PYTHON NOTES SECTION A

➤ Introduction to Python:

- Python is a general purpose dynamically typed high level language developed by Guido van Rossum in the year 1991 at CWI in Netherlands.
- **Syntax:**
```
print("Hello")
```

➤ History of Python (Origins Late 1980s – 1991)

- Python was created by Guido van Rossum in the late 1980s.
- Guido was working at CWI (Centrum Wiskunde & Informatica) in the Netherlands.
- He wanted a language that was:
    1. Easy to read and write
    2. Powerful but simple
- Python was inspired by the ABC language.
- The name "Python" comes from Monty Python's Flying Circus (not the snake).
- Python 1.0 was officially released in 1991.

➤Early Growth (1990s)

1. <u>Python gained popularity for:</u>

- Clean syntax
- Built in data structures (Lists, dictionaries)

2.<u>Supported multiple programming styles:</u>

- Procedural
- Object Oriented
- Functional

3.Used in scripting, automation and academic research.

➤ <u>Python 2 Era (2000 – 2010)</u>

*Python 2.0 released in 2000
*<u>Introduced:</u>

1. List comprehensions
2. Garbage collection

3. Unicode support

* Python adoption grew rapidly in:

- Web development
- System administration

➤ **Python 3 Revolution (2008 – Present)**

- Python 3.0 released in 2008
- Not backward compatible with Python 2

Major improvements:

1. Better unicode handling
2. Cleaner syntax
3. Improved libraries

Python 2 officially reached end of life in 2020.

Modern Python (2015 – Today)

Used in:

- Web Development (Django, Flask, Fast API)
- Data Science & AI (NumPy, Pandas, TensorFlow, PyTorch)
- Automation & Scripting
- Game development
- DevOps & Cloud

➤**Different Versions of Python:**

1. Python 1.x (1991) – Now obsolete
2. Python 2.x (2000) – End of Life January 2020
3. Python 3.x (2008 – Present) – Currently active

Major Improvements:

- Better unicode support
- Cleaner syntax
- Improved performance & security

Always use Python 3
Stable versions: Python 3.10 / 3.11 / 3.12

➤<u>**Installing Python:**</u>

Step 1 – Download Python

- Go to python.org
- Download latest Python 3.x

Step 2 – Install Python

- Check “Add Python to PATH”
- Click Install Now

Step 3 – Verify Installation
Open command prompt and type:

```
python --version
```

➤<u>**Setting up Python in Local Environment:**</u>

Local environment means:

- Python installed on your computer
- Runs on your machine, not the internet

Basic Setup Components:

1. Python Interpreter
2. Text editor / IDE
3. Terminal or Command Prompt

Popular Editors / IDEs:

- IDLE
- VS Code
- PyCharm
- Sublime Text

➤<u>**Python IDLE:**</u>

- Python's default editor
- Installed automatically with Python
- Best for beginners

IDLE has two modes:

1. **<u>Interactive Mode</u>**
    i. Directly type python commands
    ii. Executes immediately

   <u>Example</u> :

   ```
   >>> print(“Hello Pyhton”)
     Output-    Hello python
   ```

   <u>Used for</u>

- Testing code   * Learning Basics

2. Script Mode

- Write Python programs in file
- File extension: `.py`

Steps:

1. Open IDLE
2. Click File → New File
3. Write code
4. Save as example.py
5. Press F5 to run

➤**<u>Executing python from a file :</u>**

Creating a Python File

Creating a file named hello.py :

Example:

```
print("Hello World")
print("Python is running from a file")
```

Runing the file (using terminal):

```
python hello.py
output appears in the terminal
```

➤ **<u>Python Command Line & Interactive Mode:</u>**

Starting interactive mode.

Open terminal and type: python

You’ll see: >>> This is python interactive shell

Example:

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

➤ Strengths of Python

1. Simple, Readable and Expressive Syntax

Python was designed with readability as a core principle.

Why this is powerful:

- Code looks like plain English
- Indentation enforces clean structure
- Reduces cognitive load

Impact:

- Faster onboarding for beginners
- Easier maintenance for teams.
- Lower long term project cost

** This is Why python is widely used in education, start-ups, and research.

2. High Developer Productivity

Python allows developers to build more with less code.

Reasons:

- Dynamic typing
- Automatic memory management
- Rich standard library
- Third party libraries

Real world effect:
A feature that takes weeks in Java/C++ may take days in Python.

Ideal for:

- MVPs
- Prototypes
- Hackathons
- Rapid iteration

This is why companies like Google, Netflix, Dropbox and Instagram use Python.

3. Massive Ecosystem & Libraries:

- Python has one of the largest ecosystem of any language.

Key domains:

- Web development: Django, Flask, FastAPI
- AI / ML: TensorFlow, PyTorch, Scikit-learn
- Data Science: NumPy, Pandas, Matplotlib
- Automation: Requests, BeautifulSoup
- DevOps: Ansible, Fabric

Why this matters:

- No need to build from scratch
- Most problems already have solutions
- Accelerates development exponentially

4.Multi-Paradigm Language:

Supports:

1. Procedural Programming
2. Object Oriented Programming (OOP)
3. Functional Programming

- This flexibility makes python suitable for many types of projects.

Benefits:

- Choose best style for problem
  - Scripts → Procedural
  - Applications → OOP
  - Data Processing → Functional

5. Cross Platform Compatibility:

Python runs almost identically on:

1. Windows
2. Linux
3. MacOS

Impact:

- Write once, run anywhere
- Great for cloud, servers and CI/CD pipelines

➤**Weakness of Python:**

1. Slower Execution Speed

Python is interpreted language, not compiled to machine code.

Why slow:

- Code executed line by line
- Dynamic typing runtime overhead

Comparison:

- Python 10–50x slower than C/C++
- Slower than Java, Go, Rust

Not ideal for:

- Game engines
- Real time systems
- High frequency trading
- Low latency applications

2. High Memory Usage

Python objects consume more memory.

Example:
Integer in Python uses more memory than in C.

Why:

1. Object metadata
2. Garbage collection

3. Dynamic typing

Not suitable for:

- Embedded systems
- Mobile apps
- Memory constrained environments

3. Dynamic Typing Runtime Errors

Python checks types at runtime.

Example:

```
x = "10"
y = x + 5   # runtime error
```

Impact:

- Bugs discovered late
- Requires strong testing discipline

4.Not ideal for mobile development:

Python lacks strong native support for:

1. Android
2. iOS

Why:

- Performance constraints
- Platform tooling favors Java / Swift / Kotlin

Python is usually used for:

- Backend services
- APIs
- Mobile app servers

5.Dependency & Packaging Complexity:

Python packaging can be confusing:

1. pip

2. virtualenv
3. venv
4. poetry
5. conda

Problems:

- Dependency conflicts
- Environment inconsistencies

Python Variables :-

- A variable in Python is a named memory location used to store data values.
- It acts as a container or placeholder that holds information which can be used and modified during program execution.

Important points about variables

1. Variables store data temporarily during program execution.
2. The value of a variable can change anytime.
3. Python does not require declaration of variable type before use.
4. Python is dynamically typed – type is automatically assigned.
5. The assignment operator (=) is used to assign value.

Example

```
x = 10
name = 'Sham'
price = 99.5

print(x)
print(name)
print(price)
```

Output

```
10
Tushar
99.5
```

Types of Variables

1 : Numeric Types

(a) Integer (Int)

Stores whole numbers

```
x = 10
```

(b) Float

Stores decimal numbers

```
y = 10.5
```

(c) Complex

Stores decimal numbers

```
z = 2 + 3j
```

---

2 : String Types

Stores sequence of characters enclosed in quotes.

```
name = "Python"
```

---

3 : Boolean Type

Stores True or False values.

```
flag = True
```

---

4 : Sequence Types

- List
- Tuple
- Range

Example :

```
list1 = [1,2,3]
tuple1 = (4,5,6)
```

---

5 : Set Type

Stores unique values

```
s = {1,2,3}
```

---

6 : Dictionary Type

Stores Key – Value pairs

```
d = {"name":"Sham","age":20}
```

Python Identifiers :-

- An identifier is a name given to variables, functions, classes, modules, or any other object in Python.
- It is used to identify an object uniquely in the program.

Rules for naming identifiers

1. Must start with alphabet (a-z or A-Z) or underscore (_).
2. Cannot start with a digit.
3. Can contain letters, digits and underscore.
4. No special symbols like @, #, $, %, etc.
5. No spaces allowed.
6. Cannot be a keyword.
7. Python is case sensitive.

Valid identifiers

```
name
roll_no
x1
_myVar
total_marks
```

Invalid identifiers

```
1name (starts with digit)
total marks (contains space)
for (keyword)
name@ (special character)
```

Reserved Keywords

- Keywords are reserved words in Python that have special meaning and are part of Python syntax.
- They cannot be used as variable names or identifiers.

Important points about keywords

1. Keywords are predefined words.
2. Each keyword has fixed meaning.
3. They cannot be redefined.
4. They form the structure of Python programs.

Example

```
if      else     elif
for     while    break
continue pass    return
def     class    lambda
try     except   finally
True    False    None
and     or       from
not     import   global
```

Data Type :-

- A data type represents the type of value that a variable can store.
- It tells Python what kind of data is being handled and what operations can be performed on it.

Important Points

1. Data type defines the nature of data (number, text, collection, etc).
2. It determines the memory allocation and operations allowed on the data.
3. Python automatically assigns data types based on the value given.
4. Python is dynamically typed language, so we do not need to declare data types explicitly.

Example

```
x = 10        # int
y = 5.5       # float
z = "Hello"   # string
```

Types

1 : Numeric Types :-

Numeric data types store the numerical values.

Python Supports :

- int
- float
- complex

---

a : Integer (int)

- Integer represents whole numbers (positive, negative or zero) without decimal points.

Points :

1. Can store unlimited length numbers.
2. Used for counting and indexing.
3. Does not contain fractional values.

Example

```
x = 10
print(type(x))
```

Output

```
<class 'int'>
```

---

b : Float

- Float represents decimal numbers (numbers with fractional part)

Points :

1. Contains decimal point.
2. Accurate up to 15 decimal points.
3. Used in scientific and mathematical calculations.

Example

```
y = 5.75
print(type(y))
```

Output

```
<class 'float'>
```

---

c : Complex

Complex numbers are written in the form x + yj.

Points :

1. x = real part
2. y = imaginary part
3. Used in advanced mathematics

Example

```
z = 2 + 3j
print(type(z))
```

Output

```
<class 'complex'>
```

---

2 : String data type :-

- A string is a sequence of characters enclosed in single or double quotes.
- Strings are immutable (cannot be changed)
- Can contain letters, numbers and special characters.
- Indexed starting from 0.

Example

```
str1 = "Hello"
print(str1)
print(str1[0])
```

Output

```
Hello
H
```

---

3 : List data type :-

- A list is an ordered, mutable collection of elements enclosed in square brackets [ ].
- Lists are changeable (mutable).
- Can store multiple data types.

Example

```
list1 = [10,"Hello",5.5]
print(list1)
list1.append("Python")
print(list1)
```

Output

```
[10,'Hello',5.5]
[10,'Hello',5.5,'Python']
```

---

4 : Tuple data type

- A tuple is an ordered, immutable collection of elements enclosed in parentheses ( ).
- Cannot modify after creation.
- Faster than lists.
- Used when data should not change.

Example

```
t = (1,2,3)
print(t)
print(t[1])
```

Output

```
(1,2,3)
2
```

---

5 : Set data type :-

- A set is an unordered collection of unique elements enclosed in curly brackets { }.
- Does not allow duplicate values.
- Unordered
- Mutable

Example

```
s = {1,2,2,3}
print(s)
```

Output

```
{1,2,3}
```

---

6 : Dictionary data type :-

- A dictionary data type is a collection of key value pairs enclosed in curly braces { }.

Points :

- Keys must be unique
- Keys must be immutable (string, number, tuple)
- Mutable collection

Example

```
student = {
"name":"Rahul",
"roll":101,
"marks":95
}

print(student)
print(student["name"])
```

Output :

```
{'name':'Rahul','roll':101,'marks':95}
Rahul
```

➤ Conditional statement / decision making statements :-

- Conditional statements are also known as decision making statements.
- Conditional statements are used to make decisions that allow us to control the path of execution.
- Conditional statements typically involve decision steps. These steps rely on conditions that are evaluated as either true or false.
- Types of conditional statement are :-

---

1. If statement :-

- The if statement checks the condition.
- If the condition is true, the code inside (if) runs.

Syntax

```
if condition :
    statement
```

Program

```
age = int(input("Enter your age : "))

if age >= 18 :
    print("You are eligible to vote")
```

Output

```
Enter your age : 22
You are eligible to vote
```

---

2. If else statement :-

- The if else statement is a conditional statement that executes one block of code when the condition is true and another block of code when the condition is false.

Syntax

```
if condition :
    statement
else :
    statement
```

Program

```
age = int(input("Enter your age : "))

if age >= 18 :
    print("You are eligible for vote")
else :
    print("You are not eligible for vote")
```

Output

```
Enter your age : 20
You are eligible for vote
```

---

3. Else if statement :-

- The else if statement is used to check multiple conditions in sequence.
- It executes the block of code associated with the first true condition.

Syntax

```
if condition1 :
    statement
elif condition2 :
    statement
else :
    statement
```

Program

```
age = int(input("Enter your age : "))

if age < 0 :
    print("Invalid age")

elif age >= 18 :
    print("You are eligible to vote")

else :
    print("You are not eligible to vote")
```

Output

```
Enter your age : 25
You are eligible to vote
```

---

4. Nested if statement :-

- A nested if statement is an if statement placed inside another if or else block.
- It is used when a condition depends upon another condition.

Syntax

```
if condition1 :
    if condition2 :
        statement
    else :
        statement
else :
    statement
```

Program

```
num = float(input("Enter a number : "))

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

Output

```
Enter a number : 12
Positive number
```

➤ Looping statement :-

- Looping statement are used to control the repetition in a program.
- It allows the programmer to repeat a specified block of code until a give condition is met.

Types

---

1. While Loop

- A while loop is a pre test conditional loop in python that repeatedly executes a block of statements as long as a given condition is true.
- The condition is checked before executing the loop body.
- When the condition false, the loop terminates.

Syntax

```
while <condition> :
    statement
```

Program

```
print("first five whole numbers")

x = 0
while(x < 5) :
    print(x)
    x = x + 1
```

Output

```
0
1
2
3
4
```

➤ While loop with else statement

- In python, an else block can be used with a while loop.
- The else block is executed when the loop condition becomes false.

Program

```
count = 0

while count < 5 :
    print(count , " is less than 5")
    count = count + 1

else :
    print(count , " is not less than 5")
    print("Exit program")
```

Output

```
0 is less than 5
1 is less than 5
2 is less than 5
3 is less than 5
4 is less than 5
5 is not less than 5
Exit program
```

---

2. For loop :-

- A for loop in python is used to iterate over a sequence such as a list, tuple, string, or range.
- It executes the loop body once for each element in the sequence.

Syntax

```
for <variable> in <sequence> :
    statement(s)
```

Program

```
print numbers from 0 to 4

for i in range(5) :
    print(i)
```

Output

```
0
1
2
3
4
```

Program : Sum of numbers in a list

```
numbers = [2, 3, 4, 6, 10, 12]

sum = 0

for val in numbers :
    sum = sum + val

print("The sum is" , sum)
```

Output

```
The sum is 37
```

➤ **For loop with else**

- The else block in a for loop executes after the loop finishes normally (without break).

Program

```
fruits = ['apple' , 'mango' , 'grapes' , 'orange']

for i in fruits :
    print("I like" , i)

else :
    print("No fruit left")
```

Output

```
I like apple
I like mango
I like grapes
I like orange
No fruit left
```

---

3. Nested Loop :-

- A nested loop is a loop inside another loop.
- The inner loop executes completely for each iteration of the outer loop.

Syntax

```
while condition1 :
    while condition2 :
        statement
```

Program

```
print prime number from 2 to 20

i = 2
while i <= 20 :
    j = 2
    while j <= i // 2 :
        if i % j == 0 :
            break
        j = j + 1
    if j > i // 2 :
        print(i)
    i = i + 1
```

Output

```
2
3
5
7
11
13
17
19
```

➤ Break Statement :-

- The break statement is a loop control statement used to terminate the execution of a loop immediately when a specified condition is satisfied.
- After break is executed, the control of the program moves outside the loop.
- It can be used inside both for and while loops.

Syntax

```
break
```

Program

```
Using break in a for loop

for x in ['a' , 'b' , 'c' , 'd' , 'e'] :
    if x == 'c' :
        print("Element found")
        break
    print(x)
```

Output

```
a
b
Element found
```

---

2. Continue Statement

- The continue statement is a loop control statement used to skip the current iteration of the loop and immediately move to the next iteration.
- Unlike break, it does not terminate the loop.
- It can be used in both for and while loops.

Syntax

```
continue
```

Program

```
Print only odd numbers using continue
```

```
x = 0

while x < 10 :
    x = x + 1
    if x % 2 == 0 :
        continue
    print(x)
```

Output

```
1
3
5
7
9
```

Explanation

- Even numbers are skipped using continue
- Only odd numbers are printed
- Loop continues until condition becomes false

➤ Operator Precedence :-

Operator precedence refers to the order or priority in which operators are evaluated in an expression.

In python, when an expression contains more than one operator, operators are evaluated according to their precedence level.

Operators with higher precedence are evaluated before operators with lower precedence.

Important rules of operator precedence

- If two operators of the same precedence appear in an expression, the operator that appears first is evaluated first.
- Parantheses () can be used to change the order of evaluation
- Expressions inside the innermost paranthese are evaluated first.

➤ Operator Precedence Table (High → Low)

Operators	Description
`**`	Exponentiation (Power)
`~ + -`	Bitwise complement, unary plus and minus
`* / % //`	Multiplication, Division, Modulus, Floor Division
`+ -`	Addition and Subtraction
`<< >>`	Bitwise shift operators
`&`	Bitwise AND
`^`	Bitwise XOR
`< <= > >=`	Relational operators
`== !=`	Equality operators
`is is not`	Identity operators
`not`	Logical NOT
`and`	Logical AND
`or`	Logical OR

Program

1. Same Precedence Operators

```
i = 10 * 2 + 4
print(i)
```

Output

```
24
```

- `*` and + have different precedence.
  Multiplication (*) is evaluated first, then addition (+).

---

2. Using Parentheses

```
i = 10 * (2 + 4)
print(i)
```

Output

```
60
```

Expression inside parentheses (2 + 4) is evaluated first.

---

3. To illustrate operator precedence

```
a = 10
b = 5
c = 15
d = 20

i = (a + b) * c / d
print(i)

i = a + (b * c) / d
print(i)
```

Output

```
11.25
13.75
```

Explanation

- In the first expression, (a + b) is evaluated first due to parentheses.
- In the second expression (b * c) is evaluated first.
- This shows how parentheses change the result.

Data Type Conversion :

- Data type conversion in Python is the process of converting a value from one data type to another.
- Python provides built in functions to perform data type conversion.

There are two types of data type conversion in Python :

---

1. Implicit Type Conversion

- Implicit type conversion is the automatic conversion of one data type into another by the python interpreter without any user intervention.
- Python always converts a smaller data type into a larger data type to avoid loss of data.

Program

```
x = 20
y = 20.6
z = x + y

print("x is of type :", type(x))
print("y is of type :", type(y))
print(z)
print("z is of type :", type(z))
```

Output

```
x is of type : <class 'int'>
y is of type : <class 'float'>
40.6
z is of type : <class 'float'>
```

Explanation

Here, integer x is automatically converted into float while adding with y.
This is called implicit type conversion.

---

2. Explicit Type Conversion :

- Explicit type conversion is the manual conversion of one data type into another by the programmer using built in functions.

Common explicit conversion functions

Function	Description
int()	converts data to integer
float()	converts data to float
str()	converts data to string
list()	converts to list
tuple()	converts to tuple
set()	converts to set
dict()	converts key-value pairs to dictionary
ord()	converts character to ASCII value

Program

```
s = "1101"

a = int(s,2)
print("After converting to integer base 2 :", a)

b = float(s)
print("After converting to float :", b)
```

Output

```
After converting to integer base 2 : 13
After converting to float : 1101.0
```

Operators :-

- An operator is a special symbol that performs a specific operation on one or more operands.
- The data elements, upon which an operator will perform a particular operation, are called operands.
- Operators that take one operand are called unary operators.
- Operators that take two operands are called binary operators.

Types of Operators

1. Arithmetic Operator :

Arithmetic operators are used to perform basic mathematical operations.

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
**	Exponentiation
//	Floor Division

Program

```
A = 10
B = 20

print(A + B)
print(A - B)
print(A * B)
print(A / B)
print(A % B)
print(A ** 2)
print(10 // 3)
```

Output

```
30
-10
200
0.5
```

```
10
100
3
```

(b) Relational Operators :

Relational operators are used to compare two values and return either True or False.

Operator	Meaning
==	Equal to
!=	Not equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal

Program

```
A = 10
B = 20

print(A == B)
print(A != B)
print(A > B)
print(A < B)
print(A >= B)
print(A <= B)
```

Output

```
False
True
False
True
False
```

```
True
```

---

3. Assignment Operators :

Assignment operators are used to assign values to variables.

Operator	Example
=	x = 5
+=	x += 1
-=	x -= 1
*=	x *= 2
/=	x /= 2
%=	x %= 2

Program

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

Output

```
15
```

---

4. Logical Operators :

Logical operators are used to combine conditional statements.

Operator	Meaning
and	True if both conditions are true
or	True if any one condition is true

Operator	Meaning
not	Reverses the condition

Program

```
print(5 > 4 and 6 < 9)
print(5 < 4 or 6 > 9)
print(not(5 < 4))
```

Output

```
True
False
True
```

---

5. Membership Operators :

Membership operators test whether a value is present in a sequence.

Operator	Meaning
in	Present in sequence
not in	Not present in sequence

Program

```
list = [10, 20, 30, 40]

print(10 in list)
print(50 not in list)
```

Output

```
True
True
```

---

6. Identity Operators :

Identity operators compare the memory location of two objects.

Operator	Meaning
is	Same memory location
is not	Different memory location

Program

```
a = 10
b = 10
c = 20

print(a is b)
print(a is not c)
```

Output

```
True
True
```

---

7. Bitwise Operators :

Bitwise operators perform operations bit by bit on binary numbers.

Operator	Operation
&	AND
^	XOR
~	NOT
<<	Left Shift
>>	Right Shift
|	OR

Program

```
a = 60
b = 13

print(a & b)
print(a | b)
print(a ^ b)
print(~a)
print(a << 2)
print(a >> 2)
```

Output

```
12
61
49
-61
240
15
```

Functions

- A function is a named block of code that performs a specific and well defined task.
- It is an independent block of code that executes only when it is called.
- In Python, a function is a small sub program that supports modular programming.
- The main program calls the function whenever required, and after execution, control returns to the calling point.
- Creating a function is just like hiring an individual and giving him a particular task to perform.

Need of function

- The most important reason to use the function is to make program handling easier as only a small part of the program is dealt with a time.
- Easy code reusability: You just have to call the function by its name to use it.
- Functions can be really useful when making large programs.

- Functions decompose the large program into smaller segments which makes program easy to understand, maintain and debug.
- It facilitates top–down modular programming. In this program style, the high level logic of the overall problem is solved first while the details of each lower level function are addressed later.
- A function can be shared by many programs.

Types of functions

1. Built in functions :

Functions that come inbuilt into the python itself called built in functions.

Examples:
`print() , input() , len() , type() , eval()`

Each built in function performs a specific task.

---

2. User defined functions :

Functions that are created by the user to perform a specific task are called user defined functions.

Python language allows programmers to define their own functions according to their requirement.

Steps for writing user defined function in python

1. Function definition

- Function definition contains programming codes that implements what the function does.

Rules to define a function in python are:

1. The declaration of user defined function in python begins with the keyword **def**.
2. It is followed by the function name.

3. The function takes arguments as input with the opening and closing parentheses.
4. The function name followed by a colon (:) Colon is compulsory.
5. After defining the function name and arguments a block of program statement start at the next line and these statement must be indented.
6. First statement of a function can be optional docstring.

Syntax

```
def function_name (parameter_list):
    statement-1
    statement-2
```

Components

- function_name → Name of function
- parameter_list → List of parameters. It is like a placeholder
- function body → Statements inside function
- return statement → optional

---

2. Function Calling :

- Control of the program cannot be transferred to the user defined function unless it is called.
- After declaring and defining a function, you will have to call it to perform the defined task.
- When a program calls a function, program control is transferred to the called function.
- To execute a function, it is called.
- A called function performs the defined task and when its return statement is encountered it returns program control back to the calling point where it was invoked or called from.

Syntax

```
function_name (arguments)
```

Example

Function without parameters

```
def show():
    print("Hello Students")
    print("This is Python")

show()
```

Output

```
Hello Students
This is Python
```

---

3. Function with parameters :

- Function can accept values called arguments.
- Functions with parameters become much more powerful and useful because it allows us to do something slightly different each time we call the function by passing in different parameters within the parentheses.

Example

```
def sum(x,y):
    s = x + y
    print("Sum of two numbers is")
    print(s)

sum(10,20)
sum(20,30)
```

Output

```
Sum of two numbers is
30
Sum of two numbers is
50
```

---

4. Return statement :

- The return statement is used to send a value back to the calling program.
- If no value is given, it returns None.
- It terminates the function execution.
- The data type of the return expression must match that of the declared return-type for the function.

Example

```
def sum(x,y,z):
    return x + y + z

total = sum(10,20,30)
print("Sum:", total)

avg = total / 3
print("Average:", avg)
```

Output

```
Sum: 60
Average: 20.0
```

The anonymous functions :-

- Python supports the creation of anonymous functions at runtime, using a construct called lambda.
- An anonymous function is a function without a name.
- In Python, anonymous functions are created using the lambda keyword.
- We use lambda functions when we require a nameless function for a short period of time.
- Lambda functions are mainly used in combination with the functions filter(), map(), and reduce().
- The lambda feature was added to Python due to the demand from Lisp programmers.
- Lambda functions can have any number of parameters.
- It contains only one expression.
- Does not use return keyword.

Syntax

```
lambda arg1, arg2 : expression
```

Example

```
square = lambda x : x ** 2
print(square(5))
```

Output

```
25
```

Use of lambda function

1. filter() :

- The filter(function, list) function in python takes in a function and a list as parameters.
- It offers an elegant way to filter out all elements of a list, for which the function returns true.

Example

```
my_list = [11,5,24,6,10,17,9]
new_list = list(filter(lambda x : x % 2 == 0,
my_list))
print(new_list)
```

Output

```
[24,6,10]
```

---

2. map() :

- The map(fun, li) function in python takes in a function and a list as parameters.
- The first parameter fun is the name of a function and the second a list li.
- A map() applies the function fun to all the elements of the list sequence li.
- It returns a new list with the elements changed by fun.

Example

```
my_list = [1,5,4,6,8]
new_list = list(map(lambda x : x * 2, my_list))
print(new_list)
```

Output

```
[2,10,8,12,16]
```

Scope of variables :

- Scope of a variables refers to the area of the program where the variable can be accessed.

There are two types

1. Local variable

- Variables declared inside a function are called local variables.
- Accessible only inside function.
- Created when function is called.
- Destroyed when function ends.

Example

```
def show():
    a = 10
    print("Value of a is:", a)

show()
print(a)
```

Output

```
Value of a is: 10
NameError: name 'a' is not defined
```

2. Global variable :

- Variables declared outside the function are called Global variables.
- Accessible throughout program.
- Not destroyed after function call.

Example

```
b = 20

def show():
    a = 10
    print("Value of a is:", a)
    print("Value of b is:", b)

show()
print(b)
```

Output

```
Value of a is: 10
Value of b is: 20
20
```

Keyword argument (Parameter) :

- Keyword arguments are arguments passed to a function by explicitly specifying the parameter name along with its value during the function call.
- In this function/method, the order of arguments does not matter because values are assigned based on parameter names.
- When calling a function, arguments can be passed into two ways:

1. Positional arguments (order matters)
2. Keyword arguments (order does not matter)

In keyword arguments, we use:

```
parameter_name = value
```

Syntax

```
function_name(parameter1 = value1 , parameter2 =
value2)
```

Example

```
def student(name, age, course):
    print("Name:", name)
    print("Age:", age)
    print("Course:", course)

student(name = "Tushar", age = 20, course = "BCA")
```

Output

```
Name: Tushar
Age: 20
Course: BCA
```

2.Optional parameters (Default parameters)

- Optional parameters are parameters that have default values assigned during functional definition.
- If the user does not provide a value while calling the function, the default value is used automatically.
- In Python, we can assign a default value to a parameter. Such parameter becomes optional while calling the function.

Syntax

```
def function_name (parameter = default_value):
    statements
```

Example

```
def greet(name = 'Student'):
    print('Hello', name)

greet('Mohan')
greet()
```

Output

```
Hello Mohan
Hello Student
```

STRING operations IN PYTHON

1. String (Definition)

A string in Python is a sequence of characters enclosed within single quotes (' '), double quotes (" "), or triple quotes (''' ''').

- A string can contain letters, numbers, symbols, and spaces.
- Strings are immutable, which means their values cannot be changed after creation.
- Each character in a string is stored at a specific position called index.

---

2. Creating a String

Strings are created by assigning characters inside quotes to a variable.

- Python allows both single and double quotes to define a string.
- Triple quotes are used for multi-line strings.
- Quotes inside strings can also be handled using alternate quote types.

Example Program

```
s1 = "Hello World"
s2 = 'Python Programming'

print(s1)
print(s2)
```

Output

```
Hello World
```

---

3. Length of a String – len()

The len() function is used to find the number of characters in a string.

- It counts all characters including spaces.
- It returns an integer value.
- It helps to determine the size of the string.

Example Program

```
s = "Hello Python"
print(len(s))
```

Output

```
12
```

---

4. Indexing in String

Indexing is used to access individual characters of a string.

- Indexing starts from 0 in forward direction.
- Backward indexing starts from -1.
- Characters are accessed using square brackets [ ].

Example Program

```
s = "Python"
print(s[0])
print(s[-1])
```

Output

```
P
n
```

---

5. Slicing the String

Slicing is used to extract a part (substring) of a string.

- It uses the syntax [start : stop : step].
- Start index is inclusive and stop index is exclusive.
- Step defines the increment between indices.

Example Program

```
s = "Hello Python"
print(s[0:5])
print(s[6:])
print(s[-2:])
```

Output

```
Hello
Python
on
```

---

6. Concatenation of Strings

Concatenation means joining two or more strings.

- It is done using + operator.
- It returns a new string.
- Original strings remain unchanged.

Example Program

```
s1 = "Hello "
s2 = "Python"
s3 = s1 + s2
print(s3)
```

Output

```
Hello Python
```

---

7. Comparing Strings

Strings can be compared using relational operators like == , < , >.

- Comparison is based on ASCII or Unicode values.
- It returns True or False.
- Used in decision-making statements.

Example Program

```
s1 = "Bat"
s2 = "Bag"

if s1 == s2:
    print("Strings are same")
else:
    print("Strings are not same")
```

Output

```
Strings are not same
```

---

8. Replacing a String – replace()

The replace() method replaces a substring with another substring.

- It returns a new string.
- Original string remains unchanged.
- Used to modify string content.

Example Program

```
s = "I like Java"
new = s.replace("Java","Python")
print(new)
```

Output

```
I like Python
```

---

9. Splitting String – split()

The split() method divides a string into a list.

- It separates based on delimiter.
- Default delimiter is space.
- Returns list of substrings.

Example Program

```
s = "apple,mango,orange"
```

```
print(s.split(","))
```

Output

```
['apple', 'mango', 'orange']
```

---

10. Joining String – join()

The join() method combines list elements into a string.

- It uses a separator.
- Works on iterable objects.
- Returns a single string.

Example Program

```
list = ['apple','mango','orange']
print("-".join(list))
```

Output

```
apple-mango-orange
```

---

11. Changing Case of String

a) upper()

- Converts all characters into uppercase.
- Returns a new string.
- Original string remains unchanged.

```
s = "python"
print(s.upper())
```

Output:

```
PYTHON
```

---

b) lower()

- Converts all characters into lowercase.
- Opposite of upper().

- Returns new string.

```
s = "PYTHON"
print(s.lower())
```

Output:

```
python
```

---

c) swapcase()

- Converts uppercase to lowercase and vice versa.
- Changes each character’s case.
- Returns a new string.

```
s = "Welcome Python"
print(s.swapcase())
```

Output:

```
wELCOME pYTHON
```

---

d) title()

- Converts first letter of each word to uppercase.
- Useful for formatting text.
- Returns new string.

```
s = "welcome to python"
print(s.title())
```

Output:

```
Welcome To Python
```

String Testing Methods in Python

String testing methods are used to check the nature of characters present in a string.
These methods return Boolean values:

- True – if condition is satisfied

- False – if condition is not satisfied

---

1. isalnum() Method

Definition:

- The isalnum() method checks whether all characters in a string are alphanumeric.
- A string is called alphanumeric if it contains only alphabets (A–Z, a–z) and numbers (0–9).
- This method returns True if the string contains only letters and digits and contains at least one character; otherwise, it returns False.

Syntax:

```
string.isalnum()
```

Example:

```
s = "Python123"
print(s.isalnum())
```

Output:

```
True
```

---

2. isalpha() Method

Definition:

- The isalpha() method checks whether all characters in a string are alphabets only.
- It returns True if the string contains only letters (A–Z or a–z).
- If the string contains numbers, spaces, or special characters, then it returns False.

Syntax:

```
string.isalpha()
```

Example:

```
s = "Python"
print(s.isalpha())
```

Output:

```
True
```

---

3. isdigit() Method

Definition:

- The isdigit() method checks whether all characters in the string are digits (0–9).
- It returns True if the string contains only numeric characters.
- If the string contains alphabets, spaces, or symbols, it returns False.

Syntax:

```
string.isdigit()
```

Example:

```
s = "12345"
print(s.isdigit())
```

Output:

```
True
```

---

4. isupper() Method

Definition:

- The isupper() method checks whether all the alphabetical characters in the string are in uppercase.
- It returns True if all letters are capital letters.
- If any lowercase letter is present, it returns False.

Syntax:

```
string.isupper()
```

Example:

```
s = "PYTHON"
print(s.isupper())
```

Output:

```
True
```

---

5. islower() Method

Definition:

- The islower() method checks whether all the alphabetical characters in the string are in lowercase.
- It returns True if all letters are small letters.
- If any uppercase letter is present, then it returns False.

Syntax:

```
string.islower()
```

Example:

```
s = "python"
print(s.islower())
```

Output:

```
True
```

Here are your **same to same notes in written form** from the **Python Modules (Section – B)** 👇

Python Modules

Section – B

- A Python module is a file containing Python code such as functions, variables, and classes that can be reused in other Python programs.
- It helps organize large programs into smaller, manageable, and reusable components.
- Python has built in modules, but you can also create your own.

- A module is a part of a program that contains code for a specific function or task.
- Each module is independent, meaning it has everything needed to perform one job.

Important Points

1. A module is simply a .py file
2. It allows code reusability across multiple programs.
3. It improves code organization and readability.
4. Each module has its own namespace to avoid naming conflicts.
5. Python provides built in modules and also allows user defined modules.

Purpose of Module

1. **Code Organization :**
   Encapsulates related functions, classes and variables into a single file for better structure.
2. **Reusability :**
   Promotes the reuse of code across multiple programs reducing redundancy.
3. **Modularity :**
   Allows developers to break down complex programs into manageable logical units.
4. **Namespace Management :**
   Creates separate namespaces to avoid naming conflicts b/w identifiers.
5. **Built in Modules :**
   Provides access to a rich collection of predefined modules for common tasks (eg: math operations, file handling)

Advantages

- Less code has to be written
- The code is stored across multiple files
- Code is short, simple and easy to understand
- The same code can be used in many applications
- The scoping of variables can easily be controlled
- Errors can easily be identified, as they are localized to a function

Importing Modules

Python provides different ways to import modules.

---

1) Using import statement

- The import statement should be in the starting of code.
- The import keyword is followed by one or more Python modules specifies, separated by commas.

When the interpreter encounters an import statement, it imports the modules if the module is present in the search path.

- A search path is a list of directories that the interpreter searches before importing a module.

Syntax

```
import module1 , module2 , module
```

Example

```
import math
print(math.sqrt(16))
print(math.pow(2,3))
```

Output

```
4.0
8.0
```

---

2) Using from ..... import statement

- It imports the specific functions.
- No need to use module name prefix.
- Saves memory and improves readability.

Syntax

```
from module_name import function_name
```

Example

```
from math import sqrt
print(sqrt(25))
```

Output

```
5.0
```

---

3) Using from ..... import * statement

- Imports all functions from module.
- No need to use module name.
- Not recommended for large programs (namespace confusion).

Syntax

```
from <module_name> import *
```

Example

```
from math import *
print(factorial(5))
```

Output

```
120
```

Importing Multiple Modules

Syntax

```
import module1 , module2
```

Example

```
import math , calendar
print(math.sqrt(36))
print(calendar.isleap(2024))
```

Output

```
6.0
```

```
True
```

Python Standard Modules

Python provides many built in modules called Standard modules.

These modules are already available in Python and perform common tasks.

---

1) math module

It is used for mathematical operations.

Example

```
import math
print(math.ceil(5.2))
print(math.floor(5.8))
print(math.sqrt(9))
```

Output

```
6
5
3.0
```

Function	Description
ceil(n)	Returns the next integer no. of the given no.
sqrt(n)	Returns the square root of the given number
exp(n)	Returns the natural logarithm e raised to the given no.
floor(n)	Returns the previous integer number of the given no.
pow(base, exp)	Returns base to raised to the exp power

---

2) datetime module

- It is used for date and time handling.

Commonly used classes in the datetime module are:

- **date** – An idealized naive date, assuming the current Gregorian calendar always was, and always will be, in effect.
  Its attributes are year, month and day.
- **time** – Its attributes are hour, minute, second, microsecond.
- **datetime** – It is a combination of date and time along with the attributes year, month, day, hour, minute, second, microsecond.

Example of date

```
import datetime
x = datetime.date.today()
print(x)
```

Output

```
2026-02-12
```

Example of time

```
from datetime import time
a = time(10,33,46,234566)
print('a = ', a)
```

Output

```
a = 10:33:46.234566
```

---

3) calendar module

- It is used for calendar operations.

Example

```
import calendar
print(calendar.month(2022,3))
```

Output

```
March 2022
```

```
Mo Tu We Th Fr Sa Su
      1  2  3  4  5
6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31
```

The dir() Function

- The dir() function is a built in Python function that returns a sorted list of names (attributes, functions, variables) defined in a module or object.
- It is mainly used to explore what is available inside a module.

Important Points

1. dir() returns a list of strings.
2. It shows all attributes defined in a module.
3. Names starting with double underscores (_ _) are special Python system defined attributes.
4. It is useful for learning unknown modules.
5. It helps in debugging and inspection of objects.

Syntax

```
dir(module_name)
or
dir(object_name)
```

Example

```
import math
print(dir(math))
```

Output

```
['__doc__', '__loader__', '__name__', '__package__',
'acos', 'asin', 'atan', 'ceil', 'cos', 'floor',
'factorial', 'log', 'pi', 'pow', 'sqrt', 'tan']
```

Python File Handling

Section – B

➤ Opening File Using open( ) Method

- File handling in Python allows us to store data permanently in files instead of memory. To perform any operation like reading, writing, or appending, we must first open the file.
- Python provides a built in function `open()` to open a file.
- The `open()` function is a built in Python function used to open a file and return a file object.
- This file object is then used to perform operations such as `read()`, `write()`, `append()`, and `close()`.

Syntax

```
file_object = open(file_name, access_mode,
buffer_size, encoding)
```

Parameters

1. file_name

- Name of the file to be opened
- Must be given as a string
- Example : `"data.txt"`

---

2. Access_mode

- Defines how the file will be opened
- Optional parameter
- Default mode is `"r"` (read mode)

---

3. buffer_size

- Specifies buffer size for the file operations
- 0 → No buffering
- 1 → Line buffering
- Positive number → Custom buffer size
- Negative → Default system buffering

---

4. Encoding

- Specifies encoding type
- Example : UTF-8, cp1252

---

1) Creating Text File

- Creating a text file means generating a new file using Python so that data can be stored permanently.
- If the file does not exist, Python creates it.
- If the file exists and opened in `"w"` mode, it overwrites the file.

Program

```
f = open("news.txt","w")
f.write("Welcome to Python file handling")
f.close()
print("File created successfully")
```

Output

```
File created successfully
```

---

2) Controlling File Access Modes

- File access modes control how a file is opened and what operations can be performed on it such as reading, writing or appending.
- Python provides various file access modes.

Types of file modes

Mode	Description
r	Read only (default)
w	Write only (overwrites file)
a	Append data
r+	Read and Write
w+	Write and Read (overwrite)
rb	Read in binary
wb	Write in binary
ab	Append in binary

Important Points

1. If mode is not specified → default is `"r"`.
2. `"w"` mode deletes previous content.
3. `"a"` mode adds data at end of file.
4. Binary mode (b) is used for images, audio files, etc.
5. Text mode returns string data.

Example : Writing and Reading file

```
f = open("data.txt","w")
f.write("Python is easy")
f.close()

f = open("data.txt","r")
content = f.read()
print(content)
f.close()
```

Output

```
Python is easy
```

---

3) Working with file object attributes

- When a file is opened using `open()`, Python returns a file object.
- This file object contains several attributes that provides information about the file.

Important file attributes

Attribute	Description
file.name	Returns file name
file.mode	Returns mode in which file is opened
file.closed	Returns True if file is closed
file.encoding	Returns encoding type

Example

```
fo = open("news.txt","w")
print("File Name = ", fo.name)
print("File Mode = ", fo.mode)
print("File Closed = ", fo.closed)
print("Encoding = ", fo.encoding)
fo.close()
print("File Closed After Closing = ", fo.closed)
```

Output

```
File Name = news.txt
File Mode = w
File Closed = False
Encoding = cp1252
```

```
File Closed After Closing = True
```

➤ close( ) Method

- The `close()` method is used to close a file after performing operations.
- It releases system resources and ensures data is saved properly.

Important Points

1. It flushes unsaved data.
2. No further writing can be done after closing.
3. Good programming practice is to always close file.
4. Prevents memory leaks.

Example

```
f = open("sample.txt","w")
f.write("Hello")
f.close()

print("Is file closed ? ", f.closed)
```

Output

```
Is file closed ? True
```

➤ Reading File

Python provides different methods to read data from a file after opening it in read mode ("r").

(a) read( ) method

- The `read()` method is used to read the entire content of a file or a specified number of characters from the file.
- If no argument is given, it reads the complete file.

- If a number is given (eg. read(5)), it reads only that many characters.
- After reading, the file pointer moves to the end of the read content.

Syntax

```
file_object.read(size)
```

Example

```
f = open("demo.txt","r")
data = f.read()
print(data)
f.close()
```

Output

```
Hello Students
Welcome to Python
```

(b) readline( ) method

- The `readline()` method reads one line at a time from the file including the newline character (\n).
- It reads only a single line.
- Each call reads the next line.
- Useful when processing files line by line.

Example

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

Output

```
Hello Students
Welcome to Python
```

(c) readlines( ) method

- The `readlines()` method reads all lines of a file and returns them as a list.
- Each line becomes an element in the list.
- New line characters remain in the list.
- Useful when you want to process lines as a list.

Example

```
f = open("demo.txt","r")
lines = f.readlines()
print(lines)
f.close()
```

Output

```
['Hello Students\n', 'Welcome to Python\n']
```

➤ Writing files in Python

- To write into a file we open it in `"w"` or `"a"` mode.

---

write( ) method

- The `write()` method is used to insert text or data into a file.
- `"w"` mode overwrites existing data.
- If file does not exist, it creates a new file.
- It does not automatically add a newline (\n).

Example

```
f = open("demo.txt","w")
f.write("Hello Students\n")
f.write("Python Programming")
```

```
f.close()
```

Output

```
Hello Students
Python Programming
```

---

writelines( ) method

- The `writelines()` method writes multiple lines at once from a list into a file.
- Takes a list of strings as input.
- Does not automatically add newline.
- Used for writing bulk data.

Example

```
f = open("demo.txt","w")
lines = ["Line 1\n","Line 2\n","Line 3"]
f.writelines(lines)
f.close()
```

Output

```
Line 1
Line 2
Line 3
```

➤ File Built in Methods

(a) tell( ) method

- The `tell()` method returns the current position of the file pointer in bytes.
- Shows where reading/writing is happening.
- Useful in large files.
- Position starts from 0.

Example

```
f = open("demo.txt","r")
print(f.read(5))
print(f.tell())
f.close()
```

Output

```
Hello
5
```

(b) truncate( ) method

- The `truncate()` method reduces the size of the file to the specified no. of bytes.
- Used to shorten file content.
- Works only in write mode.
- Removes extra data after given size.

Example

```
f = open("demo.txt","r+")
f.truncate(5)
f.close()
```

(File now contains only first 5 characters)

(c) rename( ) Method

- Requires old file name and new file name.
- If old file does not exist, error occurs.

Example

```
import os
os.rename("demo.txt","newdemo.txt")
```

```
print("File Renamed")
```

Output

```
File Renamed
```

➤ Sending Output To STDOUT Using print( )

- By default, the `print()` function sends output to the console (standard output – stdout).
- It can also redirect output to a file.
- STDOUT means standard output stream.
- Output can be redirected to a file.
- Useful for logging data.

Example

```
import sys
f = open("output.txt","w")
sys.stdout = f
print("This is written to file")
sys.stdout = sys.__stdout__
f.close()
print("Done")
```

Output

```
Done
```

(File contains : This is written to file)

➤ Reading Input Using input( ) Method

- This `input()` function is used to take input from the user at runtime.

Important Points

- It always returns input as a string.
- To convert to integer, use `int()`.
- Used for dynamic data entry.

Example

```
name = input("Enter your name:")
print("Hello",name)
```

Output

```
Enter your name : Sham
Hello Sham
```

Here is your **same-to-same written content** from the notes you uploaded 👇

Python Collections

Section – A

- A sequence is a data structure in python used to store multiple values in an ordered form.
- Each element in the sequence is stored at a specific position called an index which allows easy access and manipulation of data.
- Sequence stores elements in ordered manner. Indexing starts from 0.
- It supports
  (i) Indexing (ii) Slicing (iii) Iteration
- Sequence can be :
  • Homogeneous • Heterogeneous
- Sequences are used for storing collection of data.

Types of Sequence

1 : List :-

- A list is a built in data structure used to store multiple items in a single variable.
- It is an ordered and mutable collection of elements enclosed within square brackets [ ] and separated by commas.

Features of List

1 : Ordered :

- Elements are stored in the order in which they are inserted.

2 : Mutable :

- Elements of a list can be changed, added, or removed after creation.

3 : Heterogeneous :

- A list can contain elements of different data types such as int, float and string.

4 : Dynamic Size :

- Size of the list can increase or decrease during execution.

Syntax

```
list_name = [value1, value2, value3]
```

Example

```
list1 = ["Hello", 7.5, 29, "abcd"]
print(list1)
print(list1[2])
```

Output

```
["Hello", 7.5, 29, "abcd"]
29
```

List (Sequences) in Python

1 : Creating and accessing lists :-

- List elements can be accessed using index operator [ ].
- Index starts from 0.
- Each element has a unique index.
- Index must be integer value.
- Out of range index gives error.

Example

```
list2 = ["Hello", 7.5, 29, "abcd", 786]
print(list2[3])
```

Output

```
abcd
```

Nested List

- A nested list is a list that contains another list as its element.
- Used for multi-dimensional data.
- Outer index → Inner index
- Accessed using multiple indexes.
- Helps in representing matrices.

Example

```
list1 = [[1,2,3],[4,5,6],[7,8,9]]
print(list1[0][1])
```

Output

```
2
```

Negative Indexing

- Negative indexing is used to access elements from the end of the list.
- Index -1 refers to last element.
- Index -2 refers to second last.
- Used for reverse accessing.

- Works from right to left.

Program

```
list1 = ["Hello", 7.5, 29, "abcd", 786]
print(list1[-1])
```

Output

```
786
```

---

2 : List Slicing

- List slicing is used to extract a portion of elements from a list.
- Uses colon (:) operator
- Start index is inclusive
- Stop index is exclusive
- Step is optional

Syntax

```
list[start : stop : step]
```

Program

```
list1 = ["Hello", 7.5, 29, "abcd", 786]
print(list1[2:4])
```

Output

```
[29, "abcd"]
```

---

3 : List Operations

(a) Concatenation (+)

- It is used to join two lists.

```
list1 = [1,2]
list2 = [3,4]
print(list1 + list2)
```

Output

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

(b) Repetition (*)

- It is used to repeat list elements.

```
list1 = [10,20]
print(list1 * 2)
```

Output

```
[10,20,10,20]
```

(c) Updation of List

- List elements can be modified using assignment operator.

```
list1 = ["Hello", 7.5, 29, "abcd"]
list1[3] = 999
print(list1)
```

Output

```
["Hello", 7.5, 29, 999]
```

(d) Deletion of List

- Elements can be deleted using del keyword.

```
list1 = ["Hello", 7.5, 29, "abcd"]
del list1[1]
print(list1)
```

Output

```
["Hello", 29, "abcd"]
```

List Functions

1 : len()

- This function returns the length of list.

```
list = ["Hello", 7.5]
len(list)
```

Output

```
2
```

---

2 : max()

- This function returns the elements from the python list with maximum value.

```
list = [45,67,89,83]
max(list)
```

Output

```
89
```

---

3 : min()

- This function returns the elements from the python list with minimum value.

```
list = [45,63,2]
min(list)
```

Output

```
2
```

---

4 : list()

- This function converts the sequence such as python tuple into python list.

```
tuple = ('a','z','b','y')
list1 = list(tuple)
list1
```

Output

```
['a','z','b','y']
```

List Methods

1 : append()

- The append() method adds a single item to the existing list.

Syntax

```
list.append(item)
```

Example

```
fruit = ['apple','mango','guava']
fruit.append('orange')
fruit
```

Output

```
['apple','mango','guava','orange']
```

---

2 : count()

- The method count() returns count of how many times the particular item occurs.

Syntax

```
list.count(item)
```

Example

```
fruit = ['apple','mango','guava','orange','mango']
fruit.count('mango')
```

Output

```
2
```

---

3 : extend()

- The method extend() appends the contents of sequence to list.

Syntax

```
list.extend(seq)
```

Example

```
fruit = ['apple','mango','guava']
veg = ['carrot','onion']
fruit.extend(veg)
fruit
```

Output

```
['apple','mango','guava','carrot','onion']
```

---

4 : index()

- The method index() returns the index value of the object.

Syntax

```
list.index(obj)
```

Program

```
fruit = ['apple','mango','guava','orange','mango']
fruit.index('orange')
```

Output

```
3
```

---

5 : insert()

- The method insert() inserts an object at the given index.

Syntax

```
list.insert(index, object)
fruit.insert(2,'orange')
fruit
```

Output

```
['apple','mango','orange','guava']
```

---

6 : pop()

- The method pop() removes and returns last object or particular object from the list.

Syntax

```
list.pop()
list.pop(index)
```

Example

```
fruit = ['apple','mango','guava','orange','mango']
fruit.pop()
```

Output

```
mango
```

---

7 : remove()

- This method removes the object from the given list.

Syntax

```
list.remove(obj)
```

Example

```
fruit = ['apple','mango','guava','orange']
fruit.remove('mango')
fruit
```

Output

```
['apple','guava','orange']
```

---

8 : sort()

- This method sorts the elements of the list.

Syntax

```
list.sort()
```

Program

```
list = [78,12,99,56,19]
list.sort()
list
```

Output

```
[12,19,56,78,99]
```

List Loops

- List looping in python is the process of iterating through each element of a list one by one using loop statements.
- It allows us to access, modify and perform operations on each item present in the list.

- Python provides the for loop to iterate over list elements.
- The loop variable automatically takes the value of each element during iteration.
- Loop continues execution until all elements in the list are processed.

Syntax

```
for variable in list_name:
    statements
```

Program

```
fruits = ["apple", "mango", "grapes", "orange"]

for i in fruits:
    print(i)
```

Output

```
apple
mango
```

```
grapes
orange
```

Range of length

- The range(len(list)) function is used to iterate through the list by using index values of elements instead of directly accessing the values.
- len() returns the total number of elements in the list.
- range() generates index values from 0 to length-1.
- Elements can be accessed using indexing inside loop.

Syntax

```
for i in range(len(list)):
    statements
```

Program

```
fruits = ["apple", "mango", "grapes", "orange"]

for i in range(len(fruits)):
    print(i, fruits[i])
```

Output

```
0 apple
1 mango
2 grapes
3 orange
```

Enumerate Function

- enumerate() is a built in python function used to iterate over a list while retrieving both index and value of each element.
- It returns index and elements together.
- Eliminates the need of using range().
- Makes code more readable and efficient.

Syntax

```
for index, value in enumerate(list):
```

Program

```
fruits = ["apple", "mango", "grapes", "orange"]

for i, name in enumerate(fruits):
    print(i, name)
```

Output

```
0 apple
1 mango
2 grapes
3 orange
```

Aliasing and Cloning of Lists

Aliasing

- Aliasing occurs when two or more variables refer to the same memory location of a list. Any modification made through one variable will reflect in the other.
- Assignment operator (=) creates aliasing.
- No new list is created in memory.
- Changes in one list affect the other list.

Program

```
list1 = [10,20,30,40,50]
list2 = list1

list1[1] = 100

print("List1:", list1)
print("List2:", list2)
```

Output

```
List1: [10,100,30,40,50]
List2: [10,100,30,40,50]
```

Cloning

- Cloning is the process of creating an exact copy of list in a new memory location so that changes in one list do not affect the other.
- Creates a new list in memory.
- Original list remains unchanged.
- Can be done using slicing or list() function.

Program

```
list1 = [10,20,30,40,50]
list2 = list1[:]

list1[2] = 200

print("Original List:", list1)
print("Cloned List:", list2)
```

Output

```
Original List: [10,20,200,40,50]
Cloned List: [10,20,30,40,50]
```

List Comprehension

- List comprehension is a concise method used to create a new list by applying an expression to each element of an existing list using a single line of code.
- Used to generate new lists from existing lists.
- Can include conditions for filtering elements.
- Makes code shorter and more efficient.

Syntax

```
new_list = [expression for item in list if condition]
```

Program

```
sample = [10,20,30,40]

cubes = [x**3 for x in sample]

print(cubes)
```

Output

```
[1000,8000,27000,64000]
```

Tuples

- A tuple in python is an ordered and immutable collection of elements enclosed in parentheses ( ).
- Immutable makes tuples useful for storing fixed collection of related data.

Features of Tuple

1 : Immutable

- After the creation of a tuple its elements can not be changed.
- This makes tuples useful for storing fixed data safely.

2 : Ordered Collection

- The elements of a tuple follow a specific order and can be accessed using indexing.

3 : Allow Duplicate Values

- Tuples can contain duplicate elements.

Heterogeneous data types

- A tuple can store elements of different data types such as integer, float, string etc.

Benefits of Tuple

- Tuples are immutable, so data remains secure and can not be accidentally modified.
- Tuples are faster than lists during execution.
- Tuples can be used as dictionary keys, whereas lists can not.

Creating Tuples

- A tuple is created by placing elements inside parentheses and separating them with commas.

Program

```
tup1 = (10,20,30,40,50)
tup2 = ('apple','mango','orange')

print(tup1)
print(tup2)
```

Output

```
(10,20,30,40,50)
('apple','mango','orange')
```

Accessing Tuple Elements

1 : Indexing

- Indexing is used to access elements of a tuple.
- Index value always starts from 0.

Program

```
tup = (1,3,5,7,9)

print(tup[2])
```

Output

```
5
```

---

2 : Slicing

- Slicing is used to access multiple elements of a tuple using start and end index values.

Program

```
tup = (1,3,5,7,9)

print(tup[:])
print(tup[1:4])
print(tup[-2:])
```

Output

```
(1,3,5,7,9)
(3,5,7)
(7,9)
```

Tuple Operations

1 : Concatenation (+)

- Concatenation is used to combine two tuples into a single tuple using (+) operator.

Example

```
t1 = (1,2,3)
t2 = (4,5)

t3 = t1 + t2

print(t3)
```

Output

```
(1,2,3,4,5)
```

---

2 : Repetition (*)

- Repetition operator * is used to repeat the elements of a tuple.

Program

```
t = (1,2,3)

print(t * 2)
```

Output

```
(1,2,3,1,2,3)
```

---

3 : Membership Test (in / not in)

- Membership operators are used to check whether an element is present in a tuple or not.

Program

```
tup = (10,20,30,40)

print(20 in tup)
print(50 in tup)
```

Output

```
True
False
```

Tuple Functions

1 : len()

- The len() function returns the total no. of elements present in a tuple.

Program

```
tup = (10,20,30,40,50)

print(len(tup))
```

Output

```
5
```

---

2 : max()

- The max() function returns the largest element from the tuple.

Program

```
tup = (45,67,89,34,73)

print(max(tup))
```

Output

```
89
```

---

3 : min()

- The min() function returns the smallest element from the tuple.

Program

```
tup = (45,67,89,34,73)

print(min(tup))
```

Output

```
34
```

---

4 : count()

- The count() function returns how many times a particular element occurs in the tuple.

Program

```
tup = (45,67,89,34,89)

print(tup.count(89))
```

Output

```
2
```

---

5 : index()

- The index() function returns the index position of the given element.

Program

```
tup = ("apple","mango","orange")

print(tup.index("mango"))
```

Output

```
1
```

Dictionary :-

- A dictionary in Python is a built in data type used to store data in the form of key value pairs.
- Dictionary is written inside curly braces { }.
- Each element consists of a key and value separated by a colon.
- Keys must be unique and immutable (string, number, tuple).
- Values can be of any data type.
- Dictionary is mutable, so values can be changed after creation.

Syntax

```
dict_name = { key : value }
```

Operating dictionary in Python

1 : Creating and Accessing dictionary

- Dictionary is created using { }.
- Values are accessed using keys inside square brackets [ ].
- If the key is not present, it gives an error.

Program

```
student = {
    "name" : "Sham",
    "class" : "BCA",
    "roll_no" : 358,
    "marks" : 78,
    "subject" : "math"
}

print(student)
print(student["name"])
```

Output

```
{'name' : 'Sham', 'class' : 'BCA', 'roll_no' : 358,
'marks' : 78, 'subject' : 'math'}
Sham
```

---

2 : Updating dictionary

- New key value pairs can be added.
- Existing values can be modified.
- Dictionary allows dynamic changes.

Program

```
student["marks"] = 89
student["grade"] = "A"
print(student)

student["marks"] = 95
student["grade"] = "A+"
print(student)
```

Output

```
{'marks' : 89, 'grade' : 'A'}
{'marks' : 95, 'grade' : 'A+'}
```

---

3 : Deleting dictionary

- Individual elements can be deleted using del.
- Entire dictionary can also be deleted.
- After deleting dictionary, it can not be accessed.

Program

```
student = {
    'name' : 'Sham',
    'class' : 'BCA',
    'roll_no' : 358,
    'subject' : 'math'
}

del student['subject']
print(student)
```

Output

```
{'name' : 'Sham', 'class' : 'BCA', 'roll_no' : 358}
```

Dictionary functions

1 : len()

- Returns the no. of items in dictionary.

Program

```
student = {
    'name' : 'Ram',
    'class' : 'BBA'
}

print(len(student))
```

Output

```
2
```

---

2 : str()

- Converts dictionary into a string.

Program

```
student = {
    'name' : 'Shamu',
    'class' : 'BBA',
    'roll_no' : 358,
    'subject' : 'math',
    'marks' : 78
}

print(str(student))
```

Output

```
{'class' : 'BBA', 'marks' : 78, 'name' : 'Shamu',
'roll_no' : 358, 'subject' : 'math'}
```

Dictionary methods

1 : keys()

- Returns all the keys of the dictionary.

Syntax

```
dict.keys()
```

Program

```
student = {"name" : "Tushar", "age" : 20}

print(student.keys())
```

Output

```
dict_keys(['name', 'age'])
```

---

2 : values()

- The method values() returns all the values of the dictionary.

Syntax

```
dict.values()
```

Program

```
student = {"name" : "Tushar", "age" : 20}

print(student.values())
```

Output

```
dict_values(['Tushar', 20])
```

---

3 : items()

- Returns all key value pairs as tuple.

Syntax

```
dict.items()
```

Program

```
student = {"name" : "Tushar", "age" : 20}

print(student.items())
```

Output

```
dict_items([('name', 'Tushar'), ('age', 20)])
```

---

4 : update (dictionary 2)

- The method update() adds items of dictionary 2 to first dictionary.

Syntax

```
dict1.update(dict2)
```

Program

```
dict1 = {'a' : 1, 'b' : 2}
dict2 = {'c' : 3}

dict1.update(dict2)
print(dict1)
```

Output

```
{'a' : 1, 'b' : 2, 'c' : 3}
```

---

5 : clear()

- Removes all elements from the dictionary.

Syntax

```
dict.clear()
```

Program

```
student = {'name' : 'Tushar', 'roll_no' : 18}

student.clear()
print(student)
```

Output

```
{}
```

---

6 : fromkeys(sequence)

- Creates a new dictionary using given keys and same value.

Syntax

```
dict.fromkeys(seq[,values])
```

Program

```
keys = ('a','b','c')
value = 0

d = dict.fromkeys(keys,value)
print(d)
```

Output

```
{'a' : 0, 'b' : 0, 'c' : 0}
```

---

7 : copy()

- Creates a copy of an existing dictionary.

Syntax

```
new_dict = dict.copy()
```

Program

```
d1 = {'a' : 1, 'b' : 2}
d2 = d1.copy()

print(d2)
```

Output

```
{'a' : 1, 'b' : 2}
```

Dictionaries and looping

- The use of for loop while accessing dictionaries is very similar to using this loop with lists.

---

1 : Looping through all keys in dictionary

- By default looping over a dictionary gives access to all the keys.

Program

```
student = {'name' : 'Ram', 'age' : 20}

for key in student.keys():
    print(key)
```

Output

```
name
age
```

---

2 : Looping through all values in a dictionary

- The dict.values() is used to iterate over values of a dictionary.

Program

```
student = {'name' : 'Ram', 'age' : 20}

for value in student.values():
    print(value)
```

Output

```
Ram
20
```

---

3 : Looping through key value pairs

- The dict.items() is used to iterate over both keys and values of a dictionary.

Program

```
student = {'name' : 'Ram', 'age' : 20}

for key, value in student.items():
    print(key, value)
```

Output

```
name Ram
age 20
```

---

4 : Looping through a dictionary in order

Program

```
student = {'b' : 2, 'a' : 1, 'c' : 3}

for key, value in sorted(student.items()):
    print(key, value)
```

Output

```
a 1
b 2
c 3
```

Set :-

- A set in Python is an unordered collection of unique elements.
- It does not allow duplicate values and its elements must be immutable.
- Set is mutable, which means we can add or remove elements.
- Set elements are unique (duplicate values are automatically removed).
- Set elements must be immutable (like numbers, strings, tuples).
- Sets are defined using curly braces { }.
- Sets are used to perform mathematical operations like union, intersection etc.

Example

```
S1 = {5,6,7,4,5,2,5}
S2 = {'H','E','L','L','O'}
S3 = {1.0,'Hello',(1,2,3)}

print(S1)
print(S2)
print(S3)
```

Output

```
{2,4,5,6,7}
{'L','E','O','H'}
{1.0,'Hello',(1,2,3)}
```

Operations on Set

(a) Adding Element → add()

- Used to add a single element to a set.

```
Syntax : set_name.add(value)
```

(b) Adding Multiple Elements → update()

- Used to add multiple elements at once.

```
Syntax : set_name.update(list / tuple)
```

(c) Removing Element → remove()

- Used to remove a specific element.

```
Syntax : set_name.remove(value)
```

Program

```
list1 = [1,3,5]
S = set(list1)
```

```
print(S)
S.add(2)
print(S)

S.update([6,7,8])
print(S)

S.remove(3)
print(S)
```

Output

```
{1,3,5}
{1,2,3,5}
{1,2,3,5,6,7,8}
{1,2,5,6,7,8}
```

Frozen set :-

- A frozen set is an immutable version of a set. Once it is created, elements can not be added or removed from it.
- Frozen set is immutable.
- We cannot add or remove elements after creation.
- It is created using frozen set() function.
- Used when data should remain fixed or constant.

Program

```
S1 = {10,20,30,40,50}
list1 = [1,2,3,4]
vowels = ('a','e','i','o','u')

fs1 = frozenset(S1)
fs2 = frozenset(list1)
fs3 = frozenset(vowels)

print(fs1)
print(fs2)
print(fs3)
```

Output

```
frozenset({40,50,10,20,30})
frozenset({1,2,3,4})
frozenset({'a','i','e','o','u'})
```