Manonmaniam Sundaranar University,
Directorate of Distance & Continuing Education,
Tirunelveli - 627 012 Tamilnadu, India

OPEN AND DISTANCE LEARNING(ODL) PROGRAMMES
(FOR THOSE WHO JOINED THE PROGRAMMES FROM THE ACADEMIC YEAR 2023–2024)

III YEAR

B.Sc. Physics

Course Material

Python Programming and Basics of AI & Data Science

Prepared

By

Dr. S. Shailajha
Dr. P. Hema
Assistant Professor
Department of Physics
Manonmaniam Sundaranar University
Tirunelveli – 12

PYTHON PROGRAMMING AND BASICS OF AI & DATA SCIENCE

UNIT I: BASICS

Python introduction – Tokens: literals, Variables, Reserved Words, Operators, Delimiters and Escape sequences – Standard Data Types – Expressions – Comments in Python – Input and Output functions – Simple Physics formula-based programming in Python.

UNIT II: CONTROL STATEMENTS

Control Flow Statements and Syntax with examples – Looping statements – String operations – LISTS: List – list slices – list Methods – list loop – Tuples assignment – sets – Dictionaries.

UNIT III: FUNCTIONS

Definition and types – Passing parameters to a Function – Scope – Type conversion – Passing Functions to a Function – Modules – Standard Modules – Inbuilt Function – Scope of Variables.

UNIT IV: OBJECT ORIENTED FEATURES

Introduction –Defining Classes – Public and Private Data member – Creating Object – Accessing class members – Using Objects. Constructors – Destructors – Introduction of simple Inheritance – Introduction of simple Polymorphism.

ERROR HANDLING: Run Time Errors – Exception Model.

UNIT V: ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

Introduction – History of AI – Applications of AI – Defining Algorithm – A* Algorithm.

DATA SCIENCE: Introduction – Defining Data, Information and Data structure – Basic Concept of Probability and Statistics.

TEXT BOOKS

1. Fundamental of Pythons – First Program by Kenneth A. Lambert.
2. Python Programming – A modular approach by Pearson – Sheetal Taneja.
3. Handson AI for beginners by Patric D. Smith.
4. Introduction to Data Science by Dr. Sushil Dohare, Dr. V. Selva Kumar Sachin Raval.

UNIT I: BASICS

Python introduction – Tokens: literals, Variables, Reserved Words, Operators, Delimiters and Escape sequences – Standard Data Types – Expressions – Comments in Python – Input and Output functions – Simple Physics formula-based programming in Python.

1.1 PYTHON INTRODUCTION

Python is a high-level scripting language which can be used for a wide variety of text processing, system administration and internet-related tasks. Unlike many similar languages, its core language is very small and easy to master, while allowing the addition of modules to perform a virtually limitless variety of tasks. Python is a true object-oriented language, and is available on a wide variety of platforms. There's even a python interpreter written entirely in Java, further enhancing python's position as an excellent solution for internet-based problems.

Python was developed in the early 1990's by Guido van Rossum, then at CWI in Amsterdam, and currently at CNRI in Virginia. In some ways, python grew out of a project to design a computer language which would be easy for beginners to learn, yet would be powerful enough for even advanced users. This heritage is reflected in python's small, clean syntax and the thoroughness of the implementation of ideas like object-oriented programming, without eliminating the ability to program in a more traditional style. So, python is an excellent choice as a first programming language without sacrificing the power and advanced capabilities that users will eventually need. Although pictures of snakes often appear on python books and websites, the name is derived from Guido van Rossum's favourite TV show, "Monty Python's Flying Circus". For this reason, lots of online and print documentation for the language has a light and humorous touch. Interestingly, many experienced programmers report that python has brought back a lot of the fun they used to have programming, so van Rossum's inspiration may be well expressed in the language itself.

PYTHON VERSIONS

- Python 1.0
- Python 2.0
- Python 3.0

PYTHON FEATURES

- Easy to learn, easy to read and easy to maintain.

- Portable: It can run on various hardware platforms and has the same interface on all platforms.
- Extendable: You can add low-level modules to the Python interpreter.
- Scalable: Python provides a good structure and support for large programs. Python has support for an interactive mode of testing and debugging.
- Python has a broad standard library cross-platform.
- Everything in Python is an object: variables, functions, even code. Every object has an ID, a type, and a value.
- Python provides interfaces to all major commercial databases.
- Python supports functional and structured programming methods as well as OOP.
- Python provides very high-level dynamic data types and supports dynamic type checking.
- Python supports GUI applications
- Python supports automatic garbage collection.
- Python can be easily integrated with C, C++, and Java.

APPLICATIONS OF PYTHON

- Machine Learning
- GUI Applications (like Kivy, Tkinter, PyQt etc.)
- Web frameworks like Django (used by YouTube, Instagram, Dropbox)
- Image processing (like OpenCV, Pillow)
- Web scraping (like Scrapy, BeautifulSoup, Selenium)
- Test frameworks
- Multimedia
- Scientific computing
- Text processing

1.2 TOKENS IN PYTHON

In Python, when you write a code, the interpreter needs to understand what each part of your code does. Tokens are the smallest units of code that have a specific purpose or meaning. Each token, like a keyword, variable name, or number, has a role in telling the computer what to do.

For Example, let's break down a simple Python code example into tokens:

```
# Example Python code
variable = 5 + 3
print ("Result: ", variable)
```

Now, let's identify the tokens in this code:

- Keywords (like 'if' or 'while') tell the computer about decision-making or loops.
- Variable names (identifiers) are like labels for storing information.
- Numbers and text (literals) represent actual values.
- Operators (like + or –) are symbols that perform actions on values.

When the interpreter reads and processes these tokens, it can understand the instructions in your code and carry out the intended actions. The combination of different tokens creates meaningful instructions for the computer to execute. Tokens are generated by the Python tokenizer, after reading the source code of a Python program. It breaks, the code into smaller parts. The tokenizer ignores whitespace and comments and returns a token sequence to the Python parser. The Python parser then uses the tokens to construct a parse tree, showing the program's structure. The parse tree is then used by the Python interpreter to execute the program.

Types of Tokens:

When working with the Python language, it is important to understand the different types of tokens that make up the language. Python has different types of Tokens, including literals, variables, reserved Words, operators, delimiters and escape sequences. Each token type fulfills a specific function and plays an important role in the execution of a Python script.

1.Literals:

Literals are constant values that are directly specified in the source code of a program. They represent fixed values that do not change during the execution of the program. Python supports various types of literals, including string literals, numeric literals, boolean literals, and special literals such as None.

Numeric literals can be integers, floats, or complex numbers. Integers are whole numbers without a fractional part, while floats are numbers with a decimal point. Complex numbers consist of a real part and an imaginary part, represented as "x + yj'', where "x" is the real part and "y" is the imaginary part.

String literals are sequences of characters enclosed in single quotes (") or double quotes (""). They can contain any printable characters, including letters, numbers, and special characters. Python also supports triple-quoted strings, which can span multiple lines and are often used for docstrings, multi-line comments, or multi-line strings.

Boolean literals represent the truth values "True" and "False''. They are used in logical expressions and control flow statements to make decisions based on certain conditions. Boolean literals are often the result of comparison or logical operations.

Special literals include None, which represents the absence of a value or the null value. It is often used to indicate that a variable has not been assigned a value yet or that a function does not return anything.

2. Variables:

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.

Rules for Python variables:

- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Variable names are case-sensitive (age, Age and AGE are three different 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 −

```
a= 100              # An integer assignment
b = 1000.0          # A floating point
c = "John"          # A string
print (a)
print (b)
print (c)
```

This produces the following result −

```
100
1000.0
John
```

Multiple Assignment:

Python allows you to assign a single value to several variables simultaneously.

For example:

a = b = c = 1

Here, an integer object is created with the value 1, and all three variables are assigned to the same memory location. You can also assign multiple objects to multiple variables.

For example −

a, b, c = 1,2,"mrcet “

To combine both text and a variable, Python uses the “+” character:

Example

x = "awesome" print ("Python is " + x)

Output

Python is awesome

You can also use the + character to add a variable to another variable:

Example

x = "Python is " y = "awesome" z = x + y print(z)

Output:

Python is awesome

3. Reserved Words: (Keywords)

Keywords are reserved words in Python that have a special meaning and are used to define the syntax and structure of the language. These words cannot be used as identifiers for variables, functions, or other objects. Python has a set of 35 keywords, each serving a specific purpose in the language.

There are 35 keywords in Python 3.11. They are:

and	as	assert	async	continue
else	if	not	while	def
except	import	or	with	del
finally	in	pass	yield	elif
for	is	raise	await	false
from	lambda	return	break	none
global	nonlocal	try	class	true

4. Operators:

Operators are like little helpers in Python, using symbols or special characters to carry out tasks on one or more operands. Python is generous with its operators, offering a diverse set. These include the everyday arithmetic operators, those for assignments, comparison operators, logical operators, identity operators, membership operators, and even those for handling bits.

Type of Operator	Description	Example
Arithmetic Operators	Perform mathematical operations such as addition, subtraction, multiplication, division, modulus, and exponentiation.	+, -, *, /, %, **
Assignment Operators	Assign values to variables, including the equal sign and compound assignment operators.	=, +=, -=, *=, /=, %=
Comparison Operators	Compare two values and return a boolean (True or False) based on the comparison.	==, !=, >, <, >=, <=
Logical Operators	Combine conditions and perform logical operations like AND, OR, and NOT.	and, or, not
Identity Operators	Compare the memory addresses of objects to check if they are the same or different.	is, is not
Membership Operators	Test if a value is present in a sequence (e.g., list, tuple, string).	in, not in
Bitwise Operators	Perform bit-level operations on binary numbers, allowing manipulation of individual bits.	&

5.Delimiters:

Delimiters are characters or symbols used to separate or mark the boundaries of different elements in Python code. They are used to group statements, define function or class bodies, enclose string literals, and more. Python uses various delimiters, including parentheses '()', commas ',', brackets '[]', braces '{}', colons ':', and semicolons.

Punctuation Mark	Usage
Parentheses	Define function arguments, control the order of operations, and create tuples.
Brackets	Create lists, which are mutable sequences of values.
Braces	Define sets (unordered collections of unique elements) and dictionaries (key-value pairs).
Commas	Separate elements in tuples, lists, sets, and dictionaries. It is also used to separate function arguments and create multiple variable assignments.
Colons	Define the body of control flow statements like if, else, for, while, and def.
Semicolons	Separate multiple statements on a single line for brevity or to combine related statements.

6. Escape sequences:

Specifying a backslash (\) in front of the quote character in a string“escapes” it and causes Python to suppress its usual special meaning. It is then interpreted simply as a literal single quote character:

```
>>> print("mrcet is an autonomous (\') college")
mrcet is an autonomous (') college
>>> print('mrcet is an autonomous (\") college')
mrcet is an autonomous (") college
```

The following is a table of escape sequences which cause Python to suppress the usual special interpretation of a character in a string:

```
>>> print('a\
....b')
a....b
>>> print('a\
24
b\
```

c')
abc
>>> print('a \n b')
a
b

>>> print("mrcet \n college")
mrcet

college Escape Sequence	Usual Interpretation of Character(s) After Backslash	"Escaped" Interpretation
\'	Terminates string with single quote opening delimiter	Literal single quote (') character
\"	Terminates string with double quote opening delimiter	Literal double quote (") character
\newline	Terminates input line	Newline is ignored
\\	Introduces escape sequence	Literal backslash (\) character

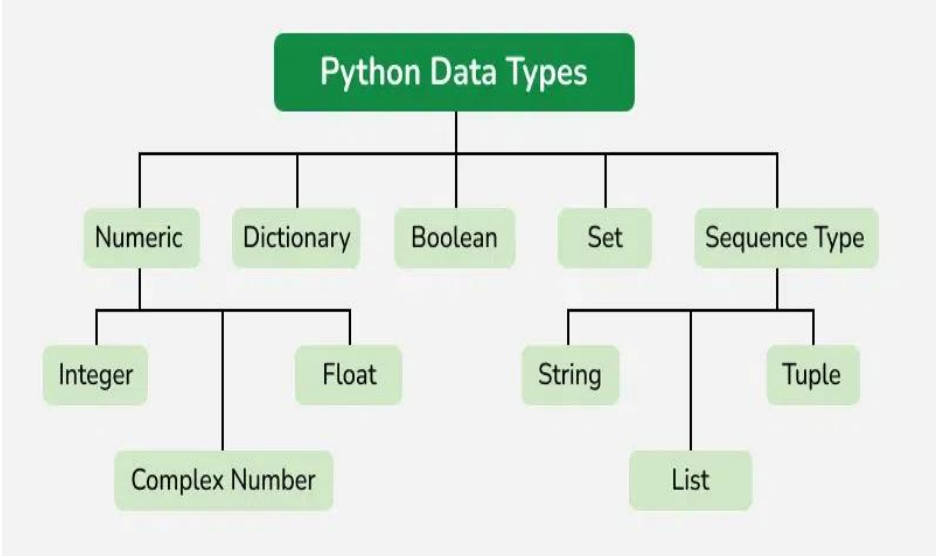
In Python (and almost all other common computer languages), a tab character can be specified by the escape sequence \t:
>>> print("a\tb")
a b

1.3. STANDARD DATA TYPES:

Data types in Python are a way to classify data items. They represent the kind of value, which determines what operations can be performed on that data. Since everything is an object in Python programming, Python data types are classes and variables are instances (objects) of these classes.

The following are standard or built-in data types in Python:

- Numeric: int, float, complex
- Sequence Type: string, list, tuple
- Mapping Type: dict
- Boolean: bool
- Set Type: set, frozenset
- Binary Types: bytes, bytearray, memoryview



1. Numeric Data Types

Python numbers represent data that has a numeric value. A numeric value can be an integer, a floating number or even a complex number. These values are defined as int, float and complex classes.

- Integers: value is represented by int class. It contains positive or negative whole numbers (without fractions or decimals). There is no limit to how long an integer value can be.
- Float: value is represented by float class. It is a real number with a floating-point representation. It is specified by a decimal point. Optionally, character e or E followed by a positive or negative integer may be appended to specify scientific notation.
- Complex Numbers: It is represented by a complex class. It is specified as (real part) + (imaginary part)j. For example - 2+3j

```
a = 5
print(type(a))

b = 5.0
```

```
print(type(b))

c = 2 + 4j
print(type(c))
```

Output

```
<class 'int'>
<class 'float'>
<class 'complex'>
```

2. Sequence Data Types

A sequence is an ordered collection of items, which can be of similar or different data types. Sequences allow storing of multiple values in an organized and efficient fashion. There are several sequence data types of Python:

String Data Type:

Python Strings are arrays of bytes representing Unicode characters. In Python, there is no character data type, a character is a string of length one. It is represented by str class. Strings in Python can be created using single quotes, double quotes or even triple quotes. We can access individual characters of a String using index.

```
s = 'Welcome to the Geeks World'
print(s)

# check data type
print(type(s))

# access string with index
print(s[1])
print(s[2])
print(s[-1])
```

Output

```
Welcome to the Geeks World
<class 'str'>
e
l
d
```

List Data Type:

Lists are similar to arrays found in other languages. They are an ordered and mutable collection of items. It is very flexible as items in a list do not need to be of the same type.

Creating a List in Python:

Lists in Python can be created by just placing sequence inside the square brackets[].

```
# Empty list
```

```
a = []

# list with int values
a = [1, 2, 3]
print(a)

# list with mixed values int and String
b = ["Geeks", "For", "Geeks", 4, 5]
print(b)
```

Output

```
[1, 2, 3]
['Geeks', 'For', 'Geeks', 4, 5]
```

Access List Items:

In order to access the list items refer to index number. In Python, negative sequence indexes represent positions from end of the array. Instead of having to compute offset as in List[len(List)-3], it is enough to just write List[-3]. Negative indexing means beginning from end, -1 refers to last item, -2 refers to second-last item, etc.

```
a = ["Geeks", "For", "Geeks"]
print("Accessing element from the list")
print(a[0])
print(a[2])

print("Accessing element using negative indexing")
print(a[-1])
print(a[-3])
```

Output

```
Accessing element from the list
Geeks
Geeks
Accessing element using negative indexing
Geeks
Geeks
```

Tuple Data Type:

Tuple is an ordered collection of Python objects. The only difference between a tuple and a list is that tuples are immutable. Tuples cannot be modified after it is created.

Creating a Tuple in Python:

In Python, tuples are created by placing a sequence of values separated by a 'comma' with or without the use of parentheses for grouping data sequence. Tuples can contain any number of elements and of any datatype (like strings, integers, lists, etc.).

```
# initiate empty tuple
```

```
tup1 = ()
tup2 = ('Geeks', 'For')
print("\nTuple with the use of String: ", tup2)
```

Output
Tuple with the use of String: ('Geeks', 'For')

Access Tuple Items:
In order to access tuple items refer to the index number. Use the index operator [ ] to access an item in a tuple.

```
tup1 = (1, 2, 3, 4, 5)

# access tuple items
print(tup1[0])
print(tup1[-1])
print(tup1[-3])
```

Output
1
5
3

3. Boolean Data Type:

Python Boolean Data type is one of the two built-in values, True or False. Boolean objects that are equal to True are truthy (true) and those equal to False are falsy (false). However non-Boolean objects can be evaluated in a Boolean context as well and determined to be true or false. It is denoted by class bool.

```
print(type(True))
print(type(False))
print(type(true))
```

Output
<class'bool'>
<class 'bool'>
Traceback(mostrecentcalllast):
File"/home/7e8862763fb66153d70824099d4f5fb7.py",line8,in
print(type(true))
NameError: name 'true' is not defined

Truthy and Falsy Values:

In Python, truthy and falsy values are values that evaluate to True or False in a Boolean context. Truthy values behave like True, while falsy values behave like False when used in conditions.

```
if 1:
   print("1 is truthy")
if not 0:
   print("0 is falsy")
```

Output

```
1 is truthy
0 is falsy
```

4. Set Data Type:

In Python Data Types, Set is an unordered collection of data types that is iterable, mutable, and has no duplicate elements. The order of elements in a set is undefined though it may consist of various elements.

Create a Set in Python:

Sets can be created by using the built-in set() function with an iterable object or a sequence by placing the sequence inside curly braces, separated by a 'comma'. The type of elements in a set need not be the same, various mixed-up data type values can also be passed to the set.

```
# initializing empty set
s1 = set()

s1 = set("GeeksForGeeks")
print("Set with the use of String: ", s1)

s2 = set(["Geeks", "For", "Geeks"])
print("Set with the use of List: ", s2)
```

Output

```
Set with the use of String:  {'s', 'o', 'F', 'G', 'e', 'k', 'r'}
Set with the use of List:  {'Geeks', 'For'}
```

Access Set Items:

Set items cannot be accessed by referring to an index, since sets are unordered the items have no index. But we can loop through the set items using a for loop, or ask if a specified value is present in a set, by using the keyword in.

```
set1 = set(["Geeks", "For", "Geeks"]) #Duplicates are removed automatically
print(set1)

# loop through set
for i in set1:
  print(i, end=" ") #prints elements one by one
```

```
# check if item exist in set
print("Geeks" in set1)
```

Output
{'For', 'Geeks'}
For Geeks True

5. Dictionary Data Type:

A dictionary in Python is a collection of data values, used to store data values like a map, unlike other Python Data Types, a Dictionary holds a key: value pair. Key-value is provided in dictionary to make it more optimized. Each key-value pair in a Dictionary is separated by a colon : , whereas each key is separated by a 'comma'.

Create a Dictionary in Python:

Values in a dictionary can be of any datatype and can be duplicated, whereas keys can't be repeated and must be immutable. The dictionary can also be created by the built-in function dict().

```
# initialize empty dictionary
d = {}

d = {1: 'Geeks', 2: 'For', 3: 'Geeks'}
print(d)

# creating dictionary using dict() constructor
d1 = dict({1: 'Geeks', 2: 'For', 3: 'Geeks'})
print(d1)
```

Output
{1: 'Geeks', 2: 'For', 3: 'Geeks'}
{1: 'Geeks', 2: 'For', 3: 'Geeks'}

Accessing Key-value in Dictionary:

In order to access items of a dictionary refer to its key name. Key can be used inside square brackets. Using get() method we can access dictionary elements.

```
d = {1: 'Geeks', 'name': 'For', 3: 'Geeks'}

# Accessing an element using key
print(d['name'])

# Accessing a element using get
print(d.get(3))
```

Output
For
Geeks

1.4. EXPRESSIONS:

An expression is a combination of values, variables, and operators. If you type an expression on the command line, the interpreter evaluates it and displays the result:

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

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

Confusingly, evaluating an expression is not quite the same thing as printing a value.

```
>>> message = "What's your name?"
>>> message
"What's your name?"
>>> print(message)
What's your name?
```

When the Python shell displays the value of an expression, it uses the same format you would use to enter a value. In the case of strings, that means that it includes the quotation marks. But the print statement prints the value of the expression, which in this case is the contents of the string. In a script, an expression all by itself is a legal statement, but it doesn't do anything. The script

```
17
3.2
"Hello, World!" 1 + 1
```

produces no output at all.

1.5. PYTHON COMMENTS:

Comments in Python are the lines in the code that are ignored by the interpreter during the execution of the program.

- Comments enhance the readability of the code.
- Comment can be used to identify functionality or structure the code-base.
- Comment can help understanding unusual or tricky scenarios handled by the code to prevent accidental removal or changes.
- Comments can be used to prevent executing any specific part of your code, while

print("Python Comments") """

Comments in Python

In Python, single line comments start with hashtag symbol #.

 # sample comment

name = "geeksforgeeks"

print(name)

Output

geeksforgeeks

Multi-Line Comments

Python does not provide the option for multiline comments. However, there are different ways through which we can write multiline comments.

Multiline comments using multiple hashtags (#)

We can multiple hashtags (#) to write multiline comments in Python. Each and every line will be considered as a single-line comment.

 # Python program to demonstrate

 # multiline comments

print("Multiline comments")

Using String Literals as Comment

Python ignores the string literals that are not assigned to a variable. So, we can use these string literals as Python Comments.

'Single-line comments using string literals'

""" Python program to demonstrate

multiline comments"""

print("Multiline comments")

Best Practice to Write Comments

These are some of the tips you can follow, to make your comments effective are:

1. Comments should be short and precise.
2. Use comments only, when necessary, don't clutter your code with comments.
3. Avoid writing generic or basic comments.
4. Write comments that are self-explanatory.

1.6. INPUT AND OUTPUT FUNCTIONS:

Understanding input and output operations is fundamental to Python programming. With the print() function, we can display output in various formats, while the input() function enables interaction with users by gathering input during program execution.

Taking input in Python

Python's input() function is used to take user input. By default, it returns the user input in form of a string.

Example:

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

Output

```
Enteryourname:GeeksforGeeks
Hello, GeeksforGeeks ! Welcome!
```

The code prompts the user to input their name, stores it in the variable "name" and then prints a greeting message addressing the user by their entered name.

Printing Output using print() in Python

At its core, printing output in Python is straightforward, thanks to the print() function. This function allows us to display text, variables and expressions on the console. Let's begin with the basic usage of the print() function:

In this example, "Hello, World!" is a string literal enclosed within double quotes. When executed, this statement will output the text to the console.

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

Output:

Hello, World!

Printing Variables

We can use the print() function to print single and multiple variables. We can print multiple variables by separating them with commas. Example:

```
s = "Bob"
print(s)
```

```
s = "Alice"
age = 25
city = "New York"
print(s, age, city)
```

Output:

Bob

Alice 25 New York

Take Multiple Input in Python

We are taking multiple input from the user in a single line, splitting the values entered by the user into separate variables for each value using the split() method. Then, it prints the values with corresponding labels, either two or three, based on the number of inputs provided by the user.

```
x, y = input("Enter two values: ").split()
print("Number of boys: ", x)
print("Number of girls: ", y)
 x, y, z = input("Enter three values: ").split()
print("Total number of students: ", x)
print("Number of boys is : ", y)
print("Number of girls is : ", z)
```

Output

```
Enter two values:5   10
Number of boys: 5
Number of girls: 10
Enter three values: 5 10 15
Total number of students: 5
Number of boys is : 10
Number of girls is : 15
```

Change the Type of Input in Python

By default input() function helps in taking user input as string. If any user wants to take input

as int or float, we just need to typecast it.

Print Names in Python

The code prompts the user to input a string (the color of a rose), assigns it to the variable color and then prints the inputted color.

```
color = input("What color is rose?: ")
print(color)
```

Output

What color is rose?: Red

Red

Print Numbers in Python

The code prompts the user to input an integer representing the number of roses, converts the input to an integer using typecasting and then prints the integer value.

```
n = int(input("How many roses?: "))
print(n)
```

Output

How many roses?: 88

88

Print Float or Decimal Number in Python

The code prompts the user to input the price of each rose as a floating-point number, converts the input to a float using typecasting and then prints the price.

```
price = float(input("Price of each rose?: "))
print(price)
```

Output

Price of each rose?: 50.3050.3

50.3050.3

Find DataType of Input in Python

In the given example, we are printing the type of variable x. We will determine the type of an object in Python.

```
a = "Hello World"
b = 10
```

```
c = 11.22
d = ("Geeks", "for", "Geeks")
e = ["Geeks", "for", "Geeks"]
f = {"Geeks": 1, "for":2, "Geeks":3}
print(type(a))
print(type(b))
print(type(c))
print(type(d))
print(type(e))
print(type(f))
```

Output

```
<class 'str'>
<class 'int'>
<class 'float'>
<class 'tuple'>
<class 'list'>
<class 'dict'>
```

1.7. SIMPLE PHYSICS FORMULA-BASED PROGRAMMING IN PYTHON:

Here we can find the acceleration (a), final velocity(v), initial velocity(u) and time(t) using the formula a = (v-u)/t.

At first, functions are defined for all four types of calculations, in which they will accept three inputs and assign the value in three different variables. Then the fourth value is calculated using the acceleration formula and the calculated value is returned. We are going to use the same acceleration formula in different approaches.

Approach:

- In the first approach, we will find initial velocity by using the formula **"u = (v-a*t)"**
- In the second approach, we will find final velocity by using formula **"v = u + a*t"**
- In the third approach, we will find acceleration by using formula **"a = (v - u)/t"**
- In the fourth approach, we will find time by using formula **"t = (v - v)/a"**

Example 1: Initial velocity (u) is calculated.

```
# code
# Enter final velocity in m/s:
finalVelocity = 10
# Enter acceleration in m per second square
acceleration = 9.8
#Enter time taken in second
time = 1
initialVelocity = finalVelocity - acceleration * time
print("Initial velocity = ", initialVelocity)
```

Output:

Initial velocity = 0.1999999999999993

Example 2: Final velocity (v) is calculated.

```
# code
# initial velocity in m/s:
initialVelocity = 10

# acceleration in m per second square
acceleration = 9.8

# time taken in second
time = 1
finalVelocity = initialVelocity + acceleration * time
print("Final velocity = ", finalVelocity)
```

Output:

Final velocity = 19.8

Example 3: Acceleration (a) is calculated.

```
#code
# initial velocity in m/s
initialVelocity = 0
```

```
# final velocity in m/s
finalVelocity = 9.8
# time in second
time = 1
acceleration = (finalVelocity - initialVelocity) / time
print("Acceleration = ", acceleration)
```

Output:

```
Acceleration =  9.8
```

Example 4: Time (t) is calculated.

```
# code
#final velocity in m/s
finalVelocity = 10

#initial velocity in m/s
initialVelocity = 0
#acceleration in meter per second square
acceleration = 9.8
time = (finalVelocity - initialVelocity) / acceleration
print("Time taken = ", time)
```

Output:

```
Time taken =  1.0204081632653061
```

Problem:

Questions based on the equation, v = u + at

A cyclist speeds up at a constant rate from rest to 8 m/s in 6s. Find the acceleration of the cyclist.

Solution:

u = 0 m/s [body starts from rest]

v = 8 m/s

t = 6 s

We know: v = u + at

=> a = (v-u)/t = (8–0)/6 = 4/3 m/s2 = 1.33 m/s²

Code

```
while True:
    u = float(input('Enter Initial Velocity of cyclist: '))
    v = float(input('Enter final Velocity of cyclist: '))
    t = float(input('Enter elapsed time in second spent by cyclist: '))

    # Find the acceleration of the cyclist.
    a = (v-u) / t
    print('The acceleration of the cyclist: %.2f ' %a)
    print('\n')

```

Output:

```
Enter Initial Velocity of cyclist: 0
Enter final Velocity of cyclist: 8
Enter elapsed time in second spent by cyclist: 6
The acceleration of the cyclist: 1.33
```

Explanation of the above code

In this code, we are trying to find the acceleration of a cyclist. Let me break it down step by step:

1. The code starts with a while True: loop, which means it will keep running forever until we stop it manually. Inside this loop, we will ask the user to input three values:
2. a. u: This is the initial velocity of the cyclist. b. v: This is the final velocity of the cyclist. c. t: This is the elapsed time in seconds spent by the cyclist.
3. After getting the input values from the user, the code calculates the acceleration of the cyclist using the formula:
4. acceleration (a) = (final velocity (v) - initial velocity (u)) / elapsed time (t)
5. Then, it prints the calculated acceleration with two decimal places using the line:

6. Print ('The acceleration of the cyclist: %.2f ' %a)
7. The code then adds an extra newline for better formatting using:
8. print('\n')
9. After displaying the result, the loop restarts, and the process repeats, asking the user for new values to calculate the acceleration again.

This code allows you to find the acceleration of the cyclist by providing the initial velocity, final velocity, and elapsed time. Remember that acceleration measures how fast the velocity of an object changes over time.

UNIT II: CONTROL STATEMENTS

Control Flow Statements and Syntax with examples – Looping statements – String operations – LISTS: List – list slices – list Methods – list loop – Tuples assignment – sets – Dictionaries.

2.1. CONTROL FLOW STATEMENTS:

A statement is an instruction that the Python interpreter can execute. We have normally two basic statements, the assignment statement and the print statement. Some other kinds of statements that are if statements, while statements, and for statements generally called as control flows.

Control Flow Statements in Python are fundamental building blocks that dictate the execution order of a program. They enable developers to create logical pathways and make decisions in their code, using structures like if, for, and while.

These statements are essential for adding complexity and functionality to Python scripts, allowing for conditional execution and repetitive tasks. This introduction will briefly explore how these control flow mechanisms enhance the versatility and efficiency of Python programming.

The flow control statements are divided into three categories

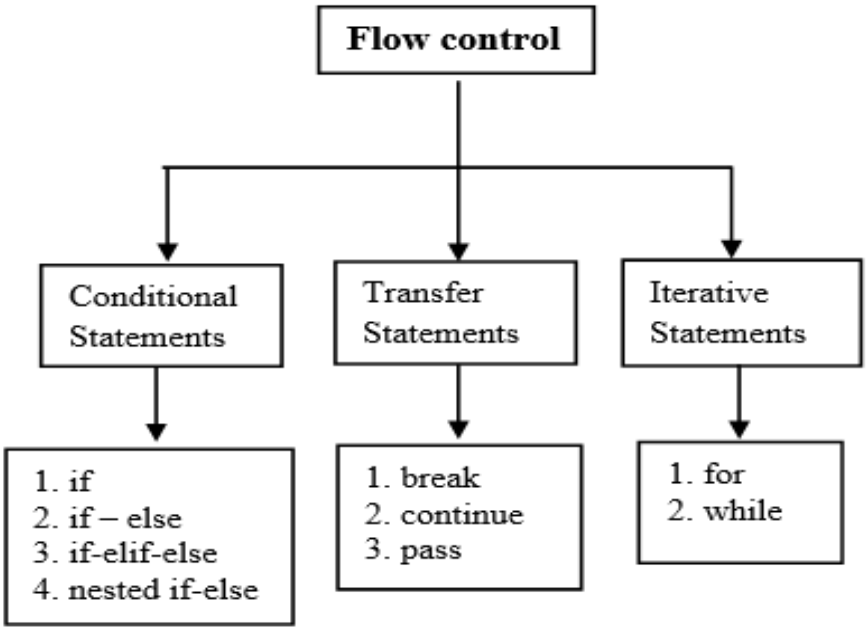
1. Conditional statements
2. Iterative statements.
3. Transfer statements

Conditional statements:

In Python, condition statements act depending on whether a given condition is true or false. You can execute different blocks of codes depending on the outcome of a condition. Condition statements always evaluate to either True or False.

There are three types of conditional statements.

1. if statement
2. if-else
3. if-elif-else
4. nested if-else



Python control flow statements

Iterative statements

In Python, iterative statements allow us to execute a block of code repeatedly as long as the condition is True. We also call it a loop statements. Python provides us the following two loop statement to perform some actions repeatedly

1. for loop
2. while loop

Transfer statements

In Python, transfer statements are used to alter the program's way of execution in a certain manner. For this purpose, we use three types of transfer statements.

1. break statement
2. continue statement
3. pass statements

If statement in Python

In control statements, The if statement is the simplest form. It takes a condition and evaluates to either True or False. If the condition is True, then the True block of code will be executed, and if the condition is False, then the block of code is skipped, and The controller moves to

the next line

Syntax of the if statement

```
if condition:
statement 1
statement 2
statement n
```

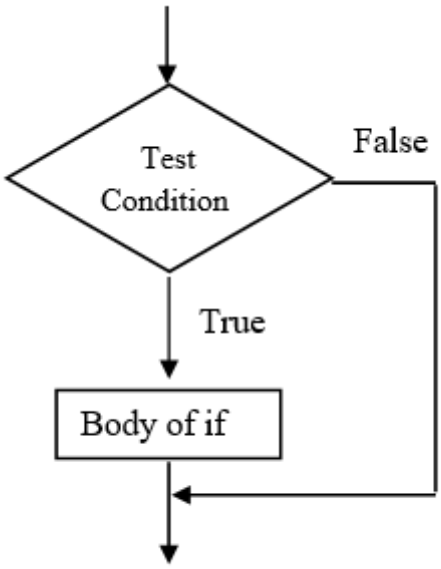


Fig. Flowchart of if statement

Example

```
number = 6
if number > 5:
    # Calculate square
    print(number * number)
print('Next lines of code')
```

Run

Output

```
36
Next lines of code
```

If – else statement

The if-else statement checks the condition and executes the if block of code when the condition is True, and if the condition is False, it will execute the else block of code.

Syntax of the if-else statement

```
if condition:
    statement 1
else:
    statement 2
```

If the condition is True, then statement 1 will be executed If the condition is False, statement 2 will be executed. See the following flowchart for more detail.

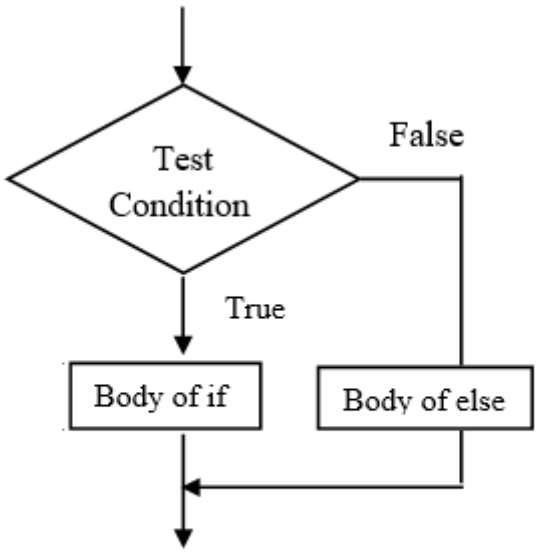


Fig. Flowchart of if-else

Example

```
password = input('Enter password ')
if password == "PYnative@#29":
 print("Correct password")
else:
    print("Incorrect Password")
```

Output 1:

Enter password PYnative@#29

Correct password

Output 2:

Enter password PYnative

Incorrect Password

Chain multiple if statement in Python

In Python, the if-elif-else condition statement has an elif blocks to chain multiple conditions one after another. This is useful when you need to check multiple conditions.

With the help of if-elif-else we can make a tricky decision. The elif statement checks multiple conditions one by one and if the condition fulfills, then executes that code.

Syntax of the if-elif-else statement:

```
if condition-1:
    statement 1
elif condition-2:
    stetement 2
elif condition-3:
    stetement 3
    ...
else:
    statement
```

Example

```
def user_check(choice):
   if choice == 1:
      print("Admin")
   elif choice == 2:
      print("Editor")
   elif choice == 3:
      print("Guest")
```

```
    else:
        print("Wrong entry")
user_check(1)
user_check(2)
user_check(3)
user_check(4)
```

Output:

```
Admin
Editor
Guest
Wrong entry
```

Nested if-else statement

In Python, the nested if-else statement is an if statement inside another if-else statement. It is allowed in Python to put any number of if statements in another if statement. Indentation is the only way to differentiate the level of nesting. The nested if-else is useful when we want to make a series of decisions.

Syntax of the nested-if-else:

```
if conditon_outer:
    if condition_inner:
        statement of inner if
    else:
        statement of inner else:
    statement ot outer if
else:
    Outer else
statement outside if block
```

Example: Find a greater number between two numbers

```
num1 = int(input('Enter first number '))
num2 = int(input('Enter second number '))
```

```
if num1 >= num2:
    if num1 == num2:
        print(num1, 'and', num2, 'are equal')
    else:
        print(num1, 'is greater than', num2)
else:
    print(num1, 'is smaller than', num2)
```

Output 1:

Enter first number 56

Enter second number 15

56 is greater than 15

Output 2:

Enter first number 29

Enter second number 78

29 is smaller than 78

Single statement suites

Whenever we write a block of code with multiple if statements, indentation plays an important role. But sometimes, there is a situation where the block contains only a single line statement. Instead of writing a block after the colon, we can write a statement immediately after the colon.

Example

```
number = 56
if number > 0: print("positive")
else: print("negative")
```

Similar to the if statement, while loop also consists of a single statement, we can place that statement on the same line.

Example

```
x = 1
while x <= 5: print(x,end=" "); x = x+1
```

Output

1 2 3 4 5

for loop in Python

Using for loop, we can iterate any sequence or iterable variable. The sequence can be string, list, dictionary, set, or tuple.

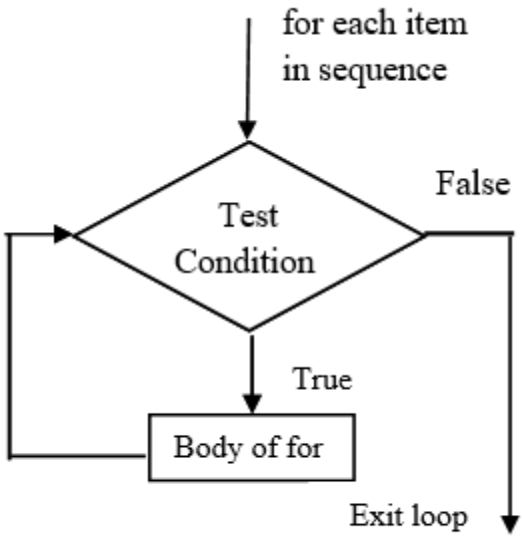


Fig. Flowchart of for loop

Syntax of for loop:

for element in sequence:

body of for loop

Example to display first ten numbers using for loop

```
for i in range(1, 11):
    print(i)
```

Run

Output

1

2

3

4

5

6

7

8

9

10

While loop in Python

In Python, the while loop statement repeatedly executes a code block while a particular condition is true. In a while-loop, every time the condition is checked at the beginning of the loop, and if it is true, then the loop's body gets executed. When the condition became False, the controller comes out of the block.

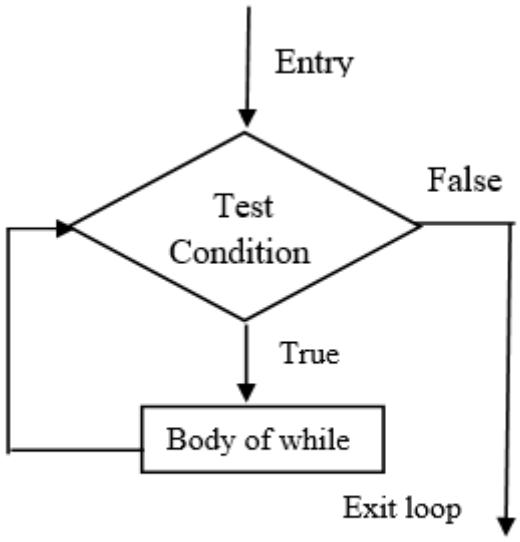


Fig. Flowchart of while loop

Syntax of while-loop

```
while condition :
    body of while loop
```

Example to calculate the sum of first ten numbers

```
num = 10
sum = 0
```

```
i = 1
while i <= num:
    sum = sum + i
    i = i + 1
print("Sum of first 10 number is:", sum)
```

Run

Output

```
Sum of first 10 number is: 55
```

Break Statement in Python

The break statement is used inside the loop to exit out of the loop. It is useful when we want to terminate the loop as soon as the condition is fulfilled instead of doing the remaining iterations. It reduces execution time. Whenever the controller encountered a break statement, it comes out of that loop immediately Let's see how to break a for a loop when we found a number greater than 5.

Example of using a break statement

```
for num in range(10):
    if num > 5:
        print("stop processing.")
        break
    print(num)
```

Run

Output

```
0
1
2
3
4
5
stop processing.
```

Continue statement in python

The continue statement is used to skip the current iteration and continue with the next iteration.

Let's see how to skip a for a loop iteration if the number is 5 and continue executing the body of the loop for other numbers.

Example of a continue statement

```
for num in range(3, 8):
   if num == 5:
      continue
   else:
      print(num)
```

Run

Output

3

4

6

7

Pass statement in Python

The pass is the keyword In Python, which won't do anything. Sometimes there is a situation in programming where we need to define a syntactically empty block. We can define that block with the pass keyword.

A pass statement is a Python null statement. When the interpreter finds a pass statement in the program, it returns no operation. Nothing happens when the pass statement is executed.

It is useful in a situation where we are implementing new methods or also in exception handling. It plays a role like a placeholder.

Example

```
months = ['January', 'June', 'March', 'April']
for mon in months:
   pass
print(months)
```

Run

Output

['January', 'June', 'March', 'April']

2.2. LOOPING STATEMENTS:

In Python, looping statements are control structures that allow a block of code to run repeatedly, either for a specific number of times or until a certain condition is met.

Types of Loops in Python:

Python gives us two primary ways to repeat actions each with its own style and use case. Let's break them down:

1. The for Loop

The for loop in Python is used when you want to repeat a block of code for each item in a sequence (like a list, string, or range of numbers). Unlike some other languages, Python's for loop is more like a "for-each" loop—it directly iterates over items.

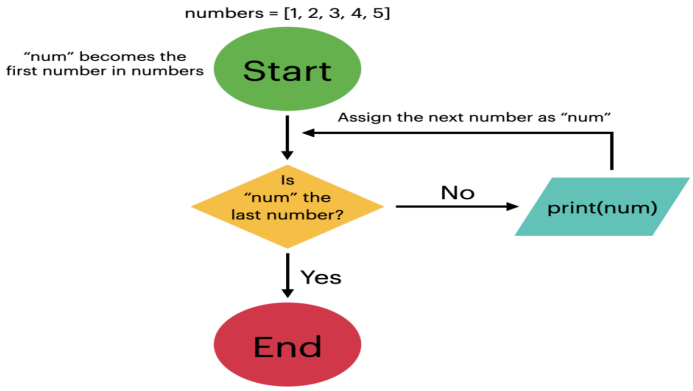
- Best used when you know in advance how many times you want to repeat something, or when you're iterating over a collection (like a list, string, or dictionary).
- Example: Looping through a list of names or printing numbers from 1 to 10.

Syntax:

for variable in sequence:

code to be executed

- variable → takes each value from the sequence one by one.
- sequence → can be a list, tuple, string, or even a range()



Sample code: Loop through a list

```
# Sample code: Loop through a list
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print("I like", fruit)
```

Expected Output:

I like apple

I like banana

I like cherry

The range() Function

Most of the time, you'll use range() with a for loop when you want to repeat something a specific number of times.

2. The while Loop:

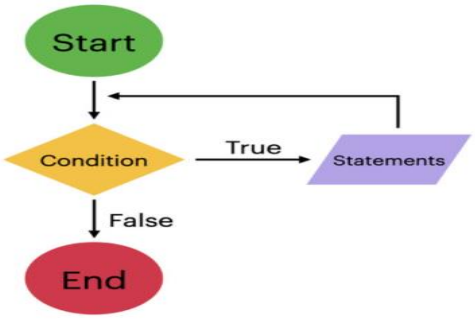
A while loop is used when you want to keep executing a block of code as long as a certain condition is true. Unlike the for loop, which runs for a fixed number of iterations, the while loop is controlled by a condition.

- Best used when you don't know exactly how many times the code should run, but you do know the condition that controls when it should stop.
- Example: Asking a user for input until they type the correct password.

Syntax:

```
while condition:
# code to be executed
```

- The loop continues to run as long as the condition evaluates to True.
- If the condition becomes False, the loop stops.



Sample Code: User Input Until Correct Password

```
# Sample code: While loop for password checking
password = ""
while password != "python123":
    password = input("Enter password: ")
print("Access granted!")
```

Expected Output:

```
Enter password: python
Enter password: python@123
Enter password: python123
Access granted!
```

2.3. STRING OPERATIONS:

Strings are an essential part of Python used to handle text data. We can create them using single, double, or triple quotes. Strings help store names, sentences, or any sequence of characters in a program. Strings in Python support various operations like slicing, indexing, concatenation, and formatting, making it easy to manage and modify text. In this, we will learn using the example of a string in Python for better understanding.

Working with Strings in Python:

We need to place the sequence of letters within single quotes or double quotes to create a string and assign it to a variable. There is also an option of giving variables a string of characters or different characters. We can assign sequences of single and double quotes.

1. Creating a String in Python:

In Python, we create a string by placing text inside single (') or double (") quotes. This is the basic way we define short messages or names. For longer or multi-line text, we use triple quotes (''' or """). These are helpful when writing docstrings or storing paragraphs. Strings in Python are used to handle and manipulate text data in programs.

Example:

```
st1 = "hello world"        # String using double quotes
st2 = 'bye'                # String using single quotes
st3 = """Welcome to WsCubeTech""" # multi-line string using triple quotes
```

```
print(st1)
print(st2)
print(st3)
```

Run Code

Output:

```
hello world
bye
Welcome to WsCubeTech
```

2. Accessing Characters in Python String:

In Python, we can access individual characters in a string using indexing. Each character has a position, starting from 0 for the first character. Strings in Python support both positive and negative indexing. Positive indexes start from the left, while negative indexes start from the right.

Example:

```
text = "WsCube Tech"
print(text[0])   # First character
print(text[3])   # Fourth character
print(text[-1])  # Last character
print(text[-4])  # Fourth character from end
```

Run Code

Output:

```
W
u
h
T
```

3. String Slicing in Python:

In Python, slicing helps us to get a part of a string by giving a start and end index using the format string[start:end]. It includes characters from the start index but excludes the end index. We can also slice using negative indexes to count from the end. Strings in Python allow flexible slicing, which is useful when we need to extract specific words or letters.

Example:

```
text = "WsCube Tech"
print(text[0:6])     # From index 0 to 5
print(text[7:])      # From index 7 to end
print(text[-4:-1])   # Using negative indexing
```

Run Code

Output:

```
WsCube
Tech
Tec
```

4. String Length in Python:

We use the len() function to find how many characters are in a string, including spaces.

Example:

```
text = "WsCube Tech"
print(len(text))
```

Run Code

Output:

```
11
```

5. String Concatenation:

String concatenation refers to joining two strings together into a single line. We can join two strings using the + operator.

Example:

```
str1 = "WsCube"
str2 = "Tech"
result = str1 + " " + str2
print(result)
```

Run Code

Output:

```
WsCube Tech
```

6. Strings Indexing and Splitting:

Indexing helps us access individual characters in a string by their position number, starting from 0. We can use positive or negative indexes to get characters from the start or end.Strings in Python also let us split text into parts using the .split() method, which divides the string based on a specified separator, like a space.

Example:

```
text = "WsCube Tech"
print(text[0])        # Get first character
print(text[-1])       # Get last character
print(text.split())   # Split string by space
```

Run Code

Output:

```
W
h
['WsCube', 'Tech']
```

7. String Immutability in Python:

Strings in Python cannot be changed once created. If we try to modify a character, it will cause an error because strings are immutable.

Example:

```
text = "WsCube"
text[0] = "W"
print(text)
```

Run Code

Output:

TypeError: 'str' object does not support item assignment

8. Deleting a String in Python:

In Python, we can delete an entire string variable using the del keyword. This removes the reference to the string from memory. Strings in Python are objects, so deleting a string means we cannot use it afterward, or it will cause an error.

Example:

```
text = "WsCube Tech"
del text
# print(text)  # This will cause an error because text is deleted
```

Run Code

Output:

```
NameError: name 'text' is not defined
```

9. Updating a String in Python:

We cannot change individual characters in a string because strings are immutable. However, we can create a new string by combining parts of the old string with new content.

This is how we update the content of a string in Python by forming a modified version of it.

Example:

```
text = "WsCube Tech"
updated_text = "Hello " + text[7:]
print(updated_text)
```

Run Code

Output:

```
Hello Tech
```

10. Multi-line Strings in Python:

We use triple quotes (''' or """) to create strings that span multiple lines. This helps when we want to store paragraphs, messages, or long text blocks. A multiline string in Python is also useful for writing docstrings or comments that need multiple lines.

Example:

```
msg = """Welcome to
WsCube Tech.
Learn Python easily!"""
print(msg)
```

Run Code

Output:

```
Welcome to
```

WsCube Tech.

Learn Python easily!

11. String Formatting in Python:

We format strings to insert variables or values into text easily.

Method	Description	Example
f-strings	Easy way to add variables in a string	f"Hello, {name}!"
.format()	Fills values in placeholders {}	"Hello, {}!".format(name)

Example:

```
name = "WsCube"
print(f"Welcome to {name} Tech")          # Using f-string
print("Welcome to {} Tech".format(name))    # Using .format()
```

Run Code

Output:

```
Welcome to WsCube Tech
Welcome to WsCube Tech
```

12. Looping Through a String in Python:

We can use a for loop to go through each character in a string one by one. This helps us process or display characters from Python strings easily.

Example:

```
text = "WsCube Tech"
for char in text:
    print(char)
```

Run Code

Output:

```
W
s
C
```

u

b

e

T

e

c

h

13. String Comparison in Python:

In Python, we can compare two strings using comparison operators like ==, !=, <, >, <=, and >=. These operators check if strings are equal, not equal, or which one comes first alphabetically.

Python strings comparison is case-sensitive, so "Hello" and "hello" are treated as different.

Example:

```
str1 = "WsCube Tech"
str2 = "WsCube tech"
print(str1 == str2)
print(str1 != str2)
print(str1 < str2)
```

Run Code

Output:

```
False
True
True
```

Methods of Python String:

Apart from the ones mentioned above, there are various other string methods in Python, which we have listed below:

Methods	Description
upper()	Converts the string to uppercase
lower()	Converts the string to lowercase
capitalize()	Capitalizes the first character of the string
partition()	Splits string into a tuple at the first match of the separator.
replace()	Replaces part of the string with another value
encode()	Encodes string to bytes (UTF-8 by default)
find()	Returns the index of the first occurrence of a substring
title()	Converts string to title case (each word starts with a capital letter)
rstrip()	Removes trailing whitespace or characters
split()	Splits the string from left
startswith()	Checks if a string begins with the specified string or not.
isnumeric()	checks numeric characters
index()	returns the index of a substring.

Examples of Python String Methods:

```
text = "wsCube Tech"

# 1. upper()

print(text.upper())
```

```
# 2. lower()
print(text.lower())
# 3. capitalize()
print(text.capitalize())
# 4. partition()
print(text.partition(" "))
# 5. replace()
print(text.replace("Tech", "Python"))
# 6. encode()
print(text.encode())
# 7. find()
print(text.find("Tech"))
# 8. title()
print(text.title())
# 9. rstrip()
text2 = "Python    "
print(text2.rstrip())
# 10. split()
print(text.split())
# 11. startswith()
print(text.startswith("ws"))
# 12. maketrans() and translate()
trans = str.maketrans("abc", "123")
text3 = "abcde"
print(text3.translate(trans))
# 13. isnumeric()
num = "12345"
print(num.isnumeric())
# 14. index()
print(text.index("Tech"))
```

Run Code

Output:

WSCUBE TECH

wscube tech

Wscube tech

('wsCube', ' ', 'Tech')

wsCube Python

b'wsCube Tech'

7

Wscube Tech

Python

['wsCube', 'Tech']

True

123de

True

2.4. LISTS:

- It is a general purpose most widely used in data structures.
- List is a collection which is ordered and changeable and allows duplicate members. (Grow and shrink as needed, sequence type, sortable).
- To use a list, you must declare it first. Do this using square brackets and separate values with commas.
- We can construct / create list in many ways.

Ex:

```
>>> list1=[1,2,3,'A','B',7,8,[10,11]]
>>> print(list1)
[1, 2, 3, 'A', 'B', 7, 8, [10, 11]]
>>> x=list()
>>> x
>>> tuple1=(1,2,3,4)
```

```
>>> x=list(tuple1)
>>> x
[1, 2, 3, 4]
```

List operations:

These operations include indexing, slicing, adding, multiplying, and checking for membership

Basic List Operations:

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

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

Indexing, Slicing, and Matrixes

Because lists are sequences, indexing and slicing work the same way for lists as they do for strings.

Assuming following input

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

2.5. LIST SLICES:

```
>>> list1=range(1,6)
>>> list1
range(1, 6)
```

```
>>> print(list1)
range(1, 6)
>>> list1=[1,2,3,4,5,6,7,8,9,10]
>>> list1[1:]
[2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list1[:1]
>>> list1[2:5]
[3, 4, 5]
>>> list1[:6]
[1, 2, 3, 4, 5, 6]
>>> list1[1:2:4]
[2]
>>> list1[1:8:2]
[2, 4, 6, 8]
```

2.6. LIST METHODS:

The list data type has some more methods. Here are all of the methods of list objects:

- Del()
- Append()
- Extend()
- Insert()
- Pop()
- Remove()
- Reverse()
- Sort()

Delete: Delete a list or an item from a list

```
>>> x=[5,3,8,6]
>>> del(x[1]) #deletes the index position 1 in a list
>>> x
```

```
[5, 8, 6]
----------
>>> del(x)
>>> x # complete list gets deleted
```

Append: Append an item to a list

```
>>> x=[1,5,8,4]
>>> x.append(10)
>>> x
[1, 5, 8, 4, 10]
```

Extend: Append a sequence to a list.

```
>>> x=[1,2,3,4]
>>> y=[3,6,9,1]
>>> x.extend(y)
>>> x
[1, 2, 3, 4, 3, 6, 9, 1]
```

Insert: To add an item at the specified index, use the insert () method:

```
>>> x=[1,2,4,6,7]
>>> x.insert(2,10) #insert(index no, item to be inserted)
>>> x
[1, 2, 10, 4, 6, 7]
------------------------
>>> x.insert(4,['a',11])
>>> x
[1, 2, 10, 4, ['a', 11], 6, 7]
```

Pop: The pop() method removes the specified index, (or the last item if index is not specified) or simply pops the last item of list and returns the item.

```
>>> x=[1, 2, 10, 4, 6, 7]
>>> x.pop()
7
>>> x
```

```
[1, 2, 10, 4, 6]
---------------------------------
>>> x=[1, 2, 10, 4, 6]
>>> x.pop(2)
10
>>> x
[1, 2, 4, 6]
```

Remove: The **remove()** method removes the specified item from a given list.

```
>>> x=[1,33,2,10,4,6]
>>> x.remove(33)
>>> x
[1, 2, 10, 4, 6]
>>> x.remove(4)
>>> x
[1, 2, 10, 6]
```

Reverse: Reverse the order of a given list.

```
>>> x=[1,2,3,4,5,6,7]
>>> x.reverse()
>>> x
[7, 6, 5, 4, 3, 2, 1]
```

Sort: Sorts the elements in ascending order

```
>>> x=[7, 6, 5, 4, 3, 2, 1]
>>> x.sort()
>>> x
[1, 2, 3, 4, 5, 6, 7]
----------------------
>>> x=[10,1,5,3,8,7]
>>> x.sort()
>>> x
[1, 3, 5, 7, 8, 10]
```

2.7. LIST LOOP:

Loops are control structures used to repeat a given section of code a certain number of times or until a particular condition is met.

Method #1: For loop

```
#list of items
list = ['M','R','C','E','T']
i = 1
#Iterating over the list
for item in list:
print ('college ',i,' is ',item)
i = i+1
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/lis.py
college 1 is M
college 2 is R
college 3 is C
college 4 is E
college 5 is T
```

Method #2: For loop and range()

In case we want to use the traditional for loop which iterates from number x to number y.

```
# Python3 code to iterate over a list
list = [1, 3, 5, 7, 9]
# getting length of list
length = len(list)
# Iterating the index
# same as 'for i in range(len(list))'
for i in range(length):
print(list[i])
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/listlooop.py
```

1

3

5

7

9

Method #3: using while loop

```
# Python3 code to iterate over a list
list = [1, 3, 5, 7, 9]
# Getting length of list
length = len(list)
i = 0
# Iterating using while loop
while i < length:
print(list[i])
i += 1
```

Mutability:

A mutable object can be changed after it is created, and an immutable object can't.

Append: Append an item to a list

```
>>> x=[1,5,8,4]
>>> x.append(10)
>>> x
[1, 5, 8, 4, 10]
```

Extend: Append a sequence to a list.

```
>>> x=[1,2,3,4]
>>> y=[3,6,9,1]
>>> x.extend(y)
>>> x
```

Delete: Delete a list or an item from a list

```
>>> x=[5,3,8,6]
>>> del(x[1]) #deletes the index position 1 in a list
```

```
>>> x
[5, 8, 6]
```

Insert: To add an item at the specified index, use the insert () method:

```
>>> x=[1,2,4,6,7]
>>> x.insert(2,10) #insert(index no, item to be inserted)
>>> x
[1, 2, 10, 4, 6, 7]
------------------------
>>> x.insert(4,['a',11])
>>> x
[1, 2, 10, 4, ['a', 11], 6, 7]
```

Pop: The pop() method removes the specified index, (or the last item if index is not specified) or simply pops the last item of list and returns the item.

```
>>> x=[1, 2, 10, 4, 6, 7]
>>> x.pop()
7
>>> x
[1, 2, 10, 4, 6]
---------------------------------
>>> x=[1, 2, 10, 4, 6]
>>> x.pop(2)
10
>>> x
[1, 2, 4, 6]
```

Remove: The **remove()** method removes the specified item from a given list.

```
>>> x=[1,33,2,10,4,6]
>>> x.remove(33)
>>> x
[1, 2, 10, 4, 6]
>>> x.remove(4)
```

```
>>> x
[1, 2, 10, 6]
```

Reverse: Reverse the order of a given list.

```
>>> x=[1,2,3,4,5,6,7]
>>> x.reverse()
>>> x
[7, 6, 5, 4, 3, 2, 1]
```

Sort: Sorts the elements in ascending order

```
>>> x=[7, 6, 5, 4, 3, 2, 1]
>>> x.sort()
>>> x
[1, 2, 3, 4, 5, 6, 7]
----------
>>> x=[10,1,5,3,8,7]
>>> x.sort()
>>> x
[1, 3, 5, 7, 8, 10]
```

2.8. TUPLES:

- A tuple is a collection which is ordered and unchangeable. In Python tuples are written with round brackets.
- Supports all operations for sequences.
- Immutable, but member objects may be mutable.
- If the contents of a list shouldn't change, use a tuple to prevent items from accidently being added, changed, or deleted.
- Tuples are more efficient than list due to python's implementation.

We can construct tuple in many ways:

```
X=() #no item tuple
X=(1,2,3)
X=tuple(list1)
```

X=1,2,3,4

Example:

```
>>> x=(1,2,3)
>>> print(x)
(1, 2, 3)
>>> x
(1, 2, 3)
----------------------
>>> x=()
>>> x
()
---------------------------
>>> x=[4,5,66,9]
>>> y=tuple(x)
>>> y
(4, 5, 66, 9)
---------------------------
>>> x=1,2,3,4
>>> x
(1, 2, 3, 4)
```

Some of the operations of tuple are:

- Access tuple items
- Change tuple items
- Loop through a tuple
- Count()
- Index()
- Length()

Access tuple items: Access tuple items by referring to the index number, inside square brackets

```
>>> x=('a','b','c','g')
>>> print(x[2])
```

c

Change tuple items: Once a tuple is created, you cannot change its values. Tuples are unchangeable.

```
>>> x=(2,5,7,'4',8)
>>> x[1]=10
Traceback (most recent call last):
File "<pyshell#41>", line 1, in <module>
x[1]=10
Type Error: 'tuple' object does not support item assignment
>>> x
(2, 5, 7, '4', 8) # the value is still the same
```

Loop through a tuple: We can loop the values of tuple using for loop

```
>>> x=4,5,6,7,2,'aa'
>>> for i in x:
print(i)
4
5
6
7
2
aa
```

Count (): Returns the number of times a specified value occurs in a tuple

```
>>> x=(1,2,3,4,5,6,2,10,2,11,12,2)
>>> x.count(2)
4
```

Index (): Searches the tuple for a specified value and returns the position of where it was found

```
>>> x=(1,2,3,4,5,6,2,10,2,11,12,2)
>>> x.index(2)
1
```

(Or)

```
>>> x=(1,2,3,4,5,6,2,10,2,11,12,2)
>>> y=x.index(2)
>>> print(y)
1
```

Length (): To know the number of items or values present in a tuple, we use len().

```
>>> x=(1,2,3,4,5,6,2,10,2,11,12,2)
>>> y=len(x)
>>> print(y)
12
```

Tuple Assignment

Python has tuple assignment feature which enables you to assign more than one variable at a time. In here, we have assigned tuple 1 with the college information like college name, year, etc. and another tuple 2 with the values in it like number (1, 2, 3… 7).

For Example,

Here is the code,

```
 >>> tup1 = ('mrcet', 'eng college','2004','cse', 'it','csit');
 >>> tup2 = (1,2,3,4,5,6,7);
 >>> print(tup1[0])
 mrcet
>>> print(tup2[1:4])
(2, 3, 4)
```

Tuple 1 includes list of information of mrcet

Tuple 2 includes list of numbers in it

We call the value for [0] in tuple and for tuple 2 we call the value between 1 and 4

Run the above code- It gives name mrcet for first tuple while for second tuple it gives number (2, 3, 4)

Tuple as return values:

A Tuple is a comma separated sequence of items. It is created with or without (). Tuples are immutable.

```
# A Python program to return multiple values from a method using tuple
```

```
# This function returns a tuple
def fun():
str = "mrcet college"
x = 20
return str, x; # Return tuple, we could also
# write (str, x)
# Driver code to test above method
str, x = fun() # Assign returned tuple
print(str)
print(x)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/tupretval.py

mrcet college

20

Functions can return tuples as return values.

```
def circleInfo(r):
""" Return (circumference, area) of a circle of radius r """
c = 2 * 3.14159 * r
a = 3.14159 * r * r
return (c, a)
print(circleInfo(10))
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/functupretval.py

(62.8318, 314.159)

```
def f(x):
y0 = x + 1
y1 = x * 3
y2 = y0 ** y3
return (y0, y1, y2)
```

Tuple methods:

Count (): Returns the number of times a specified value occurs in a tuple

```
>>> x=(1,2,3,4,5,6,2,10,2,11,12,2)
>>> x.count(2)
4
```

Index (): Searches the tuple for a specified value and returns the position of where it was found

```
>>> x=(1,2,3,4,5,6,2,10,2,11,12,2)
>>> x.index(2)
1
```

(Or)

```
>>> x=(1,2,3,4,5,6,2,10,2,11,12,2)
>>> y=x.index(2)
>>> print(y)
1
```

2.9. SETS:

A set is a collection which is unordered and unindexed with no duplicate elements. In Python sets are written with curly brackets.

- To create an empty set we use set()
- Curly braces '{}' or the set() function can be used to create sets

We can construct tuple in many ways:

```
X=set()
X={3,5,6,8}
X=set(list1)
```

Example:

```
>>> x={1,3,5,6}
>>> x
{1, 3, 5, 6}
----------
>>> x=set()
```

```
>>> x
set()
----------
>>> list1=[4,6,"dd",7]
>>> x=set(list1)
>>> x
{4, 'dd', 6, 7}
```

- We cannot access items in a set by referring to an index, since sets are unordered the items has no index.
- But you can loop through the set items using a for loop, or ask if a specified value is present in a set, by using the in keyword.

Some of the basic set operations are:

- Add()
- Remove()
- Len()
- Item in x
- Pop
- Clear

Add (): To add one item to a set use the add () method. To add more than one item to a set use the update () method.

```
>>> x={"mrcet","college","cse","dept"}
>>> x.add("autonomous")
>>> x
{'mrcet', 'dept', 'autonomous', 'cse', 'college'}
----
>>> x={1,2,3}
>>> x.update("a","b")
>>> x
{1, 2, 3, 'a', 'b'}
----------
```

```
>>> x={1,2,3}
>>> x.update([4,5],[6,7,8])
>>> x
{1, 2, 3, 4, 5, 6, 7, 8}
```

Remove (): To remove an item from the set we use remove or discard methods.

```
>>> x={1, 2, 3, 'a', 'b'}
>>> x.remove(3)
>>> x
{1, 2, 'a', 'b'}
```

Len (): To know the number of items present in a set, we use len().

```
>>> z={'mrcet', 'dept', 'autonomous', 'cse', 'college'}
>>> len(z) PYTHON PROGRAMMING III YEAR/I SEM MRCET
43
```

5

Item in X: you can loop through the set items using a for loop.

```
>>> x={'a','b','c','d'}
>>> for item in x:
print(item)
c
d
a
b
```

pop ():This method is used to remove an item, but this method will remove the last item. Remember that sets are unordered, so you will not know what item that gets removed.

```
>>> x={1, 2, 3, 4, 5, 6, 7, 8}
>>> x.pop()
1
>>> x
{2, 3, 4, 5, 6, 7, 8}
```

Clear (): This method will the set as empty.

```
>>> x={2, 3, 4, 5, 6, 7, 8}
>>> x.clear()
>>> x
set()
```

The set also consist of some mathematical operations like:

Intersection AND &

Union OR |

Symmetric Diff XOR ^

Diff In set1 but not in set2 set1-set2

Subset set2 contains set1 set1<=set2

Superset set1 contains set2 set1>=set2

Some examples:

```
>>> x={1,2,3,4}
>>> y={4,5,6,7}
>>> print(x|y)
{1, 2, 3, 4, 5, 6, 7}
```

```
>>> x={1,2,3,4}
>>> y={4,5,6,7}
>>> print(x&y)
{4}
```

```
>>> A = {1, 2, 3, 4, 5}
>>> B = {4, 5, 6, 7, 8}
>>> print(A-B)
{1, 2, 3}
```

```
>>> B = {4, 5, 6, 7, 8}
>>> A = {1, 2, 3, 4, 5}
```

```
>>> print(B^A)
{1, 2, 3, 6, 7, 8}
```

2.10. DICTIONARIES:

A dictionary is a collection which is unordered, changeable and indexed. In Python dictionaries are written with curly brackets, and they have keys and values.

- Key-value pairs
- Unordered

We can construct or create dictionary like:

```
X={1:'A',2:'B',3:'c'}
X=dict([('a',3) ('b',4)]
X=dict('A'=1,'B' =2)
```

Examples:

```
>>> dict1 = {"brand":"mrcet","model":"college","year":2004}
>>> dict1
{'brand': 'mrcet', 'model': 'college', 'year': 2004}
```

To access specific value of a dictionary, we must pass its key,

```
>>> dict1 = {"brand":"mrcet","model":"college","year":2004}
>>> x=dict1["brand"]
>>> x
'mrcet'
```

To access keys and values and items of dictionary:

```
>>> dict1 = {"brand":"mrcet","model":"college","year":2004}
>>> dict1.keys()
dict_keys(['brand', 'model', 'year'])
>>> dict1.values()
dict_values(['mrcet', 'college', 2004])
>>> dict1.items()
dict_items([('brand', 'mrcet'), ('model', 'college'), ('year', 2004)])
```

```
>>> for items in dict1.values():
print(items)
mrcet
college
2004
>>> for items in dict1.keys():
print(items)
brand
model
year
>>> for i in dict1.items():
print(i)
('brand', 'mrcet')
('model', 'college')
('year', 2004)
```

Some of the operations are:

- Add/change
- Remove
- Length
- Delete

Add/change values: You can change the value of a specific item by referring to its key name

```
>>> dict1 = {"brand":"mrcet","model":"college","year":2004}
>>> dict1["year"]=2005
>>> dict1
{'brand': 'mrcet', 'model': 'college', 'year': 2005}
```

Remove(): It removes or pop the specific item of dictionary.

```
>>> dict1 = {"brand":"mrcet","model":"college","year":2004}
>>> print(dict1.pop("model"))
college
```

```
>>> dict1
{'brand': 'mrcet', 'year': 2005}
```

Delete: Deletes a particular item.

```
>>> x = {1:1, 2:4, 3:9, 4:16, 5:25}
>>> del x[5]
>>> x
```

Length: we use len() method to get the length of dictionary.

```
>>>{1: 1, 2: 4, 3: 9, 4: 16}
{1: 1, 2: 4, 3: 9, 4: 16}
>>> y=len(x)
>>> y
4
```

Iterating over (key, value) pairs:

```
>>> x = {1:1, 2:4, 3:9, 4:16, 5:25}
>>> for key in x:
print(key, x[key])
1 1
2 4
3 9
4 16
5 25
>>> for k,v in x.items():
print(k,v)
1 1
2 4
3 9
4 16
5 25
```

List of Dictionaries:

```
>>> customers = [{"uid":1,"name":"John"},
```

```
{"uid":2,"name":"Smith"},
{"uid":3,"name":"Andersson"},
]
>>> >>> print(customers)
[{'uid': 1, 'name': 'John'}, {'uid': 2, 'name': 'Smith'}, {'uid': 3, 'name': 'Andersson'}]
## Print the uid and name of each customer
>>> for x in customers:
print(x["uid"], x["name"])
1 John
2 Smith
3 Andersson
## Modify an entry, This will change the name of customer 2 from Smith to Charlie
>>> customers[2]["name"]="charlie"
>>> print(customers)
[{'uid': 1, 'name': 'John'}, {'uid': 2, 'name': 'Smith'}, {'uid': 3, 'name': 'charlie'}]
## Add a new field to each entry
>>> for x in customers:
x["password"]="123456" # any initial value
>>> print(customers)
[{'uid': 1, 'name': 'John', 'password': '123456'}, {'uid': 2, 'name': 'Smith', 'password': '123456'},
{'uid': 3, 'name': 'charlie', 'password': '123456'}]
## Delete a field
>>> del customers[1]
>>> print(customers)
[{'uid': 1, 'name': 'John', 'password': '123456'}, {'uid': 3, 'name': 'charlie', 'password': '123456'}]
>>> del customers[1]
>>> print(customers)
[{'uid': 1, 'name': 'John', 'password': '123456'}]
## Delete all fields
>>> for x in customers:
```

```
del x["uid"]
>>> x
{'name': 'John', 'password': '123456'}
```

UNIT III: FUNCTIONS

Definition and types – Passing parameters to a Function – Scope – Type conversion – Passing Functions to a Function – Modules – Standard Modules – Inbuilt Function – Scope of Variables.

3.1. DEFINITION OF FUNCTION:

A function in Python is a block of code that performs a specific task. It takes zero or more inputs (arguments), performs some operations on those inputs, and may return a result. Functions provide a way to break down a large program into smaller, more manageable pieces.

Functions are very essential part of any programming language. They help our code to make modular, reusable, and organized. Instead of writing the same code again and again, we can create a function and call it whenever it needed.

Types of Functions in Python:

Python supports various types of functions, each serving different purposes in programming. Here are the main types of functions in Python, along with examples:

1. Built-in Functions

These functions are pre-defined in Python and can be used directly without any further declaration.

Example

```
# Using the built-in len() function
my_list = [1, 2, 3, 4, 5]
print(len(my_list))
# Output: 5
```

2. User-defined Functions

These are functions that users create to perform specific tasks.

Example

```
def add_numbers(a, b):
    return a + b
result = add_numbers(3, 5)
print(result)
# Output: 8
```

3.Anonymous Functions (Lambda Functions)

These are small, unnamed functions defined using the lambda keyword. They are typically used for short, simple operations.

Example

```
add = lambda x, y: x + y
print(add(3, 5))
 # Output: 8
```

4. Recursive Functions

These are functions that call themselves within their definition. They help solve problems that can be broken down into smaller, similar problems.

Example

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)
print(factorial(5))
# Output: 120
```

5. Higher-Order Functions

These functions can take other functions as arguments or return them as results. Examples include map(), filter(), and reduce().

Example

```
def square(x):
    return x * x
numbers = [1, 2, 3, 4, 5]
squared_numbers = list(map(square, numbers))
print(squared_numbers)
 # Output: [1, 4, 9, 16, 25]
```

6. Generator Functions

These functions yield values one at a time and can produce a sequence of values over time, using the yield keyword.

Example

```
def generate_numbers():
    for i in range(1, 6):
        yield i
for number in generate_numbers():
    print(number)
 # Output: 1 2 3 4 5
```

3.2. PASSING PARAMETERS TO A FUNCTION:

In Python, you can pass parameters to a function using several methods, including positional arguments, keyword arguments, default arguments, and variable-length arguments.

1. Positional Arguments

Positional arguments are passed to the function in the order they are defined. The number and order of arguments must match the parameters in the function definition.

Python

```
def greet(first_name, last_name):
    print(f"Hello, {first_name} {last_name}!")
greet("Jane", "Doe")
# Output: Hello, Jane Doe!
```

2. Keyword Arguments

Keyword arguments allow you to specify the parameter names when calling the function. This means you can pass arguments in any order, which improves readability.

Python

```
def display_info(first_name, last_name):
    print(f"First Name: {first_name}")
    print(f"Last Name: {last_name}")
# Order doesn't matter with keyword arguments
display_info(last_name="Smith", first_name="John")
# Output:
# First Name: John
```

```
# Last Name: Smith
```

3. Default Arguments

You can provide default values for parameters in the function definition. If an argument is not provided during the function call, the default value is used.

Python

```
def calculate_interest(principal, time, rate=0.15):
    interest = principal * time * rate
    print(f"Interest: {interest}")
calculate_interest(1000, 2)
# Output: Interest: 300.0 (uses default rate of 0.15)
calculate_interest(1000, 2, rate=0.1)
# Output: Interest: 200.0 (overrides the default rate)
```

4. Variable-Length Arguments (*args and **kwargs)

Python allows you to pass a variable number of arguments using special syntax:

***args (non-keyword arguments):** This is used to pass a variable number of positional arguments. The arguments are collected into a tuple inside the function.

Python

```
def find_sum(*numbers):
    result = sum(numbers)
    print(f"Sum: {result}")
find_sum(1, 2, 3) # Output: Sum: 6
find_sum(4, 5)    # Output: Sum: 9
```

****kwargs (keyword arguments):** This is used to pass a variable number of keyword arguments. The arguments are collected into a dictionary inside the function.

Python

```
def display_details(**details):
    for key, value in details.items():
        print(f"{key}: {value}")
display_details(name="Alice", age=30, city="New York")
```

Output:

name: Alice

age: 30

city: New York

3.3. SCOPE:

A variable is only available from inside the region it is created. This is called scope. Scope is the region of code where a name is visible. When you use a name, Python looks it up in this order (LEGB):

- Local: the current function's scope.
- Enclosing: any outer function scopes (for nested functions).
- Global: the module's top level (module namespace).
- Built-in: names defined by Python in builtins (for example, len, print).

Names live in namespaces (think dictionaries mapping names to objects). Scope is about which namespaces Python consults to resolve a name at a given point in code.

Local scope:

Names assigned inside a function are local to that function unless declared otherwise. They are not visible outside the function.

```
def show_order_id():
    order_id = 42
    print("inside function:", order_id)
show_order_id()
print("outside function:", order_id)  # NameError
```

Python decides scope at compile time. If a function assigns to a name anywhere in its body, all uses of that name in the function are treated as local-leading to a common Unbound Local Error when you read before assigning:

```
discount_rate = 0.10  # module-level (global)
def price_with_discount(amount_cents):
    print("configured discount:", discount_rate)  # looks local because of the assignment below
    discount_rate = 0.20  # assignment makes 'discount_rate' local in this function
```

```
    return int(amount_cents * (1 - discount_rate))
# UnboundLocalError: cannot access local variable 'discount_rate' where it is not associated with a value.
```

Enclosing scope (closures):

Nested functions can see names from their immediately enclosing function. To rebind such a name (not just read it), declare it nonlocal.

```
def make_step_counter():
    count = 0  # enclosing scope for 'increment'
    def increment():
        nonlocal count  # rebind the 'count' in the nearest enclosing function
        count += 1
        return count
    return increment
step = make_step_counter()
print(step())  # 1
print(step())  # 2
```

Without nonlocal, assigning to count inside increment() would create a new local name and leave the outer count unchanged.

Global scope:

Names assigned at the top level of a module live in the module's global namespace. Any function can read them. To assign to a module-level name from inside a function, declare it global.

```
greeting = "Hello"
def greet_city(city_name):
    print(greeting, city_name)  # reads global
def set_greeting(new_greeting):
    global greeting
    greeting = new_greeting     # rebinds global
greet_city("Nairobi")  # Hello Nairobi
set_greeting("Hi")
```

```
greet_city("Nairobi")  # Hi Nairobi
```

Use global sparingly. Prefer passing values as parameters and returning results to keep code testable and predictable.

Built-in scope (and a note on keywords):

The built-in scope contains names like len, print, and Exception. Avoid shadowing them, or you'll lose access to the built-in for that scope.

```
list = [1, 2, 3]     # shadows the built-in 'list' constructor
list("abc")          # TypeError: 'list' object is not callable
del list             # fix by deleting the shadowing name
```

3.4. TYPE CONVERSION:

Type conversion means changing the data type of a value. **For example**, converting an integer (5) to a float (5.0) or a string ("10") to an integer (10). In Python, there are two types of type conversion:

1. **Implicit Conversion:** Python changes the data type by itself while running the code, to avoid mistakes or data loss.
2. **Explicit Conversion:** You change the data type on purpose using functions like int(), float() or str().

Implicit Type Conversion:

In **implicit conversion**, Python automatically converts one data type into another during expression evaluation. This usually happens when a smaller data type like int is combined with a larger one like float in an operation. **Example:**

```
x = 10          # Integer
y = 10.6        # Float
z = x + y
print("x:", type(x))
print("y:", type(y))
print("z =", z)
print("z :", type(z))
```

Output

```
x: <class 'int'>
```

y: <class 'float'>

z = 20.6

z : <class 'float'>

Explanation: x is an integer and **y** is a float. In **z = x + y**, Python automatically converts **x** to float to avoid data loss, making **z** a float 0.6.

Explicit Type Conversion:

Explicit conversion (or type casting) is when you manually convert the data type of a value using Python's built-in functions. This is helpful when you want to control how the data is interpreted or manipulated in your code. Some common type casting functions include:

- **int()** converts a value to an integer
- **float()** converts a value to a floating point number
- **str()** converts a value to a string
- **bool()** converts a value to a Boolean (True/False)

Example:

```
s = "100"  # String
a = int(s)
print(a)
print(type(a))
```

Output

100

<class 'int'>

Explanation: a = int(s), we explicitly convert it to an integer. This manual type change is called explicit type conversion and a becomes 100 of type <class 'int'>.

Examples of Common Type Conversion Functions

Example 1: Converting a binary string

```
s = "10010"
a = int(s, 2)
print(a)
b= float(s)
print(b)
```

Output

18

10010.0

Explanation:

- **int(s, 2)** interprets the binary string '10010' as the integer 18.
- **float(s)** converts the string to a floating-point number.

Example 2: ASCII, Hexadecimal and Octal Conversion

```
c = '4'
print("ASCII of '4':", ord(c))
print("56 in Hex:", hex(56))
print("56 in Octal:", oct(56))
```

Output

ASCII of '4': 52

56 in Hex: 0x38

56 in Octal: 0o70

Explanation:

- **ord(c)** returns the ASCII code of the character '4'.
- **hex()** and **oct()** convert the integer 56 to its hexadecimal and octal representations, respectively.

Example 3: String to Tuple, Set and List

```
s = 'geeks'
print("To tuple:", tuple(s))
print("To set:", set(s))
print("To list:", list(s))
```

Output

To tuple: ('g', 'e', 'e', 'k', 's')

To set: {'e', 'g', 'k', 's'}

To list: ['g', 'e', 'e', 'k', 's']

Explanation:

- **tuple(s)** keeps all characters including duplicates in order.

- **set(s)** removes duplicates and returns an unordered collection.
- **list(s)** returns a list of characters from the string.

Example 4: Other Conversions – Complex, String, Dictionary

```
a = 1
tup = (('a', 1), ('f', 2), ('g', 3))
print("To complex:", complex(1, 2))
print("To string:", str(a))
print("To dict:", dict(tup))
```

Output

```
To complex: (1+2j)
To string: 1
To dict: {'a': 1, 'f': 2, 'g': 3}
```

Explanation:

- **complex(1, 2)** creates a complex number with real part 1 and imaginary part 2.
- **str(a)** converts the integer 1 to the string "1".
- **dict(tup)** creates a dictionary from a sequence of key-value tuples.

3.5. PASSING FUNCTIONS TO A FUNCTION:

In Python, functions are first-class objects meaning they can be assigned to variables, passed as arguments and returned from other functions. This enables **higher-order functions**, **decorators** and lambda expressions. By passing a function as an argument, we can modify a function's behavior dynamically without altering its implementation. **For Example:**

```
def process(func, text):  # applies a function to text
   return func(text)
def uppercase(text):  # converts text to uppercase
   return text.upper()
print(process(uppercase, "hello"))
```

Output

```
HELLO
```

Explanation: **process()** applies a given function to **text** and **uppercase()** converts text to uppercase. Passing uppercase to process with "hello" results in "HELLO".

Higher Order Functions:

A higher-order function takes or returns another function, enabling reusable and efficient code. It supports functional programming with features like callbacks, decorators, and utilities such as map(), filter(), and reduce().

Example 1 : Basic function passing

```
# higher-order function
def fun(func, number):
    return func(number)
# function to double a number
def double(x):
    return x * 2
print(fun(double, 5))
```

Output

10

Explanation: fun() takes **double**() as an argument and applies it to 5, returning 10.

Example 2: Passing Built-in Functions

```
# function to apply an operation on a list
def fun(func, numbers):
    return [func(num) for num in numbers]
# using the built-in 'abs' function
a = [-1, -2, 3, -4]
print(fun(abs, a))
```

Output

[1, 2, 3, 4]

Explanation: abs() is passed to **fun()**, which applies it to each element in the list, converting negative numbers to positive.

Lambda Functions:

A lambda function in Python is a small, anonymous function with a single expression, defined using lambda. It's often used in higher-order functions for quick, one-time operations.

Example: Lambda Function as an Argument

```
# function that applies an operation to a number
def fun(func, number):
    return func(number)
# passing a lambda function
print(fun(lambda x: x ** 2, 5))
```

Output

25

Explanation: lambda x: x ** 2 is passed to **fun()**, which squares the input 5 to produce 25.

Wrapper Functions(Decorators)

A wrapper function (decorator) enhances another function's behavior without modifying it. It takes a function as an argument and calls it within the wrapper.

Example 1 : Simple decorator

```
# simple decorator example
def decorator_fun(original_fun):
    def wrapper_fun():
        print("Hello, this is before function execution")
        original_fun()
        print("This is after function execution")
    return wrapper_fun
@decorator_fun
def display():
    print("This is inside the function !!")
# calling the decorated function
display()
```

Output

```
Hello, this is before function execution
This is inside the function !!
This is after function execution
```

Explanation: decorator_fun wraps the **display()** function, adding pre- and post-execution messages.

Example 2: Lambda Wrapper Function

```
def apply_lambda(func, value):
    return func(value)
square = lambda x: x ** 2
print("Square of 2 is:", apply_lambda(square, 2))
```

Output

Square of 2 is: 4

Explanation: apply_lambda() function applies the lambda function **lambda x: x ** 2** to 2, returning 4.

Built-in Functions using function arguments:

Python provides built-in functions that take other functions as arguments .

Example 1 : map()

```
a = [1, 2, 3, 4]
res = list(map(lambda x: x * 2, a))
print(res)
```

Output

[2, 4, 6, 8]

Explanation: map() function applies **lambda x: x * 2** to each element in **a**, doubling all values.

Example 2 : filter()

```
a = [1, 2, 3, 4, 5]
res = list(filter(lambda x: x % 2 == 0, a))
print(res)
```

Output

[2, 4]

Explanation: filter() selects even numbers from the list using **lambda x: x % 2 == 0**.

Example 3: reduce()

```
from functools import reduce
a = [1, 2, 3, 4]
res = reduce(lambda x, y: x + y, a)
```

```
print(res)
```

Output

10

Explanation: reduce() function applies **lambda x, y: x + y** cumulatively to elements in a, summing them to 10.

3.6. MODULES:

A Module in python is a file containing definitions and statements. A module can define functions, classes and variables. Modules help organize code into separate files so that programs become easier to maintain and reuse. Instead of writing everything in one place, related functionality can be grouped into its own module and imported whenever needed.

Create a Python Module:

To create a Python module, write the desired code and save that in a file with .py extension. Let's understand it better with an example:

Example: Let's create a calc.py in which we define two functions, one add and another subtract.

```
# calc.py
def add(x, y):
   return (x+y)
def subtract(x, y):
   return (x-y)
```

This is all that is required to create a module.

Import module:

Modules can be used in another Python file using the import statement. When Python sees an import, it loads the module if it exists in the interpreter's search path.

Syntax:

import module

Example: Now, we are importing the calc that we created earlier to perform add operation.

```
import calc
print(calc.add(10, 2))
```

Output

12

Explanation: import calc loads the module and calc.add() accesses a function through dot notation.

Types of Import Statements:

1. Import From Module: This allows importing specific functions, classes, or variables rather than the whole module.

```
from math import sqrt, factorial
print(sqrt(16))
print(factorial(6))
```

Output

4.0

720

Explanation: Only sqrt and factorial are brought into the local namespace, so the prefix math. is not required.

2. Import All Names: * imports everything from a module into the current namespace.

```
from math import *
print(sqrt(16))
print(factorial(6))
```

Output

4.0

720

Explanation: Every public name of math becomes directly accessible. (Not recommended in large projects due to namespace conflicts.)

3. Import With Alias: You can shorten a module's name using as.

```
import math as m
print(m.pi)
```

Output

3.141592653589793

Explanation: math is accessed through the shorter alias m.

Types of Modules in Python:

Python provides several kinds of modules. Each type plays a different role in application development.

1. Built-in Modules: These come bundled with Python and require no installation - e.g., math, random, os.

```
import random
print(random.randint(1, 5))
```

Output

4

Explanation: random.randint() returns a random number within the given range.

2. User-Defined Modules: These are modules you create yourself, such as calc.py.

```
import calc
print(calc.sub(20, 5))
```

Output

15

Explanation: The module is created manually and then imported into another script.

3. External (Third-Party) Modules: These modules are installed using pip - e.g., NumPy, Pandas, Requests.

```
import requests
r = requests.get("https://example.com")
print(r.status_code)
```

Output

200

Explanation: requests is installed separately (pip install requests) and provides HTTP utilities.

4. Package Modules: A package is a directory containing multiple modules, usually with an __init__.py file.

Example Directory

mypkg/

__init__.py

calc.py

utils.py

Using a module from a package

```
from mypkg import utils
print(utils.some_func())
```

calls a function named some_func(), the output will be whatever that function returns.

If utils.py contains something like:

```
def some_func():
    return "Hello"
```

Output

Hello

Locating a Module

Python searches for modules in a predefined list of directories known as the module search path. You can view this list using sys.path.

```
import sys
for p in sys.path:
    print(p)
```

Output

```
/home/guest/sandbox
/usr/local/lib/python313.zip
/usr/local/lib/python3.13
/usr/local/lib/python3.13/lib-dynload
/usr/local/lib/python3.13/site-packages
```

Explanation: Python checks each path in order until it finds the module you're trying to import.

3.7. STANDARD MODULES:

Python's "Standard Library" is a vast collection of built-in modules that are installed with Python itself. These modules provide pre-written functionality to handle common programming tasks, so you don't have to reinvent the wheel.

Here are some of the most frequently used standard modules, categorized by their function:

1. System and Operating System Interaction:

- **sys**: Provides access to system-specific parameters and functions. Useful for interacting with the Python runtime environment.
    - *Common uses*: Accessing command-line arguments (sys.argv), exiting the program (sys.exit()), checking the Python version (sys.version).
- **os**: A portable way of interacting with the operating system (e.g., Windows, macOS, Linux).
    - *Common uses*: File and directory management (os.listdir(), os.mkdir(), os.remove(), os.path.join()), environment variables.
- **pathlib**: (Recommended over os.path in modern Python) An object-oriented way to handle filesystem paths.
    - *Common uses*: Creating, parsing, and manipulating file paths cleanly.

2. Mathematics and Data Types:

- **math**: Provides access to common mathematical functions and constants beyond basic arithmetic.
    - *Common uses*: math.sqrt(), math.pi, math.ceil(), trigonometric functions.
- **random**: Generates pseudo-random numbers, sequences, and choices.
    - *Common uses*: Simulating dice rolls (random.randint()), shuffling lists (random.shuffle()), selecting random elements (random.choice()).
- **datetime**: Classes for working with dates and times.
    - *Common uses*: Getting the current time, formatting dates, calculating time differences.

3. Data Storage and Serialization:

- **json**: Encodes and decodes data using the JavaScript Object Notation (JSON) format, which is common for data interchange on the web.
    - *Common uses*: Reading from or writing to .json files, interacting with APIs.
- **pickle**: Python's native way to "serialize" (convert) Python objects into a byte stream, allowing them to be saved to a file or transferred over a network.
    - *Common uses*: Saving the state of a complex object or game session.
- **csv**: Provides tools for reading from and writing to CSV (Comma Separated Values)

- files easily.

4. Internet and Networking:

- **requests**: (While not strictly *built-in*, it is the de facto standard for HTTP requests and is recommended for robust web interaction).
- **urllib / urllib.request**: Python's built-in modules for fetching URLs (websites).
- **http.server**: Allows you to quickly spin up a basic web server (often used for serving static files locally).

5. Utilities and Debugging:

- **re**: Provides full support for regular expressions (regex), a powerful tool for pattern matching and text manipulation.
- **logging**: A flexible and comprehensive framework for emitting log messages from Python programs.
- **unittest**: The built-in framework for writing automated tests for your code.
- **argparse**: Handles the parsing of command-line arguments and options passed to your scripts, creating user-friendly command-line interfaces (CLIs).

Accessing the Standard Library:

The official Python Standard Library documentation is the definitive resource for discovering all available modules and how to use them.

You can import any of these modules into your code using the simple import statement:

python

```
import os
import math
import json
import random
print(math.sqrt(16))
print(os.getcwd())
```

3.8. INBUILT FUNCTION:

Python's **built-in functions** are a set of core functions that are readily available for use in any Python program without needing to import any external libraries. These functions

provide basic functionalities, ranging from simple data manipulation to advanced operations, making Python an efficient and versatile programming language.

List of Python Built-in Functions:

As of Python 3.12.2 version, the list of built-in functions is given below −

Sr.No.	Function & Description
1	**Python aiter() function** Returns an asynchronous iterator for an asynchronous iterable.
2	**Python all() function** Returns true when all elements in iterable is true.
3	**Python anext() function** Returns the next item from the given asynchronous iterator.
4	**Python any() function** Checks if any Element of an Iterable is True.
5	**Python ascii() function** Returns String Containing Printable Representation.
6	**Python bin() function** Converts integer to binary string.
7	**Python bool() function** Converts a Value to Boolean.
8	**Python breakpoint() function** This function drops you into the debugger at the call site and calls sys.breakpointhook().
9	**Python bytearray() function** Returns array of given byte size.

10	**Python bytes() function** Returns immutable bytes object.
11	**Python callable() function** Checks if the Object is Callable.
12	**Python chr() function** Returns a Character (a string) from an Integer.
13	**Python classmethod() function** Returns class method for given function.
14	**Python compile() function** Returns a code object.
15	**Python complex() function** Creates a Complex Number.
16	**Python delattr() function** Deletes Attribute From the Object.
17	**Python dict() function** Creates a Dictionary.
18	**Python dir() function** Tries to Return Attributes of Object.
19	**Python divmod() function** Returns a Tuple of Quotient and Remainder.
20	**Python enumerate() function** Returns an Enumerate Object.

21	**Python eval() function** Runs Code Within Program.
22	**Python exec() function** Executes Dynamically Created Program.
23	**Python filter() function** Constructs iterator from elements which are true.
24	**Python float() function** Returns floating point number from number, string.
25	**Python format() function** Returns formatted representation of a value.
26	**Python frozenset() function** Returns immutable frozenset object.
27	**Python getattr() function** Returns value of named attribute of an object.
28	**Python globals() function** Returns dictionary of current global symbol table.
29	**Python hasattr() function** Returns whether object has named attribute.
30	**Python hash() function** Returns hash value of an object.
31	**Python help() function** Invokes the built-in Help System.

32	**Python hex() function** Converts to Integer to Hexadecimal.
33	**Python id() function** Returns Identify of an Object.
34	**Python input() function** Reads and returns a line of string.
35	**Python int() function** Returns integer from a number or string.
36	**Python isinstance() function** Checks if a Object is an Instance of Class.
37	**Python issubclass() function** Checks if a Class is Subclass of another Class.
38	**Python iter() function** Returns an iterator.
39	**Python len() function** Returns Length of an Object.
40	**Python list() function** Creates a list in Python.
41	**Python locals() function** Returns dictionary of a current local symbol table.
42	**Python map() function** Applies Function and Returns a List.

43	**Python memoryview() function** Returns memory view of an argument.
44	**Python next() function** Retrieves next item from the iterator.
45	**Python object() function** Creates a featureless object.
46	**Python oct() function** Returns the octal representation of an integer.
47	**Python open() function** Returns a file object.
48	**Python ord() function** Returns an integer of the Unicode character.
49	**Python print() function** Prints the Given Object.
50	**Python property() function** Returns the property attribute.
51	**Python range() function** Returns a sequence of integers.
52	**Python repr() function** Returns a printable representation of the object.
53	**Python reversed() function** Returns the reversed iterator of a sequence.

54	**Python set() function** Constructs and returns a set.
55	**Python setattr() function** Sets the value of an attribute of an object.
56	**Python slice() function** Returns a slice object.
57	**Python sorted() function** Returns a sorted list from the given iterable.
58	**Python staticmethod() function** Transforms a method into a static method.
59	**Python str() function** Returns the string version of the object.
60	**Python super() function** Returns a proxy object of the base class.
61	**Python tuple() function** Returns a tuple.
62	**Python type() function** Returns the type of the object.
63	**Python vars() function** Returns the __dict__ attribute.
64	**Python zip() function** Returns an iterator of tuples.

65	**Python __import__() function** Function called by the import statement.
66	**Python unichr() function** Converts a Unicode code point to its corresponding Unicode character.
67	**Python long() function** Represents integers of arbitrary size.

Built-in Mathematical Functions:

There are some additional built-in functions that are used for performing only mathematical operations in Python, they are listed below −

Sr.No.	Function & Description
1	**Python abs() function** The abs() function returns the absolute value of x, i.e. the positive distance between x and zero.
2	**Python max() function** The max() function returns the largest of its arguments or largest number from the iterable (list or tuple).
3	**Python min() function** The function min() returns the smallest of its arguments i.e. the value closest to negative infinity, or smallest number from the iterable (list or tuple)
4	**Python pow() function** The pow() function returns x raised to y. It is equivalent to x**y. The function has third optional argument mod. If given, it returns (x**y) % mod value

5	**Python round() Function** round() is a built-in function in Python. It returns x rounded to n digits from the decimal point.
6	**Python sum() function** The sum() function returns the sum of all numeric items in any iterable (list or tuple). An optional start argument is 0 by default. If given, the numbers in the list are added to start value.

3.9. SCOPE OF VARIABLES:

The **scope of a variable** in Python is defined as the specific area or region where the variable is accessible to the user. The scope of a variable depends on where and how it is defined. In Python, a variable can have either a global or a local scope.

Types of Scope for Variables in Python:

On the basis of scope, the Python variables are classified in three categories

- Local Variables
- Global Variables
- Nonlocal Variables

Local Variables:

A local variable is defined within a specific function or block of code. It can only be accessed by the function or block where it was defined, and it has a limited scope. In other words, the scope of local variables is limited to the function they are defined in and attempting to access them outside of this function will result in an error. Always remember, multiple local variables can exist with the same name.

Example

The following example shows the scope of local variables.

```
def myfunction():
  a = 10
  b = 20
```

```
  print("variable a:", a)
  print("variable b:", b)
  return a+b
  print (myfunction())
```

In the above code, we have accessed the local variables through its function. Hence, the code will produce the following output −

variable a: 10

variable b: 20

30

Global Variables:

A global variable can be accessed from any part of the program, and it is defined outside any function or block of code. It is not specific to any block or function.

Example

The following example shows the scope of global variable. We can access them inside as well as outside of the function scope.

```
#global variables
name = 'TutorialsPoint'
marks = 50
def myfunction():
  # accessing inside the function
  print("name:", name)
  print("marks:", marks)
# function call
myfunction()
```

The above code will produce the following output −

name: TutorialsPoint

marks: 50

Nonlocal Variables:

The Python variables that are not defined in either local or global scope are called nonlocal

variables. They are used in nested functions.

Example

The following example demonstrates the how nonlocal variables works.

```
def yourfunction():
  a = 5
  b = 6
  # nested function
  def myfunction():
    # nonlocal function
    nonlocal a
    nonlocal b
    a = 10
    b = 20
    print("variable a:", a)
    print("variable b:", b)
    return a+b
  print (myfunction())
yourfunction()
```

The above code will produce the below output −

```
variable a: 10
variable b: 20
30
```

UNIT IV: OBJECT ORIENTED FEATURES

Introduction –Defining Classes – Public and Private Data member – Creating Object – Accessing class members – Using Objects. Constructors – Destructors – Introduction of simple Inheritance – Introduction of simple Polymorphism.

ERROR HANDLING: Run Time Errors – Exception Model.

4.1. OBJECT ORIENTED FEATURES – INTRODUCTION:

Object-Oriented Programming (OOP) is a powerful way of writing software that models real-world concepts into reusable code components called **objects** and **classes**. Python is a full-featured, object-oriented language, meaning it provides all the necessary constructs to implement this paradigm effectively.

The primary goal of OOP is to bind data (attributes) and the functions that operate on that data (methods) into single units, making code more organized, reusable, and maintainable.

Core Terminology:

The foundation of OOP relies on these basic concepts:

- **Class:** A blueprint or template that defines the structure and behavior for a specific category of objects.
- **Object:** A specific instance of a class. The class is the blueprint; the object is the actual item built from that blueprint.
- **Attribute:** A variable stored within an object that represents its state or data (e.g., a Car object might have a color attribute).
- **Method:** A function defined within a class that defines the actions or behaviors an object can perform (e.g., a Car object might have a drive() method).

The Four Pillars of OOP in Python:

Python's object-oriented nature is built upon four fundamental principles (often remembered by the acronym A PIE: Abstraction, Polymorphism, Inheritance, Encapsulation):

1. Encapsulation:

Encapsulation is the bundling of data and the methods that operate on that data within a single class unit.

- **How Python uses it:** The primary goal is to hide the internal workings and state of an object from the outside world. This protects data from being accidentally modified. In Python, this is achieved by defining methods that act as the only interface for modifying

internal attributes.

2. Abstraction:

Abstraction is about simplifying complexity by providing a simple, clean interface while hiding the complicated implementation details.

- **How Python uses it:** Users interact with an object through a simple set of methods without needing to understand the underlying code. The len() function is an abstraction: you use it to find the length of a list, string, or dictionary, without caring *how* it calculates the length for each specific type. Python allows developers to define abstract base classes (ABCs) using the standard library's abc module to enforce this structure.

3. Inheritance:

Inheritance is a mechanism for creating a new class (the *child* or *subclass*) that derives properties and behaviors from an existing class (the *parent* or *superclass*).

- **How Python uses it:** It promotes code reuse and establishes a hierarchical "is-a" relationship (e.g., a Dog *is a* kind of Animal).

Python

```
class Animal: # Parent Class
    def speak(self):
        return "Animal sound"
class Dog(Animal): # Child class inherits speak()
    def speak(self):
        return "Woof!"
```

Use code with caution.

4. Polymorphism:

Polymorphism (meaning "many forms") allows objects of different classes to respond to the same method call in different ways.

- **How Python uses it:** Python utilizes "Duck Typing" for polymorphism. As long as different objects share the same method name (interface), your code can treat them interchangeably. A single function can accept a Dog object or a Cat object and call a .speak() method on either one, getting the appropriate unique response.

4.2. DEFINING CLASSES:

In Python, we define a class using the class keyword, followed by the class name. Class names are traditionally written using **CapWords** or **CamelCase** convention.

Classes serve as blueprints for creating objects (instances).

Basic Class Definition:

A minimal class definition in Python looks like this:

Python

```
class Dog:
    pass # 'pass' is a placeholder when you have no attributes or methods yet
```

Use code with caution.

Adding Attributes and Methods:

Classes typically contain two primary components: attributes (data/state) and methods (functions/behavior).

Class Attributes

These attributes are shared by all instances of the class. They are defined directly within the class body but outside any method.

Python

```
class Dog:
    # Class Attribute (shared by all dogs)
    species = "Canis familiaris"
```

Use code with caution.

Instance Attributes and the __init__ Method:

Instance attributes are unique to each specific object created from the class. They are defined inside a special method called __init__.

The __init__ method (short for initialization) is a *constructor*. It is automatically called every time you create a new instance of the class.

- **self Parameter:** The first parameter of any method in a class definition *must* be self. When you call a method on an instance (e.g., my_dog.bark()), Python automatically passes the instance itself to the self parameter. It refers to the specific object that called the method.

Python

```
class Dog:
    species = "Canis familiaris"
    # The constructor method
    def __init__(self, name, breed):
        # Instance attributes (unique to each dog object)
        self.name = name
        self.breed = breed
```

Use code with caution.

Methods:

Methods are functions defined within the class. They always take self as their first argument, giving them access to the instance's specific data.

Python

```
class Dog:
    species = "Canis familiaris"
    def __init__(self, name, breed):
        self.name = name
        self.breed = breed
    # An instance method (defines behavior)
    def bark(self):
        return f"{self.name} says Woof!"
    def get_info(self):
        return f"{self.name} is a {self.breed} of species {self.species}."
```

Use code with caution.

Creating and Using Class Instances:

Once the class is defined, you can create objects (instances) by calling the class name as if it were a function. The arguments you pass (e.g., "Buddy", "Golden Retriever") are passed to the __init__ method.

Python

```
# Create two instances of the Dog class
```

```
dog1 = Dog("Buddy", "Golden Retriever")
dog2 = Dog("Max", "German Shepherd")
# Access attributes using dot notation
print(dog1.name)
print(dog2.breed)

# Call methods using dot notation
print(dog1.bark())
print(dog2.get_info())
# Access the class attribute (same for both)
print(dog1.species)
print(Dog.species)
```

Use code with caution.

Output:

```
Buddy
German Shepherd
Buddy says Woof!
Max is a German Shepherd of species Canis familiaris.
Canis familiaris
Canis familiaris
```

4.3. PUBLIC AND PRIVATE DATA MEMBER:

In Python, the distinction between "public" and "private" data members (attributes and methods) is implemented differently than in languages like Java or C++. Python does not have strict access modifiers (public, private, protected). Instead, it relies heavily on **naming conventions** and a mechanism called **name mangling** to suggest or enforce privacy.

By default, all members in a Python class are considered **public**.

1. Public Members:

Public members are accessible from anywhere, both inside the class methods and outside the class when an instance is used. All attributes and methods in Python are public by default.

Python

```
class Company:
    def __init__(self, name, employees):
        # Public attributes
        self.name = name
        self.employees = employees

    # Public method
    def display_info(self):
        print(f"Company: {self.name}, Employees: {self.employees}")
# Create an instance
my_company = Company("TechCorp", 150)
# Access public attributes directly from outside
print(f"Company name is: {my_company.name}")
# Call the public method
my_company.display_info()
```

Use code with caution.

2. Private Members (Conventions and Name Mangling):

Python uses specific naming conventions to indicate that a member should not be accessed directly from outside the class.

A. The Convention: Single Underscore Prefix (_member_name)

Prefixing an attribute or method name with a single underscore (_) is the standard *Pythonic* way to indicate a "protected" or "internal use only" member.

- **Behavior:** This is purely a convention for the developer. Python does not technically prevent external access; it signals to the user of the class that they should treat this member as internal implementation detail and ideally not touch it directly.
- **Use Case:** Often used when you intend for a method to be used by subclasses (inheritance) but not by general users of the class.

Python

```
class BankAccount:
```

```
    def __init__(self, initial_balance):
        self.balance = initial_balance  # Public attribute
        self._internal_log = []         # Protected attribute (Convention only)
    def deposit(self, amount):
        self.balance += amount
        self._log_transaction(f"Deposited {amount}")
    def _log_transaction(self, message):
        # This method should only be called internally
        self._internal_log.append(message)
        print(f"[LOGGED]: {message}")
my_account = BankAccount(100)
my_account.deposit(50)
# You CAN access the protected member, but you SHOULDN'T
print(my_account._internal_log)
```

Use code with caution.

B. Strong Privacy: Double Underscore Prefix (__member_name):

Prefixing a name with a *double* underscore (__) initiates a mechanism called **name mangling**.

- **Behavior:** Python renames the attribute internally to _ClassName__attribute_name. This makes accessing it from outside the class difficult (though not impossible). It serves as a strong signal for a "private" member.
- **Use Case:** Prevents accidental modification from outside the class and, more importantly, avoids name collisions if a subclass uses the same attribute name.

Python

```
class DataProcessor:
    def __init__(self, data):
        self.__raw_data = data  # Private attribute using name mangling
    def process(self):
        # Can be accessed normally inside the class
        return self.__analyze_data()
    def __analyze_data(self):
```

```
    # Private method using name mangling
    return f"Analyzing {self.__raw_data}"
processor = DataProcessor("financials")
# This works fine
print(processor.process())
# This will raise an AttributeError: 'DataProcessor' object has no attribute '__raw_data'
# print(processor.__raw_data)
# You can access it using the mangled name, but this is a hack:
# print(processor._DataProcessor__raw_data)
```

4.4. CREATING OBJECT:

In Python, an object is an instance of a class, which acts as a blueprint for creating objects. Each object contains data (variables) and methods to operate on that data. Python is object-oriented, meaning it focuses on objects and their interactions. For a better nderstanding of the concept of objects in Python. Let's consider an example, many of you have played CLASH OF CLANS, so let's assume base layout as the class which contains all the buildings, defenses, resources, etc. Based on these descriptions we make a village, here the village is the object in Python.

Creating an object:

When creating an object from a class, we use a special method called the constructor, defined as __init__(), to initialize the object's attributes. Example:

```
class Car:
   def __init__(self, model, price):
      self.model = model
      self.price = price
Audi = Car("R8", 100000)
print(Audi.model)
print(Audi.price)
```

Output

R8

100000

Explanation: Car class defines a blueprint for car objects. The __init__() constructor initializes the model and price attributes, using self to refer to the current object. When Audi = Car("R8", 100000) is executed, "R8" is assigned to model and 100000 to price. These attributes are accessed via dot notation, like Audi.model and Audi.price.

Accessing class members:

In Python, you can access both instance variables and methods of a class using an object. Instance variables are unique to each object, while methods define the behavior of the objects. Below are examples demonstrating how to access and interact with class members:

Example 1: In this example, we use methods to access and modify the car's attributes.

```
class Car:
    def __init__(self, model):
        self.model = model
    def setprice(self, price):
        self.price = price
    def getprice(self):
        return self.price
Audi = Car("R8")
Audi.setprice(1000000)
print(Audi.getprice())
```

Output

1000000

Explanation: Car class defines a blueprint for car objects with a constructor (__init__()) to initialize the model attribute. The setprice() method assigns a price and getprice() retrieves it. When Audi = Car("R8") is executed, the model is set to "R8", the price is set using setprice() and the price is accessed with getprice().

Example 2: In this example, we create multiple car objects and access the model and price attributes directly using the objects, without the need for methods.

```
class Car:
    vehicle = 'Car'
    def __init__(self, model, price):
        self.model = model
        self.price = price
Audi = Car("R8", 100000)
BMW = Car("I8", 10000000)
print(Audi.model, Audi.price)
print(BMW.model, BMW.price)
```

Output

R8 100000

I8 10000000

Explanation: Car class defines a blueprint with a class variable vehicle and a constructor to initialize model and price. When Audi = Car("R8", 100000) and BMW = Car("I8", 10000000) are executed, the attributes are set and accessed directly, like Audi.model and Audi.price.

Self keyword in Python objects:

In Python objects, the self keyword represents the current instance of the class. It is automatically passed to instance methods and is used to access and modify the object's own attributes and methods. By using self, each object can maintain its own separate state, ensuring that operations are performed on the correct instance. Example:

```
class Test:
    def __init__(self, a, b):
        self.country = a
        self.capital = b
    def fun(self):
        print("Capital of " + self.country + " is " + self.capital)
x = Test("India", "Delhi")
x.fun()
```

Output

Capital of India isDelhi

Explanation: Test class uses the __init__() constructor to initialize the country and capital attributes with self. When x is created with "India" and "Delhi", x.country and x.capital are set. The fun() method then accesses these attributes via self .

Deleting an object:

You can delete objects, variables or object properties using the del keyword. This removes the reference to the object or attribute from memory, allowing Python's garbage collector to reclaim the memory if no other references exist. Example:

```
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model
        Audi = Car("Audi", "A6") # creating obj
del Audi # deleting obj
print(Audi.brand)
```

Output

```
Hangup (SIGHUP)
Traceback (most recent call last):
  File "/home/guest/sandbox/Solution.py", line 10, in <module>
    print(Audi.brand)
          ^^^^
NameError: name 'Audi' is not defined
```

Explanation: After creating the Audi object, the del keyword deletes it. Attempting to access Audi.brand afterward results in an error because the object no longer exists.

4.5. ACCESSING CLASS MEMBERS:

In Python, class members (attributes and methods) can be accessed using dot notation (.), both from within the class definition and externally via object instances. The rules for access differ slightly depending on whether the member is a class member (shared by all

instances) or an instance member (unique to a specific object).

Accessing Instance Members:

Instance members are unique to each object and are accessed using the instance name followed by the dot operator.

1. Accessing from Outside the Class

Once an object is created, you can read or modify its instance attributes directly.

Python

```
class Student:
    def __init__(self, name, grade):
        self.name = name      # Instance attribute
        self.grade = grade    # Instance attribute
    def display_student(self):
        print(f"{self.name} is in grade {self.grade}")
# Create an instance
student1 = Student("Alice", 10)
# Accessing attributes externally
print(f"Student 1 Name: {student1.name}")
print(f"Student 1 Grade: {student1.grade}")
# Modifying attributes externally
student1.grade = 11
print(f"Student 1 new Grade: {student1.grade}")

# Calling a method externally
student1.display_student()
```

Use code with caution.

2. Accessing from Inside the Class (Methods):

Within the class's methods (like __init__ or display_student above), instance members are accessed using the self keyword, which refers to the current object instance.

Python

```
class Student:
```

```
    def __init__(self, name, grade):
        # Accessing instance attributes using 'self'
        self.name = name
        self.grade = grade
```

Use code with caution.

Accessing Class Members:

Class members are defined directly in the class body and are shared across all instances.

1. Accessing using the Class Name (Recommended)

The most direct and explicit way to access a class member is by using the ClassName.member_name syntax.

Python

```
class Car:
    # Class attribute
    wheels = 4
# Accessing the class attribute directly using the class name
print(f"All cars have {Car.wheels} wheels.")
```

Use code with caution.

2. Accessing using an Instance Name:

You can also access class members via an instance (instance.member_name). If the instance doesn't have an instance attribute with that name, Python automatically looks up the value in the class scope.

Python

```
my_car = Car()
your_car = Car()
print(f"My car wheels: {my_car.wheels}")
print(f"Your car wheels: {your_car.wheels}")
```

Use code with caution.

3. Accessing from Inside the Class (Methods):

You can access class members inside a method using either self. member_name or ClassName.member_name.

Python

```
class Car:
    wheels = 4
    def display_wheels(self):
        # Access using self (Python looks up the chain)
        print(f"Using self: {self.wheels} wheels")
        # Access using Class Name (Explicit reference)
        print(f"Using Car: {Car.wheels} wheels")
my_car = Car()
my_car.display_wheels()
```

4.6. USING OBJECTS:

Using objects in Python involves creating instances of a class and then interacting with their attributes (data) and methods (behaviors). Here is a step-by-step guide on how to effectively use objects in Python, building on the concepts of defining classes and creating instances.

Prerequisites: Defining the Class:

Before using an object, you must define its blueprint (the class). Let's use a Book class as an example:

Python

```
class Book:
    # Class attribute
    material_type = "paperback"
    # Constructor method to initialize instance attributes
    def __init__(self, title, author, pages):
        self.title = title    # Instance attribute
```

```
        self.author = author # Instance attribute
        self.pages = pages   # Instance attribute
        self.is_opened = False # Initial state
    # Instance method
    def open_book(self):
        if not self.is_opened:
            self.is_opened = True
            return f"Opening '{self.title}'."
        else:
            return f"'{self.title}' is already open."
    # Instance method to get info
    def get_summary(self):
        return f"'{self.title}' by {self.author}, {self.pages} pages."
```

Step 1: Create an Object (Instantiation)

You create a concrete object (an instance) from the class blueprint:

Python

```
# Create two different instances of the Book class
book1 = Book("The Hitchhiker's Guide to the Galaxy", "Douglas Adams", 192)
book2 = Book("Dune", "Frank Herbert", 412)
```

Use code with caution.

Step 2: Accessing Attributes

You can access the data stored within an object using the **dot notation** (.). You can read the values or change them.

Reading Attributes:

Python

```
print(f"Book 1 Title: {book1.title}")
print(f"Book 2 Author: {book2.author}")
print(f"Book 1 Current State (is_opened): {book1.is_opened}")
# Output:
# Book 1 Title: The Hitchhiker's Guide to the Galaxy
```

 # Book 2 Author: Frank Herbert

 # Book 1 Current State (is_opened): False

Modifying Attributes:

You can change the state of a specific object instance by reassigning its attribute value:

Python

book1.pages = 200 *# Update the number of pages for book1*

print(f"Book 1 new pages: {book1.pages}")

 # Output:

 # Book 1 new pages: 200

Use code with caution.

Step 3: Calling Methods

Methods are functions defined within the class that define the object's behaviors. You call them using dot notation, just like attributes, but with parentheses () to execute them.

Python

 # Call the 'open_book' method on book1

message1 = book1.open_book()

print(message1)

 # Call the 'get_summary' method on book2

summary2 = book2.get_summary()

print(summary2)

 # Check the state after calling the method

print(f"Book 1 New State (is_opened): {book1.is_opened}")

 # Output:

 # Opening 'The Hitchhiker's Guide to the Galaxy'.

 # 'Dune' by Frank Herbert, 412 pages.

 # Book 1 New State (is_opened): True

Step 4: Using Objects in Data Structures

Objects are first-class citizens in Python, meaning you can store them in lists, dictionaries, or pass them as arguments to functions.

Python

```
# Store objects in a list
my_books = [book1, book2]
# Loop through the list of objects
for book in my_books:
    print(f"Summary: {book.get_summary()}")
    # We can call methods and access attributes within the loop
```

4.7. CONSTRUCTORS:

In Python, a constructor is a special method that is called automatically when an object is created from a class. Its main role is to initialize the object by setting up its attributes or state. The method __new__ is the constructor that creates a new instance of the class while __init__ is the initializer that sets up the instance's attributes after creation. These methods work together to manage object creation and initialization.

__new__ Method:

This method is responsible for creating a new instance of a class. It allocates memory and returns the new object. It is called before __init__.

```
class ClassName:
    def __new__(cls, parameters):
        instance = super(ClassName, cls).__new__(cls)
        return instance
```

To learn more, please refer to "__new__" method

__init__ Method:

This method initializes the newly created instance and is commonly used as a constructor in Python. It is called immediately after the object is created by __new__ method and is responsible for initializing attributes of the instance.

Syntax:

```
class ClassName:
    def __init__(self, parameters):
        self.attribute = value
```

Note: It is called after __new__ and does not return anything (it returns None by default).

To learn more, please refer to "__init__" method

Differences Between __init__ and __new__

__new__ method:

- Responsible for creating a new instance of the class.
- Rarely overridden but useful for customizing object creation and especially in singleton or immutable objects.

__init__ method:

- Called immediately after __new__.
- Used to initialize the created object.

Types of Constructors:

Constructors can be of two types.

1. Default Constructor

A default constructor does not take any parameters other than self. It initializes the object with default attribute values.

```
class Car:
    def __init__(self):

        #Initialize the Car with default attributes
        self.make = "Toyota"
        self.model = "Corolla"
        self.year = 2020
# Creating an instance using the default constructor
car = Car()
print(car.make)
print(car.model)
print(car.year)
```

Output

```
Toyota
```

Corolla

2020

2. Parameterized Constructor:

A parameterized constructor accepts arguments to initialize the object's attributes with specific values.

```
class Car:
    def __init__(self, make, model, year):
            #Initialize the Car with specific attributes.
        self.make = make
        self.model = model
        self.year = year
# Creating an instance using the parameterized constructor
car = Car("Honda", "Civic", 2022)
print(car.make)
print(car.model)
print(car.year)
```

Output

Honda

Civic

2022

4.8. DESTRUCTORS:

Destructors are called when an object gets destroyed. In Python, destructors are not needed as much as in C++ because Python has a garbage collector that handles memory management automatically.

The **__del__()** method is a known as a destructor method in Python. It is called when all references to the object have been deleted i.e when an object is garbage collected.

Syntax of destructor declaration:

```
def __del__(self):
  # body of destructor
```

Note : A reference to objects is also deleted when the object goes out of reference or when the program ends.

Example 1 : Here is the simple example of destructor. By using del keyword, we deleted the all references of object 'obj', therefore destructor invoked automatically.

```
# Python program to illustrate destructor
class Employee:
    # Initializing
    def __init__(self):
        print('Employee created.')
    # Deleting (Calling destructor)
    def __del__(self):
        print('Destructor called, Employee deleted.')
obj = Employee()
del obj
```

Output

```
Employee created.
Destructor called, Employee deleted.
```

Note : The destructor was called **after the program ended** or when all the references to object are deleted i.e when the reference count becomes zero, not when object went out of scope.

Example 2: This example gives the explanation of above-mentioned note. Here, notice that the destructor is called after the 'Program End...' printed.

```
# Python program to illustrate destructor
 class Employee:
     # Initializing
    def __init__(self):
        print('Employee created')
     # Calling destructor
    def __del__(self):
        print("Destructor called")
```

```
def Create_obj():
   print('Making Object...')
   obj = Employee()
   print('function end...')
   return obj
print('Calling Create_obj() function...')
obj = Create_obj()
print('Program End...')
```

Output

```
Calling Create_obj() function...
Making Object...
Employee created
function end...
Program End...
Destructor called
```

Example 3: Now, consider the following example :

```
# Python program to illustrate destructor
class A:
   def __init__(self, bb):
      self.b = bb
class B:
   def __init__(self):
      self.a = A(self)
   def __del__(self):
      print("die")
def fun():
   b = B()
fun()
```

Output

die

In this example when the function fun() is called, it creates an instance of class B which passes itself to class A, which then sets a reference to class B and resulting in a **circular reference**. Generally, Python's garbage collector which is used to detect these types of cyclic references would remove it but in this example the use of custom destructor marks this item as "uncollectable". Simply, it doesn't know the order in which to destroy the objects, so it leaves them. Therefore, if your instances are involved in circular references they will live in memory for as long as the application run.

NOTE : The problem mentioned in example 3 is resolved in newer versions of python, but it still exists in version < 3.4 .

Example: Destruction in recursion

In Python, you can define a destructor for a class using the __del__() method. This method is called automatically when the object is about to be destroyed by the garbage collector. Here's an example of how to use a destructor in a recursive function:

```
class RecursiveFunction:
    def __init__(self, n):
        self.n = n
        print("Recursive function initialized with n =", n)
    def run(self, n=None):
        if n is None:
            n = self.n
        if n <= 0:
            return
        print("Running recursive function with n =", n)
        self.run(n-1)
    def __del__(self):
        print("Recursive function object destroyed")
# Create an object of the class
```

```
obj = RecursiveFunction(5)
# Call the recursive function
obj.run()
# Destroy the object
del obj
```

Output

```
('Recursive function initialized with n =', 5)
('Running recursive function with n =', 5)
('Running recursive function with n =', 4)
('Running recursive function with n =', 3)
('Running recursive function with n =', 2)
('Running recursive function with n =', 1)
Recursive function object destroyed
```

In this example, we define a class RecursiveFunction with an __init__() method that takes in a parameter n. This parameter is stored as an attribute of the object.

We also define a run() method that takes in an optional parameter n. If n is not provided, it defaults to the value of self.n. The run() method runs a recursive function that prints a message to the console and calls itself with n-1.

We define a destructor using the __del__() method, which simply prints a message to the console indicating that the object has been destroyed.

We create an object of the class RecursiveFunction with n set to 5, and call the run() method. This runs the recursive function, printing a message to the console for each call.

Finally, we destroy the object using the del statement. This triggers the destructor, which prints a message to the console indicating that the object has been destroyed.

Note that in this example, the recursive function will continue running until n reaches 0. When n is 0, the function will return and the object will be destroyed by the garbage collector. The destructor will then be called automatically.

Advantages of using destructors in Python:

- **Automatic cleanup**: Destructors provide automatic cleanup of resources used by an

- object when it is no longer needed. This can be especially useful in cases where resources are limited, or where failure to clean up can lead to memory leaks or other issues.
- **Consistent behavior:** Destructors ensure that an object is properly cleaned up, regardless of how it is used or when it is destroyed. This helps to ensure consistent behavior and can help to prevent bugs and other issues.
- **Easy to use:** Destructors are easy to implement in Python, and can be defined using the __del__() method.
- **Supports object-oriented programming:** Destructors are an important feature of object-oriented programming, and can be used to enforce encapsulation and other principles of object-oriented design.
- **Helps with debugging:** Destructors can be useful for debugging, as they can be used to trace the lifecycle of an object and determine when it is being destroyed.

4.9. INTRODUCTION OF SIMPLE INHERITANCE:

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows a class (called a child or derived class) to inherit attributes and methods from another class (called a parent or base class). In this article, we'll explore inheritance in Python.

Example: Here, we create a parent class Animal that has a method info(). Then we create a child classes Dog that inherit from Animal and add their own behavior.

```
class Animal:
    def __init__(self, name):
        self.name = name
    def info(self):
        print("Animal name:", self.name)
class Dog(Animal):
    def sound(self):
        print(self.name, "barks")
d = Dog("Buddy")
d.info()      # Inherited method
```

d.sound()

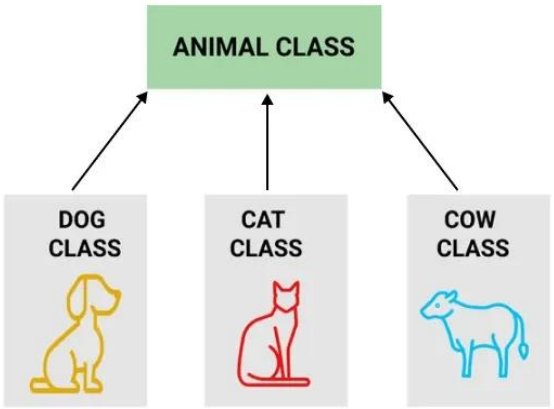
Output

Animal name: Buddy

Buddy barks

Explanation:

- class Animal: Defines the parent class.
- info(): Prints the name of the animal.
- class Dog(Animal): Defines Dog as a child of Animal class.
- d.info(): Calls parent method info() and d.sound(): Calls child method.



Why do we need Inheritance

- Promotes code reusability by sharing attributes and methods across classes.
- Models real-world hierarchies like Animal -> Dog or Person -> Employee.
- Simplifies maintenance through centralized updates in parent classes.
- Enables method overriding for customized subclass behavior.
- Supports scalable, extensible design using polymorphism.

super() Function

super() function is used to call the parent class's methods. In particular, it is commonly used in the child class's __init__() method to initialize inherited attributes. This way, the child class can leverage the functionality of the parent class.

Example: Here, Dog uses super() to call Animal's constructor

```
# Parent Class: Animal
class Animal:
```

```
    def __init__(self, name):
        self.name = name
    def info(self):
        print("Animal name:", self.name)
# Child Class: Dog
class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name)   # Call parent constructor
        self.breed = breed
    def details(self):
        print(self.name, "is a", self.breed)
d = Dog("Buddy", "Golden Retriever")
d.info()      # Parent method
d.details()   # Child method
```

Output

```
Animal name: Buddy
Buddy is a Golden Retriever
```

Explanation:

- The super() function is used inside __init__() method of Dog to call the constructor of Animal and initialize inherited attribute (name).
- This ensures that parent class functionality is reused without needing to rewrite the code in the child class.

Types of Python Inheritance:

Inheritance be used in different ways depending on how many parent and child classes are involved. They help model real-world relationships more effectively and allow flexibility in code reuse.

Python supports several types of inheritance, let's explore it one by one:

1. Single Inheritance

In single inheritance, a child class inherits from just one parent class.

Example: This example shows a child class Employee inheriting a property from the parent class Person.

```
class Person:
    def __init__(self, name):
        self.name = name
class Employee(Person):  # Employee inherits from Person
    def show_role(self):
        print(self.name, "is an employee")
emp = Employee("Sarah")
print("Name:", emp.name)
emp.show_role()
```

Output

Name: Sarah

Sarah is an employee

Explanation: Here Employee inherits name from Person, it also defines its own method show_role().

2. Multiple Inheritance

In multiple inheritance, a child class can inherit from more than one parent class.

Example: This example demonstrates Employee inheriting properties from two parent classes: Person and Job.

```
class Person:
    def __init__(self, name):
        self.name = name
class Job:
    def __init__(self, salary):
        self.salary = salary
class Employee(Person, Job):  # Inherits from both Person and Job
    def __init__(self, name, salary):
        Person.__init__(self, name)
        Job.__init__(self, salary)
    def details(self):
        print(self.name, "earns", self.salary)
emp = Employee("Jennifer", 50000)
```

```
emp.details()
```

Output

Jennifer earns 50000

Explanation: Here Employee gets attributes from both Person and Job and It can access both name and salary.

3. Multilevel Inheritance

In multilevel inheritance, a class is derived from another derived class (like a chain).

Example: This example shows Manager inheriting from Employee, which in turn inherits from Person.

```
class Person:
    def __init__(self, name):
        self.name = name
class Employee(Person):
    def show_role(self):
        print(self.name, "is an employee")
class Manager(Employee):  # Manager inherits from Employee
    def department(self, dept):
        print(self.name, "manages", dept, "department")
mgr = Manager("Joy")
mgr.show_role()
mgr.department("HR")
```

Output

Joy is an employee

Joy manages HR department

Explanation: Here Manager inherits from Employee and Employee inherits from Person. So Manager can use methods from both parent and grandparent.

4. Hierarchical Inheritance

In hierarchical inheritance, multiple child classes inherit from the same parent class.

Example: This example demonstrates two child classes (Employee and Intern) inheriting from a single parent class Person.

```
class Person:
```

```
    def __init__(self, name):
        self.name = name
class Employee(Person):
    def role(self):
        print(self.name, "works as an employee")
class Intern(Person):
    def role(self):
        print(self.name, "is an intern")
emp = Employee("David")
emp.role()
intern = Intern("Eva")
intern.role()
```

Output

David works as an employee

Eva is an intern

Explanation: Both Employee and Intern inherit from Person. They share the parent's property (name) but implement their own methods.

5. Hybrid Inheritance

Hybrid inheritance is a combination of more than one type of inheritance.

Example: This example demonstrates TeamLead inheriting from both Employee (which inherits Person) and Project, combining multiple inheritance types.

```
class Person:
    def __init__(self, name):
        self.name = name
        class Employee(Person):
    def role(self):
        print(self.name, "is an employee")
class Project:
    def __init__(self, project_name):
        self.project_name = project_name
class TeamLead(Employee, Project):  # Hybrid Inheritance
```

```
    def __init__(self, name, project_name):
        Employee.__init__(self, name)
        Project.__init__(self, project_name)
    def details(self):
        print(self.name, "leads project:", self.project_name)
lead = TeamLead("Sophia", "AI Development")
lead.role()
lead.details()
```

Output

Sophia is an employee

Sophia leads project: AI Development

Explanation: Here TeamLead inherits from Employee (which already inherits Person) and also from Project. This combines single, multilevel and multiple inheritance -> hybrid.

4.10. INTRODUCTION OF SIMPLE POLYMORPHISM:

Polymorphism means "many forms". It refers to the ability of an entity (like a function or object) to perform different actions based on the context. Technically, in Python, polymorphism allows same method, function or operator to behave differently depending on object it is working with. This makes code more flexible and reusable.

Why do we need Polymorphism?

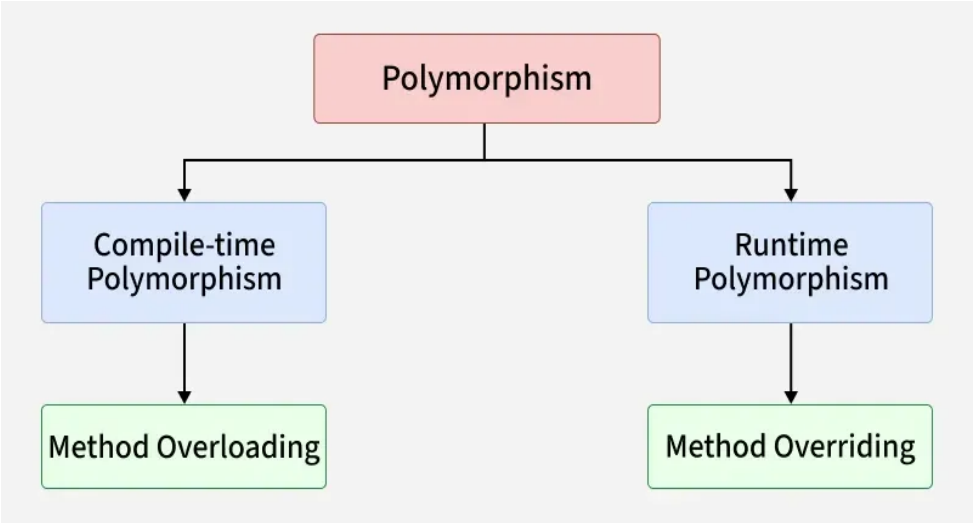
- Ensures consistent interfaces across different classes.
- Allows objects to respond differently to the same method call.
- Promotes loose coupling by relying on shared behavior, not specific types.
- Enables writing flexible, reusable code that works across types.
- Simplifies testing and future extension of code.

Example of Polymorphism:

A remote control can operate multiple devices like a TV, AC or music system. You press the power button and each device responds differently TV turns on, AC starts cooling, music system plays music. Polymorphism here means same interface (power button), but different behavior based on device (object).

Types of Polymorphism:

Polymorphism in Python refers to ability of the same method or operation to behave differently based on object or context. It mainly includes compile-time and runtime polymorphism.



1. Compile-time Polymorphism:

Compile-time polymorphism means deciding which method or operation to run during compilation, usually through method or operator overloading. Languages like Java or C++ support this. But Python doesn't because it's dynamically typed it resolves method calls at runtime, not during compilation. So, true method overloading isn't supported in Python, though similar behavior can be achieved using default or variable arguments.

Example:

This code demonstrates method overloading in Python using default and variable-length arguments. The multiply() method works with different numbers of inputs, mimicking compile-time polymorphism.

```
class Calculator:
    def multiply(self, a=1, b=1, *args):
        result = a * b
        for num in args:
            result *= num
        return result
```

```
# Create object
calc = Calculator()
# Using default arguments
print(calc.multiply())
print(calc.multiply(4))
# Using multiple arguments
print(calc.multiply(2, 3))
print(calc.multiply(2, 3, 4))
```

Output

```
1
4
6
24
```

2. Runtime Polymorphism (Overriding):

Runtime polymorphism means that the behavior of a method is decided while program is running, based on the object calling it. In Python, this happens through Method Overriding a child class provides its own version of a method already defined in the parent class. Since Python is dynamic, it supports this, allowing same method call to behave differently for different object types.

Example:

This code shows runtime polymorphism using method overriding. The sound() method is defined in base class Animal and overridden in Dog and Cat. At runtime, correct method is called based on object's class.

```
class Animal:
    def sound(self):
        return "Some generic sound"
class Dog(Animal):
    def sound(self):
        return "Bark"
class Cat(Animal):
    def sound(self):
```

```
        return "Meow"
# Polymorphic behavior
animals = [Dog(), Cat(), Animal()]
for animal in animals:
    print(animal.sound())
```

Output

Bark

Meow

Some generic sound

Explanation: Here, sound method behaves differently depending on whether object is a Dog, Cat or Animal and this decision happens at runtime. This dynamic nature makes Python particularly powerful for runtime polymorphism.

Polymorphism in Built-in Functions:

Python's built-in functions like len() and max() are polymorphic they work with different data types and return results based on type of object passed. This showcases it's dynamic nature, where same function name adapts its behavior depending on input.

Example:

This code demonstrates polymorphism in Python's built-in functions handling strings, lists, numbers and characters differently while using same function name.

```
print(len("Hello"))  # String length
print(len([1, 2, 3]))  # List length
print(max(1, 3, 2))  # Maximum of integers
print(max("a", "z", "m"))  # Maximum in strings
```

Output

5

3

3

z

Polymorphism in Functions:

In Python, polymorphism lets functions accept different object types as long as they support needed behavior. Using duck typing, Python focuses on whether an object has right method

not its type allowing flexible and reusable code.

Example:

This code demonstrates polymorphism using duck typing as perform_task() function works with different object types (Pen and Eraser), as long as they have a .use() method showing flexible and reusable function design.

```
class Pen:
    def use(self):
        return "Writing"
class Eraser:
    def use(self):
        return "Erasing"
def perform_task(tool):
    print(tool.use())
perform_task(Pen())
perform_task(Eraser())
```

Output

Writing

Erasing

Polymorphism in Operators:

In Python, same operator (+) can perform different tasks depending on operand types. This is known as operator overloading. This flexibility is a key aspect of polymorphism in Python.

Example:

This code shows operator polymorphism as + operator behaves differently based on data types adding integers, concatenating strings and merging lists all using same operator.

```
print(5 + 10)  # Integer addition
print("Hello " + "World!")  # String concatenation
print([1, 2] + [3, 4])  # List concatenation
```

Output

15

Hello World!

[1, 2, 3, 4]

4.11. ERROR HANDLING:

Error handling in Python allows programs to continue running smoothly even when unexpected errors or exceptional situations (like a file not being found, or a user entering text instead of a number) occur.

The primary mechanism for handling these events is the **try...except** block.

The try...except Block:

The basic structure involves placing the code that might raise an error inside a try block. If an error occurs, the execution immediately stops within the try block and jumps to the code within the except block.

Basic Syntax:

Python

```
try:
    # Code that might cause an error
    result = 10 / 0
except ZeroDivisionError:
    # Code to run if that specific error occurs
    print("Error: Cannot divide by zero!")
```

Use code with caution.

Example 1: Handling Invalid User Input

A common scenario is dealing with user input, where a ValueError might occur if you try to convert non-numeric text to an integer.

Python

```
def get_age():
    try:
        # Try to convert user input to an integer
        age = int(input("Enter your age: "))
        print(f"Your age is: {age}")
    except ValueError:
```

```
    # Run this block if the conversion fails
    print("Invalid input. Please enter a numerical value for your age.")
get_age()
```

Use code with caution.

Catching Specific Exceptions

It is best practice to catch specific types of errors rather than using a generic except block. This helps in debugging and prevents catching unexpected errors you didn't intend to handle.

Python

```
try:
   # Example: Accessing a non-existent file
   with open('non_existent_file.txt', 'r') as f:
      content = f.read()
except FileNotFoundError:
   print("The file was not found. Please check the file path.")
except IOError:
   # Handles general input/output errors
   print("An I/O error occurred.")
```

Use code with caution.

You can also use the as e syntax to access the error message provided by Python itself:

Python

```
try:
   my_list = [1, 2]
   print(my_list[3]) # This causes an IndexError
except IndexError as e:
   print(f"An index error occurred: {e}")
   # Output: An index error occurred: list index out of range
```

Use code with caution.

The else and finally Blocks:

The try...except block can be extended with two optional blocks: else and finally.

The else Block:

The else block runs only if the code inside the try block executes successfully *without* raising any exceptions.

Python

```
try:
    # This might fail if denominator is 0
    numerator = 10
    denominator = 2
    result = numerator / denominator
except ZeroDivisionError:
    print("Cannot divide by zero.")
else:
    # This runs only if NO error occurred in 'try'
    print(f"The result is: {result}")
```

Use code with caution.

The finally Block:

The finally block always executes, regardless of whether an exception was raised or handled. It is typically used for cleanup operations that must happen (e.g., closing a file or network connection).

Python

```
try:
    f = open("myfile.txt", "w")
    f.write("Some data")
except IOError:
    print("Error writing to file.")
finally:
    # This will always run, ensuring the file is closed
    if 'f' in locals() and not f.closed:
        f.close()
        print("File closed successfully in finally block.")
```

Use code with caution.

Raising Your Own Exceptions:

You can manually trigger an error using the raise keyword. This is useful for enforcing business logic or indicating a specific error condition in your own functions.

Python

```
def validate_age(age):
    if age < 0:
        raise ValueError("Age cannot be negative!")
    if age >= 100:
        raise ValueError("Age seems too high.")
    return age
# Use the function within a try block to handle your custom error
try:
    validate_age(-5)
except ValueError as e:
    print(f"Validation failed: {e}")
    # Output: Validation failed: Age cannot be negative!
```

4.12. RUN TIME ERRORS:

Run-time errors, also known as **exceptions**, occur when a program is syntactically correct (it passes Python's parser) but encounters a problem during execution. These errors interrupt the normal flow of the program if they are not handled properly using try...except blocks.

Here are the most common types of run-time errors (exceptions) encountered in Python:

1. Name Error:

This error occurs when you try to use a variable or function name that Python does not recognize (hasn't been defined or imported).

python

```
# Example: Typos or using an undefined variable
x = 10
print(y)
# NameError: name 'y' is not defined
# Example: Forgetting to import a module
```

math.pi

 # NameError: name 'math' is not defined

Use code with caution.

2. Type Error:

This error occurs when an operation or function is applied to an object of an inappropriate type.

python

 # Example: Adding a string and an integer

result = 5 + "10"

 # Type Error: unsupported operand type(s) for +: 'int' and 'str'

 # Example: Calling a function with too many arguments

def greet(name):

print (f" Hello, {name}")

greet("Alice", "Bob")

 # Type Error: greet() takes 1 positional argument but 2 were given

Use code with caution.

3. Value Error:

This error occurs when a function receives an argument of the correct *type* but an inappropriate *value*. This is common when converting data.

python

 # Example: Trying to convert a non-numeric string to an integer

age = int("hello")

 # ValueError: invalid literal for int() with base 10: 'hello'

Use code with caution.

4. ZeroDivisionError:

This error occurs when you try to divide a number by zero.

python

 # Example: Basic division by zero

result = 10 / 0

 # ZeroDivisionError: division by zero

Use code with caution.

5. Index Error:

This error occurs when you try to access an index that is outside the bounds of a list, tuple, or string.

python

 # Example: Accessing a non-existent list item

my_list =

print(my_list)

 # IndexError: list index out of range

Use code with caution.

6. KeyError:

This error occurs when you try to access a non-existent key in a dictionary.

python

 # Example: Accessing a non-existent dictionary key

my_dict = {'a': 1, 'b': 2}

print(my_dict)

 # KeyError: 'c'

Use code with caution.

7. FileNotFoundError:

This error occurs when the code tries to open a file that does not exist at the specified location.

python

 # Example: Trying to open a file that isn't there

f = open("missing_file.txt", "r")

 # FileNotFoundError: [Errno 2] No such file or directory: 'missing_file.txt'

Use code with caution.

Handling Run-Time Errors:

To prevent these errors from crashing your program, you must use the try...except block mechanism described in Error Handling:

Python

try:

```
    # Code that might crash
    print(10 / 0)
except ZeroDivisionError:
    # Code that runs instead of crashing
    print("Caught a division by zero error!")
print("Program continues running.")
```

4.12. EXCEPTION MODEL:

The exception model in Python is a robust and flexible system for managing run-time errors. It is built around the fundamental concepts of exceptions as objects, a hierarchy of exception classes, and the structured flow of control using the try...except...finally block mechanism.

1. Exceptions are Objects (First-Class Citizens):

In Python, errors are not just fatal events; they are instances of specific classes (objects) that inherit from a common base class called BaseException.

When an error occurs, Python "raises" an instance of the corresponding exception class (e.g., ValueError, ZeroDivisionError, IndexError). This object contains information about the error, such as the type of error, the line number where it occurred, and an associated error message.

2. The Exception Hierarchy:

All built-in exceptions form a hierarchy, which allows for granular or broad error handling. The base of most standard errors is the Exception class.

BaseException

```
├── Exception
    ├── StandardError
    |   ├── ArithmeticError
    |   |   ├── ZeroDivisionError
    |   ├── ValueError
    |   ├── IndexError
    |   ├── KeyError
    |   └── ... and many more
```

└── ... (SystemExit, KeyboardInterrupt etc.)

This hierarchy is crucial for the except block logic:

- Handling Broadly: An except Exception: block will catch almost all common run-time errors.
- Handling Specifically: An except ValueError: block only catches that specific type of error.

When you catch a parent class in the hierarchy (e.g., ArithmeticError), you automatically catch all its children (e.g., ZeroDivisionError).

3. The try...except...finally Flow (The Exception Handling Mechanism):

The mechanism Python uses to handle exceptions is the try...except block, which alters the standard flow of control when an error is encountered:

1. try block: Contains the code that might raise an exception. Execution proceeds normally here unless an error occurs.
2. except block(s): If an error occurs in the try block, Python immediately stops execution there and searches for a matching except block to handle the specific exception type raised.
3. else block (optional): Code that executes *only* if the try block completed successfully without any errors.
4. finally block (optional): Code that *always* executes, regardless of whether an exception occurred, was caught, or was left unhandled. This is used for cleanup operations.

python

```
try:
    # 1. Code attempts to run
    result = 10 / 0
except ZeroDivisionError as e:
    # 2. Execution jumps here if error occurs
    print(f'Handled error: {e}")
else:
    # 3. Code runs ONLY if no error in try
    print(f'Calculation successful: {result}")
finally:
```

```
    # 4. Code runs ALWAYS at the end
    print("Execution of try block concluded.")
```

Use code with caution.

4. Unhandled Exceptions (Propagation):

If an exception is raised in a function and there is no try...except block to catch it immediately, the exception propagates up the call stack.

- The function stops executing, and the exception is passed to the function that called it.
- This continues up the chain until an appropriate except block is found.
- If the exception reaches the top level of the program without being handled, the program terminates, and Python prints a Traceback message detailing where the error occurred.

5. Raising Exceptions (raise keyword):

You can manually initiate an exception using the raise keyword. This allows you to enforce validation rules or signal specific error conditions in your own code, integrating your custom errors into Python's standard exception model.

python

```
def check_positive(number):
    if number < 0:
        raise ValueError("The number must be positive.")
    return number
```

UNIT V: ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

Introduction – History of AI – Applications of AI – Defining Algorithm – A* Algorithm. **DATA SCIENCE:** Introduction – Defining Data, Information and Data structure – Basic Concept of Probability and Statistics.

5.1. INTRODUCTION OF ARTIFICIAL INTELLIGENCE:

In today's world, technology is growing very fast, and we are getting in touch with different new technologies day by day. Here, one of the booming technologies of computer science is Artificial Intelligence which is ready to +create a new revolution in the world by making intelligent machines. The Artificial Intelligence is now all around us. It is currently working with a variety of subfields, ranging from general to specific, such as self-driving cars, playing chess, proving theorems, playing music, Painting, etc. AI is one of the fascinating and universal fields of Computer science which has a great scope in future. AI holds a tendency to cause a machine to work as a human.

Artificial Intelligence is composed of two words Artificial and Intelligence, where Artificial defines "man-made," and intelligence defines "thinking power", hence AI means "a man-made thinking power." So, we can define AI as: "It is a branch of computer science by which we can create intelligent machines which can behave like a human, think like humans, and able to make decisions." Artificial Intelligence exists when a machine can have human-based skills such as learning, reasoning, and solving problems With Artificial Intelligence you do not need to preprogram a machine to do some work, despite that you can create a machine

with programmed algorithms which can work with own intelligence, and that is the awesomeness of AI. It is believed that AI is not a new technology, and some people says that as per Greek myth, there were Mechanical men in early days which can work and behave like humans.

Why Artificial Intelligence?

Before Learning about Artificial Intelligence, we should know that what is the importance of AI and why should we learn it. Following are some main reasons to learn about AI:

- With the help of AI, you can create such software or devices which can solve real-world problems very easily and with accuracy such as health issues, marketing, traffic issues, etc.
- With the help of AI, you can create your personal virtual Assistant, such as Cortana, Google Assistant, Siri, etc.
- With the help of AI, you can build such Robots which can work in an environment where survival of humans can be at risk. AI opens a path for other new technologies, new devices, and new Opportunities.

Advantages of Artificial Intelligence:

Following are some main advantages of Artificial Intelligence:

- High Accuracy with less errors: AI machines or systems are prone to less errors and high accuracy as it takes decisions as per pre-experience or information.
- High-Speed: AI systems can be of very high-speed and fast-decision making, because of that AI systems can beat a chess champion in the Chess game.
- High reliability: AI machines are highly reliable and can perform the same action multiple times with high accuracy.
- Useful for risky areas: AI machines can be helpful in situations such as defusing a bomb, exploring the ocean floor, where to employ a human can be risky.
- Digital Assistant: AI can be very useful to provide digital assistant to the users such as AI technology is currently used by various Ecommerce websites to show the products as per customer requirement.
- Useful as a public utility: AI can be very useful for public utilities such as a self-driving car which can make our journey safer and hassle-free, facial recognition for security purpose, Natural language processing to communicate with the human in human-

language, etc.

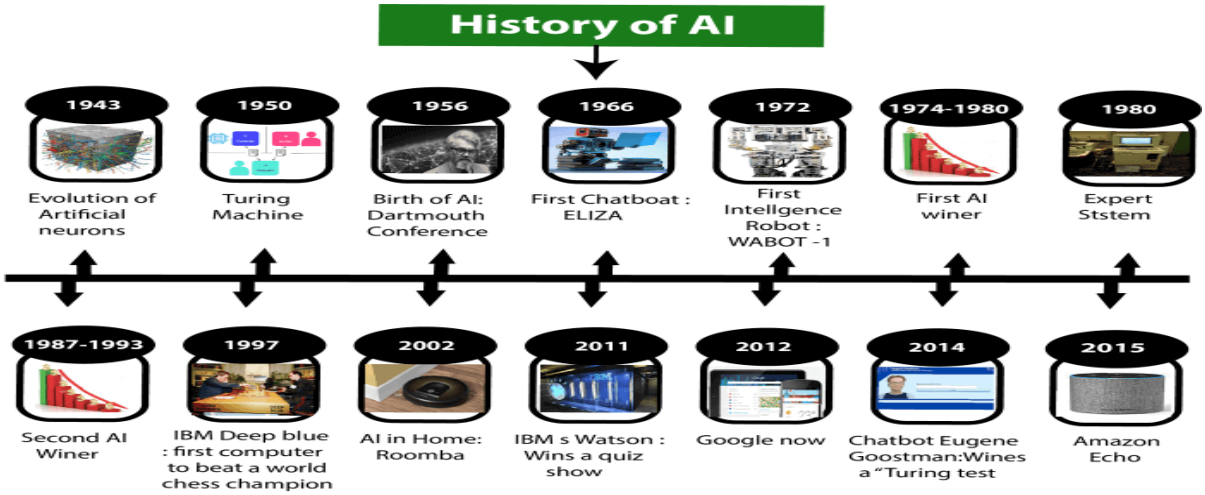
Disadvantages of Artificial Intelligence:

Every technology has some disadvantages, and the same goes for Artificial intelligence. Being so advantageous technology still, it has some disadvantages which we need to keep in our mind while creating an AI system. Following are the disadvantages of AI:

- High Cost: The hardware and software requirement of AI is very costly as it requires lots of maintenance to meet current world requirements.
- Can't think out of the box: Even we are making smarter machines with AI, but still they cannot work out of the box, as the robot will only do that work for which they are trained, or programmed.
- No feelings and emotions: AI machines can be an outstanding performer, but still it does not have the feeling so it cannot make any kind of emotional attachment with human, and may sometime be harmful for users if the proper care is not taken.
- Increase dependency on machines: With the increment of technology, people are getting more dependent on devices and hence they are losing their mental capabilities.
- No Original Creativity: As humans are so creative and can imagine some new ideas but still AI machines cannot beat this power of human intelligence and cannot be creative and imaginative

5.2. HISTORY OF AI:

Artificial Intelligence is not a new word and not a new technology for researchers. This technology is much older than you would imagine.



Maturation of Artificial Intelligence (1943-1952):

- **Year 1943:** The first work which is now recognized as AI was done by Warren McCulloch and Walter pits in 1943. They proposed a model of **artificial neurons**.
- **Year 1949:** Donald Hebb demonstrated an updating rule for modifying the onnection strength between neurons. His rule is now called **Hebbian learning**.
- **Year 1950:** The Alan Turing who was an English mathematician and pioneered Machine learning in 1950. Alan Turing publishes **"Computing Machinery and Intelligence"** in which he proposed a test. The test can check the machine's ability to exhibit intelligent behavior equivalent to human intelligence, called a **Turing test**.

The birth of Artificial Intelligence (1952-1956):

- **Year 1955:** An Allen Newell and Herbert A. Simon created the "first artificial intelligence program"Which was named as **"Logic Theorist"**. This program had proved 38 of 52 Mathematics theorems, and find new and more elegant proofs for some theorems.
- **Year 1956:** The word "Artificial Intelligence" first adopted by American Computer scientist John McCarthy at the Dartmouth Conference. For the first time, AI coined as an academic field.

At that time high-level computer languages such as FORTRAN, LISP, or COBOL were invented. And the enthusiasm for AI was very high at that time.

The golden years-Early enthusiasm (1956-1974):

- **Year 1966:** The researchers emphasized developing algorithms which can solve mathematical problems. Joseph Weizenbaum created the first chatbot in 1966, which was named as ELIZA.
- **Year 1972:** The first intelligent humanoid robot was built in Japan which was named as WABOT-1.

The first AI winter (1974-1980):

- The duration between years 1974 to 1980 was the first AI winter duration. AI winter refers to the time period where computer scientist dealt with a severe shortage of funding from government for AI researches.

- During AI winters, an interest of publicity on artificial intelligence was decreased.

A boom of AI (1980-1987):

- **Year 1980:** After AI winter duration, AI came back with "Expert System". Expert systems were programmed that emulate the decision-making ability of a human expert.
- In the Year 1980, the first national conference of the American Association of Artificial Intelligence **was held at Stanford University**.

The second AI winter (1987-1993):

- The duration between the years 1987 to 1993 was the second AI Winter duration.
- Again Investors and government stopped in funding for AI research as due to high cost but not efficient result. The expert system such as XCON was very cost effective.

The emergence of intelligent agents (1993-2011):

- **Year 1997:** In the year 1997, IBM Deep Blue beats world chess champion, Gary Kasparov, and became the first computer to beat a world chess champion.
- **Year 2002:** for the first time, AI entered the home in the form of Roomba, a vacuum cleaner.
- **Year 2006:** AI came in the Business world till the year 2006. Companies like Facebook, Twitter, and Netflix also started using AI.

Deep learning, big data and artificial general intelligence (2011-present):

- **Year 2011:** In the year 2011, IBM's Watson won jeopardy, a quiz show, where it had to solve the complex questions as well as riddles. Watson had proved that it could understand natural language and can solve tricky questions quickly.
- **Year 2012:** Google has launched an Android app feature "Google now", which was able to provide information to the user as a prediction.
- **Year 2014:** In the year 2014, Chatbot "Eugene Goostman" won a competition in the infamous "Turing test."
- **Year 2018:** The "Project Debater" from IBM debated on complex topics with two master debaters and also performed extremely well.
- Google has demonstrated an AI program "Duplex" which was a virtual assistant and which had taken hairdresser appointment on call, and lady on other side didn't notice

that she was talking with the machine.

5.3. APPLICATIONS OF AI:

1. AI in E-Commerce:

a) Personalized Shopping

Artificial Intelligence technology is used to create recommendation engines through which you can engage better with your customers. These recommendations are made in accordance with their browsing history, preference, and interests. It helps in improving your relationship with your customers and their loyalty towards your brand.

b) AI-powered Assistants

Virtual shopping assistants and chatbots help improve the user experience while shopping online. Natural Language Processing is used to make the conversation sound as human and personal as possible. Moreover, these assistants can have real-time engagement with your customers. Did you know that on amazon.com, soon, customer service could be handled by chatbots?

c) Fraud Prevention

Credit card frauds and fake reviews are two of the most significant issues that Ecommerce companies deal with. By considering the usage patterns, AI can help reduce the possibility of credit card frauds taking place. Many customers prefer to buy a product or service based on customer reviews. AI can help identify and handle fake reviews.

2. AI in Navigation:

Based on research from MIT, GPS technology can provide users with accurate, timely, and detailed information to improve safety. The technology uses a combination of Convolutional Neural Network and Graph Neural Network, which makes lives easier for users by automatically detecting the number of lanes and road types behind obstructions on the roads. AI is heavily used by Uber and many logistics companies to improve operational efficiency, analyze road traffic, and optimize routes.

3. AI in Robotics:

Robotics is another field where artificial intelligence applications are commonly used. Robots powered by AI use real-time updates to sense obstacles in its path and pre-plan its journey

instantly.

It can be used for -

- Carrying goods in hospitals, factories, and warehouses
- Cleaning offices and large equipment
- Inventory management

4. AI in Human Resource

Did you know that companies use intelligent software to ease the hiring process? Artificial Intelligence helps with blind hiring. Using machine learning software, you can examine applications based on specific parameters. AI drive systems can scan job candidates' profiles, and resumes to provide recruiters an understanding of the talent pool they must choose from.

5. AI in Healthcare:

Artificial Intelligence finds diverse applications in the healthcare sector. AI is used in healthcare to build sophisticated machines that can detect diseases and identify cancer cells. AI can help analyze chronic conditions with lab and other medical data to ensure early diagnosis. AI uses the combination of historical data and medical intelligence for the discovery of new drugs.

6. AI in Agriculture:

Artificial Intelligence is used to identify defects and nutrient deficiencies in the soil. This is done using computer vision, robotics, and machine learning, AI can analyze where weeds are growing. AI bots can help to harvest crops at a higher volume and faster pace than human laborers.

7. AI in Gaming:

Another sector where Artificial Intelligence applications have found prominence is the gaming sector. AI can be used to create smart, human-like NPCs to interact with the players.It can also be used to predict human behavior using which game design and testing can be improved. The Alien Isolation games released in 2014 uses AI to stalk the player throughout the game. The game uses two Artificial Intelligence systems - ‘Director AI’ that frequently knows your location and the ‘Alien AI,’ driven by sensors and behaviors that continuously hunt the player.

8. AI in Automobiles:

Artificial Intelligence is used to build self-driving vehicles. AI can be used along with the vehicle's camera, radar, cloud services, GPS, and control signals to operate the vehicle. AI can improve the in-vehicle experience and provide additional systems like emergency braking, blind-spot monitoring, and driver assist steering.

9. AI in Social Media:

Instagram

On Instagram, AI considers your likes and the accounts you follow to determine what posts you are shown on your explore tab.

Facebook

Artificial Intelligence is also used along with a tool called Deep Text. With this tool, Facebook can understand conversations better. It can be used to translate posts from different languages automatically.

Twitter

AI is used by Twitter for fraud detection, removing propaganda, and hateful content. Twitter also uses AI to recommend tweets that users might enjoy, based on what type of tweets they engage with.

10. AI in Marketing:

Artificial intelligence applications are popular in the marketing domain as well.

- Using AI, marketers can deliver highly targeted and personalized ads with the help of behavioral analysis, pattern recognition, etc. It also helps with retargeting audiences at the right time to ensure better results and reduced feelings of distrust and annoyance.
- AI can help with content marketing in a way that matches the brand's style and voice. It can be used to handle routine tasks like performance, campaign reports, and much more.
- Chatbots powered by AI, Natural Language Processing, Natural Language Generation, and Natural Language Understanding can analyze the user's language and respond in the ways humans do.
- AI can provide users with real-time personalizations based on their behavior and can

- be used to edit and optimize marketing campaigns to fit a local market's needs.

5.4. DEFINING ALGORITHM:

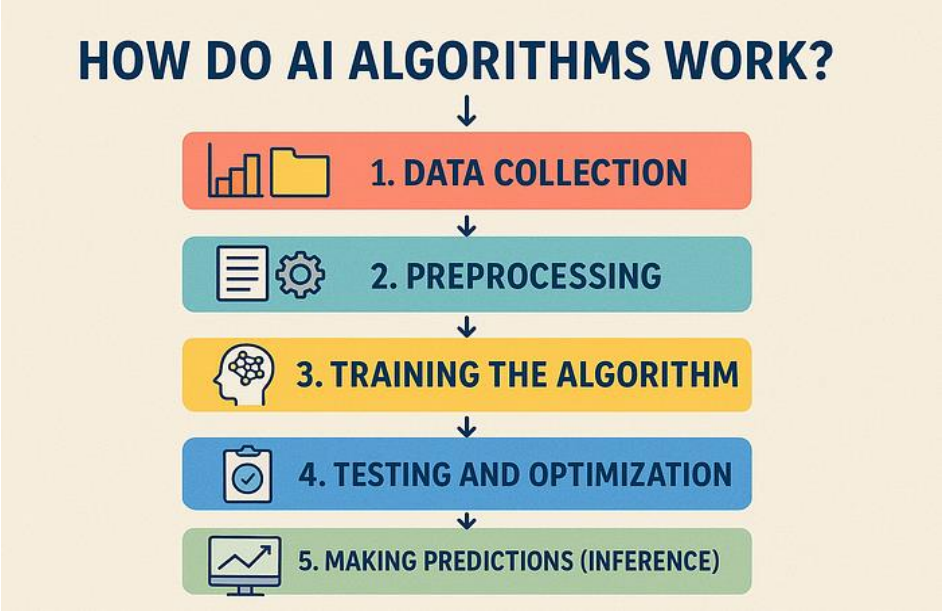
An AI algorithm is a set of instructions or rules a machine follows to perform a specific task. Think of it like a recipe — step-by-step instructions that take inputs (like data) and deliver outputs (like decisions or predictions).

What makes these algorithms special is that they learn and improve over time. Unlike traditional programming, where humans define every rule, AI algorithms uncover patterns and insights in data to "train" themselves and adapt.

For example:

- Input: Pictures of cats and dogs in a dataset.
- Process: Teach the algorithm to tell which image is a cat or a dog by labelling them.
- Output: Once trained, the AI can predict whether a new image it's given is a cat or a dog.

How Do AI Algorithms Work?



AI algorithms work by identifying patterns and applying learned knowledge to new data. This process is broken down into key steps: data collection, training, and inference.

1. Data Collection

AI algorithms need data—and lots of it! Their performance depends on having sufficient examples to identify patterns.

For instance:

- Netflix analyzes your watch history to recommend movies.
- E-commerce stores analyze past purchases to suggest products you might like.

2. Preprocessing

Raw data isn't always neat — it usually contains missing values, duplicates, or noise. Preprocessing cleans and organizes data so that the AI can work efficiently.

Techniques include:

- Normalization (scaling data to a standard format)
- Handling missing data (filling gaps or removing problematic examples)

3. Training the Algorithm

When training an AI algorithm, you teach it to understand patterns. You can do this by feeding it labeled data (for supervised learning; more on this shortly) or unlabeled data (for unsupervised learning).

4. Testing and Optimization

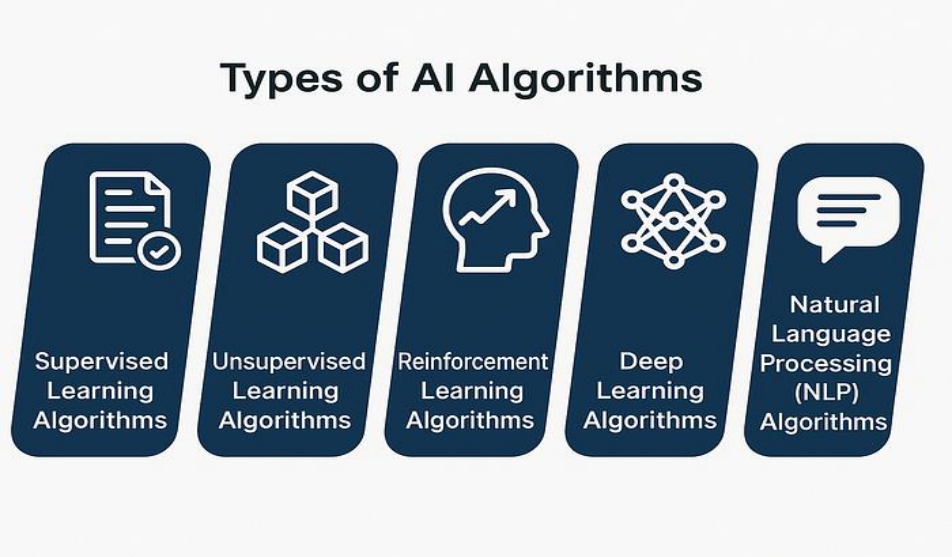
Once trained, the algorithm's performance is tested on new data. This ensures it understands patterns without simply memorizing examples. If necessary, developers tweak its parameters to improve accuracy or efficiency.

5. Making Predictions (Inference)

Once the algorithm is trained and tested, it can perform real-world tasks. For example:

- Self-driving cars use AI inference to make real-time decisions on the road.
- Sentiment analysis algorithms evaluate the tone of customer reviews to determine whether they're negative or positive.

Types of AI Algorithms (With Examples!)



AI algorithms come in various Flavors depending on their purpose and what type of data they work with. Here are the primary types:

1. Supervised Learning Algorithms

Supervised algorithms learn from labelled datasets. They're told what the outputs should look like, enabling them to map inputs to outputs.

Example: A spam email filter. By training on examples of spam vs. legitimate emails, the AI learns to classify emails in the future.

- Common Algorithms: Linear Regression, Support Vector Machines, and Neural Networks

2. Unsupervised Learning Algorithms

Unsupervised learning works with unlabeled data. Instead of pre-defined categories, the algorithm identifies patterns on its own.

Example: Clustering customers based on purchasing habits to send personalized recommendations.

- Common Algorithms: K-Means Clustering and Principle Component Analysis (PCA)

3. Reinforcement Learning Algorithms

With reinforcement learning, AI learns by trial and error — rewarded for desired behaviors and penalized for undesired ones.

Example: AI in gaming (e.g., AlphaGo) trains itself to beat human players using reward-based feedback.

- Popular Technique: Q-Learning

4. Deep Learning Algorithms

Deep Learning focuses on neural networks with multiple layers (hence "deep") to simulate how our brains process information. This AI algorithm is particularly powerful for tasks like speech and image recognition.

Example: Virtual assistants like Alexa recognize your voice using deep learning techniques.

5. Natural Language Processing (NLP) Algorithms

NLP algorithms specialize in understanding human language — written or spoken.

Example: ChatGPT uses NLP to generate responses that feel conversational and human-like.

- Tools: Transformers and Recurrent Neural Networks (RNNs)

5.5. A* ALGORITHM:

The A* (A-star) algorithm is a popular and widely used pathfinding and graph traversal algorithm in artificial intelligence.

What is A* Algorithm?

A* is an informed search algorithm, meaning it uses both the actual cost to reach a node and an estimated cost from that node to the goal to find the shortest path efficiently.

How does A* work?

- It maintains two main sets:
  - **Open Set:** Nodes that need to be evaluated.
  - **Closed Set:** Nodes that have already been evaluated.
- For each node, it calculates a **total cost function f(n)**: $f(n)=g(n)+h(n)$ where,
  - $g(n)$ is the actual cost from the start node to the current node n.
  - $h(n)$ is the heuristic estimated cost from node n to the goal.
- The algorithm selects the node from the open set with the smallest $f(n)$ value and explores its neighbors.
- It updates the costs if a cheaper path to a neighbor is found.

Key Components:

- **g(n):** Actual cost to reach node n.
- **h(n):** Heuristic cost estimate to reach the goal from node n.
- **f(n):** Estimated total cost of the cheapest solution through n.

Heuristic Function:

- The heuristic function $h(n)$ helps guide the search.
- It must be admissible (it never overestimates the true cost).
- Common heuristics include Euclidean distance, Manhattan distance, etc., depending on the problem context.

Steps of the A* Algorithm:

1. Initialize the open set with the start node.
2. Repeat:
    - Choose the node with the lowest $f(n)$ from the open set.
    - If this node is the goal, reconstruct and return the path.
    - Move this node from open to closed set.
    - For each neighbor:
        - Calculate g, h, and f.
        - If neighbor in closed set with higher cost, ignore it.
        - If neighbor in open set with higher cost, update it.
        - Otherwise, add neighbor to open set.

Applications:

- Robotics path planning
- Game AI
- Network routing
- Puzzle solving

This algorithm efficiently finds the shortest path especially when a good heuristic is available.

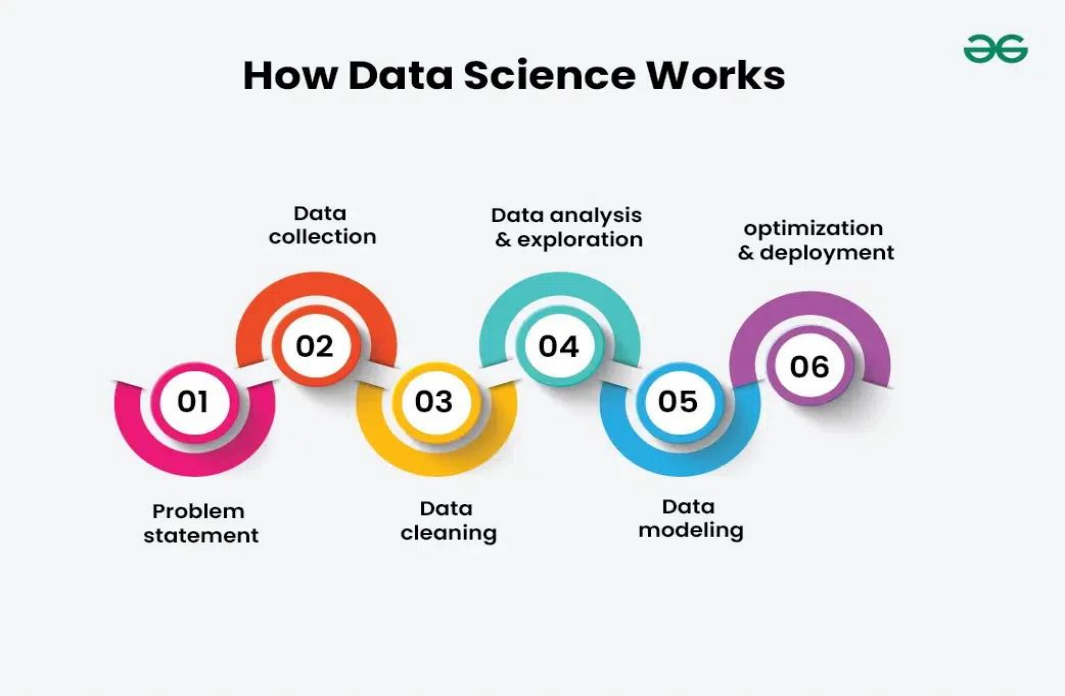
5.6. DATA SCIENCE: INTRODUCTION

Data science is an interdisciplinary field that uses scientific methods, processes,

algorithms, and systems to extract knowledge and insights from structured and unstructured data. In simpler terms, data science is about obtaining, processing, and analyzing data to gain insights for many purposes. In short, data science empowers the industries to make smarter, faster, and more informed decisions. In order to find patterns and achieve such insights, expertise in relevant domain is required. With expertise in Healthcare, a data scientists can predict patient risks and suggest personalized treatments.

Data Science Life Cycle:

Data science is not a one-step process such that you will get to learn it in a short time and call ourselves a Data Scientist. It's passes from many stages and every element is important. One should always follow the proper steps to reach the ladder. Every step has its value and it counts in your model.



1. Problem Statement:

No work starts without motivation; Data science is no exception though. It's really important to declare or formulate your problem statement very clearly and precisely. Your whole model and it's working depend on your statement. Many scientists consider this as the main and much important step of Date Science. So, make sure what's your problem statement and how well can it add value to business or any other organization.

2. Data Collection:

After defining the problem statement, the next obvious step is to go in search of data that you might require for your model. You must do good research, find all that you need. Data can be in any form i.e unstructured or structured. It might be in various forms like videos, spreadsheets, coded forms, etc. You must collect all these kinds of sources.

3. Data Cleaning:

As you have formulated your motive and also you did collect your data, the next step to do is cleaning. Yes, it is! Data cleaning is the most favorite thing for data scientists to do. Data cleaning is all about the removal of missing, redundant, unnecessary and duplicate data from your collection. There are various tools to do so with the help of programming in either R or Python. It's totally on you to choose one of them. Various scientist has their opinion on which to choose. When it comes to the statistical part, R is preferred over Python, as it has the privilege of more than 12,000 packages. While python is used as it is fast, easily accessible and we can perform the same things as we can in R with the help of various packages.

4. Data Analysis and Exploration:

It's one of the prime things in data science to do and time to get inner Holmes out. It's about analyzing the structure of data, finding hidden patterns in them, studying behaviors, visualizing the effects of one variable over others and then concluding. We can explore the data with the help of various graphs formed with the help of libraries using any programming language. In R, GGplot is one of the most famous models while Matplotlib in Python.

5. Data Modelling:

Once you are done with your study that you have formed from data visualization, you must start building a hypothesis model such that it may yield you a good prediction in future. Here, you must choose a good algorithm that best fit to your model. There different kinds of algorithms from regression to classification, SVM (Support vector machines), Clustering, etc. Your model can be of a Machine Learning algorithm. You train your model with the train data and then test it with test data. There are various methods to do so. One of them is the K-fold method where you split your whole data into two parts, one is Train and the other is test data. On these bases, you train your model.

6. Optimization and Deployment:

You followed each and every step and hence build a model that you feel is the best fit. But how can you decide how well your model is performing? This where optimization comes. You test your data and find how well it is performing by checking its accuracy. In short, you check the efficiency of the data model and thus try to optimize it for better accurate prediction. Deployment deals with the launch of your model and let the people outside there to benefit from that. You can also obtain feedback from organizations and people to know their need and then to work more on your model.

5.7. DEFINING DATA, INFORMATION AND DATA STRUCTURE:

1. Data:

Data is the raw form of information, a collection of facts, figures, symbols or observations that represent details about events, objects or phenomena. By itself, data may appear meaningless, but when organized, processed and interpreted, it transforms into valuable insights that support decision-making, problem-solving and innovation.

Importance of Data:

- Decision-making and insights: Organizations use data to make better decisions. Raw data becomes useful when transformed into insights with the help of analytics.
- AI/ML and Innovation: Data is the fuel for artificial intelligence and machine learning. More and higher-quality data means better training, more accurate predictions.
- Digital transformation: The rise of Big Data has enabled new capabilities i.e from real-time analytics to personalized services.

Types of Data:

Data can be categorized in different ways depending on how it is collected, stored and represented. Broadly, it falls into the following types:

1. Quantitative Data:

Quantitative data is information that can be measured, counted and expressed in numerical form. It provides objective values that can be analyzed statistically to identify patterns, trends and relationships.

- Represents numbers and measurable values.
- Can be divided into: Discrete data (Whole numbers) and Continuous data (Values on a scale).
- Widely used in research, finance, engineering and business analytics.

Example: Age of people, number of customers visiting a store, temperature readings, sales revenue.

2. Qualitative Data:

Qualitative data is descriptive, non-numeric information that explains qualities, characteristics or categories rather than quantities. It helps understand opinions, experiences and meanings behind behaviors.

- Focuses on qualities, attributes and categories rather than numbers.
- Often collected through surveys, interviews or observations.
- Useful for understanding opinions, motivations and behaviors.

Example: Customer feedback ("satisfied", "unsatisfied"), product colors, interview transcripts, social media comments.

3. Structured Data:

Structured data is information organized into a predefined format (rows and columns) that makes it easily searchable and manageable by traditional databases.

- Stored in relational databases or spreadsheets.
- Easy to process with SQL and other tools.
- Best suited for tasks requiring accuracy and consistency.

Example: Bank transactions, employee records, product inventories.

4. Unstructured Data:

Unstructured data is raw information that does not follow a predefined structure or format making it harder to organize and analyze with conventional tools.

- Accounts for over 80% of data generated globally.
- Requires advanced tools (AI, NLP, computer vision) to extract insights.
- Common in social media, multimedia and IoT applications.

Example: Emails, images, videos, voice recordings, PDF documents.

5. Semi-Structured Data:

Semi-structured data combines aspects of structured and unstructured data. It does not reside in traditional tables but still contains tags or markers that provide a loose structure.

- Provides a balance between flexibility and structure.
- Easier to analyze than unstructured data, but less rigid than structured data.
- Often used in web applications, IoT devices and log systems.

Example: JSON files, XML documents, NoSQL databases, sensor logs.

Big Data:

When datasets grow in size, complexity and speed, traditional methods don't suffice. Big Data refers to datasets that are too large, too varied or too fast to be handled by traditional data processing tools.

2.Information:

The raw data is collected; after processing this raw data the outcome is information. This information can be defined as when the data is processed, organized, and presented in a specific context to serve its use is called information. The information doesn't have any existence without data, mostly information have measuring unit like quantity, time, etc. There are also a lot of differences between data and information. For information to be useful, the process data has the following characteristics which are:

- **Time -** Information should be available at any point in time whenever it is required.
- **Accuracy -** Information should be actual and organized only then it can serve its purpose.
- **Completeness -** Information should be finite and consistent.

Some examples of information:

1. Information about transportation systems such as train schedules.
2. Geographical information such as direction.
3. Payslips
4. Bank passbook
5. Printed documents.

Role of Information in Today's Generation:

- The information helps in generating new information which can be new theories, a new idea's or new discovery.
- The information helps in the duplication of research because it gives an idea of what already has been discovered.
- Information technology helps to build and emerge the growth of commerce and business sector and generate maximum possible output.
- Information technology has played a great role in the creation of employment.
- Data analysts, systems architect, hardware engineer, and software developers, and web designers all beholden their jobs to information technology. Without such advancement, these jobs would not be entertained.

3. Data Structure:

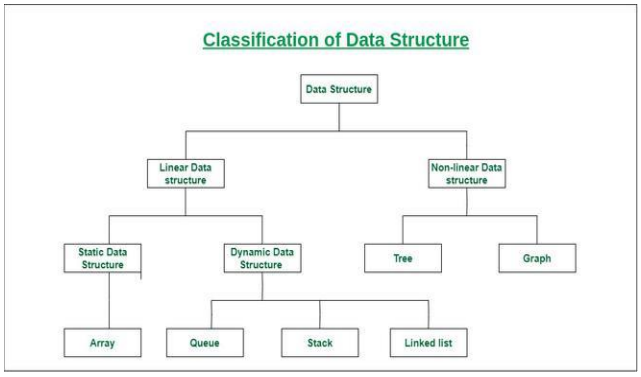
A data structure is a way of organizing and storing data in a computer so that it can be accessed and used efficiently. It refers to the logical or mathematical representation of data, as well as the implementation in a computer program.

Classification:

Data structures can be classified into two broad categories:

- **Linear Data Structure:** A data structure in which data elements are arranged sequentially or linearly, where each element is attached to its previous and next adjacent elements, is called a linear data structure. Examples are array, stack, queue, etc.
- **Non-linear Data Structure:** Data structures where data elements are not placed sequentially or linearly are called non-linear data structures. Examples are trees and graphs.

  Classification of Data Structure:



Applications of Data Structures:

Data structures are used in a wide range of computer programs and applications, including:

- **Databases:** Data structures are used to organize and store data in a database, allowing for efficient retrieval and manipulation.
- **Operating systems:** Data structures are used in the design and implementation of operating systems to manage system resources, such as memory and files.
- **Computer graphics:** Data structures are used to represent geometric shapes and other graphical elements in computer graphics applications.
- **Artificial intelligence:** Data structures are used to represent knowledge and information in artificial intelligence systems.

Advantages of Data Structures:

The use of data structures provides several advantages, including:

- **Efficiency:** Data structures allow for efficient storage and retrieval of data, which is important in applications where performance is critical.
- **Flexibility:** Data structures provide a flexible way to organize and store data, allowing for easy modification and manipulation.
- **Reusability:** Data structures can be used in multiple programs and applications, reducing the need for redundant code.
- **Maintainability:** Well-designed data structures can make programs easier to understand, modify, and maintain over time.

5.8. BASIC CONCEPT OF PROBABILITY AND STATISTICS:

Basic Concept of Probability:

Probability is defined as the likelihood of the occurrence of any event. It gives a numerical value to the chance or likelihood of something happening. Probability is generally denoted by **P(E)**, where **E** represents the event.

It is expressed as a number between 0 and 1:

- 0 means the event is impossible,
- 1 means the event is certain,
- Values between 0 and 1 represent partial chances

Concepts of Probability are used in various real-life scenarios:

- **Stock Market:** Investors and analysts use probability models to understand trends and patterns in the movement of stock prices.
- **Insurance:** Insurance companies use probability models to estimate the likelihood of various events and to manage risks, which helps in setting premiums.
- **Weather Forecasting:** Meteorologists use probability to predict the likelihood of weather events, such as rain, snow, storms, or temperature changes.

Formula for Probability:

The probability formula is defined as the ratio of the number of favorable outcomes to the total number of outcomes.

Probability of Event P(E) = [Number of Favorable Outcomes] / [Total Number of Outcomes]

The probability of an event E, denoted by P(E), is a number between 0 and 1 that represents the likelihood of E occurring.

- If P(E) = 0, the event E is impossible.
- If P(E) = 1, the event E is certain to occur.
- If 0 < P(E) < 1, the event E is possible but not guaranteed.

Note: *The sum of the probabilities of all events in a sample space is always equal to 1.*

For example, when we toss a coin, there are only two possible outcomes: Heads (H) or Tails (T). However, if we toss two coins simultaneously, there will be four possible outcomes: (H, H), (H, T), (T, H), and (T, T).

Sample Space and Event:

- **Sample Space and Events Sample Space**: The sample space, often denoted by S, is the set of all possible outcomes of an experiment. **For example**, when rolling a six-sided die, the sample space is S = {1, 2, 3, 4, 5, 6}.
- **Event**: An event is any subset of the sample space. It represents a specific outcome or a combination of outcomes. There are many different types of events in Probability such as Impossible and Sure Events, Mutually Exclusive Events, Exhaustive Events, Dependent and Independent Events. For example, rolling an even number E = {2, 4, 6}) is an event in the context of rolling a die.

Basic Probability Rules:

- **Addition Rule**: **P(A∪B) = P(A) + P(B) - P(A∩B)**, where A∪B denotes the union of events A and B.
- **Multiplication Rule for Independent Events: P(A∩B) = P(A) × P(B)**, where A and B are independent events.
- **Complement Rule**: **P (A ′) = 1 - P(A)**, where ′ A ′ denotes the complement of event A.

Probability Distribution:

A probability distribution describes how the values of a random variable are spread or distributed. It tells us the probability of each possible outcome in a sample space and can be either discrete (for countable outcomes) or continuous (for measurable outcomes like height or time).

Types of Probability Distributions:

- Bernoulli Distribution
- Binomial Distribution
- Poisson Distribution
- Uniform Distribution
- Normal Distribution (Gaussian)
- Exponential Distribution

Applications of Probability:

Some of the common events for which we can use applications of probability to check the results are:

- Choosing a card from the deck of cards
- Flipping a coin
- Throwing a dice in the air
- Pulling a red ball out of a bucket of red and white balls
- Winning a lucky draw

Basic Concept of Statistics:

Statistics is the science of collecting, analyzing, and interpreting data to uncover patterns and make decisions. In data science, it acts as the backbone for understanding data and building

reliable models.

- Summarizes data using measures like mean, median, and variance
- Models uncertainty with probability and distributions
- Tests hypotheses (e.g., A/B testing)
- Finds relationships through regression and correlation

Types of Statistics:

There are commonly two types of statistics, which are discussed below:

1. **Descriptive Statistics:** Descriptive Statistics helps us simplify and organize big chunks of data. This makes large amounts of data easier to understand.
2. **Inferential Statistics:** Inferential Statistics is a little different. It uses smaller data to conclude a larger group. It helps us predict and draw conclusions about a population.
3. Basic formulas of statistics are,

Parameters	Definition	Formulas
Population Mean (μ)	Average of the entire group.	$\frac{\Sigma x}{N}$
Sample Mean	Average of a subset of the population	$\frac{\Sigma x}{N}$
Sample/Population Standard Deviation	Measures how spread out the data is from the mean	Population $\sigma=\sqrt{\frac{1}{N}\sum_{i=1}^{n}(xi-\mu)^2}$ Sample $s=\sqrt{\frac{1}{N-1}\sum_{i=1}^{n}(xi-\bar{x})^2}$
Sample/Population Variance	Shows how far values are from the mean, squared	Variance (Population) $=\frac{(\Sigma(xi-\bar{x})^2}{n}$ Variance (Sample) $=\frac{(\Sigma(xi-\bar{x})^2}{n-1}$

Class Interval (CI)	Range of values in a group	CI = Upper Limit − Lower Limit
Frequency(f)	How often a value appears	Count of occurrences
Range (R)	Difference between largest and smallest values	Range = Max−Min

Role in Data Science:

Probability and statistics are crucial throughout the data science process. They help data scientists:

- Understand and prepare data using statistical methods.
- Build models based on statistical principles.
- Assess uncertainty and risk in models.
- Make data-driven decisions using techniques like A/B testing.