UNIT-1

Lecture-1

Rapid Introduction to Procedural Programming

1. What is Procedural Programming?

Procedural Programming is a programming paradigm (style of programming) that is based on the concept of procedures (also called A functions, routines, or subroutines).

- program is divided into small blocks of code (procedures) that perform specific tasks.
- These procedures can be reused throughout the program.
- It focuses on step-by-step execution (sequence of instructions).

Python, although it supports multiple paradigms (procedural, object-oriented, functional), is commonly introduced through procedural programming.

2. Key Features of Procedural Programming

- Top-Down Approach → Programs are written step by step in a logical order.
- Use of Functions → Code is organized into reusable blocks (functions).
- Modularity → The program is divided into smaller, manageable parts.
- Variables → Used to store data for processing.
- Control Structures → If-else, loops, etc. are used to control flow.

3. Steps in Procedural Programming

- Define the problem clearly.
- Break it into smaller tasks (procedures).
- Write functions for each task.
- Execute them in a step-by-step sequence.

4. Advantages

- Easy to learn and use.
- Code is readable and structured.
- Functions promote code reuse.
- Debugging is simpler (smaller chunks of code).

5. Limitations

- Not suitable for very large projects (hard to manage functions).
- Code can become less flexible compared to OOP. Example Program: Sum of Two Numbers#

Example: Procedural approach in Python

Step 1: Define a procedure (function) for input def get_numbers():

```
# Step 1: Take input
a = int(input("Enter first number: "))
b = int(input("Enter second number: "))

# Step 2: Define a procedure for calculation
def calculate_sum(x, y):
    return x + y

# Step 3: Define a procedure for output
def display_result(result):
    print("The sum is:", result)

# Calling the functions
result = calculate_sum(a, b)
display_result(result)

# Main program (step-by-step execution)
num1, num2 = get_numbers()   # Input result = calculate_sum(num1, num2)   # Processing
display_result(result)   # Output
```

6.Sample Output

```
Enter first number: 10
Enter second number: 20
The sum is: 30
```

Lecture-2

Data Types – Identifiers and Keywords in Python

1. Identifiers in Python

Identifiers are the names given to variables, functions, classes, or objects in a program. They help us identify different elements in code.

Rules for Identifiers

1. Identifiers can contain letters (a–z, A–Z), digits (0–9), and the underscore (_) character.
   - Example: name, roll_number, age2.
2. They cannot begin with a digit.
   - ❌ 2age (invalid)
   - ✅ age2 (valid)
3. Identifiers are case-sensitive.
   - Example: Name and name are different identifiers.
4. Reserved keywords cannot be used as identifiers.
   - ❌ if = 10 (invalid, because if is a keyword).
5. No special characters allowed except underscore (_).
   - ❌ student-name
   - ✅ student_name

2. Keywords in Python

Keywords are predefined words in Python that have special meaning and cannot be used as identifiers.

For example: if, else, while, class, import, etc. List of Python Keywords (Python 3.12+)

Some important ones are:

False, True, None,and, or, not, if, else, elif,

while, for, break, continue, pass,

def, return, class, import, from, as,

try, except, finally, raise, assert,

in, is, global, nonlocal, lambda, yield

3. Differences Between Identifiers and Keywords

Feature	**User-defined names (variables, etc.)**	**Predefined reserved words**
Identifier / Keyword	Identifier	Keyword
Meaning	Names chosen by the programmer	Words fixed by Python for syntax
Customizable? / Example	Yes / age, student_name	No / if, while, def

4. Example Program – Using Identifiers and Keywords

```
# Example: Identifiers and Keywords in Python
# Valid identifiers
student_name = "Alice"
roll_number = 25
marks = 85.5

# Using keywords in program (not as identifiers!)
if marks >= 40:   # 'if' and '>=' are keywords/operators
    result = "Pass"
else:
    result = "Fail"

# Display the results
print("Student Name:", student_name)
print("Roll Number:", roll_number)
print("Marks:", marks)
print("Result:", result)
```

Output:

Student Name: Alice Roll Number: 25 Marks: 85.5

Result: Pass

Lecture-3

Integral Types in Python

1. What are Integral Types?

In Python, integral types refer to whole numbers (integers) that do not have a fractional or decimal part.

They are represented using the built-in int data type.

Example: -100, 0, 42, 999999

2. Characteristics of Integers in Python

- No Size Limit – Unlike languages like C/Java, Python's int can store arbitrarily large numbers.
- Example: 10000000000000000000000000000000 is valid.
- Positive and Negative Numbers – Integers can be positive, negative, or zero.
- Example: -45, 0, 78.
- Automatic Conversion – Python automatically converts large integers into long integers (internally), so you don't need to worry about overflow.
- Immutable – Like all basic data types in Python, integers are immutable (cannot be changed after creation).

3. Types of Integral Literals

Python integers can be represented in different number systems:

- 1. Decimal (Base 10) – Normal numbers (default system). o Example: a = 100
- 2. Binary (Base 2) – Prefix with 0b or 0B.
- Example: b = 0b1010 # 10 in decimal 3. Octal (Base 8) – Prefix with 0o or 0O.
- Example: c = 0o12 # 10 in decimal 4. Hexadecimal (Base 16) – Prefix with 0x or 0X.
- Example: d = 0xA # 10 in decimal4. Useful Integer Operations

    - Arithmetic: +, -, *, /, //, %, **
    - Comparison: <, >, ==, !=
    - Bitwise: &, |, ^, ~, <<, >>

Example Program – Demonstrating Integral Types

Example: Integral Types in Python

```
# Decimal integer
a = 42

# Binary, Octal, and Hexadecimal representations
b = 0b101010     # binary
c = 0o52         # octal
d = 0x2A         # hexadecimal

# Arithmetic operations
sum_val = a + 10
product = a * 2
power = a ** 2

# Printing results
print("Decimal (a):", a)
print("Binary (b = 0b101010):", b)
print("Octal (c = 0o52):", c)
print("Hexadecimal (d = 0x2A):", d)
print("Sum:", sum_val)
print("Product:", product)
print("Power:", power)
```

6. Sample Output

```
Decimal (a): 42
Binary (b = 0b101010): 42
Octal (c = 0o52): 42
Hexadecimal (d = 0x2A): 42
Sum: 52
Product: 84
Power: 1764
```

Lecture-4

Floating Point Types and Strings in Python

Part 1: Floating Point Types

1. What are Floating Point Types?

Floating point types are used to represent numbers with a decimal point.

In Python, this is handled by the built-in float data type.

Example:

pi = 3.14159

temperature = -7.5

2. Characteristics of Floats

- Decimal Numbers – Can store numbers like 0.5, -2.75, 100.0.
- 2. Scientific Notation – Python supports e notation for very large or small numbers. o Example: 1.2e3 → 1200.0, 3.5e-2 → 0.035
- Immutable – Float values cannot be changed; operations return new float values.
- Precision – Python floats are implemented using 64-bit double precision.

3. Operations on Floats

- Arithmetic: +, -, *, /, // (floor division), %, **
- Comparison: <, >, ==, !=4. Example Program – Floating Point Numbers

```
# Floating Point Example

a = 5.5

b = 2.0


# Arithmetic operations

sum_val = a + b

difference = a - b
```

```
product = a * b

division = a / b

power = a ** b


# Printing results

print("a + b =", sum_val)

print("a - b =", difference)

print("a * b =", product)

print("a / b =", division)

print("a ** b =", power)
```

Output:

```
a + b = 7.5

a - b = 3.5

a * b = 11.0

a / b = 2.75

a ** b = 30.25
```

Part 2: Strings in Python

1. What is a String?

A string is a sequence of characters enclosed in single (' ') or double (" ") quotes. Strings can contain letters, numbers, symbols, or spaces.

Examples:

```
name = "Alice"

greeting = 'Hello, World!'
```

2. Characteristics of Strings

- Immutable – Once created, the string cannot be changed.
- 2. Indexing – Characters in a string can be accessed using indices (0-based). o Example: name[0] → 'A'

- 3. Slicing – Substrings can be extracted.o Example: name[1:4] → 'lic'

- 4. Concatenation – Strings can be joined using +.
- Example: "Hello " + "Alice" → "Hello Alice" 5. Repetition – Strings can be repeated using *.
- Example: "Hi! " * 3 → "Hi! Hi! Hi! "

3. Example Program – Strings

```
# String Example

greeting = "Hello"

name = "Alice"


# Concatenation

message = greeting + ", " + name + "!"

print(message)


# Indexing

print("First character:", name[0])


# Slicing

print("Substring:", name[1:4])


# Repetition

print("Repeat greeting:", greeting * 3)
```

Output:

```
Hello, Alice!

First character: A

Substring: lic

Repeat greeting: HelloHelloHello
```

Lecture-5

Comparing Strings in Python

1. Introduction

Comparing strings means checking if one string is equal to, less than, or greater than another string.

Python allows string comparison using relational operators and built-in methods.

2. Methods of Comparing Strings

A. Using Relational Operators

Python supports the following operators for string comparison:

Operator & Meaning	**Example Usage**
== → Checks if two strings are equal	"apple" == "apple" → True
!= → Checks if two strings are not equal	"apple" != "banana" → True
< → Less than	
> → Greater than	"apple" < "banana" → True
<= → Less than or equal	"apple" >= "apple" → True
>= → Greater than or equal	

Lexicographical Order:

Strings are compared character by character based on ASCII values.

Example: 'apple' < 'banana' → True because 'a' < 'b'.

B. Using Built-in Methods

1. str.startswith(substring) – Returns True if string starts with the given substring.
2. str.endswith(substring) – Returns True if string ends with the given substring.
3. str.casefold() or str.lower() – Converts strings to lowercase before comparison (case- insensitive comparison).

3. Example Program – Comparing Strings

```
# String Comparison Example
str1 = "Apple"
str2 = "Banana"
str3 = "apple"

# Using relational operators
print("str1 == str2:", str1 == str2)
print("str1 != str2:", str1 != str2)
print("str1 < str2:", str1 < str2)
print("str1 > str2:", str1 > str2)

# Case-insensitive comparison
print("str1 equals str3 (case-insensitive):", str1.lower() == str3.lower())

# Using startswith() and endswith()
print("str2 starts with 'Ba':", str2.startswith("Ba"))
print("str2 ends with 'na':", str2.endswith("na"))
```

4. Sample Output

```
str1 == str2: False
str1 != str2: True
str1 < str2: True
str1 > str2: False
str1 equals str3 (case-insensitive): True
str2 starts with 'Ba': True
str2 ends with 'na': True
```

5. Summary

- Strings can be compared using relational operators.
- Comparison is lexicographical (ASCII-based).
- Case-insensitive comparison avoids issues with capital letters.
- Built-in methods like startswith() and endswith() make comparisons easy.

Lecture-6

Slicing and Striding Strings in Python

1. Introduction

In Python, strings are sequences of characters, which allows us to access individual characters or parts of strings using indexing, slicing, and striding.

2. Indexing

- Each character in a string has an index.
- Indexing starts from 0 for the first character and -1 for the last character.

Example:

```
text = "Python"

print(text[0])   # P

print(text[-1])  # n
```

3. Slicing

Definition

Slicing allows you to extract a substring from a string using the syntax: string[start:end]

- start → index to start slicing (inclusive)
- end → index to stop slicing (exclusive)

Example:

```
text = "PythonProgramming"


# Slicing examples

print(text[0:6])    # Python

print(text[6:17])   # Programming
```

```
# If start is omitted → starts from index 0
print(text[:6])     # Python

# If end is omitted → goes up to the end of the string
print(text[6:])     # Programming
```

4. Striding

Definition

Striding allows you to skip characters while slicing using the syntax: string[start:end:step]

- step → number of characters to skip

Example:

```
text = "PythonProgramming"
print(text[0:17:2])  # Picks every 2nd character
print(text[::-1])    # Reverses the string
```

5. Negative Indexing with Slicing

- Python allows negative indices in slicing.
- Example: text[-5:-1] slices from 5th last to 2nd last character.

```
text = "PythonProgramming"
print(text[-11:-1])  # "Programming"
```

6. Example Program – Slicing and Striding Strings # Slicing and Striding Example

```
text = "PythonProgramming"

# Basic slicing
slice1 = text[0:6]
slice2 = text[6:]

# Striding
stride1 = text[0:17:2]
```

```
reverse_text = text[::-1]


# Printing results
print("Original text:", text)
print("First word (slice):", slice1)
print("Second word (slice):", slice2)
print("Every 2nd character (stride):", stride1)
print("Reversed text:", reverse_text)
```

7. Sample Output

Original text: PythonProgramming

First word (slice): Python

Second word (slice): Programming

Every 2nd character (stride): PtoPormig

Reversed text: gnimmargorPmnohtyP

8. Summary

- Indexing accesses single characters.
- Slicing extracts substrings using [start:end].
- Striding allows skipping characters with [start:end:step].
- Negative indices make it easy to slice from the end of the string.

Lecture-7

String Operators and Methods in Python

1. Introduction

Strings in Python support various operators and built-in methods to perform operations like concatenation, repetition, searching, and formatting.

2. String Operators

A. Concatenation (+)

Joins two or more strings together.

```
str1 = "Hello"
str2 = "World"
result = str1 + " " + str2
print(result)  # Hello World
```

B. Repetition (*)

Repeats a string multiple times.

```
str1 = "Hi! "
print(str1 * 3)  # Hi! Hi! Hi!
```

C. Membership (in, not in)

Checks if a substring exists in a string.

```
text = "Python Programming"
print("Python" in text)     # True
print("Java" not in text)    # True
```

D. Comparison (==, !=, <, >)

Compares strings lexicographically (ASCII-based).

print("apple" < "banana") # True print("apple" == "Apple") # False

3. String Methods

Python provides many built-in string methods. Here are the most commonly used:

Method & Description	**Example Usage / Notes**
upper() → Converts string to uppercase	"hello".upper() → "HELLO"
lower() → Converts string to lowercase	"HELLO".lower() → "hello"
capitalize() → Capitalizes first character	"hello world".capitalize() → "Hello world"
title() → Capitalizes first character of each word	"hello world".title() → "Hello World"
strip() → Removes leading/trailing spaces	" hello ".strip() → "hello"
replace(old, new) → Replaces a substring with another	"apple".replace("a", "o") → "opple"
split(separator) → Splits string into list	"a,b,c".split(",") → ['a','b','c']
join(iterable) → Joins elements into string	",".join(['a','b','c']) → "a,b,c"
find(sub) → Index of first occurrence	"apple".find("p") → 1
count(sub) → Number of occurrences	"apple".count("p") → 2

4. Example Program – String Operators and Methods

String Operators

str1 = "Hello"

str2 = "World"

Concatenation

concat = str1 + " " + str2

Repetition

repeat = str1 * 3

Membership

check1 = "Hello" in concat

```
check2 = "Python" not in concat

# String Methods
text = "   python programming   "
upper_text = text.upper()
lower_text = text.lower()
capital_text = text.capitalize()
title_text = text.title()
strip_text = text.strip()
replace_text = text.replace("python", "Java")
split_text = text.split()
join_text = "-".join(split_text)

# Printing results
print("Concatenation:", concat)
print("Repetition:", repeat)
print("'Hello' in concat?", check1)
print("'Python' not in concat?", check2)
print("Uppercase:", upper_text)
print("Lowercase:", lower_text)
print("Capitalized:", capital_text)
print("Title Case:", title_text)
print("Stripped:", strip_text)
print("Replace 'python' with 'Java':", replace_text)
print("Split into list:", split_text)
print("Join list with '-':", join_text)
```

5. Sample Output

```
Concatenation: Hello World
Repetition: HelloHelloHello
'Hello' in concat? True
```

'Python' not in concat? True

Uppercase: PYTHON PROGRAMMING

Lowercase: python programming

Capitalized: Python programming

Title Case: Python Programming

Stripped: python programming

Replace 'python' with 'Java': Java programming

Split into list: ['python', 'programming']

Join list with '-': python-programming

6. Summary

- Operators +, *, in, not in, ==, <, > are commonly used with strings.
- Python provides many string methods for changing case, trimming, replacing, splitting, joining, and searching.
- Strings are immutable, so methods always return a new string.

Lecture-8

String Formatting with str.format in Python

1. Introduction

String formatting allows you to insert values into strings dynamically.

The str.format() method is a powerful way to format strings in Python.

Syntax:

"string with placeholders {}".format(values)

- Placeholders are defined using curly braces {}.
- Values inside .format() are inserted in order or by keyword.

2. Positional Formatting

Values are inserted based on their position in .format().

```
name = "Alice"
age = 20
print("My name is {} and I am {} years old.".format(name, age))
```

Output:

My name is Alice and I am 20 years old.

3. Indexed Placeholders

You can use index numbers inside {} to control the order.

```
print("My name is {0} and I am {1} years old. {0} loves Python.".format(name, age))
```

Output:

My name is Alice and I am 20 years old. Alice loves Python.

4. Keyword Formatting

You can pass keyword arguments to .format().

print("My name is {n} and I am {a} years old.".format(n=name, a=age))

Output:

My name is Alice and I am 20 years old.

5. Formatting Numbers

You can format integers, floats, and percentages:

pi = 3.14159265

print("Value of pi: {:.2f}".format(pi)) # 2 decimal places

print("Value of pi: {:10.2f}".format(pi)) # width 10, 2 decimals

Output:

Value of pi: 3.14

Value of pi: 3.14

- {:.2f} → float with 2 decimal places
- {:10.2f} → float with width 10 and 2 decimals (right-aligned)

6. Example Program – str.format

String Formatting Example

name = "Alice"

age = 20

score = 95.567

Positional formatting

print("Hello, {}. You scored {} marks.".format(name, score))

Indexed formatting

print("Name: {0}, Age: {1}, Name again: {0}".format(name, age))

Keyword formatting

print("Student {n} is {a} years old.".format(n=name, a=age))

Number formatting

print("Score rounded: {:.1f}".format(score))print("Score padded: {:10.2f}".format(score))

7. Sample Output

Hello, Alice. You scored 95.567 marks.

Name: Alice, Age: 20, Name again: Alice

Student Alice is 20 years old.

Score rounded: 95.6

Score padded: 95.57

8. Summary

- str.format() allows dynamic insertion of values into strings.
- Supports positional, indexed, and keyword formatting.
- Can format numbers with precision and width.
- Provides a clean and readable way to create strings dynamically.

Lecture-9

Collections Data Types – Tuples, Lists, Sets, Dictionaries in Python

1. Introduction

Collections are data types that can store multiple values. Python provides four main collection types:

Collection Type & Description	Mutable? & Syntax Example
List → Ordered sequence of elements	Yes → [1, 2, 3]
Tuple → Ordered sequence of elements	No → (1, 2, 3)
Set → Unordered collection of unique elements	Yes → {1, 2, 3}
Dictionary → Key-value pairs	Yes → {'a': 1, 'b': 2}

2. Lists

- Ordered and mutable.
- Can contain elements of different types.
- Common operations: append, insert, remove, pop, slicing.

List Example

fruits = ["apple", "banana", "cherry"] fruits.append("orange")

fruits.remove("banana") print(fruits)

Output:

['apple', 'cherry', 'orange']3. Tuples

- Ordered but immutable.
- Useful for fixed data like coordinates or RGB values.
- Can be nested or sliced.

Tuple Example

point = (10, 20)

print("X:", point[0], "Y:", point[1])

Output:

X: 10 Y: 20

4. Sets

- Unordered, unique elements, mutable.
- Useful for removing duplicates and set operations.

Set Example

numbers = {1, 2, 3, 3, 4}

numbers.add(5)

numbers.remove(2)

print(numbers)

Output:

{1, 3, 4, 5}

 Set Operations: union (|), intersection (&), difference (-)

a = {1, 2, 3}

b = {3, 4, 5}

print("Union:", a | b)

print("Intersection:", a & b)

Output:

Union: {1, 2, 3, 4, 5} Intersection: {3}5. Dictionaries

- Key-value pairs, unordered, mutable.
- Keys must be unique and immutable.
- Values can be of any type.

Dictionary Example

student = {"name": "Alice", "age": 20, "marks": 95} print(student["name"]) # Access value by key

Add or update

student["grade"] = "A"

student["marks"] = 98

```
print(student)
```

Output:

Alice

{'name': 'Alice', 'age': 20, 'marks': 98, 'grade': 'A'}

6. Summary

- Lists: Ordered, mutable, allows duplicates.
- Tuples: Ordered, immutable, allows duplicates.
- Sets: Unordered, mutable, unique elements only.
- Dictionaries: Unordered, mutable, key-value pairs.

Lecture-10

Iterating and Copying Collections in Python

1. Introduction

Collections (lists, tuples, sets, dictionaries) are often iterated to process elements. Python also provides ways to copy collections without affecting the original data.

2. Iterating Collections

A. Using for Loop

fruits = ["apple", "banana", "cherry"] for fruit in fruits:

print(fruit)

Output:

apple

banana

cherry

B. Using while Loop

numbers = [1, 2, 3, 4]

i = 0

```
while i < len(numbers):
    print(numbers[i])
    i += 1
```

Output:

1

2

3

4

C. Iterating with enumerate()

```
# Using enumerate() to get index and element
fruits = ["apple", "banana", "cherry"]

for index, fruit in enumerate(fruits):
    print(index, fruit)
```

Output:

0 apple

1 banana

2 cherry

D. Iterating Dictionaries

```
student = {"name": "Alice", "age": 20}
for key, value in student.items():
print(key, ":", value)
```

Output:

name : Alice

age : 20

3. Copying Collections

A. Shallow Copy

- Creates a new collection but nested elements refer to same objects.
- Methods:

o list.copy() → List

o dict.copy() → Dictionary

o copy.copy() → All collections

import copy

original = [1, 2, [3, 4]] shallow = copy.copy(original) shallow[2][0] = 99

print("Original:", original) print("Shallow:", shallow)

Output:

Original: [1, 2, [99, 4]]

Shallow: [1, 2, [99, 4]]

Nested list affected due to shallow copy.

B. Deep Copy

- Creates a completely independent copy, including nested elements.
- Method: copy.deepcopy()

import copy

original = [1, 2, [3, 4]]

deep = copy.deepcopy(original)

deep[2][0] = 99

print("Original:", original)

print("Deep:", deep)

Output:

Original: [1, 2, [3, 4]]

Deep: [1, 2, [99, 4]]

4. Summary

- Iteration: Use for, while, enumerate(), or .items() for dictionaries.
- Copying:
- Shallow copy: top-level copy; nested objects are shared. o Deep copy: complete independent copy.
- Deep copy: complete independent copy.

Lecture-11

Introduction to PIP (Python Package Installer)

1. What is PIP?

PIP stands for “Pip Installs Packages”.

It is the default package manager for Python that allows you to install, manage, and uninstall Python packages from the Python Package Index (PyPI).

- PIP helps you extend Python functionality by adding third-party libraries.
- Most Python installations (Python 3.4+) come with PIP pre-installed.

2. Checking if PIP is Installed

pip –version

Output Example:

pip 23.2.1 from /usr/local/lib/python3.12/site-packages/pip (python 3.12)

If not installed, you can install it using:

python -m ensurepip –upgrade

3. Installing Packages

Use the install command to install a package from PyPI.

Syntax:

pip install package_name

Example:

pip install requests

- Installs the Requests library for HTTP requests in Python.

4. Upgrading Packages

pip install --upgrade package_name

Example:

pip install --upgrade requests

- Upgrades the package to the latest version.

5. Uninstalling Packages

pip uninstall package_name

Example:

pip uninstall requests

- Removes the installed package.

6. Listing Installed Packages

pip list

Output Example:

Package Version

--- ---

requests 2.31.0

numpy 1.26.0

pandas 2.1.0

7. Example Program Using an Installed Package

After installing requests, you can use it in Python:

```
# Example: Using requests package
import requests
response = requests.get("https://api.github.com")
print("Status Code:", response.status_code)
print("Response JSON:", response.json())
```

Sample Output:

Status Code: 200

Response JSON: {...JSON data from GitHub API...}

This shows how PIP helps you install external packages and use them in your Python programs.

8. Summary

- PIP is the Python package manager.
- Commands: install, uninstall, upgrade, list.
- Allows Python to use third-party libraries like requests, numpy, pandas, etc.
- Essential for real-world Python development.