रा.इ.सू.प्रौ.सं
NIELIT
National Institute of Electronics & Information Technology

INFOMAX COMPUTER ACADEMY

O LEVEL
NIELIT CERTIFICATION

EXAM CAPSULE

FOR

O LEVEL NIELIT EXAM

PYTHON PROGRAMMING

COMPACT. CONCISE. COMPLETE.

Your Quick Guide to Exam Success

Exam Focused
Topic-wise coverage as per O Level syllabus

Quick Revision
Key points, summaries & important concepts

Practice Oriented
Solved examples & practice questions

Exam Ready
Important questions & exam tips

BASED ON LATEST SYLLABUS | TRUSTED BY STUDENTS | LEARN SMART SCORE HIGH | STEP CLOSER TO SUCCESS

Mob.: 8874588766 Web: www.infomax.org.in

PYTHON CHEAT SHEET

Introduction to Programming

INFOMAX COMPUTER ACADEMY

Topics: The basic Model of computation, algorithms, flowcharts, Programming Languages, compilation, testing & debugging and documentation.

1. THE BASIC MODEL OF COMPUTATION

A computer follows the **IPO cycle** to solve a problem.

- **Input:** Data is provided to the computer.
- **Process:** The computer processes the data according to instructions.
- **Output:** The result is produced.

Example (Add two numbers)

Input	Process	Output
a = 5 b = 3	sum = a + b	sum = 8

2. ALGORITHMS

An **algorithm** is a finite sequence of unambiguous steps to solve a problem.

Characteristics

- Input : Zero or more inputs
- Output : At least one output
- Definiteness : Each step is clear and unambiguous
- Finiteness : Must terminate after finite steps
- Effectiveness : Each step is basic and feasible

Example (Algorithm to find the largest of two numbers)

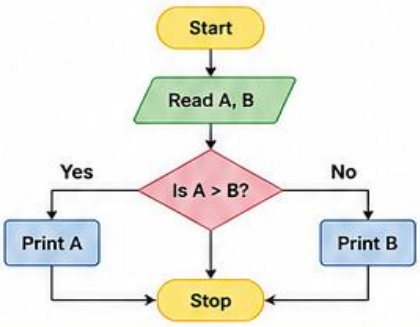
1. Start
2. Read two numbers A and B
3. If A > B, print A
4. Else print B
5. Stop

3. FLOWCHARTS

A **flowchart** is a graphical representation of an algorithm.

Common Symbols	
Terminator	Start / Stop
Input / Output	Read / Print
Process	Instruction
Decision	Yes / No
Flow Line	Direction

Example (Largest of two numbers)



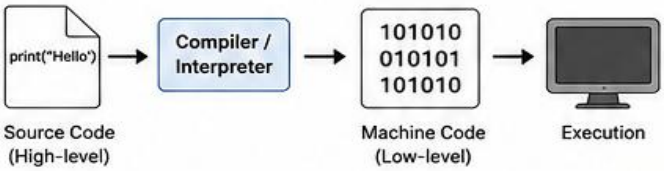
4. PROGRAMMING LANGUAGES

A programming language is a formal language used to write instructions for a computer.

- **Low-level Languages** : Closer to machine language (e.g., Assembly)
- **High-level Languages** : Closer to human language (e.g., Python, C, Java)
- **Python** is a high-level, interpreted, general-purpose programming language.

5. COMPILATION

Compilation is the process of converting source code written in a high-level language into machine code.



Python Note

Python is an interpreted language.
The interpreter executes the code line-by-line.
No separate compilation step is required.

6. TESTING & DEBUGGING

- **Testing** ensures the program works as expected.
- **Debugging** is the process of finding and fixing errors.

Common Types of Errors

- **Syntax Error** – Mistake in code structure
- **Runtime Error** – Occurs during execution
- **Logical Error** – Program runs but gives wrong output

Debugging Tips

- Read error messages carefully
- Use print() to check values
- Test with small inputs
- Fix one error at a time
- Re-run and retest

```
# Example: Debugging
a = 10
b = 0
try:
    c = a / b    # Runtime Error
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

7. DOCUMENTATION

Documentation is the process of writing and maintaining information about a program.

- Helps others understand your code
- Helps you in future updates
- Improves readability and maintenance

In Python

- Use comments (#) to explain code
- Use docstrings (""" """) for modules, functions, classes
- Keep documentation neat and updated

```
def add(a, b):
    """This function returns
       the sum of two numbers."""
    return a + b
```

Key Takeaway

Programming is about understanding the problem, designing the solution, writing the code, testing it, and documenting it well.

Happy Coding!

ALGORITHMS AND FLOWCHARTS
TO SOLVE PROBLEMS
Plan → Design → Implement → Test → Solve

INFOMAX

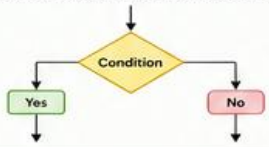
1. FLOW CHART SYMBOLS

SYMBOL	MEANING
	Terminator Start / Stop
	Process Processing or Instruction
	Input / Output Read / Write data
	Decision Condition check (Yes / No)
	Flow Line Direction of flow
	On-page Connector Connects flow within the page
	Off-page Connector Connects flow to another page

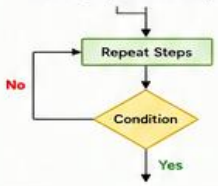
2. BASIC CONSTRUCTS

Sequence (Sequential Processing)

Selection (Decision Based Processing)



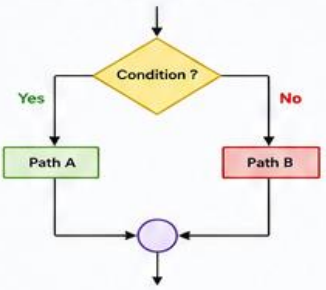
Iteration (Repetition / Looping)



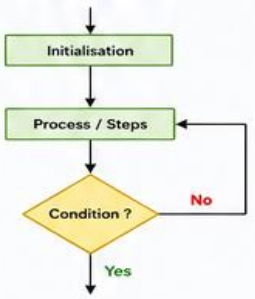
3. TYPES OF PROCESSING

SEQUENTIAL PROCESSING
Steps are executed one after another in a sequence.

DECISION BASED PROCESSING
A decision is made based on a condition and different paths are followed.



ITERATIVE PROCESSING
A set of steps is repeated until a condition is satisfied.



4. GENERAL ALGORITHM DESIGN STEPS

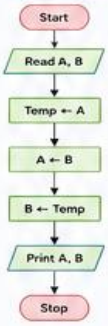
1. Identify the problem clearly.
2. Determine input and output.
3. Break the problem into smaller steps.
4. Design the algorithm (pseudocode).
5. Draw the flowchart.
6. Test with sample data.

5. PSEUDOCODE ELEMENTS

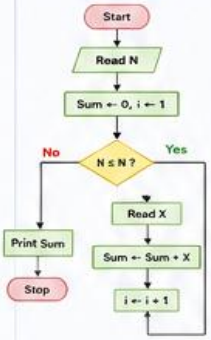
Input / Output	: READ, PRINT
Assignment	: ←
Arithmetic Ops	: +, -, *, /, %, ^
Relational Ops	: <, >, <=, >=, ==, !=
Logical Ops	: AND, OR, NOT

6. COMMON ALGORITHM FLOWCHART EXAMPLES

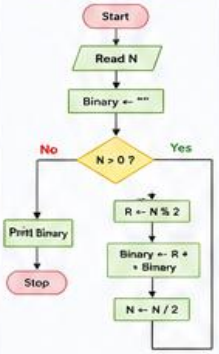
1. EXCHANGE VALUES OF TWO VARIABLES



2. SUMMATION OF N NUMBERS

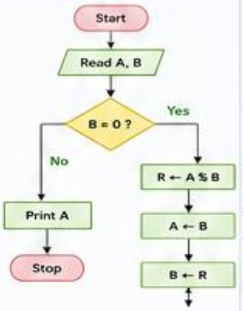


3. DECIMAL TO BINARY CONVERSION



4. REVERSE DIGITS OF AN INTEGER

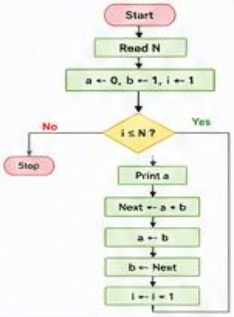
5. GCD (GREATEST COMMON DIVISOR) OF TWO NUMBERS



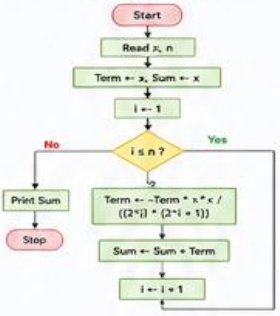
6. TEST WHETHER A NUMBER IS PRIME

7. FACTORIAL COMPUTATION

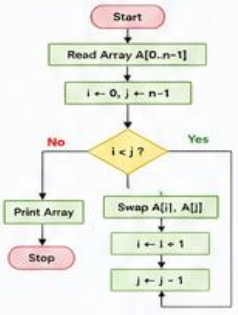
8. FIBONACCI SEQUENCE (UPTO N TERMS)



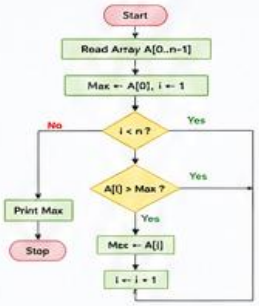
9. EVALUATE sin x AS SUM OF A SERIES



10. REVERSE ORDER OF ELEMENTS OF AN ARRAY



11. FIND LARGEST NUMBER IN AN ARRAY



12. PRINT ELEMENTS OF UPPER TRIANGULAR MATRIX

13. ALGORITHM TEMPLATE (PSEUDOCODE EXAMPLE)

```
Algorithm: Example_Algorithm
Input:  ...
Output: ...
Begin
   Read input data
   Initalise variables
   While / For / Do
      Perform operations
      If condition then
         Execute true part
      Else
         Execute false part
      End If
   End Loop
   Print output
End
```

14. TIPS FOR GOOD ALGORITHMS

- ✔ Use meaningful variable names.
- ✔ Keep steps simple and clear.
- ✔ Handle all possible cases.
- ✔ Avoid unnecessary steps.
- ✔ Test with different inputs.
- ✔ Draw neat and accurate flowcharts.

KEY POINTS
- Algorithm is a step-by-step solution to a problem.
- Flowchart is a graphical representation of an algorithm.
- Good algorithms are correct, efficient and easy to understand.

APPLICATIONS
- Solving mathematical problems
- Data processing
- Computer programs & Real-life problem solving

INTRODUCTION TO PYTHON

A Quick Summary Guide for Beginners

1. PYTHON INTRODUCTION

- Python is a high-level, interpreted, general-purpose programming language.
- Created by Guido van Rossum and first released in 1991.
- Known for its simple syntax, readability and versatility.
- Widely used in Web Development, Data Science, AI/ML, Automation, Scripting and more.

python™

Simple • Readable • Powerful

2. TECHNICAL STRENGTH OF PYTHON

Easy to Learn and Use
Simple syntax similar to English.

Interpreted Language
No compilation needed. Code runs line by line.

High-level Language
Focus on solving problems, not low-level details.

Cross-platform
Runs on Windows, macOS, Linux, Unix, etc.

Extensive Standard Library
Large collection of built-in modules and functions.

Open Source
Free to use and has a large community.

Object-Oriented
Supports OOP concepts fully.

Large Community Support
Lots of tutorials, libraries and frameworks.

3. PYTHON INTERPRETER & PROGRAM EXECUTION

Python Interpreter

- The interpreter reads your code, interprets it and executes it line by line.
- No separate compilation step.
- Errors are shown immediately.

Program Execution Process

Write Code (.py file) → Run with Interpreter → Interpreter Reads & Executes Line by Line → Output / Errors Displayed

Run Program

```
python filename.py   # Run from terminal/command prompt
 or
Run ▶ in IDE (like IDLE, VS Code, PyCharm, etc.)
```

4. USING COMMENTS

Comments are used to explain code. They are ignored by the interpreter.

Single-line Comment

```
# This is a single-line comment
```

Multi-line Comment

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

5. LITERALS & CONSTANTS

Literals

Fixed values given in a program.

```
10      # Integer literal
3.14    # Float literal
2 + 3j  # Complex literal
'Hello' # String literal
True    # Boolean literal
None    # None literal
```

Constants

Python has no built-in keyword for constants, but by convention we use UPPER_CASE names.

```
PI = 3.14159
GRAVITY = 9.8
```

6. PYTHON'S BUILT-IN DATA TYPES

Category	Data Types	Description	Example
Numeric	int	Integer numbers	10, -5, 1000
	float	Decimal numbers	3.14, -0.001
	complex	Complex numbers	2+3j, -1.2j
Sequence	str	String (text)	'hello', "Python"
	list	Ordered, mutable list	[1, 2, 3]
	tuple	Ordered, immutable	(1, 2, 3)
	range	Sequence of numbers	range(5)
Set Types	set	Unordered, unique items	{1, 2, 3}
	frozenset	Immutable set	frozenset({1,2})
Mapping	dict	Key-value pairs	{'a': 1, 'b': 2}
Boolean	bool	True or False	True, False
Special	NoneType	Represents None	None

7. NUMBERS

Integers (int)
Whole numbers (positive, negative or zero)
`x = 25, y = -10, z = 0`

Floats (float)
Decimal numbers
`a = 3.14, b = -0.001`

Complex Numbers (complex)
Numbers of the form a + bj
`c = 2 + 3j, d = -1.5j`

Real Numbers
All rational and irrational numbers (int, float)
`e = 5, f = -2.7`

Sets (set)
Unordered collection of unique items
`s = {1, 2, 3, 4}`

8. STRINGS

Slicing & Indexing

Indexing (starts at 0)

```
s = "Python"
```

Index	0	1	2	3	4	5
Char	P	y	t	h	o	n

- `s[0]` # 'P' (first character)
- `s[5]` # 'n' (last character)

Slicing

```
s[1:4]  # 'yth' (start inclusive, end exclusive)
s[:3]   # 'Pyt' (start to end-1)
s[3:]   # 'hon' (start to last)
s[-3:]  # 'hon' (last 3 chars)
```

Concatenation

```
a = "Hello"
b = "World"
c = a + " " + b   # 'Hello World'
```

Other String Operations

```
len(s)              # Length of string
s.upper()           # To Uppercase
s.lower()           # To Lowercase
s.replace('o', 'a') # Replace
s.split(',',)       # Split into list
s.strip()           # Remove spaces
'P' in s            # Membership test
s.startswith('Py')  # Starts with
s.endswith('on')    # Ends with
```

9. ACCEPTING INPUT FROM CONSOLE

input() function is used to take input from the user.

```
name = input("Enter your name: ")
age = int(input("Enter your age: "))
print("Hello", name)
print("Next year, you will be", age + 1)
```

Note: input() always returns a string. Use int(), float(), etc. to convert.

10. PRINTING STATEMENTS

print() function is used to display output.

```
print("Hello, Python!")
print(10)
print(3.14)
print("Name:", "Alice")
print("Sum =", 5 + 7)
print("A", "B", "C", sep='-')
print("Line1\nLine2")  # New line
```

11. SIMPLE PYTHON PROGRAMS (EXAMPLES)

1. Hello World

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

2. Add Two Numbers

```
a = 10
b = 20
sum = a + b      # Float (acata)
print("Sum:", sum)
```

2. Add Two Numbers

```
import math
r = float(input("Radius: "))
area = math.pi * r * r
```

4. Swap Two Numbers

```
a = 5
b = 10
a, b = b, a      # Pythonic way
print("a =", a, "b =", b)
```

5. Celsius to Fahrenheit

```
c = float(input("Celsius: "))
f = (c * 9/5) + 32
print("Fahrenheit:", f)
```

6. Check Even or Odd

```
n = int(input("Enter number: "))
if n % 2 == 0:
    print("Even")
else:
    print("Odd")
```

12. KEY POINTS

- Python is easy to learn and has a clean syntax.
- It is an interpreted and high-level language.
- Comments make code readable and maintainable.
- Python has rich data types for different needs.
- Strings are powerful and easy to manipulate.
- Input and output are simple with input() and print().
- Python is great for beginners and professionals alike.

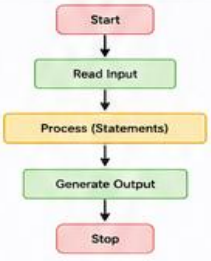
13. COMMON ESCAPE SEQUENCES IN STRINGS

Escape Sequence	Meaning
\n	New Line
\t	Tab
\'	Single Quote
\"	Double Quote
\\	Backslash
\r	Carriage Return

14. COMMON OPERATORS

Arithmetic		Comparison		Logical	
+	Addition	==	Equal	and	Logical AND
-	Subtraction	!=	Not Equal	or	Logical OR
*	Multiplication	>	Greater Than	not	Logical NOT
/	Division	<	Less Than		
%	Modulus	>=	Greater or Equal		
**	Power	<=	Less or Equal		
//	Floor Division				

15. PYTHON PROGRAM STRUCTURE



Python – The language that makes coding simple and powerful!

INTRODUCTION TO PYTHON

A Quick Summary Guide for Beginners

1. PYTHON INTRODUCTION

- Python is a high-level, interpreted, general-purpose programming language.
- Created by Guido van Rossum and first released in 1991.
- Known for its simple syntax, readability and versatility.
- Widely used in Web Development, Data Science, AI/ML, Automation, Scripting and more.

python™

Simple • Readable • Powerful

2. TECHNICAL STRENGTH OF PYTHON

Easy to Learn and Use
Simple syntax similar to English.

Interpreted Language
No compilation needed. Code runs line by line.

High-level Language
Focus on solving problems, not low-level details.

Cross-platform
Runs on Windows, macOS, Linux, Unix, etc.

Extensive Standard Library
Large collection of built-in modules and functions.

Open Source
Free to use and has a large community.

Object-Oriented
Supports OOP concepts fully.

Large Community Support
Lots of tutorials, libraries and frameworks.

3. PYTHON INTERPRETER & PROGRAM EXECUTION

Python Interpreter

- The interpreter reads your code, interprets it and executes it line by line.
- No separate compilation step.
- Errors are shown immediately.

Program Execution Process

Write Code (.py file) → Run with Interpreter → Interpreter Reads & Executes Line by Line → Output / Errors Displayed

Run Program

```
python filename.py   # Run from terminal/command prompt
 or
Run ▶ in IDE (like IDLE, VS Code, PyCharm, etc.)
```

4. USING COMMENTS

Comments are used to explain code. They are ignored by the interpreter.

Single-line Comment

```
# This is a single-line comment
```

Multi-line Comment

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

5. LITERALS & CONSTANTS

Literals

Fixed values given in a program.

```
10      # Integer literal
3.14    # Float literal
2 + 3j  # Complex literal
'Hello' # String literal
True    # Boolean literal
None    # None literal
```

Constants

Python has no built-in keyword for constants, but by convention we use UPPER_CASE names.

```
PI = 3.14159
GRAVITY = 9.8
```

6. PYTHON'S BUILT-IN DATA TYPES

Category	Data Types	Description	Example
Numeric	int	Integer numbers	10, -5, 1000
	float	Decimal numbers	3.14, -0.001
	complex	Complex numbers	2+3j, -1.2j
Sequence	str	String (text)	'hello', "Python"
	list	Ordered, mutable list	[1, 2, 3]
	tuple	Ordered, immutable	(1, 2, 3)
	range	Sequence of numbers	range(5)
Set Types	set	Unordered, unique items	{1, 2, 3}
	frozenset	Immutable set	frozenset({1,2})
Mapping	dict	Key-value pairs	{'a': 1, 'b': 2}
Boolean	bool	True or False	True, False
Special	NoneType	Represents None	None

7. NUMBERS

Integers (int)
Whole numbers (positive, negative or zero)
`x = 25, y = -10, z = 0`

Floats (float)
Decimal numbers
`a = 3.14, b = -0.001`

Complex Numbers (complex)
Numbers of the form a + bj
`c = 2 + 3j, d = -1.5j`

Real Numbers
All rational and irrational numbers (int, float)
`e = 5, f = -2.7`

Sets (set)
Unordered collection of unique items
`s = {1, 2, 3, 4}`

8. STRINGS

Slicing & Indexing

Indexing (starts at 0)

```
s = "Python"
```

Index	0	1	2	3	4	5
Char	P	y	t	h	o	n

- `s[0]` # 'P' (first character)
- `s[5]` # 'n' (last character)

Slicing

```
s[1:4]  # 'yth' (start inclusive, end exclusive)
s[:3]   # 'Pyt' (start to end-1)
s[3:]   # 'hon' (start to last)
s[-3:]  # 'hon' (last 3 chars)
```

Concatenation

```
a = "Hello"
b = "World"
c = a + " " + b   # 'Hello World'
```

Other String Operations

```
len(s)              # Length of string
s.upper()           # To Uppercase
s.lower()           # To Lowercase
s.replace('o', 'a') # Replace
s.split(',')        # Split into list
s.strip()           # Remove spaces
'P' in s            # Membership test
s.startswith('Py')  # Starts with
s.endswith('on')    # Ends with
```

9. ACCEPTING INPUT FROM CONSOLE

input() function is used to take input from the user.

```
name = input("Enter your name: ")
age = int(input("Enter your age: "))
print("Hello", name)
print("Next year, you will be", age + 1)
```

Note: input() always returns a string. Use int(), float(), etc. to convert.

10. PRINTING STATEMENTS

print() function is used to display output.

```
print("Hello, Python!")
print(10)
print(3.14)
print("Name:", "Alice")
print("Sum =", 5 + 7)
print("A", "B", "C", sep='-')
print("Line1\nLine2")  # New line
```

11. SIMPLE PYTHON PROGRAMS (EXAMPLES)

1. Hello World

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

2. Add Two Numbers

```
a = 10
b = 20
sum = a + b      # Float Locate)
print("Sum:", sum)
```

2. Add Two Numbers

```
import math
r = float(input("Radius: "))
area = math.pi * r * r
```

4. Swap Two Numbers

```
a = 5
b = 10
a, b = b, a      # Pythonic way
print("a =", a, "b =", b)
```

5. Celsius to Fahrenheit

```
c = float(input("Celsius: "))
f = (c * 9/5) + 32
print("Fahrenheit:", f)
```

6. Check Even or Odd

```
n = int(input("Enter number: "))
if n % 2 == 0:
    print("Even")
else:
    print("Odd")
```

12. KEY POINTS

- Python is easy to learn and has a clean syntax.
- It is an interpreted and high-level language.
- Comments make code readable and maintainable.
- Python has rich data types for different needs.
- Strings are powerful and easy to manipulate.
- Input and output are simple with input() and print().
- Python is great for beginners and professionals alike.

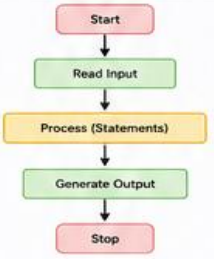
13. COMMON ESCAPE SEQUENCES IN STRINGS

Escape Sequence	Meaning
\n	New Line
\t	Tab
\'	Single Quote
\"	Double Quote
\\	Backslash
\r	Carriage Return

14. COMMON OPERATORS

Arithmetic		Comparison		Logical	
+	Addition	==	Equal	and	Logical AND
-	Subtraction	!=	Not Equal	or	Logical OR
*	Multiplication	>	Greater Than	not	Logical NOT
/	Division	<	Less Than		
%	Modulus	>=	Greater or Equal		
**	Power	<=	Less or Equal		
//	Floor Division				

15. PYTHON PROGRAM STRUCTURE



Python – The language that makes coding simple and powerful!

OPERATORS, EXPRESSIONS & PYTHON STATEMENTS

A Quick Summary Guide

1. ASSIGNMENT STATEMENT

Used to assign a value to a variable.

Syntax:

```
variable = expression
```

Examples:

```
x = 10
name = "Python"
a, b = 5, 10
x += 5      # x = x + 5
x -= 3      # x = x - 3
x *= 2      # x = x * 2
x /= 4      # x = x / 4
x //= 2     # x = x // 2
x %= 3      # x = x % 3
x **= 2     # x = x ** 2
```

2. EXPRESSIONS

Combination of values, variables and operators that evaluates to a single value.

Examples:

```
a + b

(a * b) / c

x ** 2 + 3 * x - 5

(x > 0) and (y < 10)
```

3. OPERATORS IN PYTHON

A. ARITHMETIC OPERATORS

Operator	Meaning	Example	Result
+	Addition	5 + 3	8
-	Subtraction	5 - 3	2
*	Multiplication	5 * 3	15
/	Division	5 / 2	2.5
//	Floor Division	5 // 2	2
%	Modulus	5 % 2	1
**	Exponentiation	2 ** 3	8

B. RELATIONAL (COMPARISON) OPERATORS

Operator	Meaning	Example	Result
==	Equal	5 == 5	True
!=	Not Equal	5 != 3	True
>	Greater Than	5 > 3	True
<	Less Than	5 < 3	False
>=	Greater Than or Equal	5 >= 5	True
<=	Less Than or Equal	3 <= 2	False

C. LOGICAL OPERATORS

Operator	Meaning	Example	Result
and	Logical AND	(5>3) and (2<4)	True
or	Logical OR	(5<3) or (2<4)	True
not	Logical NOT	not (5>3)	False

D. BITWISE OPERATORS

Operator	Meaning	Example (a = 5, b = 3)		Result
&	AND	a & b	1	(0101 & 0011 = 0001)
|	OR	a | b	7	(0101 | 0011 = 0111)
^	XOR	a ^ b	6	(0101 ^ 0011 = 0110)
~	NOT	~a	-6	(~0101 = ...1010)
<<	Left Shift	a << 1	10	(0101 << 1 = 1010)
>>	Right Shift	a >> 1	2	(0101 >> 1 = 0010)

4. OPERATOR PRECEDENCE

Precedence	Operators	Description
1	( )	Parentheses
2	**	Exponentiation
3	+x, -x, ~x	Unary plus, minus, bitwise NOT
4	*, /, //, %	Multiplication, Division, Floor Div, Modulus
5	+, -	Addition, Subtraction
6	<<, >>	Bitwise Shifts
7	&	Bitwise AND
8	^	Bitwise XOR
9	|	Bitwise OR
10	==, !=, >, <, >=, <=	Comparisons
11	not	Logical NOT
12	and	Logical AND
13	or	Logical OR
14	=, +=, -=, *=, /=, %=, /*=, &=, |=, ^=, <<=, >>=	Assignment

5. CONDITIONAL STATEMENTS

A. if STATEMENT

```
if condition:
    # block of code
# executes if condition is True
```

Example:

```
x = 10
if x > 5:
    print("x is greater than 5")
```

B. if-else STATEMENT

```
if condition:
    # block if True
else:
    # block if False
```

Example:

```
x = 3
if x % 2 == 0:
    print("Even")
else:
    print("Odd")
```

C. if-elif-else STATEMENT

```
if condition1:
    # block 1
elif condition2:
    # block 2
else:
    # block 3
```

Example:

```
marks = 75
if marks >= 90:
    grade = 'A'
elif marks >= 60:
    grade = 'B'
else:
    grade = 'C'
print(grade)
```

6. SIMPLE PROGRAM EXAMPLES

1. Check Positive, Negative or Zero

```
num = float(input("Enter a number: "))
if num > 0:
    print("Positive")
elif num < 0:
    print( "Negative")
else:
    print("Zero")
```

2. Find Largest of Two Numbers

```
a = int(input("Enter first number: "))
b = int(input("Enter second number: "))
if a > b:
    print(a, "is largest")
elif b > a:
    print(b, "is largest")
else:
    print("Both are equal")
```

7. ITERATIVE COMPUTATION & CONTROL FLOW

Used to repeat a block of code multiple times.

A. range() FUNCTION

Generates a sequence of numbers.

Syntax:

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

Examples:

```
range(5)          # 0 to 4
range(1, 6)       # 1 to 5
range(2, 10, 2)   # 2, 4, 6, 8
```

B. while STATEMENT

```
while condition:
    # block of code
else:
    # executes when condition
    # becomes False
```

Example:

```
i = 1
while i <= 5:
    print(i, end=' ')
    i += 1
else:
    print("\nDone!")
```

C. for LOOP

```
for variable in sequence:
    # block of code
else:
    # executes after loop
    # completes normally
```

Example:

```
for i in range(1, 6):
    print(i, end=' ')
else:
    print("\nLoop finished")
```

8. LOOP CONTROL STATEMENTS

break : Terminates the nearest loop.

```
for i in range(5):
    if i == 3:
        break          # loop stops when i == 3
```

continue : Skips the rest of the current iteration.

```
for i in range(5):
    if i == 2:
        continue       # skips printing 2
    print(i)
```

pass : Does nothing. Used as a placeholder.

```
for i in range(3):
    pass               # to be implemented later
```

9. else WITH LOOPS

The else block runs when the loop completes without a break.

Example:

```
for i in range(3):
    print(i)
else:
    print("Loop completed successfully")
```

10. assert STATEMENT

Used for debugging. Raises AssertionError if condition is False.

Syntax:

```
assert condition, "message"
```

Example:

```
x = 10
assert x > 0, "x must be positive"
print("OK")
```

11. pass STATEMENT

Used as a placeholder where syntax requires a statement.

Example:

```
def my_function():
    pass   # code to be written later

class MyClass:
    pass
```

12. SUMMARY TABLE

Topic	Key Points
Operators	Arithmetic, Relational, Logical, Bitwise with precedence.
Expressions	Combine values/operators → single value.
Conditional	if, if-else, if-elif-else for decisions.
Loops	while and for for repetition.
Control Statements	break, continue, pass, else.
Others	range() for sequences, assert for debugging.

QUICK REFERENCE

- Use operators to perform operations.
- Use expressions to compute values.
- Use conditional statements for decision making.
- Use loops for repetition.
- Use control statements to change loop flow.
- Use else with loops and assert for better programs.

REMEMBER

- ✓ Indentation is required in Python.
- ✓ Blocks are defined by indentation, not braces.
- ✓ Practice writing small programs to master concepts.

SEQUENCE DATA TYPES IN PYTHON

Lists, Tuples & Dictionary – Operations, Mutability & Examples

1. WHAT ARE SEQUENCE DATA TYPES?

A **sequence** is an ordered collection of items. In Python, the main sequence data types are:

- List – mutable, uses [ ]
- Tuple – immutable, uses ( )
- Dictionary – mutable, key-value pairs { }

All support indexing, slicing, iteration, membership test, and more.

Ordered | Iterable | Has Length

2. MUTABILITY CONCEPT

Type	Mutable?	Can Change?	Example
List	Yes	Add/Remove/Change items	[1, 2, 3]
Tuple	No	Cannot change after creation	(1, 2, 3)
Dictionary	Yes	Add/Remove/Change key-value pairs	{'a': 1, 'b': 2}

3. BASIC OPERATIONS COMPARISON

Operation	List	Tuple	Dictionary
Indexing	✓	✓	Keys only
Slicing	✓	✓	✗
Concatenation	✓	✓	⚠ (via update/merge)
Repetition (*)	✓	✓	✗
Membership (in)	✓	✓	Keys only
Length (len())	✓	✓	✓
Change Item	✓	✗	✓ (by key)
Add Item	✓ (append/insert)	✗	✓ (add key:value)
Remove Item	✓ (remove/pop)	✗	✓ (pop by key)

4. INDEXING & SLICING

Indexing (0-based)

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

Index	0	1	2	3	4
Value	10	20	30	40	50

- seq[0] -> 10 (first item)
- seq[4] -> 50 (last item)
- seq[-1] -> 50 (last item)
- seq[-2] -> 40

Slicing (end is exclusive)

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

- seq[1:4] -> [20, 30, 40]
- seq[:3] -> [10, 20, 30]
- seq[2:] -> [30, 40, 50]
- seq[::2] -> [10, 30, 50]
- seq[::-1] -> [50, 40, 30, 20, 10]

5. CONCATENATION & REPETITION

Concatenation (+)

```
List
  [1, 2] + [3, 4]
  -> [1, 2, 3, 4]

Tuple
  (1, 2) + (3, 4)

Dict
{'a':1} | {'b':2}
-> {'a':1,'b':2}
   # Python 3.9+
  (or use {**d1, **d2})
```

Repetition (*)

```
[1, 2] * 3
  (1, 2, 1, 2, 1, 2]

(1, 2] * 3
  (1, 2, 1, 2, 1, 2)
```

6. OTHER COMMON OPERATIONS

Operation	List	Tuple	Dictionary
len(x)	len(x)	len(x)	
min(x) / max(x)	min(x) (if numbers/ comparable)	min(x) (if numbers)	min(x.keys()) (for keys)
x.count(v)	x.count(v)		N/A
x.index(v)	x.index(v)		N/A
x.append(v)		✗	N/A
x.extend(iter)		✗	N/A
x.pop(i)		✗	N/A
x.remove(v)		✗	N/A
d.get(k, default)	N/A		d.get(k, default)
k in x		k in x	k in x (keys)

7. LIST EXAMPLES

```
L = [5, 2, 9, 1, 6]
```

- **Indexing:** L[2] # 9
- **Slicing:** L[1:4] # [2, 9, 1]
- **Concatenation:** L + [7, 8] # [5,2,9,1,6,7,8]
- **Repetition:** L * 2 # [5,2,9,1,6,5,2,9,1,6]
- **Append:** L.append(10)
- **Remove:** L.remove(2)
- **Length:** len(L) # 5

8. TUPLE EXAMPLES

```
T = (5, 2, 9, 1, 6)
```

- **Indexing:** T[0] # 5
- **Slicing:** T[1:4] # (2, 9, 1)
- **Concatenation:** T + (7, 8) # (5,2,9,1,6,7,8]
- **Repetition:** T * 2 # [5,2,9,1,6,5,2,9,1,6]
- **Length:** len(T) # 5

Tuples are immutable – we cannot change, add or remove items.

9. DICTIONARY EXAMPLES

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

- **Access:** D['b'] # 2
- **Add/Update:** D['d'] = 4
- **Update:** D['a'] = 10
- **Remove:** D.pop('c') # removes 'c'
- **Keys:** D.keys() # dict_keys(['a','b','d']
- **Values:** D.values() # dict_values([10,2,4])
- **Items:** D.items() # dict_items([('a',10),('b',2),('d',4)])
- **Length:** len(D) # 3

10. PRACTICAL PROBLEMS & SOLUTIONS

A. Find Maximum, Minimum & Mean

```
nums = [4, 7, 2, 9, 5, 1]

maximum = max(nums)
minimum = min(nums)
mean = sum(nums) / len(nums)

print("Max:", maximum)
print("Min:", minimum)
print("Mean:", mean)
```

Output:
Max: 9
Min: 1
Mean: 4.67

B. Linear Search in List

```
def linear_search(seq, key):
    for i, val in enumerate(seq):
        if val == key:
            return i      # found at index i
    return -1             # not found

nums = [10, 20, 30, 40, 50]
key = 30
index = linear_search(nums, key)
print(index)     # Output: 2
```

Returns index of key if found, otherwise -1.

C. Linear Search in Tuple

```
def linear_search_tuple(tup, key):
    for i, val in enumerate(tup):
        if val == key:
            return i
    return -1

t = (11, 22, 33, 44, 55)
key = 44
index = linear_search_tuple(t, key)
print(index)     # Output: 3
```

Returns index of key if found, otherwise -1.

D. Count Frequency of Elements in a List Using Dictionary

```
def count_frequency(lst):
    freq = {}
    for item in lst:
        if item in freq:
            freq[item] += 1
        else:
            freq[item] = 1
    return freq

nums = [4, 1, 2, 4, 2, 2, 3, 1, 2]
print(count_frequency(nums))
# Output: {4: 3, 1: 2, 2: 3, 3: 1}
```

Dictionary keys are unique elements and values are their frequencies.

11. METHODS SUMMARY

List Methods

append(x)	– Add item at end
extend(iter)	– Extend list
insert(i, x)	– Insert at index i
remove(x)	– Remove first occ. of x
pop([i])	– Remove & return item
clear()	– Remove all items
index(x)	– Return first index of x
count(x)	– Count occurrences of x
sort()	– Sort list
reverse()	– Reverse list

Tuple Methods

count(x)	– Count occurrences
index(x)	– Return first index

Tuples have limited methods because they are immutable.

Dictionary Methods

keys()	– All keys
values()	– All values
items()	– All (key, value) pairs
get(k, d)	– Get value for key k
update(d)	– Update with dict d
pop(k[, d])	– Remove key k
clear()	– Remove all items
len(d)	– Number of items

12. KEY TAKEAWAYS

- ✓ Lists are mutable, tuples are immutable, dictionaries store key-value pairs.
- ✓ All sequence types support indexing, slicing, length and iteration.
- ✓ Use lists for dynamic collections, tuples for fixed data, dictionaries for mappings.
- ✓ Common operations: indexing, slicing, concatenation, repetition, membership test.
- ✓ Built-in functions like max(), min(), sum(), len() are very useful.
- ✓ Dictionary is powerful for counting, grouping and fast lookups.

TIPS

- Use lists when you need to modify data.
- Use tuples when data should not change (safe & faster).
- Use dictionaries when you need to store and retrieve data by keys quickly.

PYTHON POWER

Simple to learn, powerful to build – Python makes data handling easy!

FUNCTIONS IN PYTHON

Build Reusable Code • Solve Smart • Think Modular

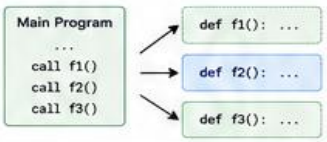
1. TOP-DOWN APPROACH Problem Solving

- Understand the Problem
- ↓
- Break into Sub-problems
- ↓
- Design Functions for each task
- ↓
- Implement & Test individually
- ↓
- Integrate & Validate Solution

2. MODULAR PROGRAMMING AND FUNCTIONS

Break a large program into small, independent functions.

- ✔ Improves readability
- ✔ Code reusability
- ✔ Easy testing & debugging
- ✔ Simpler maintenance
- ✔ Supports teamwork



3. DEFINING A FUNCTION

```
def function_name(parameters):
    """Docstring (optional)"""
    # body of function
    statements
    return value   # optional
```

Function name should be descriptive and follow PEP 8 naming rules.

4. FUNCTION PARAMETERS

- Positional Arguments (required order)
- Default Arguments
- Keyword Arguments
- VarArgs (*args) – non-keyword variable length
- KwArgs (**kwargs) – keyword variable length

```
def demo(a, b, c=10, *args, **kwargs):
    print(a, b, c)
    print(args)       # tuple of extra args
    print(kwargs)     # dict of keyword args

demo(1, 2)
demo(1, 2, 30, 40, 50, x=100, y=200)
```

5. LOCAL VARIABLES

- Variables defined inside a function are local to that function.
- They exist only during the function call and are destroyed afterward.

```
def func():
    x = 10   # local variable
    print(x)

func()
# print(x)  -> NameError
```

6. RETURN STATEMENT

- return exits a function and sends back a value.
- If no return, function returns None.

```
def add(a, b):
    sum = a + b
    return sum

res = add(5, 3)
print(res)    # 8
```

7. DOC STRINGS

- Triple-quoted string right below the function definition.

```
def square(x):
    """
    Returns the square of a number.
    Parameter:
        x (int/float): number
    Returns:
        x*x
    """
    return x * x

help(square)
```

8. GLOBAL STATEMENT

- Use global to modify a global variable inside a function.

```
count = 0
def inc():
    global count
    count += 1
inc()
print(count)    # 1
```

9. DEFAULT ARGUMENT VALUES

- Provide default value if no argument is passed.

```
def greet(name, msg="Hello"):
    print(msg, name)

greet("Alice")
greet("Bob", "Hi")
```

10. KEYWORD ARGUMENTS

- Specify arguments using parameter names.

```
def info(name, age):
    print(name, age)

info(age=25, name="John")
```

11. VARARGS PARAMETERS

***args (non-keyword)**

Receives extra positional arguments as a tuple.

```
def total(*nums):
    s = 0
    for n in nums:
        s += n
    return s

total(1,2,3,4)   # 10
```

****kwargs (keyword)**

Receives extra keyword arguments as a dict.

```
def show(**info):
    for k, v in info.items():
        print(k, ":", v)

show(name="Sam", age=20)
```

12. LIBRARY FUNCTIONS (Built-in)

- input(prompt) – read input
- eval(expression) – evaluate expression
- print(*values) – display output

```
name = input("Enter name: ")
expr = "2 + 3 * 4"
print(eval(expr))   # 14
```

13. STRING FUNCTIONS (str)

count(sub)	Count occurrences of sub string	istitle()	Check string is title case	isalpha()	All characters alphabetic
find(sub)	First index of sub string (not found: -1)	replace(old, new[, count])	Replace substring	isdigit()	All characters digits
rfind(sub)	Last index of sub string	strip()	Remove leading & trailing spaces	isalnum()	Alphanumeric check
capitalize()	First char uppercase	lstrip()	Remove leading spaces	starswith(s)	Starts with substring
title()	Each word capitalized	rstrip()	Remove trailing spaces	endswith(s)	Ends with substring
lower()	Convert to lowercase	split(sep)	Split into list by sep	encode(enc)	Encode string to bytes
upper()	Convert to uppercase	partition(sep)	Split into (before, sep, after)	decode(enc)	Decode bytes to string
swapcase()	Swap case	join(iterable)	Join iterable of strings		
islower()	Check all cased chars are lowercase	isspace()	Check all whitespace		
isupper()	Check all cased chars are uppercase				

STRING SLICING

```
s = "Python Programming"
Index:  0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
        P y t h o n   P r o g r a m m i n g

s[0:6]   -> "Python"
s[7:]    -> "Programming"
s[:6]    -> "Python"
s[-11:]  -> "Programming"
s[::2]   -> "Pto rgamn"
```

14. MEMBERSHIP OPERATORS

in, not in with sequences

```
s = "hello"
'a' in s        # True
'z' not in s    # True
```

15. PATTERN MATCHING

(Using startswith, endswith, find, etc.)

```
s = "report.pdf"
s.endswith(".pdf")    # True
s.startswith("rep")   # True
```

16. NUMERIC FUNCTIONS (Built-in)

eval(x)	Evaluate expression	random()	Random float [0.0, 1.0)
max(iterable)	Maximum value	randint(a, b)	Random int in [a, b]
min(iterable)	Minimum value	ceil(x)	Ceiling value
pow(x, y)	x raised to power y	floor(x)	Floor value
round(x, n=0)	Round to n digits	sqrt(x)	Square root
int(x)	Convert to integer		

17. DATE & TIME FUNCTIONS

```
import datetime

now = datetime.datetime.now()
print(now)
print(now.strftime("%Y-%m-%d %H:%M:%S"))
```

18. RECURSION

- A function calling itself to solve smaller sub-problems.
- Must have a base case to stop.

Example: Factorial

```
def fact(n):
    if n == 0:
        return 1   # base case
    else:
        return n * fact(n-1)

print(fact(5))   # 120
```

Call Stack for fact(3)

```
fact(3)
↳ 3 * fact(2)
  ↳ 2 * fact(1)
    ↳ 1 * fact(0)
      ↳ 1
```

19. PRACTICAL EXAMPLES

A. Max, Min & Mean

```
nums = [4, 7, 2, 9, 5]
mx = max(nums)
mn = min(nums)
mean = sum(nums)/len(nums)
print(mx, mn, mean)

# 9 2 5.4
```

B. Linear Search (List)

```
def linear_search(lst, key):
    for i, val in enumerate(lst):
        if val == key:
            return i
    return -1

print(linear_search([10,20,30,40], 30))  2
```

C. Linear Search (Tuple)

```
def linear_search_tup(tup, key):
    for i, val in enumerate(tup):
        if val == key:
            return i
    return -1

print(linear_search_tup((5,8,3,9), 3))  2
```

D. Frequency of Elements (using Dictionary)

```
def freq(lst):
    d = {}
    for x in lst:
        d[x] = d.get(x, 0) + 1
    return d

print(freq([1,2,2,3,1,4,2,3,1]))
# {1: 3, 2: 3, 3: 2, 4: 1}
```

20. KEY TAKEAWAYS

- ✔ Functions make code modular, reusable and easy to maintain.
- ✔ Understand parameters, return values and variable scope.
- ✔ Use built-in functions and string utilities to save time.
- ✔ Recursion is powerful but use with care (watch base case!).
- ✔ Practice small programs to master functions.

QUICK REFERENCE CHEAT SHEET

Parameter Types	Scope	Common Built-ins
• Positional • Default • Keyword • *args (tuple) • **kwargs (dict)	• Local – inside function • Global – outside function (use global to modify)	input(), print(), eval(), len(), max(), min(), sum(), int(), float(), str(), type(), range()

DOCSTRING EXAMPLE

```
def add(a, b):
    """
    Add two numbers.
    Args:
        a (int): first number
        b (int): second number
    Returns:
        int: sum of a and b
    """
    return a + b
```

</> Think Modular. Code Reusable. Program Powerful.

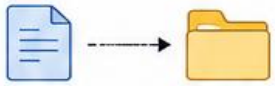
☆ Functions are the building blocks of great Python programs!

FILE PROCESSING IN PYTHON

Work with Files • Read, Write, Manage • Command Line Arguments

1. CONCEPT OF FILES

- A file is a named location on disk used to store data permanently.
- Python allows programs to create, read, update and delete files.
- Data in a file is a sequence of bytes. In text mode, Python handles encoding/decoding.
- Types of Files
  - Text Files .txt, .py, .csv, ...)
  - Binary Files (.jpg, .dat, .bin, ...)



2. OPENING FILES – MODES

Use open() to open a file. It returns a file object.

Syntax:

```
open(file, mode, encoding)
```

Mode	Description	File Must Exist?
'r'	Read (default)	Yes
'w'	Write (overwrite)	No (creates new)
'a'	Append (add at end)	No (creates new)
'x'	Create (fail if exists)	No
'b'	Binary mode	N/A
't'	Text mode (default)	N/A
'+'	Read and Write	No (creates new)

Modes can be combined, e.g. 'r+', 'w+', 'a+'
Binary modes: 'rb', 'wb', 'ab', 'rb+', etc.

3. OPENING AND CLOSING FILES

Opening:

```
f = open('data.txt', 'r')
```

Closing:

```
f.close()
```

- Always close the file after use to free resources and ensure data is saved.

Best Practice – use with statement

```
with open('data.txt', 'r') as f:
    data = f.read()
# file is automatically closed
```

Check if a file is closed:

```
f.closed  # True or False
```

4. FILE FUNCTIONS OVERVIEW

Function	Purpose
open()	Open a file
close()	Close a file
read()	Read entire content
readline()	Read one line
readlines()	Read all lines into list
write(str)	Write string to file
writelines(list)	Write list of strings
tell()	Current file position
seek(offset, whence)	Move to a position

Always open a file before using it and close it when done.

5. READING FROM A FILE

Read entire content

```
with open('data.txt', 'r') as f:
    content = f.read()
    print(content)
```

Read one line at a time

```
with open('data.txt', 'r') as f:
    line = f.readline()
    while line:
        print(line.strip())
        line = f.readline()
```

Read all lines into a list

```
with open('data.txt', 'r') as f:
    lines = f.readlines()
    for line in lines:
        print(line.strip())
```

Use .strip() to remove newline characters.

6. WRITING ONTO A FILE

Write (overwrite) – creates or replaces file

```
with open('out.txt', 'w') as f:
    f.write('Hello World!\n')
    f.write('Python File Handling\n')
```

Append – adds at end of file

```
with open('out.txt', 'a') as f:
    f.write('New line appended\n')
```

Write multiple lines

```
lines = ['Line 1\n', 'Line 2\n', 'Line 3\n']
with open('out.txt', 'w') as f:
    f.writelines(lines)
```

write() writes a single string.
writelines() writes a list/sequence of strings.

7. FILE FUNCTIONS DETAILS

Function	Description	Example
open()	Open file and return file object	f = open('a.txt','r')
close()	Close the file	f.close()
read(size=-1)	Read entire file (or size chars)	f.read()
readline()	Read one line	f.readline()
readlines()	Read all lines into list	f.readlines()
write(s)	Write string s to file	f.write('Hello')
writelines(seq)	Write list of strings	f.writelines(['a\n','b\n'])
tell()	Current cursor position	pos = f.tell()

seek(offset, whence)	Move cursor	f. seek(10)
		# whence:
		0 = start, 1 = current, 2 = end

Default cursor is at start (position 0).
seek() changes position; next read/write happens there.
tell() returns current position.

8. EXAMPLE PROGRAMS

1. Copy a File (Text)

```
with open('source.txt', 'r') as src,
     open('copy.txt', 'w') as dst:
    for line in src:
        dst.write(line)
```

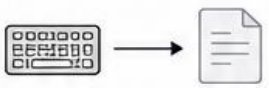
2. Count Number of Lines, Words, Characters

```
with open('data.txt', 'r') as f:
    lines = f.readlines()
lines_count = len(lines)
words_count = sum(len(line.split())
                  for line in lines)
chars_count = sum(len(line)
                  for line in lines)
print(lines_count, words_count, chars_count)
```

Output Example:
10 45 256

3. Append User Input to a File

```
text = input('Enter text: ')
with open('notes.txt', 'a') as f:
    f.write(text + '\n')
```



4. Read First N Characters

```
n = 20
with open('data.txt', 'r') as f:
    data = f.read(n)
    print(data)
```

Output:
This is the first 20

9. FILE POSITION EXAMPLE

File Content (data.txt)

```
Hello\n
Python\n
File Handling\n.
Is Easy\n
```

Code

```
f = open('data.txt', 'r')
print(f.tell())    # 0
print(f.readline(), end='')   # Hello
print(f.tell())
f.seek(0)
print(f.readline(), end='')   # Hello
f.seek(6)
print(f.readline(), end='')   # Python
f.close()
```

10. COMMAND LINE ARGUMENTS

sys module provides access to command line arguments.

sys.argv is a list where:
argv[0] is script name
argv[1]..argv[n] are arguments

```
# save as: showargs.py
import sys
print('Script name:', sys.argv[0])
print('Arguments:', sys.argv[1:])
```

```
$ python showargs.py Alice 25 test
Script name: showargs.py
Arguments: ['Alice', '25', 'test']
```

11. SUMMARY

- ✔ Use open() to open files in proper mode.
- ✔ Always close files using close() or with.
- ✔ Read using read(), readline(), readlines().
- ✔ Write using write(), writelines().
- ✔ Use tell() and seek() to control file position.
- ✔ Use sys.argv to get command line arguments.

KEY TAKEAWAYS

- Files allow permanent storage and retrieval of data.
- Choose the right mode: r, w, a, x and their combinations.
- Use built-in file functions to read, write and manage files efficiently.
- Always close files to avoid data loss and resource leaks.

COMMON FILE MODES QUICK REFERENCE

Mode	r	w	a	x	r+	w+	a+
Purpose	Read	Write	Append	Create	Read & Write	Read & Write (overwrite)	Read & Write (append)

SCOPE AND MODULES IN PYTHON

Understand Names • Organize Code • Reuse Effectively

1. SCOPE OF OBJECTS & NAMES

Scope determines where a name (variable, function, class, etc.) is visible and usable.

- Each name has a lifetime and a visibility region.
- Inner scopes can access names from outer scopes.
- Names created inside a scope are local to that scope unless declared global.

A name is looked up in the current scope, then in the enclosing scopes, and finally in the global (built-in) scope.

2. LEGB RULE

Python uses LEGB rule to resolve names (from inner to outer scope).

Rule	Scope	Where it looks
L	Local	Inside the current function
E	Enclosing	In any enclosing functions
G	Global	In the module (global level)
B	Built-in	In Python built-in names (`len`, `print`, etc.)

```
def outer():          # E
    x = 'enclosing'
    def inner():      # L
        x = 'local'
        print(x)
    inner()
    print(x)          # enclosing

x = 'global'          # G
outer()
print(x)              # global
print(len('hi'))      # built-in (B)
```

If a name is assigned in a scope, it is treated as LOCAL unless declared as global or nonlocal.

3. MODULE BASICS

- A module is a Python file (`.py`) containing Python definitions and statements.
- It helps organize code into reusable units.
- Each module has its own global namespace.
- The module name is the file name (without `.py`).

Example:

math_utils.py

```
def add(a, b): ...

def sub(a, b): ...

PI = 3.14159
```

import

main.py

```
import math_utils

result = math_utils.add(2, 3)

print(result)
```

The code in a module runs only once—when it is first imported.

4. MODULE FILES AS NAMESPACES

- All names defined in a module live in that module's namespace.
- Access names using: `module_name.name`
- Names defined in one module do not pollute another.

Example:

util.py

```
value = 100

def greet():
    return 'Hello'
```

app.py

```
import util

print(util.value)    # 100
print(util.greet()) # Hello
```

Each module has its own namespace dictionary (accessible via `module.__dict__`).

5. IMPORT MODEL

- `import` statement loads a module and binds its name.
- Python searches for modules in `sys.path`.
- The first time a module is imported, it is executed and added to `sys.modules`.
- Subsequent imports reuse the loaded module.

Forms of Import

Statement	Meaning
`import module`	Import the whole module
`from module import name`	Import specific name(s)
`from module import *`	Import all public names
`import module as alias`	Import with an alias

Example:

```
import math
from math import sqrt, pi
import math as m
print(math.pi, sqrt(16), m.e)
```

6. RELOADING MODULES

- When a module file changes during development, you may want to reload it without restarting Python.
- Use `importlib.reload(module)`.

Example:

```
import importlib
import config
print(config.VALUE)
# ... edit config.py ...
importlib.reload(config)
print(config.VALUE)     # shows updated value
```

Reloading re-executes the module code. Existing objects imported with `from module import name` will NOT update automatically.

7. PRACTICAL EXAMPLE – SCOPE

```
x = 'global'              # Global
def outer():
    y = 'enclosing'       # Enclosing
    def inner():
        z = 'local'       # Local
        print(x, y, z)    # L -> E -> G
    inner()
outer()
print(x)
```

Output:

```
global enclosing local
global
```

8. PRACTICAL EXAMPLE – MODULES

file: tools.py

```
def square(n):
    return n * n

PI = 3.14159
```

file: main.py

```
import tools
print(tools.square(5))     # 25
print(tools.PI)            # 3.14159

from tools import square
print(square(6))           # 36
```

Modules promote code reusability, better organization and separation of concerns.

9. SUMMARY

- ✔ Scope determines where names are visible.
- ✔ LEGB rule: Local → Enclosing → Global → Built-in.
- ✔ Modules group related code in files (`.py`).
- ✔ Each module creates its own namespace.
- ✔ Use `import` to access module contents.
- ✔ Modules are loaded once and cached in `sys.modules`.
- ✔ Use `importlib.reload()` to reload changed modules.

QUICK REFERENCE

SCOPE EXAMPLE

```
a = 10
def func():
    b = 20
    def inner():
        c = 30
        print (a, b, z)
    inner()
func()
# 10 20 30
```

LEGB LOOKUP ORDER

COMMON IMPORTS

```
import module
import module as alias
from module import name
from module import name as alias
from module import *

import importlib
importlib.reload(module)
```

USE CASES

- Sse scope to manage variable visibility
- Use modules to split large programs
- Use imports to reuse code
- Use reload during development

Remember: Clean scope. Organized modules. Better code.

Write once. Use anywhere.

NUMPY BASICS

Fast • Powerful • Efficient Numerical Computing in Python

NumPy

1. INTRODUCTION TO NUMPY ndarray

NumPy (Numerical Python) provides the ndarray object – a fast, flexible, multi-dimensional array for mathematical operations.

- Homogeneous: all elements must be of the same data type
- N-dimensional
- Efficient: optimized for performance
- Supports vectorized operations

```
import numpy as np
a = np.array([1, 2, 3])
print(a)
# [1 2 3]
print(type(a))
# <class 'numpy.ndarray'>
```

Index:	0	1	2
a =	1	2	3

Shape: (3,) Dtype: int64

2. DATATYPES

NumPy supports a wide range of data types.

Category	Example	np.dtype
Integers	10	np.int32
Floating	3.14	np.float64
Complex	2 + 3j	np.complex128
Boolean	True	np.bool_
Strings	'abc'	np.str_
Bytes	b'abc'	np.bytes_

```
x = np.array([1, 2, 3], dtype=np.int32)
y = np.array([1.5, 2.5], dtype=np.float64)
z = np.array([True, False], dtype=np.bool_)
print(x.dtype, y.dtype, z.dtype)
# int32 float64 bool
```

3. ARRAY ATTRIBUTES

Important attributes of ndarray:

Attribute	Description
ndim	Number of dimensions
shape	Tuple of array dimensions
size	Total number of elements
dtype	Data type of elements
itemsize	Size (in bytes) of each element
nbytes	Total bytes consumed
T	Transpose of the array

```
a = np.array([[1, 2, 3], [4, 5, 6]])
print(a.ndim)      # 2
print(a.shape)     # (2, 3)
print(a.size)      # 6
print(a.dtype)     # int64
print(a.itemsize)  # 8
print(a.nbytes)    # 48
print(a.T)
#  [[1 4]
#  [2 5]]
#  [3 6]]
```

4. ARRAY CREATION ROUTINES

Common ways to create arrays:

Function	Description	Example
np.array()	From list/tuple	np.array([1,2,3])
np.zeros()	Array of zeros	np.zeros((2,3))
np.ones()	Array of ones	np.ones((3,))
np.empty()	Uninitialized array	np.empty((2,2))
np.full()	Array with value	np.full((2,2), 7)
np.eye()	Identity matrix	np.eye(3)
np.arange()	Range with step	np.arange(0,10,2)
np.linspace()	Evenly spaced	np.linspace(0,1,5)

Examples:

```
np.zeros((2,3))      # 2x3 zeros
np.ones(5)           # 1D array of 5 ones
np.full((2,2), 9)    # 2x2 filled with 9
np.eye(3)            # 3x3 identity matrix

np.arange(0, 10, 2)  # [0 2 4 6 8]
np.linspace(0, 1, 5) # [0.  0.25 0.5 0.75 1.]
```

5. ARRAY FROM EXISTING DATA

Create arrays from existing Python data structures or other arrays.

From Python Lists / Tuples

```
lst = [1, 2, 3, 4]
a = np.array(lst)
print(a)          # [1 2 3 4]
print(a.shape)    # (4,)
```

From Nested Lists

```
nested = [[1, 2, 3], [4, 5, 6]]
b = np.array(nested)
print(b)
#  [[1 2 3]
#  [4 5 6]]
print(b.shape)    # (2, 3)
```

From Another Array (Copy / View)

```
c = np.array(a)          # copy
d = a.copy()             # explicit copy
e = a.view()             # view (no copy)
print(np.shares_memory(a, e)) # True
```

From Existing Buffer / Data

```
import array
buf = array.array('i', [1,2,3,4])
f = np.frombuffer(buf, dtype=np.int32)
print(f)   # [1 2 3 4]
```

Differences: Copy owns its data; View shares the same data. Changes in the original affect the view, not the copy.

6. ARRAY FROM NUMERICAL RANGES

Create arrays with values from numerical intervals.

np.arange() – evenly spaced values

```
a = np.arange(0, 10)      # 0 to 9
b = np.arange(1, 10, 2)   # 1 to 9 step 2
print(a)  # [0 1 2 3 5 6 7 8 9]
print(b)  # [1 3 5 7 9]
```

np.linspace() – fixed number of samples

```
x = np.linspace(0, 1, 5)
   # start, stop, num
print(x)  # [0.  0.25 0.5 0.75 1..]
```

np.logspace() – logarithmic spacing

```
y = np.logspace(0, 2, 5)
   # 10^0 to 10^2 with 5 points
print(y)  # [   1.     3.16227766
          #    10.     31.6227766
          #   100.              ]
```

np.geomspace() – geometric spacing

```
z = np.geomspace(1, 16, 5)
   # geometric progression
print(z)  # [ 1.   2.   4.   8.  16..]
```

Use arange() for integer ranges and loops, linspace() for fixed number of evenly spaced samples.

7. INDEXING & SLICING

Access and extract parts of an array.

1D Indexing

```
a = np.array([10, 20, 30, 40, 50])
print(a[0])     # 10
print(a[-1])    # 50
print(a[1:4])   # [20 30 40]
print(a[:3])    # [10 20 30]
print(a[::2])   # [10 30 50]
```

Index:	0	1	2	3	4
a =	10	20	30	40	50

a[1:4] a[::2]

2D Indexing

```
b = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]])
print(b[0, 1])   # 2
print(b[2, :])   # [7 8 9]
print(b[:, 1])   # [2 5 8]
print(b[0:2, 1:3])
#  [[2 3]
#  [5 6]]
```

Row↓ \ Col→	0	1	2
0	1	2	3
1	4	5	6
2	7	8	9

Boolean Indexing

```
a = np.array([10, 20, 30, 40, 50])
mask = a > 25
print(mask)      # [False False True True True]
print(a[mask])   # [30 40 50]
```

Fancy Indexing

```
a = np.array([10, 20, 30, 40, 50])
idx = [4, 0, 2]
print(a[idx])    # [50 10 30]
```

Slicing with Steps

```
a = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
print(a[2:8:2])   # [2 4 6]
print(a[::-1])    # reverse array
                  # [9 8 7 6 5 4 3 2 1 0]
```

General slice form:
a[start : stop : step]

- start: inclusive (default 0)
- stop : exclusive (default end)
- step : interval (default 1)

KEY TAKEAWAYS

- ✔ ndarray is the core object in NumPy.
- ✔ NumPy supports powerful array creation routines.
- ✔ Use attributes to understand array structure and memory.
- ✔ Create arrays from existing data efficiently.
- ✔ Indexing and slicing help you access data quickly.

COMMON DATA TYPES IN NUMPY

Type	np.dtype	Example
Signed Integer	np.int32	-2, 0, 10
Unsigned Integer	np.uint8	0, 255
Floating Point	np.float64	3.1415
Complex	np.complex128	1+2j

Type	np.dtype	Example
Boolean	np.bool_	True, False
String (Fixed)	np.str_	'hello'
Bytes (Fixed)	np.bytes_	b'abc'

Remember: NumPy makes Python powerful for numerical and scientific computing.

Build once. Compute anywhere.

INFOMAX COMPUTER ACADEMY

OUR COURSES

Career-Focused Learning. Future-Ready You.

Scan to Visit Our Website

Office & Online Classes Available

Get Practical Learning Exposure

One to One Classes Available

CERTIFICATION COURSES

Course Name	Duration
CCC	3 Months
BCC	3 Months
O Level	6 Months / 1 Year

BASIC COURSES

Course Name	Duration
CCA	3 Months
DCA	6 Months
DFA	6 Months
ADCA	1 Year
DOAP	1 Year
DIT	1 Year
DTP	3 Months
Tally Prime	3 Months
Advance Excel	2 Months

SPECIAL TRAINING COURSES

Course Name	Duration
Full Stack Web Development Using PHP & MYSQL	6 Months
Full Stack Web Development Using Python Django	5 Months
Diploma in Digital Marketing	3 Months
Python Data Science	6 Months
Data Analytics Using Python	4 Months
Python for AI	6 Months
Cyber Security	3 Months

UNIVERSITY COURSES

Course Name	Duration	Fees (Per Sem)
BCA	3 Years	12800/-
BBA	3 Years	12800/-
MCA	2 Years	17000/-
MBA	2 Years	17000/-
B.COM	3 Years	6500/-
M.COM	2 Years	9000/-
MSW	2 Years	9000/-
M.SC(Math)	2 Years	10000/-
BA.	3 Years	5500/-
MA	2 Years	9000/-

TECHNICAL COURSES

Course Name	Duration
C Language	2 Months
C++	3 Months
PYTHON	3 Months
DSA	2 Months
Java (Class- IX, X, XI, XII)	3-6 Months
Core Java	4 Months
HTML	1 Months
HTML & CSS	2 Months
Java Script	1 Months
SQL/MYSQL	1 Month

EXPERT FACULTY
Learn from Industry Professionals

PRACTICAL APPROACH
Hands-on Training & Live Projects

CERTIFIED COURSES
NIELIT & Industry Recognized

CAREER SUPPORT
Placement Assistance & Guidance

OFFICE:
G. R. COMPLEX (1ST FLOOR)
PREETAM NAGAR, (BESIDE RAJASTHAN SWEETS),
DHOOMANGANJ, PRAYAGRAJ U.P 211011

CONTACT:
8874588766, 9532878816, 9598948810

WEB:
WWW.INFOMAX.ORG.IN

EMAIL:
CONTACT@INFOMAX.ORG.IN