MAA OMWATI DEGREE COLLEGE(HASSANPUR)

PYTHON PROGRAMMING

Program : BCA 3RD SEM

IMPORTANT NOTES FOR EXAM

UNIT-1: Introduction to Python
Topics Covered

1. History of Python
2. Application Areas of Python
3. Structure of Python Program
4. Identifiers and Keywords
5. Operators and Precedence
6. Basic Data Types and Type Conversion
7. Statements and Expressions
8. Input and Output Statements

---

1. History of Python
Definition
Python is a **high-level, interpreted, object-oriented, and general-purpose programming language**. It is easy to learn because of its simple syntax and readability.

- **Developer:** Guido van Rossum
- **First Released:** 1991
- **Latest Versions:** Python 3.x (continuously updated)

Python was designed to make programming **simple, readable, and efficient**.

History Timeline

Year	Event
1989	Guido van Rossum started developing Python
1991	Python 0.9.0 released
2000	Python 2.0 released
2008	Python 3.0 released
Present	Python 3.x widely used

Why the Name "Python"?
The name **Python** was inspired by the British comedy TV show:
"Monty Python's Flying Circus"
It is **not named after the snake**.

Features of Python

- Easy to learn
- Simple syntax
- Open-source
- Portable
- Interpreted
- Object-oriented
- Dynamically typed
- Large standard library
- Cross-platform
- Supports GUI programming

Advantages

- Easy to read
- Less coding
- Free and open source
- Huge community support
- Platform independent

Disadvantages

- Slower than C/C++
- High memory usage
- Not ideal for mobile app development
- Limited use in low-level programming

Execute Python Syntax
As we learned in the previous page, Python syntax can be executed by writing directly in the Command Line:

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

---

2. Application Areas of Python
Python is used in many fields.

(A) Web Development
Used to build websites and web applications.
Popular Frameworks:

- Django
- Flask
- FastAPI

Example:
Instagram uses Python.

(B) Data Science
Used for analyzing and visualizing data.

Libraries:

- NumPy
- Pandas
- Matplotlib
- Seaborn

(C) Artificial Intelligence (AI)
Python is the most popular language for AI.
Libraries:

- TensorFlow
- PyTorch
- Keras

(D) Machine Learning
Used for prediction and classification.
Libraries:

- Scikit-learn
- TensorFlow

(E) Game Development
Library:

- Pygame

(F) Automation
Automates repetitive tasks like:

- File handling
- Email sending
- Web scraping

(G) Desktop Applications
Frameworks:

- Tkinter
- PyQt

(H) Cyber Security
Used for:

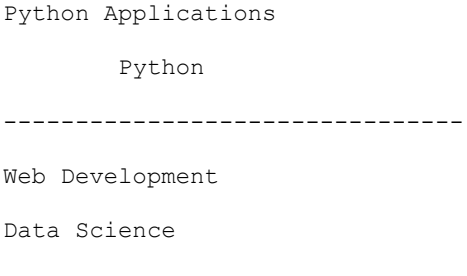
- Ethical hacking
- Network scanning
- Security testing

(I) Internet of Things (IoT)
Used with Raspberry Pi and sensors.

(J) Cloud Computing
Supported by:

- AWS
- Google Cloud
- Microsoft Azure

Diagram



```
Machine Learning

Artificial Intelligence

Automation

Game Development

Desktop Apps

Cyber Security

Cloud Computing

IoT
```

---

3. Structure of Python Program
Basic Structure

```
# Program to print Hello World

print("Hello World")
```

Parts of Python Program
1. Comments
Used to explain code.
Single-line comment

```
# This is a comment
```

Multi-line comment

```
"""
This is
a multi-line comment
"""
```

---

2. Import Statement
Imports modules.

```
import math
```

---

3. Variable Declaration

```
name = "Rahul"
age = 20
```

---

4. Statements
Instructions executed by Python.

```
print(name)
```

Program Structure Diagram

```
Comments

↓

Import Modules

↓

Variables

↓

Statements

↓
```

```
Output
```

Example Program

```
# First Program

name = "Rahul"

age = 20

print(name)

print(age)
```

Output

```
Rahul

20
```

---

4. Identifiers and Keywords

Identifiers

Definition

Identifiers are **names given to variables, functions, classes, or objects**.

Rules for Identifiers

- Must begin with a letter or underscore (_)
- Cannot begin with a digit
- No special symbols except _
- Case-sensitive
- Cannot use keywords

Valid Identifiers

```
student

student_name

_marks

total1
```

Invalid Identifiers

```
1name

student-name

class

total$
```

Keywords

Definition

Keywords are **reserved words** with predefined meanings.
They cannot be used as identifiers.

Common Python Keywords

Keyword	Meaning
if	Conditional statement
else	Alternative condition
for	Loop
while	Loop
break	Exit loop
continue	Skip iteration

Keyword	Meaning
return	Return value
class	Define class
def	Define function
import	Import module
True	Boolean true
False	Boolean false
None	No value

Difference

Identifier	Keyword
User-defined name	Reserved word
Can be created by programmer	Cannot be used as variable names
Example: age	Example: if

---

5. Operators and Operator Precedence
Definition
Operators are symbols used to perform operations on variables and values.

Types of Operators

(A) Arithmetic Operators

Operator	Meaning	Example
+	Addition	5+3
-	Subtraction	5-3
*	Multiplication	5*3
/	Division	10/2
%	Modulus	10%3
//	Floor Division	10//3
**	Exponent	2**3

Example

```
a = 10

b = 3

print(a+b)

print(a//b)

print(a%b)
```

(B) Relational Operators

Operator	Meaning
==	Equal
!=	Not Equal
>	Greater
<	Less
>=	Greater or Equal
<=	Less or Equal

(C) Logical Operators

Operator	Meaning
and	Logical AND

Operator	Meaning
or	Logical OR
not	Logical NOT

(D) Assignment Operators

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

(E) Bitwise Operators

- &
- |
- ^
- ~
- <<
-

(F) Membership Operators

- in
- not in

Example

```
"A" in "Apple"
```

(G) Identity Operators

- is
- is not

Operator Precedence

Higher precedence operators execute first.

Priority	Operators
Highest	()
	**
	*, /, //, %
	+, -
	<, >, <=, >=
	==, !=
	not
	and
Lowest	or

Example

```
2 + 3 * 4
```

Output

```
14
```

---

6. Basic Data Types and Type Conversion

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

Text Type:	`str`
Numeric Types:	`int`, `float`, `complex`
Sequence Types:	`list`, `tuple`, `range`

Mapping Type:	`dict`
Set Types:	`set`, `frozenset`
Boolean Type:	`bool`
Binary Types:	`bytes`, `bytearray`, `memoryview`
None Type:	`NoneType`

Data Types
Integer (int)
Whole numbers
```
age = 20
```

Float
Decimal numbers
```
price = 99.5
```

String
Text enclosed in quotes
```
name = "Rahul"
```

Boolean
True or False
```
isPassed = True
```

Complex
```
z = 3 + 4j
```

Data Type Table

Type	Example
int	10
float	10.5
str	"Hello"
bool	True
complex	2+3j

Type Conversion
Definition
Converting one data type into another.

Implicit Conversion
Done automatically.
```
a = 5

b = 2.5

print(a+b)
```
Output
```
7.5
```

Explicit Conversion
Done by programmer.
```
x = "10"

y = int(x)

print(y)
```

Conversion Functions

Function	Converts To

Function	Converts To
int()	Integer
float()	Float
str()	String
bool()	Boolean

---

7. Statements and Expressions

Statement

A statement is an instruction executed by Python.

Example

```
x = 10
```

Expression

An expression is a combination of variables, constants, and operators that produces a value.

Example

```
x + y
```

Difference

Statement	Expression
Performs an action	Produces a value
Example: x=5	Example: x+5

---

8. Input and Output Statements

Input

Used to accept data from the user.

Syntax

```
input()
```

Example

```
name = input("Enter Name : ")

print(name)
```

Integer Input

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

Float Input

```
salary = float(input("Enter Salary : "))
```

Output Statement

Uses the `print()` function.

Example

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

Printing Multiple Values

```
name = "Rahul"

age = 20

print(name, age)
```

Output

```
Rahul 20
```

Formatted Output (f-string)

```
name = "Rahul"

marks = 95

print(f"{name} scored {marks} marks.")
```

Output

```
Rahul scored 95 marks.
```

Advantages of Python

- Simple syntax
- Easy to learn
- Open source
- Portable
- Large library support
- Cross-platform
- Good community support

Disadvantages

- Slower execution than compiled languages
- Higher memory usage
- Not suitable for system-level programming
- Limited support for some mobile platforms

One-Page Revision (Exam Ready)

Topic	Key Point
Python	High-level, interpreted, object-oriented language developed by Guido van Rossum (1991)
Applications	Web development, AI, ML, Data Science, Automation, IoT, Cyber Security
Program Structure	Comments → Imports → Variables → Statements → Output
Identifiers	User-defined names; cannot be keywords
Keywords	Reserved words like `if`, `for`, `while`, `def`, `class`, `True`, `False`
Arithmetic Operators	`+`, `-`, `*`, `/`, `%`, `//`, `**`
Relational Operators	`==`, `!=`, `>`, `<`, `>=`, `<=`
Logical Operators	`and`, `or`, `not`
Assignment Operators	`=`, `+=`, `-=`, `*=`, `/=`
Data Types	`int`, `float`, `str`, `bool`, `complex`
Type Conversion	Implicit (automatic), Explicit (`int()`, `float()`, `str()`, `bool()`)
Statement	Executes an action (e.g., `x = 5`)
Expression	Produces a value (e.g., `x + 5`)
Input	`input()`
Output	`print()` and formatted output using `f`-strings

Strings in Python

Topics Covered

1. Introduction to Strings
2. Creating and Storing Strings
3. Built-in Functions for Strings
4. String Operators
5. String Slicing
6. String Joining
7. Formatting Strings

---

1. Introduction to Strings

Definition

A **String** is a sequence of characters enclosed within **single quotes (' ')**, **double quotes (" ")**, or **triple quotes (''' ''' or """ """)**.

A string can contain:

- Letters
- Numbers
- Symbols
- Spaces

- Special characters

In Python, strings are **immutable**, which means **their contents cannot be changed after creation**.

Examples

'hello' is the same as "hello".

```
name = "Rahul"
city = 'Delhi'
course = "Python Programming"
```

Characteristics of Strings

- Ordered collection of characters
- Immutable
- Supports indexing
- Supports slicing
- Can contain Unicode characters
- Can be concatenated and repeated

Memory Representation

```
String = "Python"

Character : P  y  t  h  o  n

Index     : 0  1  2  3  4  5

Negative  : -6 -5 -4 -3 -2 -1
```

---

2. Creating and Storing Strings

Python provides several ways to create strings.

(A) Using Single Quotes

```
name = 'Rahul'
```

Output

```
Rahul
```

(B) Using Double Quotes

```
city = "Delhi"
```

Output

```
Delhi
```

(C) Using Triple Quotes

Used for **multi-line strings**.

```
text = """Python
is
easy"""
```

Output

```
Python
is
easy
```

(D) Empty String

```
s = ""
```

Storing Strings in Variables

```
course = "Python"
```

You can display a string literal with the `print()` function:

```
print(course)
```

Output

```
Python
```

String Length

Use the `len()` function.

```
name = "Python"

print(len(name))
```
Output
```
6
```

---

3. Built-in Functions for Strings
Python provides many built-in functions to work with strings.

---

1. len()
Returns the number of characters.
```
text = "Python"

print(len(text))
```
Output
```
6
```

---

2. max()
Returns the character with the highest Unicode value.
```
print(max("Python"))
```
Output
```
y
```

---

3. min()
Returns the character with the smallest Unicode value.
```
print(min("Python"))
```
Output
```
P
```

---

4. sorted()
Returns a sorted list of characters.
```
print(sorted("python"))
```
Output
```
['h', 'n', 'o', 'p', 't', 'y']
```

---

5. type()
Returns the data type.
```
name = "Rahul"

print(type(name))
```
Output
```
<class 'str'>
```

---

6. str()
Converts data into a string.
```
age = 20

print(str(age))
```
Output
```
"20"
```

Common String Methods
upper()
Converts to uppercase.
```
text = "python"

print(text.upper())
```
Output
```
PYTHON
```

---

lower()
Converts to lowercase.
```
print("PYTHON".lower())
```

Output
```
python
```

---

capitalize()
Capitalizes the first letter.
```
print("python".capitalize())
```
Output
```
Python
```

---

title()
Capitalizes the first letter of each word.
```
print("python programming".title())
```
Output
```
Python Programming
```

---

swapcase()
Changes uppercase to lowercase and vice versa.
```
print("Python".swapcase())
```
Output
```
pYTHON
```

---

strip()
Removes spaces from both ends.
```
text = " Python "

print(text.strip())
```
Output
```
Python
```

---

replace()
Replaces a substring.
```
text = "Python"

print(text.replace("Python", "Java"))
```
Output
```
Java
```

---

find()
Finds the first occurrence of a substring.
```
text = "Python"

print(text.find("t"))
```
Output
```
2
```

---

count()
Counts occurrences of a substring.
```
text = "banana"

print(text.count("a"))
```
Output
```
3
```

---

startswith()
Checks if a string starts with a given prefix.
```
print("Python".startswith("Py"))
```
Output
```
True
```

---

endswith()
Checks if a string ends with a given suffix.
```
print("Python".endswith("on"))
```
Output

```
True
```

Summary of Important String Functions

Function	Purpose
len()	Length of string
upper()	Uppercase
lower()	Lowercase
capitalize()	First letter uppercase
title()	Capitalize every word
strip()	Remove spaces
replace()	Replace text
find()	Find substring
count()	Count occurrences
startswith()	Check beginning
endswith()	Check ending

---

4. String Operators
Operators perform operations on strings.

(A) Concatenation Operator (+)
Joins two strings.

```
first = "Python"

second = "Programming"

print(first + " " + second)
```

Output

```
Python Programming
```

(B) Repetition Operator (*)
Repeats a string.

```
print("Hi " * 3)
```

Output

```
Hi Hi Hi
```

(C) Membership Operator
Checks whether a substring exists.
in

```
print("P" in "Python")
```

Output

```
True
```

---

not in

```
print("Java" not in "Python")
```

Output

```
True
```

(D) Comparison Operators

```
print("Apple" == "Apple")
```

Output

```
True
```

Example

```
print("A" < "B")
```

Output

```
True
```

---

5. String Slicing
Definition
Slicing extracts a part of a string.

Syntax
```
string[start:stop:step]
```

Example
```
text = "Python"
```
Indexes
```
P  y  t  h  o  n

0  1  2  3  4  5
```

Examples
First three letters
```
print(text[0:3])
```
Output
```
Pyt
```

Last three letters
```
print(text[3:6])
```
Output
```
hon
```

Entire string
```
print(text[:])
```
Output
```
Python
```

Reverse string
```
print(text[::-1])
```
Output
```
nohtyP
```

Every second character
```
print(text[::2])
```
Output
```
Pto
```

Slicing Examples Table

Slice	Output
text[0:2]	Py
text[2:]	thon
text[:4]	Pyth
text[::-1]	nohtyP

---

6. String Joining
Definition
Joining combines multiple strings into a single string using the `join()` method.
Syntax
```
separator.join(iterable)
```

Example
```
words = ["Python", "Java", "C++"]

print(" ".join(words))
```
Output
```
Python Java C++
```

Using Comma
```
print(",".join(words))
```
Output
```
Python,Java,C++
```

Using Hyphen

```
print("-".join(words))
```

Output

```
Python-Java-C++
```

Join Characters

```
print("-".join("Python"))
```

Output

```
P-y-t-h-o-n
```

Advantages of join()

- Fast
- Efficient
- Cleaner than repeated concatenation
- Preferred for combining many strings

---

7. Formatting Strings

Definition

String formatting is used to insert variables or expressions into strings.

Method 1: Using %

```
name = "Rahul"

marks = 90

print("%s scored %d marks" % (name, marks))
```

Output

```
Rahul scored 90 marks
```

Method 2: Using format()

```
name = "Rahul"

marks = 90

print("{} scored {} marks".format(name, marks))
```

Output

```
Rahul scored 90 marks
```

Method 3: Using f-string (Recommended)

Available from Python 3.6 onwards.

```
name = "Rahul"

marks = 90

print(f"{name} scored {marks} marks.")
```

Output

```
Rahul scored 90 marks.
```

Formatting Numbers

```
pi = 3.14159

print(f"{pi:.2f}")
```

Output

```
3.14
```

Formatting Alignment

Left Align

```
print(f"{'Python':<10}")
```

Right Align

```
print(f"{'Python':>10}")
```

Center Align

```
print(f"{'Python':^10}")
```

Comparison of Formatting Methods

Method	Example	Recommended
% Operator	`%s`	Older method
format()	`"{}".format()`	Good
f-string	`f"{name}"`	Best and fastest

Advantages of Strings

- Easy to store text
- Immutable and safe
- Rich built-in methods
- Unicode support
- Efficient slicing and formatting

Disadvantages

- Cannot modify characters directly (immutable)
- Repeated concatenation of many strings is inefficient (use `join()` instead)

One-Page Revision (Exam Ready)

Topic	Key Point
String	Sequence of characters enclosed in quotes
String Properties	Ordered, immutable, indexable, sliceable
Creating Strings	Single quotes, double quotes, triple quotes
Built-in Functions	`len()`, `max()`, `min()`, `sorted()`, `type()`, `str()`
Important Methods	`upper()`, `lower()`, `capitalize()`, `title()`, `strip()`, `replace()`, `find()`, `count()`, `startswith()`, `endswith()`
String Operators	`+` (concatenation), `*` (repetition), `in`, `not in`, comparison operators
String Slicing	`string[start:stop:step]`
Reverse String	`string[::-1]`
String Joining	`"separator".join(iterable)`
Formatting Methods	`%`, `format()`, `f-string` (recommended)

- **Length:** `len()`
- **Case conversion:** `upper()`, `lower()`, `capitalize()`, `title()`
- **Searching:** `find()`, `count()`
- **Replacing:** `replace()`
- **Joining:** `join()`
- **Slicing:** `string[start:stop:step]`
- **Formatting:** `f"{variable}"` (most preferred in modern Python)

UNIT II

Unit-2: Control Flow Statements

Topics Covered

1. Introduction to Control Flow
2. Conditional Flow Statements
3. Loop Control Statements
4. Nested Control Flow

5. `continue` Statement
6. `break` Statement
7. `pass` Statement
8. `exit()` Function

---

1. Introduction to Control Flow
Definition
Control Flow refers to the **order in which statements are executed** in a program.
Normally, Python executes statements **from top to bottom (sequentially)**. However, using control flow statements, we can:

- Make decisions
- Repeat tasks
- Skip statements
- Stop program execution

Types of Control Flow Statements

```
                Control Flow
                     |
      ---------------------------------
      |              |              |
 Conditional      Looping       Jump Statements
 Statements      Statements     (break, continue,
                                pass, exit())
```

---

2. Conditional Flow Statements
Definition
Conditional statements allow a program to **make decisions** based on a condition.
If the condition is **True**, one block of code executes; otherwise, another block may execute.

Types of Conditional Statements

1. if Statement
2. if-else Statement
3. if-elif-else Statement
4. Nested if Statement

(A) if Statement
Definition
Executes a block of code only if the condition is **True**. If statement is used to execute a block of code only when a specified condition evaluates to True.
Syntax

```
if condition:
    statements
```

Example

```
age = 20

if age >= 18:
    print("Eligible to Vote")
```

Output

```
Eligible to Vote
```

Flowchart

```
Start
   |
Condition?
 /     \
Yes     No
 |       |
Print   Skip
 |
End
```

(B) if-else Statement

Definition

Executes one block if the condition is **True** and another block if it is **False**. If Else statement is used to execute one block of code when the condition is True and another block when the condition is False.

Syntax

```
if condition:
    statements
else:
    statements
```

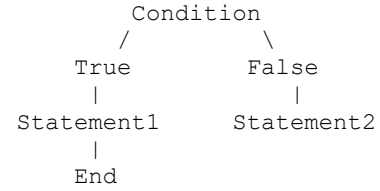
Example

```
num = 10

if num % 2 == 0:
    print("Even")
else:
    print("Odd")
```

Output

```
Even
```

Flowchart



(C) if-elif-else Statement

Definition

Used to check **multiple conditions**. statement is used to check multiple conditions in a program. It executes a block of code when its condition evaluates to True after previous conditions evaluate to False.

Syntax

```
if condition1:
    statements

elif condition2:
    statements

elif condition3:
    statements

else:
    statements
```

Example

```
marks = 75

if marks >= 90:
    print("Grade A")

elif marks >= 75:
    print("Grade B")

elif marks >= 60:
    print("Grade C")

else:
    print("Fail")
```

Output

```
Grade B
```

(D) Nested if Statement
Definition

An **if statement inside another if statement** is called a nested if. Nested if-else statement is an if-else statement placed inside another if or else block. It is used to check conditions within another condition.

Example

```
age = 20
citizen = True

if age >= 18:
    if citizen:
        print("Eligible to Vote")
```

Output

```
Eligible to Vote
```

Advantages of Conditional Statements

- Supports decision-making
- Improves program flexibility
- Easy to understand
- Reduces unnecessary execution

---

3. Loop Control Statements
Definition
Loops execute a block of code **repeatedly** until a condition becomes false. Loops are used to execute a block of code repeatedly until a condition is met or all items in a sequence are processed. The main types are For loops (iterating over sequences) and While loops (executing code based on a condition).

Types of Loops

1. `while` Loop
2. `for` Loop

(A) while Loop
Definition

Repeats statements while a condition is **True**. While loop repeatedly executes a block of code as long as the given condition remains true. When the condition becomes false, the line immediately after the loop in the program is executed.

Syntax

```
while condition:
    statements
```

Example

```
i = 1

while i <= 5:
    print(i)
    i += 1
```

Output

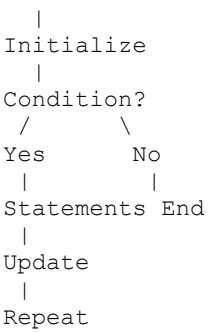
```
1
2
3
4
5
```

Explanation:

- i= 0 initializes the counter variable.
- while (i < =5): runs the loop while the condition is true.

- i += 1 increases the counter value by 1.

Flowchart



Advantages

- Best when the number of iterations is unknown.
- Useful for menus and user input validation.

(B) for Loop
Definition

Used to iterate over a **sequence** such as a string, list, tuple, or range. For loops is used to iterate over a sequence such as a list, tuple, string or range. It executes a block of code once for each item in the sequence.

Syntax

```
for variable in sequence:

    statements
```

Example

```
for i in range(1,6):
    print(i)
```

Output

```
1
2
3
4
5
```

Example with String

```
for ch in "Python":
    print(ch)
```

Output

```
P
y
t
h
o
n
```

`range()` Function

Used to generate a sequence of numbers.

Syntax

```
range(start, stop, step)
```

Example

```
for i in range(2,11,2):
```

```
    print(i)
```

Output

```
2
4
6
8
10
```

Difference Between `for` and `while`

for Loop	while Loop
Used when iterations are known	Used when iterations are unknown
Simpler for sequences	Better for condition-based repetition
Uses `range()` or collections	Uses a logical condition

---

4. Nested Control Flow
Definition

A **loop inside another loop** or an **if statement inside another control structure** is called nested control flow. A nested loop is a loop inside another loop. The inner loop executes completely for every iteration of the outer loop

---

Nested for Loop
Example

```
for i in range(1,4):
    for j in range(1,4):
        print(i, j)
```

Output

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

Nested while Loop

```
i = 1

while i <= 3:
    j = 1
    while j <= 3:
        print(i, j)
        j += 1
    i += 1
```

Applications

- Multiplication tables
- Matrix operations
- Pattern printing
- Searching in two-dimensional data

---

5. `continue` Statement
Definition

The `continue` statement **skips the current iteration** and immediately moves to the **next iteration** of the loop.

Syntax

```
continue
```

Example

```
for i in range(1,6):

    if i == 3:
        continue

    print(i)
```

Output

```
1
2
4
5
```

Flowchart

```
Loop
 |
Condition?
 |
Yes ---> continue
 |
No
 |
Execute remaining statements
 |
Next iteration
```

Advantages

- Skips unwanted values.
- Makes code cleaner than using multiple `if` statements.

---

6. `break` Statement

Definition

The `break` statement **immediately terminates the loop**.

Syntax

```
break
```

Example

```
for i in range(1,6):

    if i == 4:
        break

    print(i)
```

Output

```
1
2
3
```

Flowchart

```
Loop
 |
Condition?
 |
Yes ---> break
 |
No
 |
Execute statements
 |
Next iteration
```

Applications

- Search operations

- Menu-driven programs
- Exiting loops when the desired result is found

Difference Between `break` and `continue`

break	continue
Terminates the loop	Skips only the current iteration
Control moves outside the loop	Control moves to the next iteration
Used to stop execution	Used to ignore selected iterations

---

7. `pass` Statement

Definition

The `pass` statement is a **null (empty) statement**. It does **nothing** when executed.
It is used as a placeholder where a statement is syntactically required.

Syntax

```
pass
```

Example

```
for i in range(5):

    if i == 2:
        pass

    print(i)
```

Output

```
0
1
2
3
4
```

Example with Function

```
def future_function():
    pass
```

Uses of `pass`

- Placeholder during development
- Empty functions
- Empty classes
- Empty loops or conditions

Difference Between `pass` and `continue`

pass	continue
Does nothing	Skips the current iteration
Program continues normally	Moves directly to the next loop iteration
Can be used anywhere	Used only inside loops

---

8. `exit()` Function

Definition

The `exit()` function **terminates the entire Python program immediately**.
It is commonly used to stop execution when no further processing should occur.

Syntax

```
exit()
```

Example

```
print("Program Started")
```

```
exit()

print("Program Ended")
```

Output

```
Program Started
```

(The second `print()` statement is never executed.)

Difference Between `break`, `continue`, `pass`, and `exit()`

Statement	Purpose	Where Used
`break`	Stops the loop immediately	Loops
`continue`	Skips the current iteration	Loops
`pass`	Placeholder; performs no action	Anywhere a statement is required
`exit()`	Terminates the entire program	Entire program

Advantages of Control Flow Statements

- Improve decision-making
- Reduce code repetition
- Increase program flexibility
- Support complex logic
- Make programs interactive

Disadvantages

- Excessive nesting reduces readability.
- Incorrect loop conditions can create infinite loops.
- Overusing `break` and `continue` can make code harder to follow.

One-Page Revision (Exam Ready)

Topic	Key Point
Control Flow	Controls the order of program execution
`if`	Executes a block if the condition is `True`
`if-else`	Chooses between two blocks
`if-elif-else`	Checks multiple conditions
Nested `if`	`if` inside another `if`
`while` Loop	Repeats while a condition is `True`
`for` Loop	Iterates over a sequence or `range()`
`range()`	Generates a sequence of numbers
Nested Loops	One loop inside another
`continue`	Skips the current iteration
`break`	Terminates the loop immediately
`pass`	Placeholder that performs no action
	Ends the entire program

```
exit()
```

Functions in Python

UNIT-2: Functions in Python

Topics Covered

1. Introduction to Functions
2. Built-in Functions
3. Function Definition and Function Call
4. Scope and Lifetime of Variables
5. Default Parameters
6. Command Line Arguments

7. Lambda Functions
8. Assert Statement
9. Importing User-defined Modules

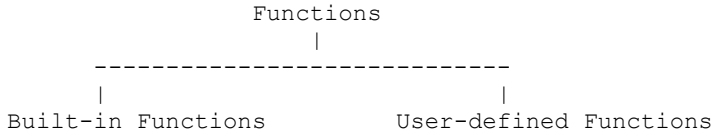
---

1. Introduction to Functions
Definition
A **function** is a **named block of reusable code** that performs a specific task. Python functions are reusable blocks of code used to perform a specific task. They help organize programs into smaller sections and execute the same logic whenever needed by calling the function.
Instead of writing the same code repeatedly, we write it once inside a function and call it whenever required.

Why Use Functions?

- Reduce code repetition
- Improve readability
- Make debugging easier
- Divide a large program into smaller parts
- Promote code reusability

Types of Functions



Advantages of Functions

- Code reusability
- Easy maintenance
- Better program organization
- Reduces program size
- Easier testing and debugging

---

2. Built-in Functions
Definition
Built-in functions are **predefined functions** provided by Python.
They are ready to use without defining them.

Common Built-in Functions
(A) print()
Displays output.

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

Output

```
Hello
```

(B) input()
Accepts input from the user.

```
name = input("Enter Name: ")
```

(C) len()
Returns the length of a string, list, tuple, etc.

```
text = "Python"
print(len(text))
```

Output

```
6
```

(D) type()
Returns the data type.

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

Output

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

(E) int()
Converts a value into an integer.
```
x = int("20")
print(x)
```
Output
```
20
```

(F) float()
Converts a value into a floating-point number.
```
print(float(5))
```
Output
```
5.0
```

(G) str()
Converts data into a string.
```
print(str(100))
```
Output
```
100
```

(H) max()
Returns the largest value.
```
print(max(10,20,30))
```
Output
```
30
```

(I) min()
Returns the smallest value.
```
print(min(10,20,30))
```
Output
```
10
```

(J) sum()
Returns the sum of elements.
```
print(sum([1,2,3]))
```
Output
```
6
```

Summary Table

Function	Purpose
print()	Display output
input()	Take input
len()	Find length
type()	Check data type
int()	Convert to integer
float()	Convert to float
str()	Convert to string
max()	Largest value
min()	Smallest value
sum()	Sum of elements

---

3. Function Definition and Function Call
Function Definition
A function is created using the **def** keyword.
Syntax
```
def function_name(parameters):
    statements
```

Example

```
def greet():
    print("Welcome to Python")
```
The function is only **defined**, not executed.

Function Call
To execute a function, call it by its name. After creating a function, call it by using the name of the functions followed by parenthesis containing parameters of that particular function.
```
greet()
```
Output
```
Welcome to Python
```

Function with Parameters
```
def add(a,b):
    print(a+b)

add(10,20)
```
Output
```
30
```

Function with Return Value
```
def square(x):
    return x*x

result = square(5)

print(result)
```
Output
```
25
```

Types of User-defined Functions
1. No Arguments, No Return Value
```
def hello():
    print("Hello")
```

---

3. Arguments, No Return Value

Arguments are values passed to a function when it is called. They allow functions to receive input data and perform operations using those values.

```
def add(a,b):
    print(a+b)
```

Types of Function Arguments

Default argument: Default argument use a predefined value when no value is passed during the function call.
Keyword Arguments: pass values using parameter names, so argument order does not matter.
Positional Arguments: values are assigned to parameters based on their order in the function call.
Arbitrary Arguments: allow functions to accept multiple values. This is done using two special symbols

---

3. No Arguments, Return Value
```
def number():
    return 100
```

---

4. Arguments and Return Value: Return is used to end a function and send a value back to the caller. It can return any data type, multiple values (packed into a tuple), or None if no value is given.

```
def multiply(a,b):
```

```
    return a*b
```

---

4. Scope and Lifetime of Variables
Scope of Variables
Definition
The **scope** of a variable is the region of the program where it can be accessed.

Types of Scope
(A) Local Variable
Declared inside a function.
Accessible only within that function.
Example
```
def test():

    x = 10

    print(x)

test()
```
Output
```
10
```

(B) Global Variable
Declared outside all functions.
Accessible throughout the program.
Example
```
x = 50

def show():

    print(x)

show()
```
Output
```
50
```

Difference

Local Variable	Global Variable
Declared inside function	Declared outside function
Accessible only inside function	Accessible everywhere
Lifetime ends after function finishes	Exists until program ends

---

Lifetime of Variables
Local Variable
Exists only while the function is executing.
Global Variable
Exists during the entire execution of the program.

---

5. Default Parameters
Definition
A **default parameter** has a predefined value.
If the caller does not provide an argument, the default value is used.

Syntax
```
def function(parameter=value):
```

Example
```
def greet(name="Student"):

    print("Hello", name)
```

```
greet()

greet("Rahul")
```
Output
```
Hello Student

Hello Rahul
```

Advantages

- Reduces the number of arguments needed.
- Makes functions more flexible.
- Simplifies function calls.

---

6. Command Line Arguments
Definition
Command line arguments are values passed to a Python program when it starts.
They are accessed using the **sys** module.

Syntax
```
import sys

print(sys.argv)
```

Example
Suppose the file is:
`demo.py`
Run from the command line:
```
python demo.py Rahul 20
```
Program
```
import sys

print(sys.argv)
```
Output
```
['demo.py', 'Rahul', '20']
```

Applications

- Passing filenames
- Passing user input
- Automation scripts

---

7. Lambda Functions
Definition
A **Lambda Function** is an **anonymous (unnamed) function**.
It is used for small, simple operations.

Syntax
```
lambda arguments : expression
```

Example
```
square = lambda x: x*x

print(square(5))
```
Output
```
25
```

Example 2
```
add = lambda a,b: a+b

print(add(10,20))
```
Output
```
30
```

Advantages

- Short code
- No need to define a named function
- Useful with `map()`, `filter()`, and `sorted()`

Disadvantages

- Limited to a single expression
- Less readable for complex logic

Difference Between Normal Function and Lambda Function

Normal Function	Lambda Function
Uses `def`	Uses `lambda`
Has a name	Usually anonymous
Can contain multiple statements	Only one expression
Better for complex logic	Best for simple operations

8. Assert Statement
Definition
The **assert** statement is used for **debugging**.
It checks whether a condition is **True**.
If the condition is **False**, Python raises an **AssertionError**.

Syntax

```
assert condition
```

Example

```
age = 20

assert age >= 18

print("Eligible")
```

Output

```
Eligible
```

Example with Error

```
x = -5

assert x > 0
```

Output

```
AssertionError
```

Advantages

- Detects programming errors early
- Simplifies debugging
- Improves code reliability

9. Importing User-defined Modules
Definition
A **module** is a Python file (`.py`) containing functions, variables, and classes.
Instead of rewriting code, we import it into another program.

Creating a Module
File: `maths.py`

```
def add(a,b):

    return a+b
```

Importing the Module

File: `main.py`

```
import maths

print(maths.add(10,20))
```

Output

```
30
```

Import Specific Function

```
from maths import add

print(add(5,10))
```

Output

```
15
```

Import with Alias

```
import maths as m

print(m.add(5,5))
```

Output

```
10
```

Advantages of Modules

- Code reusability
- Better organization
- Easier maintenance
- Faster development
- Avoids duplicate code

Difference Between Built-in and User-defined Functions

Built-in Function	User-defined Function
Already available in Python	Created by the programmer
Example: `print()`	Example: `add()`
No definition required	Must be defined using `def`

Difference Between Local and Global Variables

Local Variable	Global Variable
Declared inside a function	Declared outside a function
Limited scope	Accessible throughout the program
Temporary lifetime	Exists until the program ends

One-Page Revision (Exam Ready)

Topic	Key Point
Function	Reusable block of code defined using `def`
Built-in Functions	`print()`, `input()`, `len()`, `type()`, `max()`, `min()`, `sum()`, `int()`, `float()`, `str()`
Function Definition	`def function_name():`
Function Call	`function_name()`
Return Statement	Sends a value back to the caller
Local Variable	Declared inside a function
Global Variable	Declared outside a function
Default Parameter	Parameter with a default value
Command Line Arguments	Accessed using `sys.argv`
Lambda Function	Anonymous function using `lambda`
Assert Statement	Used to test conditions during debugging
Module	Python file containing reusable code

Topic	Key Point
Import Module	`import module`, `from module import function`, `import module as alias`

UNIT III

Unit-3: Mutable and Immutable Objects, Lists, Tuples, Dictionaries, Math and NumPy

1. Mutable and Immutable Objects in Python
Definition of Object
In Python, everything is an **object**. An object has:

- Identity (memory address)
- Type (data type)
- Value (data stored)

Example:
```
x = 10
```
Here:

- `x` is a reference
- `10` is an integer object

Mutable Objects
Definition
Mutable objects are objects whose **contents can be changed after creation** without creating a new object.
Examples:

- List
- Dictionary
- Set

Example of Mutable Object
```
numbers = [10, 20, 30]

numbers[0] = 100

print(numbers)
```
Output:
```
[100, 20, 30]
```
The original list is modified.

Features of Mutable Objects

- Can be modified after creation
- Memory location remains the same
- Faster for frequent changes
- Used when data needs updating

Immutable Objects
Definition
Immutable objects are objects whose **contents cannot be changed after creation**.
If we modify them, a new object is created.
Examples:

- Integer
- Float
- String
- Tuple

Example of Immutable Object

```
x = 10

x = x + 5

print(x)
```

Output:

```
15
```

A new integer object is created.

Difference Between Mutable and Immutable Objects

Mutable Objects	Immutable Objects
Can be changed after creation	Cannot be changed
Same memory location is modified	New object is created
Example: List, Dictionary	Example: Tuple, String
More flexible	More secure
Slightly slower	Faster

---

2. Lists in Python

Definition

A **list** is an ordered collection of elements that can store multiple values. List is a built-in data structure used to store an ordered collection of items. They are dynamic, resizable and capable of storing multiple data types.

Lists are:

- Mutable : list elements can be changed, updated, added, or removed after the list is created.
- Indexed : elements are accessed using their position, starting from index 0.
- Ordered: elements maintain the order in which they are inserted.
- Allow duplicate values
- Can store different data types

Creating a List: Lists can be created in several ways, such as using square brackets [], the list() constructor or by repeating elements.

Empty List: Square brackets [] are used to create a list directly.

```
list1 = []
```

List with Elements

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

Mixed Data Types

```
data = [10, "Python", 5.5, True]
```

Accessing List Elements: Elements in a list are accessed using indexing. Python uses zero-based indexing, meaning a[0] represents the first element. Negative indexing is also supported, where -1 accesses the last element.

Example:

```
numbers = [10,20,30]

print(numbers[0])
```

Output:

```
10
```

Negative Indexing

```
List:

10  20  30  40

-4  -3  -2  -1
```

Example:

```
print(numbers[-1])
```

Output:

```
40
```

List Operations

1. Adding Elements

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

Adds an element at the end.

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

a.append(4)

print(a)
```

Output:

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

---

insert()

Adds an element at a specific position.

```
a.insert(1,10)
```

Output:

```
[1,10,2,3]
```

---

extend()

Adds elements from another list.

```
a=[1,2]

a.extend([3,4])
```

Output:

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

---

2. Removing Elements

remove()

Removes a specific value.

```
a=[10,20,30]

a.remove(20)
```

Output:

```
[10,30]
```

---

pop()

Removes an element using index .Removes the element at a specific index or the last element if no index is specified.

```
a.pop()
```

Removes the last element.

---

clear()

Removes all elements.

```
a.clear()
```

---

3. List Slicing

Syntax:

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

Example:

```
a=[10,20,30,40,50]

print(a[1:4])
```

Output:

```
[20,30,40]
```

---

3. Tuples in Python

Definition

A tuple is an ordered collection of elements that is **immutable**.

Tuples are similar to lists, but unlike lists, they cannot be changed after their creation.

Can hold elements of different data types.

These are ordered, heterogeneous and immutable.

Creating a Tuple: A tuple is created by placing all the items inside parentheses (), separated by commas. A tuple can have any number of items.

```
t=(10,20,30)
```

Single Element Tuple

```
t=(10,)
```

Comma is required.

Accessing Tuple Elements: We can access the elements of a tuple by using indexing and slicing, similar to how we access elements in a list. Indexing starts at 0 for the first element and goes up to n-1, where n is the number of elements in the tuple. Negative indexing starts from -1 for the last element and goes backward.

```
t=(10,20,30)

print(t[1])
```

Output:

```
20
```

Tuple Characteristics

- Ordered
- Immutable
- Allows duplicates
- Faster than lists
- Supports indexing and slicing

Tuple Operations

Concatenation: Tuples can be concatenated using the + operator. This operation combines two or more tuples to create a new tuple.

```
a=(1,2)

b=(3,4)

print(a+b)
```

Output:

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

Repetition

```
t=(1,2)

print(t*3)
```

Output:

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

Slicing: Slicing a tuple means creating a new tuple from a subset of elements of the original tuple. The slicing syntax is tuple[start:stop:step].

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

print(t[1:3])
```

Output:

```
(20,30)
```

Tuple Methods

Since tuples are immutable, they have only two methods:

count()
Counts occurrences.

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

print(t.count(2))
```

Output:

```
2
```

---

index()
Returns position of an element.

```
t=(10,20,30)

print(t.index(20))
```

Output:

```
1
```

---

4. Dictionaries in Python
Definition
A dictionary is a collection of data stored as **key-value pairs.** Dictionary is a data structure that stores information in key-value pairs. While keys must be unique and immutable (like strings or numbers), values can be of any data type, whether mutable or immutable. This makes dictionaries ideal for accessing data by a specific name rather than a numeric position like in list.
Dictionary is:

- Mutable
- Unordered (before Python 3.7)
- Ordered (Python 3.7+)
- Uses unique keys

Creating a Dictionary: A dictionary is created by writing key-value pairs inside { }, where each key is connected to a value using colon (:). A dictionary can also be created using dict() function.

EXAMPLE

```
a = {"x": 1, "y": 2}
print(a)
b = dict(name="Sam", age=20)
print(b)
```

Output

```
{'x': 1, 'y': 2}
{'name': 'Sam', 'age': 20}

student = {

"name":"Rahul",

"age":20,

"branch":"CSE"

}
```

Accessing Values: A value in a dictionary is accessed by using its key. This can be done either with square brackets [ ] or with the get() method. Both return the value linked to the given key.

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

Output:

```
Rahul
```

Adding New Item: New items are added to a dictionary using the assignment operator (=) by giving a new key a value. If an existing key is used with the assignment operator, its value is updated with the new one.

```
student["marks"]=90
```

Updating Value

```
student["age"]=21
```

Removing Items: Dictionary items can be removed using built-in deletion methods that work on keys

pop()

```
student.pop("age")
```

---

del

```
del student["name"]
```

Dictionary Methods: Python dictionary methods are built-in functions that allow us to perform various operations on dictionaries, such as adding, updating, accessing and removing key-value pairs.

keys(): Returns a view object containing all dictionary keys.

Returns all keys.

```
print(student.keys())
```

---

values(): Returns a view object containing all dictionary values.

Returns all values.

```
print(student.values())
```

---

items(): Returns a view object containing all key-value pairs as tuples.

Returns key-value pairs.

```
print(student.items())
```

---

get(): Returns the value associated with the specified key. If the key is not found, it returns None (or a default value if provided).

Returns value of a key.

```
student.get("name")
```

---

update(): Updates the dictionary using another dictionary or iterable of key-value pairs.

Updates dictionary.

```
student.update({"age":22})
```

---

5. Commonly Used Functions on Lists

Function	Purpose
len()	Returns length
max()	Maximum value
min()	Minimum value
sum()	Sum of elements
sorted()	Sorts list
list()	Converts to list

Examples

len()

```
a=[10,20,30]

print(len(a))
```

Output:

```
3
```

---

max()

```
print(max([5,10,2]))
```

Output:

```
10
```

---

sum()

```
print(sum([1,2,3]))
```
Output:
```
6
```

---

6. Commonly Used Functions on Tuples

Function	Purpose
len()	Length
max()	Largest value
min()	Smallest value
sum()	Addition
sorted()	Sorting

Example:
```
t=(5,2,8)

print(max(t))
```
Output:
```
8
```

---

7. Commonly Used Functions on Dictionaries

Function	Purpose
len()	Number of key-value pairs
keys()	Returns keys
values()	Returns values
items()	Returns pairs
get()	Access value
update()	Modify dictionary
pop()	Delete item

---

8. Passing Lists, Tuples and Dictionaries as Arguments to Functions
Python allows collections to be passed as function arguments.

Passing List as Argument
Example:
```
def display(numbers):

    for n in numbers:
        print(n)


list1=[10,20,30]

display(list1)
```
Output:
```
10
20
30
```

Passing Tuple as Argument
Example:
```
def show(t):

    print(t)


tuple1=(1,2,3)

show(tuple1)
```
Output:

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

Passing Dictionary as Argument
Example:

```
def student_info(data):

    print(data["name"])


student={

"name":"Rahul",

"age":20

}

student_info(student)
```

Output:

```
Rahul
```

Advantages

- Allows passing multiple values
- Reduces number of arguments
- Makes programs flexible

---

9. Math Module in Python
Definition
The **math module** provides mathematical functions and constants.
It must be imported before use.

Importing Math Module

```
import math
```

Common Math Functions
sqrt()
Square root.

```
math.sqrt(25)
```

Output:

```
5
```

---

pow()
Power calculation.

```
math.pow(2,3)
```

Output:

```
8
```

---

factorial()

```
math.factorial(5)
```

Output:

```
120
```

---

ceil()
Rounds upward.

```
math.ceil(4.2)
```

Output:

```
5
```

---

floor()
Rounds downward.

```
math.floor(4.8)
```

Output:

Constants
pi
```
math.pi
```
Value:
```
3.14159
```

---

e
```
math.e
```
Value:
```
2.71828
```

---

10. NumPy Module for Lists and Arrays
Definition
NumPy (Numerical Python) is a Python library used for numerical computing.
It provides:

- Arrays
- Mathematical operations
- Matrix operations

Installing NumPy
```
pip install numpy
```

Importing NumPy
```
import numpy as np
```

Creating NumPy Array
```
import numpy as np

arr=np.array([1,2,3,4])

print(arr)
```
Output:
```
[1 2 3 4]
```

Advantages of NumPy Arrays

- Faster than Python lists
- Uses less memory
- Supports mathematical operations
- Supports multi-dimensional arrays

NumPy Array Operations
Addition
```
a=np.array([1,2,3])

b=np.array([4,5,6])

print(a+b)
```
Output:
```
[5 7 9]
```

Multiplication
```
print(a*2)
```
Output:
```
[2 4 6]
```

Creating Special Arrays
zeros()
```
np.zeros(5)
```
Output:

```
[0. 0. 0. 0. 0.]
```

---

ones()

```
np.ones(3)
```

Output:

```
[1. 1. 1.]
```

---

arange()

```
np.arange(1,10,2)
```

Output:

```
[1 3 5 7 9]
```

Difference Between List and NumPy Array

Python List	NumPy Array
General purpose	Numerical computing
Slower	Faster
More memory	Less memory
Supports different data types	Usually same data type
No direct mathematical operations	Supports vector operations

One-Page Revision (Exam Ready)

Topic	Key Point
Mutable Objects	Objects that can be changed after creation
Immutable Objects	Objects that cannot be modified
List	Mutable ordered collection
Tuple	Immutable ordered collection
Dictionary	Mutable key-value collection
List Methods	append(), insert(), remove(), pop(), extend()
Tuple Methods	count(), index()
Dictionary Methods	keys(), values(), items(), get(), update()
Function Arguments	Lists, tuples, dictionaries can be passed as arguments
Math Module	Provides mathematical functions like sqrt(), factorial(), pow()
NumPy	Library for numerical computing and arrays

Unit-3: Files Handling in Python

1. Introduction to File Handling

Definition

A **file** is a collection of data stored permanently on a secondary storage device such as a hard disk, SSD, or USB drive.
Python provides file handling features to:

- Create files
- Open files
- Read data from files
- Write data into files
- Update and delete files

Why File Handling is Required?

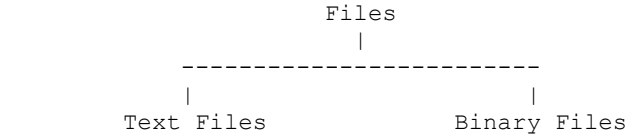
Variables store data temporarily in RAM. When the program ends, the data is lost. File handling is an important part of any web application.
Files allow data to be stored **permanently**.
Example:

- Student records
- Employee information
- Database backups

- Configuration files
- Reports

---

2. Types of Files
Files are mainly divided into two types:



A) Text Files
Definition
A text file stores data in the form of **characters**.
The data is readable by humans.
Examples:

- `.txt`
- `.csv`
- `.py`
- `.html`

Example Content

```
Name: Rahul
Age: 20
Course: Python
```

Characteristics of Text Files

- Data stored as characters
- Easy to read and edit
- Uses character encoding (ASCII/Unicode)
- Slower than binary files

B) Binary Files
Definition
Binary files store data in the form of **bytes (0s and 1s)**.
They are not directly readable by humans.
Examples:

- Images (`.jpg`, `.png`)
- Audio (`.mp3`)
- Videos (`.mp4`)
- Executable files (`.exe`)

Characteristics of Binary Files

- Stores data in binary format
- Faster processing
- Requires special software to read
- More compact than text files

Difference Between Text and Binary Files

Text Files	Binary Files
Store characters	Store bytes
Human-readable	Not human-readable
Slower	Faster
Larger size	Smaller size
Example: `.txt`, `.csv`	Example: `.jpg`, `.mp3`

---

3. File Handling Operations in Python

The main steps of file handling are:

1. Opening a file
2. Reading/Writing data
3. Closing the file

Opening a File

Python uses the `open()` function. The key function for working with files in Python is the `open()` function.
The `open()` function takes two parameters; *filename*, and *mode*.

Syntax

```
file_object = open(filename, mode)
```

Example

```
file = open("data.txt", "r")
```

File Modes in Python

Mode	Meaning
r	Read mode
w	Write mode
a	Append mode
x	Create new file
b	Binary mode
t	Text mode
r+	Read and write
w+	Write and read

Examples

Reading Mode: The `open()` function returns a file object, which has a `read()` method for reading the content of the file

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

Writing Mode: To write to an existing file, you must add a parameter to the `open()` function:
`"a"` - Append - will append to the end of the file
`"w"` - Write - will overwrite any existing content

```
file = open("student.txt","w")
```

Binary Reading Mode

```
file = open("image.jpg","rb")
```

---

4. **Creating a Text File:**
   To create a new file in Python, use the `open()` method, with one of the following parameters:
   `"x"` - Create - will create a file, returns an error if the file exists
   `"a"` - Append - will create a file if the specified file does not exists
   `"w"` - Write - will create a file if the specified file does not exists
5.

Using Write Mode

```
file = open("sample.txt","w")

file.write("Hello Python")

file.close()
```

Output file:

```
Hello Python
```

---

5. Writing Data into Text Files
write() Method

Writes a string into a file.
Example

```
file = open("data.txt","w")

file.write("Python Programming")

file.close()
```

---

writelines() Method
Writes multiple lines.
Example:

```
file = open("data.txt","w")

lines = [
"Python\n",
"Java\n",
"C++"
]

file.writelines(lines)

file.close()
```

---

6. Reading Data from Text Files
read() Method
Reads the complete file.
Example:

```
file = open("data.txt","r")

content = file.read()

print(content)

file.close()
```

---

readline() Method
Reads one line at a time.
Example:

```
file = open("data.txt","r")

print(file.readline())

file.close()
```

---

readlines() Method
Reads all lines and returns a list.
Example:

```
file = open("data.txt","r")

lines = file.readlines()

print(lines)

file.close()
```

---

7. Closing a File
The `close()` method releases the file resource.
Syntax:

```
file.close()
```

Using with Statement
Python automatically closes files using `with`.
Example:

```
with open("data.txt","r") as file:

    print(file.read())
```
Advantages:

- Automatically closes file
- Safer
- Cleaner code

---

8. Binary Files in Python
Definition
Binary files store information as bytes.
Python uses:

- `rb` → Read binary
- `wb` → Write binary

Writing Binary File
Example:
```
file = open("data.bin","wb")

file.write(b"Python")

file.close()
```

Reading Binary File
Example:
```
file = open("data.bin","rb")

data = file.read()

print(data)

file.close()
```
Output:
```
b'Python'
```

---

9. The Pickle Module
Definition
The **pickle module** is used for **serialization and deserialization** of Python objects.

Serialization
Converting Python objects into byte streams is called serialization.
Also called:
Pickling

Deserialization
Converting byte streams back into Python objects is called deserialization.
Also called:
Unpickling

Import Pickle Module
```
import pickle
```

Pickle Functions
1. dump()
Used to store objects into a binary file.
Syntax:
```
pickle.dump(object, file)
```

Example:
```
import pickle
```

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

file = open("student.dat","wb")

pickle.dump(student,file)

file.close()
```

---

2. load()
Used to read objects from a binary file.
Example:

```
import pickle

file = open("student.dat","rb")

data = pickle.load(file)

print(data)

file.close()
```

Output:

```
{'name':'Rahul','age':20}
```

Advantages of Pickle

- Stores complex Python objects
- Easy object storage
- Maintains object structure
- Useful for data persistence

Limitations of Pickle

- Only works with Python
- Not secure with unknown files
- Not human-readable

---

10. CSV Files
Definition
CSV stands for:
Comma Separated Values
A CSV file stores data in tabular form.
Example:

```
Name,Age,Course
Rahul,20,Python
Amit,21,Java
```

Uses of CSV Files

- Data exchange
- Spreadsheet storage
- Database import/export

Reading CSV Files in Python
Python provides the `csv` module.
Import:

```
import csv
```

Reading CSV File
Example:

```
import csv
```

```
file = open("student.csv","r")

reader = csv.reader(file)

for row in reader:
    print(row)

file.close()
```
Output:
```
['Name','Age']
['Rahul','20']
```

Writing CSV Files
csv.writer()
Example:
```
import csv

file = open("student.csv","w")

writer = csv.writer(file)

writer.writerow(
["Name","Age"]
)

writer.writerow(
["Rahul",20]
)

file.close()
```

Writing Multiple Rows
Using `writerows()`:
```
writer.writerows([
["Rahul",20],
["Amit",21]
])
```

CSV Reader and Writer Methods

Method	Purpose
reader()	Reads CSV data
writer()	Writes CSV data
writerow()	Writes one row
writerows()	Writes multiple rows

---

11. JSON Files
Definition
JSON stands for:
JavaScript Object Notation
It is a lightweight format used for data exchange.
Example:
```
{
"name":"Rahul",
"age":20
}
```

Advantages of JSON

- Easy to read
- Language independent
- Used in APIs

- Lightweight

JSON Module in Python
Import:
```
import json
```

Writing JSON Data
dump()
Writes Python object into JSON file.
Example:
```
import json

student={
"name":"Rahul",
"age":20
}

file=open("student.json","w")

json.dump(student,file)

file.close()
```

Reading JSON Data
load()
Reads JSON file.
Example:
```
import json

file=open("student.json","r")

data=json.load(file)

print(data)

file.close()
```
Output:
```
{'name':'Rahul','age':20}
```

JSON Functions

Function	Purpose
dump()	Write JSON data
load()	Read JSON data
dumps()	Convert object to JSON string
loads()	Convert JSON string to object

---

12. Difference Between CSV and JSON

CSV	JSON
Tabular data format	Key-value data format
Uses rows and columns	Uses objects and arrays
Less flexible	More flexible
Used in spreadsheets	Used in web applications
Easy for tables	Good for hierarchical data

---

13. Difference Between Pickle and JSON

Pickle	JSON
Python-specific	Language independent
Stores Python objects	Stores structured data

Pickle	JSON
Binary format	Text format
Not human-readable	Human-readable
More powerful	More portable

Important Exam Programs

1. Create and Write a Text File

```
file=open("test.txt","w")

file.write("Python File Handling")

file.close()
```

---

2. Read a Text File

```
file=open("test.txt","r")

print(file.read())

file.close()
```

---

3. Copy Data from One File to Another

```
f1=open("a.txt","r")

f2=open("b.txt","w")

f2.write(f1.read())

f1.close()
f2.close()
```

One-Page Revision (Exam Ready)

Topic	Key Point
File	Permanent storage of data
Text File	Stores characters
Binary File	Stores bytes
open()	Opens a file
read()	Reads complete file
readline()	Reads one line
write()	Writes data
close()	Closes file
Pickle	Stores Python objects
dump()	Serialization
load()	Deserialization
CSV	Comma-separated tabular data
JSON	Key-value data exchange format
csv.reader()	Reads CSV
csv.writer()	Writes CSV
json.load()	Reads JSON
json.dump()	Writes JSON

UNIT-4

Unit-4: Exception Handling in Python

1. Introduction to Exception Handling
Definition
An **exception** is an error or unexpected event that occurs during the execution of a program and interrupts the normal flow of execution.
Exception handling is a mechanism in Python used to handle runtime errors so that the program can continue or terminate gracefully.

Examples of Exceptions
Division by Zero
```
a = 10
b = 0

print(a/b)
```
Output:
```
ZeroDivisionError
```

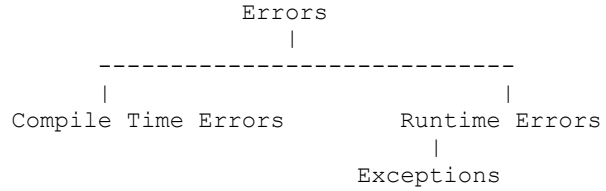
Invalid Data Type
```
x = "10"

print(x + 5)
```
Output:
```
TypeError
```

Need for Exception Handling
Without exception handling:
- Program stops suddenly
- Data may be lost
- User gets confusing error messages

With exception handling:
- Errors are handled properly
- Program execution can continue
- User receives meaningful messages

---

2. Types of Errors in Python
Python errors are mainly classified into:



A) Syntax Errors
Errors caused due to incorrect Python syntax.
Example:
```
print("Hello"
```
Output:
```
SyntaxError
```

B) Runtime Errors (Exceptions)
Errors that occur while the program is running.
Examples:
- ZeroDivisionError
- ValueError
- TypeError
- FileNotFoundError

---

3. Exception Handling Keywords
Python provides the following keywords:

Keyword	Purpose
try	Contains risky code
except	Handles exceptions
else	Executes when no exception occurs
finally	Always executes
raise	Manually generates an exception

---

4. Try-Except Block
Definition
The **try-except block** is used to handle exceptions in Python.
The code that may generate an error is placed inside the `try` block, and the handling code is placed inside the `except` block.

Syntax
```
try:
    statements
except:
    statements
```

Example
```
try:
    a = 10
    b = 0
    c = a/b

except:
    print("Cannot divide by zero")
```
Output:
```
Cannot divide by zero
```

Handling Specific Exceptions
Instead of using a general exception, we can specify the exception type.
Example
```
try:
    num = int(input("Enter number: "))
    print(num)

except ValueError:
    print("Invalid input")
```

Output:
```
Enter number: abc
Invalid input
```

Multiple Except Blocks
A single try block can have multiple except blocks.
Example
```
try:
    a = int(input("Enter number: "))
    b = int(input("Enter divisor: "))

    print(a/b)

except ValueError:
    print("Enter only numbers")

except ZeroDivisionError:
    print("Cannot divide by zero")
```

---

5. Try-Except-Else Block

Definition
The `else` block executes only when the `try` block executes successfully without generating any exception.

Syntax
```
try:
    statements

except:
    statements

else:
    statements
```

Example
```
try:
    a = 10
    b = 2

    result = a/b

except ZeroDivisionError:
    print("Error")

else:
    print("Result =", result)
```
Output:
```
Result = 5.0
```

Flow of Try-Except-Else
```
          Start
            |
          try block
            |
     ----------------
     |              |
  Exception       No Exception
     |              |
  except          else
     |
    End
```

Advantages of Else Block
- Separates normal code from error handling
- Improves readability
- Executes only when try is successful

---

6. Finally Block
Definition
The `finally` block contains code that is **always executed**, whether an exception occurs or not.
It is generally used for cleanup operations.
Examples:
- Closing files
- Releasing resources
- Closing database connections

Syntax
```
try:
    statements

except:
    statements
```

```
finally:
    statements
```

Example

```
try:
    file = open("data.txt","r")

    print(file.read())

except:
    print("File error")

finally:
    print("Execution completed")
```

Output:

```
Execution completed
```

Example Without Exception

```
try:
    print(10/2)

except:
    print("Error")

finally:
    print("Always executed")
```

Output:

```
5.0
Always executed
```

---

7. Complete Try-Except-Else-Finally Structure

Syntax

```
try:
    risky code

except Exception:
    handling code

else:
    successful execution code

finally:
    cleanup code
```

Execution Order

```
try
 |
 |---- Exception occurs
 |          |
 |       except
 |
 |
 |---- No Exception
           |
          else

finally executes always
```

---

8. Raise Statement

Definition

The `raise` statement is used to **manually generate an exception**.
It allows programmers to create custom error conditions.

Syntax

```
raise ExceptionType
```

Example 1

```
age = 15

if age < 18:
    raise ValueError("Age must be 18 or above")
```

Output:

```
ValueError: Age must be 18 or above
```

Example 2

```
number = -5

if number < 0:
    raise Exception("Negative number not allowed")
```

Uses of Raise Statement

- Validate user input
- Create custom errors
- Control program execution

---

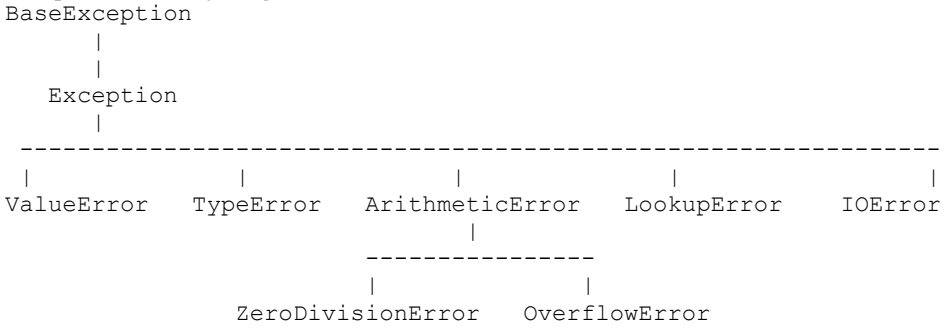
9. Hierarchy of Exceptions in Python

Definition

Python exceptions are organized in a hierarchical structure.
All exceptions are derived from the base class:

```
BaseException
```

Exception Hierarchy Diagram



Important Exception Classes

1. BaseException

The root class for all exceptions.

---

2. Exception

Base class for most built-in exceptions.

---

3. ArithmeticError

Occurs during mathematical operations.
Child classes:

- ZeroDivisionError
- OverflowError

---

4. ZeroDivisionError

Occurs when dividing by zero.
Example:

```
10/0
```

---

5. ValueError

Occurs when correct type but invalid value is provided.

Example:
```
int("abc")
```

---

6. TypeError
Occurs when operation is performed on incorrect data type.
Example:
```
"5"+10
```

---

7. IndexError
Occurs when accessing an invalid list index.
Example:
```
a=[1,2]

print(a[5])
```

---

8. KeyError
Occurs when dictionary key does not exist.
Example:
```
d={"a":1}

print(d["b"])
```

---

9. FileNotFoundError
Occurs when opening a file that does not exist.
Example:
```
open("abc.txt")
```

---

10. Adding Exceptions (Creating Custom Exceptions)
Definition
Python allows programmers to create their own exceptions by defining new exception classes.
Custom exceptions are created by inheriting from the built-in `Exception` class.

Syntax
```
class CustomException(Exception):
    pass
```

Example
```
class AgeError(Exception):
    pass


age = 12

if age < 18:
    raise AgeError("Not eligible")
```
Output:
```
AgeError: Not eligible
```

Advantages of Custom Exceptions

- Makes programs easier to understand
- Provides meaningful error messages
- Helps handle application-specific errors

---

11. Exception Handling with User Defined Exception
Example:
```
class SalaryError(Exception):
    pass


salary = -1000

try:
```

```
    if salary < 0:
        raise SalaryError("Salary cannot be negative")

except SalaryError as e:
    print(e)
```

Output:

```
Salary cannot be negative
```

---

12. Difference Between Error and Exception

Error	Exception
Usually serious problems	Runtime problems
Program may stop completely	Can be handled
Example: Syntax Error	Example: ValueError
Mostly programmer mistakes	Unexpected situations

---

13. Difference Between Finally and Else

else	finally
Executes only when no exception occurs	Always executes
Used for normal execution	Used for cleanup
Optional	Optional

---

14. Advantages of Exception Handling

1. Prevents abnormal program termination.
2. Improves program reliability.
3. Provides meaningful error messages.
4. Separates error-handling code from normal code.
5. Helps in debugging.
6. Protects important resources.

One-Page Revision Notes

Topic	Key Point
Exception	Runtime error during execution
try	Contains risky code
except	Handles errors
else	Runs when no exception occurs
finally	Always executes
raise	Manually creates exception
Exception Hierarchy	BaseException → Exception → Specific Exceptions
ZeroDivisionError	Division by zero
ValueError	Invalid value
TypeError	Invalid data type
IndexError	Invalid index
KeyError	Missing dictionary key
Custom Exception	User-created exception class

Introduction to Data Visualization

Definition

Data Visualization is the process of representing data in a graphical or visual form so that information can be easily understood, analyzed, and interpreted.

Python provides various libraries for creating visualizations.

Importance of Data Visualization

- Makes complex data easy to understand
- Helps identify patterns and trends

- Improves decision-making
- Provides better data analysis
- Helps in comparing values

Common Python Libraries for Visualization
1. Matplotlib

- Most popular Python visualization library
- Used for 2D and basic 3D plotting
- Provides functions for charts and graphs

Import:

```
import matplotlib.pyplot as plt
```

---

2. NumPy
Used for numerical calculations and generating data.

```
import numpy as np
```

---

3. Seaborn
Used for advanced statistical visualization.

```
import seaborn as sns
```

---

2. Plotting 2D Graphics
Definition
A **2D graph** represents data using two axes:

- X-axis (horizontal axis)
- Y-axis (vertical axis)

Examples:

- Line graph
- Bar graph
- Scatter plot
- Histogram
- Pie chart

2.1 Line Plot
Definition
A line plot displays data points connected by straight lines.
It is used to show changes over time.

Syntax

```
plt.plot(x,y)
```

Example

```
import matplotlib.pyplot as plt

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

y = [10,20,30,40,50]

plt.plot(x,y)

plt.xlabel("X-axis")

plt.ylabel("Y-axis")

plt.title("Line Graph")

plt.show()
```

Output
A line joining points:

```
50 |              *
```

```
40 |          *
30 |       *
20 |    *
10 | *
   -----------------
     1 2 3 4 5
```

2.2 Bar Graph
Definition
A bar graph represents data using rectangular bars.
Used for comparing different categories.

Syntax
```
plt.bar(x,height)
```

Example
```
import matplotlib.pyplot as plt

students = ["A","B","C"]

marks = [80,90,70]

plt.bar(students,marks)

plt.title("Student Marks")

plt.show()
```

Applications
- Comparing sales
- Comparing marks
- Population analysis

2.3 Scatter Plot
Definition
A scatter plot displays individual data points on a graph.
It is used to show relationships between variables.

Syntax
```
plt.scatter(x,y)
```

Example
```
import matplotlib.pyplot as plt

x=[1,2,3,4]

y=[5,10,15,20]

plt.scatter(x,y)

plt.show()
```

---

3. Plotting 3D Graphics
Definition
A **3D graph** represents data using three axes:
- X-axis
- Y-axis
- Z-axis

It is used to visualize three-dimensional data.

Creating 3D Plot
Python uses:

```
from mpl_toolkits.mplot3d import Axes3D
```

Example: 3D Line Plot

```
import matplotlib.pyplot as plt

from mpl_toolkits.mplot3d import Axes3D


fig = plt.figure()

ax = fig.add_subplot(111, projection='3d')


x=[1,2,3,4]

y=[2,4,6,8]

z=[3,6,9,12]


ax.plot(x,y,z)

plt.show()
```

Applications of 3D Visualization

- Engineering design
- Scientific research
- Medical imaging
- Geographic data analysis
- Machine learning

---

4. Histogram

Definition

A **histogram** is a graphical representation of the distribution of numerical data.
It divides data into intervals called **bins**.

Uses of Histogram

- Shows frequency distribution
- Identifies data patterns
- Analyzes large datasets
- Finds data spread

Syntax

```
plt.hist(data)
```

Example

```
import matplotlib.pyplot as plt


marks=[50,60,70,70,80,90,90,100]


plt.hist(marks)

plt.xlabel("Marks")

plt.ylabel("Frequency")

plt.title("Student Marks Distribution")

plt.show()
```

Important Terms
Bin
A range of values used to group data.
Example:
```
0-10
10-20
20-30
```

Frequency
Number of times a value appears.

Difference Between Bar Graph and Histogram

Bar Graph	Histogram
Used for categorical data	Used for numerical data
Bars can have spaces	Bars touch each other
Shows comparison	Shows distribution
Example: Sales categories	Example: Marks distribution

---

5. Pie Chart
Definition
A **pie chart** is a circular chart divided into sectors.
Each sector represents a proportion of the total data.

Formula
Sector Angle=ValueTotal×360Sector\ Angle = \frac{Value}{Total} \times 360Sector Angle=TotalValue×360

Syntax
```
plt.pie(values, labels)
```

Example
```
import matplotlib.pyplot as plt


sizes=[40,30,20,10]

labels=["Python","Java","C++","JavaScript"]


plt.pie(
sizes,
labels=labels,
autopct="%1.1f%%"
)


plt.title("Programming Languages")

plt.show()
```

Features of Pie Chart

- Shows percentage distribution
- Easy comparison
- Attractive presentation

Applications

- Market share analysis
- Budget distribution
- Survey results
- Population distribution

6. Sine and Cosine Curves
Introduction
Sine and cosine functions are mathematical trigonometric functions.
They produce smooth wave-like curves.
They are widely used in:

- Signal processing
- Physics
- Engineering
- Data analysis

6.1 Sine Curve
Mathematical Function
y=sin(x)y = sin(x)y=sin(x)
The value of sine varies between:
−1≤sin(x)≤1-1 \leq sin(x) \leq 1−1≤sin(x)≤1

Python Program

```
import numpy as np

import matplotlib.pyplot as plt


x=np.linspace(0,2*np.pi,100)


y=np.sin(x)


plt.plot(x,y)

plt.title("Sine Curve")

plt.xlabel("x")

plt.ylabel("sin(x)")


plt.grid()

plt.show()
```

Characteristics of Sine Curve

- Periodic wave
- Repeats after 2π
- Maximum value = 1
- Minimum value = -1

6.2 Cosine Curve
Mathematical Function
y=cos(x)y = cos(x)y=cos(x)
Range:
−1≤cos(x)≤1-1 \leq cos(x) \leq 1−1≤cos(x)≤1

Python Program

```
import numpy as np

import matplotlib.pyplot as plt


x=np.linspace(0,2*np.pi,100)
```

```
y=np.cos(x)

plt.plot(x,y)

plt.title("Cosine Curve")
plt.xlabel("x")
plt.ylabel("cos(x)")

plt.grid()
plt.show()
```

Characteristics of Cosine Curve

- Periodic wave
- Starts from maximum value
- Maximum value = 1
- Minimum value = -1

Difference Between Sine and Cosine Curves

Sine Curve	Cosine Curve
y = sin(x)	y = cos(x)
Starts from zero	Starts from one
Wave crosses origin	Maximum at origin
Periodic function	Periodic function

---

7. Combining Sine and Cosine Curves

Example:

```
import numpy as np
import matplotlib.pyplot as plt

x=np.linspace(0,2*np.pi,100)

plt.plot(x,np.sin(x),label="Sine")
plt.plot(x,np.cos(x),label="Cosine")

plt.legend()
plt.grid()
plt.show()
```

---

8. Important Matplotlib Functions

Function	Purpose
plot()	Creates line graph
bar()	Creates bar chart
scatter()	Creates scatter plot
hist()	Creates histogram

Function	Purpose
pie()	Creates pie chart
xlabel()	Labels x-axis
ylabel()	Labels y-axis
title()	Adds graph title
legend()	Displays labels
show()	Displays graph
grid()	Adds grid lines

---

9. Advantages of Data Visualization

1. Easy understanding of large data
2. Helps find trends and patterns
3. Improves communication
4. Supports decision-making
5. Makes reports attractive
6. Helps detect errors in data

---

10. Applications of Data Visualization

- Business analytics
- Weather forecasting
- Scientific research
- Medical analysis
- Machine learning
- Financial analysis
- Education reports

One-Page Revision Notes

Topic	Important Point
Data Visualization	Graphical representation of data
Matplotlib	Python visualization library
2D Graphics	Uses X and Y axes
3D Graphics	Uses X, Y and Z axes
Histogram	Shows frequency distribution
Pie Chart	Shows percentage distribution
Sine Curve	y = sin(x)
Cosine Curve	y = cos(x)
NumPy	Used for numerical data generation
plot()	Line graph
hist()	Histogram
pie()	Pie chart