no1 = int( input("Please enter value for number 1: "))
no2 = int( input("Please enter value for number 2: "))

temp = no1
no1 = no2
no2 = temp

print ("The Value of number 1 after swapping: ", no1)
print ("The Value of number 2 after swapping: ", no2)
```

OUTPUT
Please enter value for number 1: 10
Please enter value for number 2: 20

The Value of number 1 after swapping: 20
The Value of number 2 after swapping: 10

3. Write a Python program to display the Fibonacci series.

```
n = int(input ("How many terms the user wants to print? "))

a = 0
b = 1
count = 0
print ("The fibonacci sequence of the numbers is:")
while count < n:
    print(a)
    c = a + b
    a = b
    b = c
     count =count+ 1
```

OUTPUT

```
How many terms the user wants to print? 15
The fibonacci sequence of the numbers is:
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
```

4. Write a Python program to calculate sum of given number.

```
n=int(input("Enter a number n:"))
sum=0
while(n>0):
    rem=n%10
    sum=sum+rem
    n=n//10
print("The total sum of digits is:",sum)
```

OUTPUT

```
Enter a number n:  1234
The total sum of digits is:   10
```

5. Write a Python Program to print first prime number.

```
lower = int(input("Enter lower range: "))
upper = int(input("Enter upper range: "))

for num in range(lower,upper + 1):
   if num > 1:
       for i in range(2,num):
           if (num % i) == 0:
               break
```

```
    else:
       print(num)
```

OUTPUT

```
Enter lower range: 0
Enter upper range: 50
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
```

6. Write a Python Program to Check Armstrong Number

```
lower = int(input("Enter lower range: "))
upper = int(input("Enter upper range: "))

for num in range(lower,upper + 1):
   sum = 0
   temp = num
   while temp > 0:
      rem = temp % 10
      sum = sum + (rem ** 3 )
      temp =temp/ 10
      if num == sum:
          print(num)
```

OUTPUT

```
Enter lower range: 0
Enter upper range: 100
```

1
64

7. Write a Python Program to create a sequence of numbers using range datatype to display 1 to 30, with an increment of 2.

```
x = range(1, 30, 2)

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

OUTPUT

1
3
5
7
9
11
13
15
17
19
21
23
25
27
29

8. Write a Python Program to find area of circle.

```
PI = 3.14

radius = float(input(' Please Enter the radius of a circle: '))

area = PI * radius * radius

circumference = 2 * PI * radius

print(" Area Of a Circle = %.2f" %area)
```

```
print(" Circumference Of a Circle = %.2f" %circumference)
```

OUTPUT

```
Please Enter the radius of a circle: 3
Area Of a Circle = 28.26
Circumference Of a Circle = 18.84
```

9. Write a Python program to implement Factorial series up to user entered number.

```
lower=int(input("enter any value Lower : "))
upper=int(input("enter any value Upper : "))
fact=1

while(lower<=upper):
    fact=fact*lower
    print(" factorial is:== ",lower, fact)
    lower=lower+1
```

OUTPUT

```
enter any value lower : 1
enter any value upper : 5
('factorial is:==  ', 1, 1)
('factorial is:==  ', 2, 2)
('factorial is:==  ', 3, 6)
('factorial is:==  ', 4, 24)
('factorial is:==  ', 5, 120)
```

10. Write a Python program to check the given number is palindrome or not.

```
n=int(input("Enter number:"))
temp=n
rev=0
while(n>0):
    rem=n%10
    rev=rev*10+rem
    n=n//10
if(temp==rev):
    print("The number is a palindrome!")
else:
    print("The number isn't a palindrome!")
```

OUTPUT

```
Enter number:121
The number is a palindrome!

Enter number:123
The number isn't a palindrome!
```

11. Write a python program to display ascending and descending order from given 10 numbers.

```
NumList = []

Number = int(input("Please enter the Total Number of List Elements: "))
for i in range(1, Number + 1):
    value = int(input("Please enter the Value of %d Element : " %i))
    NumList.append(value)

for i in range (Number):
    for j in range(i + 1, Number):
        if(NumList[i] > NumList[j]):
            temp = NumList[i]
            NumList[i] = NumList[j]
            NumList[j] = temp

print("Element After Sorting List in Ascending Order is : ", NumList)

for i in range (Number):
    for j in range(i + 1, Number):
        if(NumList[i] < NumList[j]):
            temp = NumList[i]
            NumList[i] = NumList[j]
            NumList[j] = temp

print("Element After Sorting List in Descending Order is : ", NumList)
```

OUTPUT

```
Please enter the Total Number of List Elements: 10

Please enter the Value of 1 Element : 3
```

```
Please enter the Value of 2 Element : 6
Please enter the Value of 3 Element : 5
Please enter the Value of 4 Element : 2
Please enter the Value of 5 Element : 1
Please enter the Value of 6 Element : 7
Please enter the Value of 7 Element : 9
Please enter the Value of 8 Element : 8
Please enter the Value of 9 Element : 11
Please enter the Value of 10 Element : 0

Element After Sorting List in Ascending Order is :  [0, 1, 2, 3, 5, 6, 7, 8, 9, 11]
Element After Sorting List in Descending Order is :  [11, 9, 8, 7, 6, 5, 3, 2, 1, 0]
```

12. Write a Python program to print the duplicate elements of an array.

```
arr = [1, 2, 3, 4, 2, 7, 8, 8, 3,7];

print("Duplicate elements in given array: ");

for i in range(0, len(arr)):
    for j in range(i+1, len(arr)):
        if(arr[i] == arr[j]):
            print(arr[j]);
```

OUTPUT

```
Duplicate elements in given array:
2
3
7
8
```

13. Write Python programs to create functions and use functions in the program.

```
def add(num1, num2):
     return num1+num2
def sub(num1, num2):
    return num1-num2
def mul(num1, num2):
    return num1*num2
def div(num1, num2):
    return num1/num2
```

```
result=add(10,20)
print("ADDITION OF TWO NUMBERS: ",result);
result=sub(20,10)
print("SUBTRATION OF TWO NUMBERS: ",result);
result=mul(10,20)
print("MULTIPLICATION OF TWO NUMBERS: ",result);
result=div(20,10)
print("DIVISION OF TWO NUMBERS: ",result);
```

OUTPUT

```
ADDITION OF TWO NUMBERS: 30
SUBTRATION OF TWO NUMBERS: 10
MULTIPLICATION OF TWO NUMBERS: 200
DIVISION OF TWO NUMBERS: 2.0
```

14. Write Python programs to using lambda function.

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

OUTPUT
13

15. Write Python programs loading the module in Python code.

#Save this code in a file named module1.py

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

#Import the module named mymodule, and call the greeting function:

#Save this code in a file named module2.py

```
import mymodule

mymodule.greeting("Jonathan")
```

OUTPUT
Hello, CCMS COLLEGE, VADU

16. Write a program to print following pattern

1
1 2
1 2 3
1 2 3 4
1 2 3 4 5

```
rows = int(input("Enter the number of rows: "))

for i in range(1, rows+1):
    for j in range(1, i + 1):
        print(j, end=' ')
    print("")
```

OUTPUT

```
Enter the number of rows: 6
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4 5 6
```

17. Write Python programs to implement a concept of list.

```
list2 = [1,2, 3, 4, 5, 6, 7, 8, 9, 10, 11,12]
for i in list2:
    if i % 2 == 0:
        print("Even Numbers are : " ,i)
```

OUTPUT

```
Even Numbers are : 2
Even Numbers are : 4
Even Numbers are : 6
Even Numbers are : 8
Even Numbers are : 10
Even Numbers are :  12
```

18. Write Python programs to implement a concept of tuples.

```
#create a tuple
```

```
x = ("CCMS BCA COLLEGE")
# Reversed the tuple
y = reversed(x)
print(tuple(y))

#create another tuple
x = (5, 10, 15, 20)
# Reversed the tuple
y = reversed(x)
print(tuple(y))
```

OUTPUT

```
('E', 'G', 'E', 'L', 'L', 'O', 'C', ' ', 'A', 'C', 'B', ' ', 'S', 'M', 'C', 'C')
(20, 15, 10, 5)
```

19. Write a Python program to create nested list and display its elements.

```
numbers = [ [2, 3, 4],[5, 6, 7],[8, 9, 10]]

for list in numbers:
      for number in list:
        print(number, end=' ')
```

OUTPUT

```
2 3 4 5 6 7 8 9 10
```

20. Write a Python program to using multiple inheritances.

```
class Class1:
    def msg1(self):
      print("In Class1")

class Class2(Class1):
    def msg2(self):
    print("In Class2")

class Class3(Class1):
    def msg3(self):
    print("In Class3")

class Class4(Class2, Class3):
```

```
    def msg(self):
      print("In Class4")

obj = Class4()
obj.msg1()
```

OUTPUT
In Class1

21. Write a Python program to read a file bca.txt and print the contents of file along with number of vowels present in it.
Create one text file name bca.txt in same folder of python C:\Python27\bca.txt and file content is:

Bca.txt
my name is Shailesh

Patelp21.py

```
myfile = open("bca.txt", "r")

read_data = myfile.read()

# find the number of the vowels in file

vowel_count = 0

for i in read_data:

     if(i == 'A' or i == 'a' or i == 'E' or i == 'e' or i == 'I' or i == 'i' or i == 'O' or i == 'o' or i ==  'U' or i == 'u'):
            vowel_count += 1

print('The Number of Vowels in text file :', vowel_count)

myfile.close()
```

OUTPUT
The Number of Vowels in text file : 6

22. Write a Python program for Error Handling.

```
try:
  a = int(input("Enter First Number: "))
  b = int(input("Enter Second Number: "))
  c = a/b
  print("Division is : ", c)
except:
  print("Can't divide with zero")
finally:
  print("Rest of the code..")
```

OUTPUT

```
Enter First Number: 12
Enter Second Number: 3
Division is :  4.0
Rest of the code..

Enter First Number: 3
Enter Second Number: 12
Division is :  0.25
Rest of the code..

Enter First Number: 3
Enter Second Number: 0
Can't divide with zero
Rest of the code..
```

23. Write a Python program for connection with my Sql and display all record from the database.

Create Connection Start by creating a connection to the database.
Use the username and password from your MySQL database:

```
import mysql.connector
```

```
mydb = mysql.connector.connect(
  host='localhost', database='test', user='root', password='')


mycursor = mydb.cursor()

sql = "SELECT * FROM cricketor_data "

mycursor.execute(sql)
myresult = mycursor.fetchall()

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

OUTPUT

```
+____________+____________+_____________+_____________+____________+
| First_Name | Last_Name  | Date_Of_Birth | Place_Of_Birth | Country     |
+____________+____________+_____________+_____________+____________+
| Shikhar    | Dhawan     | 1981-12-05    | Delhi          | India       |
| Jonathan   | Trott      | 1981-04-22    | CapeTown       | SouthAfrica |
| Kumara     | Sangakkara | 1977-10-27    | Matale         | Srilanka    |
| Virat      | Kohli      | 1988-11-05    | Delhi          | India       |
| Rohit      | Sharma     | 1987-04-30    | Nagpur         | India       |
+____________+____________+_____________+_____________+____________+
5 rows in set (0.00 sec)
```

24. Write a Python program for modified record, display record and delete record from the database.

```
import mysql.connector

mydb = mysql.connector.connect(
  host='localhost', database='test', user='root', password='')

mycursor = mydb.cursor()
sql = "UPDATE users SET city = 'Delhi' WHERE city = 'Baroda'"

mycursor.execute(sql)

mydb.commit()
```

```
print(mycursor.rowcount, "record(s) affected")
```

OUTPUT

```
+_____+_____+_____+_____+_____+
| First_Name | Last_Name  | Date_Of_Birth | Place_Of_Birth | Country     |
+_____+_____+_____+_____+_____+
| Shikhar      | Dhawan       | 1981-12-05      | Baroda           | India      |
| Jonathan     | Trott          | 1981-04-22      | CapeTown      | SouthAfrica |
| Kumara      | Sangakkara  | 1977-10-27      | Matale          | Srilanka    |
| Virat          | Kohli          | 1988-11-05      | Delhi            | India      |
| Rohit          | Sharma       | 1987-04-30      | Nagpur          | India      |
+_____+_____+_____+_____+_____+
5 rows in set (0.00 sec)
```

25. Write a Python program for search record from the database.

```
import mysql.connector

mydb = mysql.connector.connect(

   host='localhost', database='test', user='root', password='')
mycursor = mydb.cursor()

sql = "SELECT * FROM users WHERE city ='Delhi'"

mycursor.execute(sql)

myresult = mycursor.fetchall()

for x in myresult:

   print(x)
```

OUTPUT

```
+_____+_____+_____+_____+_____+
| First_Name | Last_Name  | Date_Of_Birth | Place_Of_Birth | Country     |
+_____+_____+_____+_____+_____+
| Virat          | Kohli          | 1988-11-05      | Delhi            | India      |
+_____+_____+_____+_____+_____+

1 rows in set (0.00 sec)
```