1: Rapid Introduction to Procedural Programming

1.1 Introduction

- **Procedural Programming (PP)** is a **programming paradigm** that uses **sequences of instructions** to perform tasks.
- Focuses on **processes**, **functions**, and **step-by-step execution**.
- Python supports both **procedural** and **object-oriented** programming.

Key Idea: Solve a problem by **breaking it into procedures (functions)** that operate on data.

1.2 Characteristics of Procedural Programming

1. **Linear Execution:** Instructions executed in order.
2. **Use of Functions:** Reusable blocks of code to perform tasks.
3. **Variables and Data:** Store information that can be manipulated.
4. **Control Flow:** Decision-making and loops guide execution.
5. **Modularity:** Functions can be organized into modules for better structure.

1.3 Advantages of Procedural Programming

- Easy to learn for beginners.
- Good for **small and medium-sized programs**.
- Promotes **code reusability** through functions.
- Easier to debug step by step.

Disadvantages:

- Not ideal for very large projects.
- Less flexible for modeling real-world entities (better in OOP).

1.4 Basic Components in Procedural Programming

1.4.1 Variables

- Named storage for data.

```
x = 10
name = "Jyothi"
pi = 3.14
```

1.4.2 Data Types

- Integer (int), Float (float), String (str), Boolean (bool)

1.4.3 Operators

- Arithmetic: +, -, *, /, //, %, **
- Comparison: ==, !=, <, >
- Logical: and, or, not

1.4.4 Input & Output

- input() → Get user input
- print() → Display output

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

1.5 Control Flow

1.5.1 Conditional Statements

- Decide which code block to execute based on conditions.

```
age = 18
if age >= 18:
   print("Adult")
else:
   print("Minor")
```

1.5.2 Loops

- **For Loop:** Iterate over a sequence.

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

- **While Loop:** Repeat until a condition is False.

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

1.6 Functions

- Reusable blocks that perform specific tasks.

  Syntax:

```
def function_name(parameters):
   # code
   return value
```

Example:

```
def greet(name):
   return f"Hello, {name}!"

print(greet("Jyothi"))
```

Key Points:

- **Parameters:** Input to function.
- **Return:** Output from function.
- Functions help **avoid code repetition**.

1.7 Example Program (Procedural Style)

Problem: Calculate area of a circle using procedural programming.

```
def calculate_area(radius):
    area = 3.14 * radius * radius
    return area

r = float(input("Enter radius: "))
circle_area = calculate_area(r)
print("Area of the circle:", circle_area)
```

Explanation:

- Step 1: Define function to calculate area.
- Step 2: Take user input.
- Step 3: Call function and display result.

1.8 Summary

- Procedural programming emphasizes **step-by-step logic**, **functions**, and **control flow**.
- Ideal for small programs, scripting, and beginner-level programming.
- Understanding PP helps in transitioning to **Object-Oriented Programming** later.

Topic 2: Data Types, Identifiers, and Keywords

2.1 Introduction

- **Data Types**: Specify the type of data a variable can store.
- **Identifiers**: Names given to variables, functions, classes, etc.
- **Keywords**: Reserved words in Python that have **special meaning**.

Importance: Correct use ensures **readable, error-free code**.

2.2 Data Types in Python

Python is **dynamically typed**, meaning the variable type is inferred automatically.

2.2.1 Numeric Types

1. **Integer (int)**: Whole numbers.

```
x = 10
y = -5
```

2. **Float (float)**: Decimal numbers.

```
pi = 3.1416
temperature = -7.5
```

3. **Complex (complex)**: Numbers with real and imaginary parts.

```
z = 3 + 4j
print(z.real, z.imag)  # 3.0 4.0
```

2.2.2 Boolean (bool)

- Represents **True** or **False**.

```
a = True
b = False
print(a and b)  # False
```

2.2.3 Strings (str)

- Sequence of characters enclosed in **single ' '** or **double " "** quotes.

```
name = "Jyothi"
greeting = 'Hello'
```

2.2.4 Type Conversion

- Convert between types using int(), float(), str().

```
a = "10"
b = int(a) + 5
print(b)  # 15
```

2.3 Identifiers

- **Definition:** Names used to identify variables, functions, classes, etc.
- **Rules:**
    1. Can contain letters, digits, and underscores _.
    2. Must **start with a letter or underscore**.
    3. Cannot be a **keyword**.
    4. Case-sensitive (myVar ≠ myvar).

Examples:

```
my_var = 10
_name = "Jyothi"
age1 = 22
```

Invalid Examples:

```
1name = "Invalid"  # Cannot start with a number
for = 10          # 'for' is a keyword
```

2.4 Keywords

- Python has **reserved words** with **special meaning**.
- Cannot be used as identifiers.

Common keywords:
if, else, elif, for, while, break, continue, def, return, import, pass, True, False, None

Check all keywords in Python:

```
import keyword
print(keyword.kwlist)
```

2.5 Constants

- Variables whose value **should not change**. Python does not enforce constants but **by convention**, uppercase names indicate constants.

```
PI = 3.1416
GRAVITY = 9.8
```

2.6 Examples

```
# Variables and Data Types
name = "Jyothi"       # string
age = 22              # int
height = 5.6          # float
is_student = True     # bool

# Using identifiers
_score = 90
total_marks = 100

# Keywords cannot be used as identifiers
# if = 5  # Error

# Type conversion
a = "15"
b = int(a) + 5
print(b)  # 20
```

2.7 Summary

- **Data types** define what kind of data variables store.
- **Identifiers** are names for variables, functions, and objects.
- **Keywords** are reserved words in Python with special meaning.
- Correct naming and data type usage ensures **readable and error-free code**.

Topic 3: Integral Types

3.1 Introduction

- **Integral types** represent **whole numbers** without decimal points.
- In Python, the primary integral type is **int**.
- Integers can be **positive, negative, or zero**.

Importance: Integers are used for counting, indexing, looping, and many calculations.

3.2 Integer Type (int)

3.2.1 Definition

- An **integer** is a number **without a fractional part**.
- Python supports **arbitrary large integers** (no overflow).

```
a = 10      # positive integer
b = -5      # negative integer
c = 0       # zero
```

3.2.2 Type Checking

- Use type() to check the type of a variable:

```
x = 25
print(type(x))  # <class 'int'>
```

3.3 Numeric Operations with Integers

3.3.1 Arithmetic Operators

Operator	Description	Example	Output
+	Addition	5 + 3	8
-	Subtraction	5 - 3	2
*	Multiplication	5 * 3	15
/	Division (float)	5 / 2	2.5
//	Floor Division	5 // 2	2
%	Modulus (remainder)	5 % 2	1
**	Exponentiation	5 ** 2	25

Example:

```
a = 7
b = 3
print(a + b)  # 10
print(a // b) # 2
print(a ** b) # 343
```

3.3.2 Comparison Operators

- Compare integer values and return **Boolean (True/False)**.

```
x = 10
y = 20
print(x == y)  # False
print(x < y)   # True
print(x != y)  # True
```

3.3.3 Logical Operators

- Combine integer comparisons using and, or, not.

```
x = 10
y = 20
print(x < 15 and y > 15)  # True
print(not(x > y))         # True
```

3.4 Type Conversion

- Convert between **integers and other numeric types**.

```
a = 10
b = float(a)  # convert int to float
c = int(3.14) # convert float to int
print(b, c)   # 10.0 3
```

3.5 Working with Large Integers

- Python supports **very large integers** without overflow:

```
large_num = 12345678901234567890
print(large_num)
```

3.6 Examples

```
# Basic integer operations
x = 15
y = 4
print("Addition:", x + y)
print("Subtraction:", x - y)
print("Multiplication:", x * y)
print("Division:", x / y)
print("Floor Division:", x // y)
print("Remainder:", x % y)
print("Power:", x ** y)

# Comparison and logical
print(x > y and y > 0)  # True

# Type conversion
num = "100"
num_int = int(num)
print(num_int + 50)  # 150
```

3.7 Summary

- **Integers (int)** store whole numbers.
- Support **arithmetic, comparison, and logical operations**.
- Python integers can be **very large**.
- **Type conversion** allows flexible numeric operations.

Topic 4: Floating Point Types & Strings

4.1 Introduction

- Python supports **numeric data types** (int and float) and **text data** (str).
- **Floating point numbers** are used for **decimal values**.
- **Strings** store sequences of characters and are essential for handling **textual information**.

4.2 Floating Point Types (float)

4.2.1 Definition

- A **float** is a number with a **decimal point**.
- Example: 3.14, -7.5, 0.0

```
pi = 3.1416
temperature = -7.5
zero = 0.0
```

4.2.2 Operations with Floats

- Supports **arithmetic** operations similar to integers.

```
a = 5.5
b = 2.0
print(a + b)  # 7.5
print(a - b)  # 3.5
print(a * b)  # 11.0
print(a / b)  # 2.75
```

```
print(a ** b) # 30.25
```

4.2.3 Type Conversion

- Convert **int → float** and **float → int**

```
x = 10
y = float(x)  # 10.0
z = int(3.99) # 3
```

4.3 Strings (str)

4.3.1 Definition

- A **string** is a **sequence of characters** enclosed in **single ' '** or **double " "** quotes.

```
name = "Jyothi"
greeting = 'Hello'
sentence = "Python programming is fun!"
```

4.3.2 String Operations

1. **Concatenation (+)** → Join strings.

```
first = "Hello"
second = "World"
print(first + " " + second)  # Hello World
```

2. **Repetition (*)** → Repeat strings.

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

4.3.3 Escape Sequences

- Special characters inside strings using \:

Escape	Description
\n	New line
\t	Tab
\'	Single quote
\"	Double quote
\\	Backslash

```
print("Hello\nWorld")
print("Python\tProgramming")
```

4.4 String Indexing

- Each character in a string has a **position (index)**, starting from **0**.

```
s = "Python"
print(s[0])  # 'P'
print(s[3])  # 'h'
print(s[-1]) # 'n' (last character)
```

4.5 String Methods

- Python provides **built-in functions** for strings:

```
text = " hello world "
print(text.upper())    # " HELLO WORLD "
print(text.lower())    # " hello world "
print(text.strip())    # "hello world"
print(text.replace("world", "Python")) # " hello Python "
print(len(text))       # 13
```

4.6 Example Program

Problem: Calculate BMI and display message using floats and strings.

```
name = input("Enter your name: ")
weight = float(input("Enter weight in kg: "))
height = float(input("Enter height in meters: "))

bmi = weight / (height ** 2)
print(f"{name}, your BMI is {bmi:.2f}")

if bmi < 18.5:
   print("Underweight")
elif bmi < 25:
   print("Normal weight")
else:
   print("Overweight")
```

Explanation:

- float() converts input to decimal numbers.
- f"{bmi:.2f}" formats the float to **2 decimal places**.

4.7 Summary

- **Float:** Numbers with decimals, support arithmetic operations.
- **String:** Sequence of characters, support concatenation, repetition, indexing, and methods.
- **Escape sequences** allow special formatting in strings.
- **Type conversion** is essential for mixed data operations.

Topic 5: Comparing Strings

5.1 Introduction

- Strings are sequences of characters used to represent text.
- Comparing strings is important for **conditional checks, sorting, searching, and validation**.
- Python allows **lexicographical comparison** (dictionary order).

5.2 String Comparison Operators

Operator	Description	Example	Output
==	Equal to	"a" == "a"	True
!=	Not equal to	"a" != "b"	True
<	Less than	"apple" < "banana"	True
>	Greater than	"apple" > "banana"	False
<=	Less than or equal	"a" <= "b"	True
>=	Greater than or equal	"c" >= "b"	True

Example:

```
str1 = "apple"
str2 = "banana"

print(str1 == str2)  # False
print(str1 != str2)  # True
print(str1 < str2)   # True
print(str1 > str2)   # False
```

5.3 Case Sensitivity

- String comparisons in Python are **case-sensitive**.
- "Apple" and "apple" are **different**.

```
a = "Apple"
b = "apple"
print(a == b)   # False
print(a.lower() == b.lower())  # True
```

5.4 Using ord() and chr() for Comparison

- Strings are compared based on **Unicode values** of characters.
- ord(char) → returns Unicode of a character.
- chr(num) → returns character of Unicode value.

```
print(ord('a'))  # 97
print(ord('b'))  # 98
print('a' < 'b') # True
```

5.5 String Methods for Comparison

1. **.lower() / .upper()** → Case-insensitive comparison.

```
s1 = "Hello"
s2 = "hello"
print(s1.lower() == s2.lower())  # True
```

2. **.startswith() / .endswith()** → Check prefixes or suffixes.

```
s = "Python"
```

```
print(s.startswith("Py"))  # True
print(s.endswith("on"))    # True
```

3. **.isalpha() / .isdigit()** → Check type of characters.

```
s = "Python"
print(s.isalpha())  # True
s2 = "123"
print(s2.isdigit()) # True
```

5.6 Comparing Strings in Conditions

- Strings are often compared in **if-else statements**:

```
user_input = input("Enter yes or no: ").lower()
if user_input == "yes":
   print("You selected Yes")
elif user_input == "no":
   print("You selected No")
else:
   print("Invalid input")
```

5.7 Sorting Strings

- Strings can be **sorted alphabetically** using sorted() or list .sort().

```
names = ["Jyothi", "Anil", "Zara", "Maya"]
names.sort()  # Sorts in ascending order
print(names)  # ['Anil', 'Jyothi', 'Maya', 'Zara']

# Reverse order
names.sort(reverse=True)
print(names)  # ['Zara', 'Maya', 'Jyothi', 'Anil']
```

5.8 Example Program

Problem: Check if a word is in a list (case-insensitive)

```
words = ["Python", "Java", "C++", "JavaScript"]
search = input("Enter a language to search: ").lower()
```

```
found = False
for word in words:
    if word.lower() == search:
        found = True
        break

if found:
    print(f"{search} found in the list!")
else:
    print(f"{search} not found in the list!")
```

5.9 Summary

- String comparison uses **==, !=, <, >, <=, >=** operators.
- Comparisons are **case-sensitive**; use .lower() or .upper() for **case-insensitive checks**.
- Unicode values (ord()) determine lexicographical order.
- String methods like .startswith(), .endswith(), .isalpha() help in **validation and comparison**.

Topic 6: Slicing and Striding Strings (1/9/25)

6.1 Introduction

- **Strings** in Python are sequences of characters.
- **Slicing** allows us to extract a **part of the string**.
- **Striding** allows us to skip characters or reverse a string.
- Both are **essential for text manipulation, data extraction, and formatting**.

6.2 String Indexing

- Each character in a string has an **index** starting from **0**.
- Negative indices start from **-1** (last character).

```
s = "Python"
print(s[0])   # 'P'
print(s[3])   # 'h'
print(s[-1])  # 'n'
print(s[-3])  # 'h'
```

6.3 Slicing

6.3.1 Syntax

string[start:end]

- **start** → starting index (inclusive)
- **end** → ending index (exclusive)

Example:

```
s = "Python"
print(s[0:4])  # 'Pyth'
print(s[2:5])  # 'tho'
print(s[:3])   # 'Pyt' (from start to index 2)
print(s[3:])   # 'hon' (from index 3 to end)
```

6.3.2 Slicing with Negative Indices

```
s = "Python"
print(s[-6:-3])  # 'Pyt'
print(s[-4:-1])  # 'tho'
```

6.4 Striding (Step)

6.4.1 Syntax

string[start:end:step]

- **step** → number of characters to skip

Examples:

```
s = "Python"
print(s[::2])    # 'Pto' (every 2nd character)
print(s[1::2])   # 'yhn' (start at index 1, every 2nd character)
print(s[::-1])   # 'nohtyP' (reverse string)
print(s[5:0:-2]) # 'nhy' (reverse with step 2)
```

6.4.2 Practical Examples

1. **Extract first 3 letters**

```
word = "Programming"
print(word[:3])  # 'Pro'
```

2. **Extract last 3 letters**

```
print(word[-3:])  # 'ing'
```

3. **Reverse string**

```
print(word[::-1])  # 'gnimmargorP'
```

4. **Skip alternate letters**

```
print(word[::2])   # 'Pormig'
```

6.5 Combining Slicing and String Methods

- You can **slice and then apply methods** like .upper(), .lower().

```
text = "Python Programming"
print(text[:6].upper())  # 'PYTHON'
print(text[7:].lower())  # 'programming'
```

6.6 Example Program

Problem: Extract username from email

```
email = "jyothi123@gmail.com"
username = email[:email.index('@')]
print("Username:", username)
```

Explanation:

- email.index('@') finds the index of '@'.
- Slice from start to that index to extract the username.

6.7 Summary

- **Indexing**: Access single characters.
- **Slicing**: Extract substring using [start:end].
- **Striding**: Skip characters or reverse string using [start:end:step].
- Useful for **text parsing, formatting, and manipulation**.

Topic 7: String Operators and Methods (2/9/25)

7.1 Introduction

- Strings are **sequences of characters** in Python.
- Python provides **operators** to perform operations on strings.
- **Methods** are built-in functions to manipulate and process strings.
- Understanding both is essential for **text processing, formatting, and validation**.

7.2 String Operators

7.2.1 Concatenation (+)

- Joins two or more strings together.

```
first_name = "Jyothi"
last_name = "Kumar"
full_name = first_name + " " + last_name
print(full_name)  # Jyothi Kumar
```

7.2.2 Repetition (*)

- Repeats a string multiple times.

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

7.2.3 Membership Operators (in, not in)

- Check if a substring exists in a string.

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

7.3 String Methods

Python strings are **immutable**, so methods return **new strings**.

7.3.1 Case Conversion

Method	Description
.upper()	Converts to uppercase
.lower()	Converts to lowercase
.title()	Capitalizes first letter of words
.capitalize()	Capitalizes first letter of string

```
s = "python programming"
print(s.upper())       # 'PYTHON PROGRAMMING'
print(s.title())       # 'Python Programming'
```

7.3.2 Whitespace Handling

Method	Description
.strip()	Removes leading & trailing spaces
.lstrip()	Removes left spaces
.rstrip()	Removes right spaces

```
text = "  hello  "
print(text.strip())   # 'hello'
print(text.lstrip())  # 'hello  '
```

7.3.3 Searching and Replacing

Method	Description
.find(sub)	Returns index of first occurrence of sub (-1 if not found)
.replace(old, new)	Replaces occurrences of old with new

```
text = "Python Programming"
print(text.find("Pro"))      # 7
print(text.replace("Python", "Java"))  # 'Java Programming'
```

7.3.4 Splitting and Joining

- **Split:** Break string into a list.

```
s = "Python,Java,C++"
languages = s.split(",")
print(languages)  # ['Python', 'Java', 'C++']
```

- **Join:** Combine list elements into a string.

```
joined = "-".join(languages)
print(joined)  # Python-Java-C++
```

7.3.5 Checking String Content

Method	Description
.isalpha()	True if all characters are letters
.isdigit()	True if all characters are digits
.isalnum()	True if all characters are letters/digits

```
print("Python".isalpha())   # True
print("123".isdigit())      # True
print("Py123".isalnum())    # True
```

7.4 Example Program

Problem: Process user input and format it

```
name = input("Enter your name: ").strip().title()
print("Hello, " + name + "!")

text = "Python programming is fun"
print("Uppercase:", text.upper())
print("Replace Python with Java:", text.replace("Python", "Java"))
print("Words in text:", text.split())
```

Explanation:

- .strip() removes extra spaces.
- .title() capitalizes each word.

- .replace() and .split() manipulate strings efficiently.

7.5 Summary

- **Operators**: +, *, in, not in for combining and checking strings.
- **Methods**: .upper(), .lower(), .strip(), .replace(), .split(), .join(), .isalpha(), .isdigit().
- Strings are **immutable**, so methods always return **new strings**.
- Operators and methods make **text processing simple and powerful**.

Topic 8: String Formatting with str.format (3/9/25)

8.1 Introduction

- **String formatting** allows embedding **variables** and **expressions** into strings.
- Makes output **readable, structured, and dynamic**.
- Python provides multiple ways to format strings:
    1. Using **concatenation (+)**
    2. Using **% formatting**
    3. Using **str.format()** (focus of this topic)
    4. Using **f-strings** (Python 3.6+, advanced)

8.2 Basics of str.format()

- **Syntax:**

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

- Placeholders {} are **replaced by arguments** passed to .format().

Example:

```
name = "Jyothi"
age = 22
print("My name is {} and I am {} years old".format(name, age))
# Output: My name is Jyothi and I am 22 years old
```

8.3 Positional and Indexed Formatting

- Placeholders can be **ordered explicitly** using **index numbers**.

```
print("Name: {0}, Age: {1}".format("Jyothi", 22))
print("Age: {1}, Name: {0}".format("Jyothi", 22))
```

8.4 Keyword Formatting

- Use **named placeholders** for clarity.

```
print("Name: {n}, Age: {a}".format(n="Jyothi", a=22))
```

8.5 Formatting Numbers

8.5.1 Decimal Precision

- Round floats to **specific decimal places**.

```
pi = 3.14159265
print("Value of pi: {:.2f}".format(pi))  # 3.14
```

8.5.2 Padding and Alignment

Alignment Syntax Example

Left < {:<10}

Right > {:>10}

Center ^ {:^10}

```
text = "Hi"
print("{:<10}".format(text))  # 'Hi       '
print("{:>10}".format(text))  # '       Hi'
print("{:^10}".format(text))  # '    Hi    '
```

8.5.3 Number Formatting

- **Integer padding with zeros**

```
num = 7
```

```
print("{:03}".format(num))  # 007
```

- **Thousands separator**

```
num = 1234567
print("{:,}".format(num))   # 1,234,567
```

8.6 Combining Positional, Keyword, and Formatting

```
name = "Jyothi"
marks = 85.456
print("Student {0} scored {m:.1f} marks.".format(name, m=marks))
# Output: Student Jyothi scored 85.5 marks.
```

8.7 Example Program

Problem: Display a formatted report of students

```
students = [("Jyothi", 85), ("Anil", 92), ("Maya", 78)]
print("{:<10} {:>5}".format("Name", "Marks"))
print("-" * 17)

for student in students:
    name, marks = student
    print("{:<10} {:>5}".format(name, marks))
```

Output:

```
Name       Marks
-----------------
Jyothi      85
Anil        92
Maya         78
```

Explanation:

- :<10 → left-align name in 10 spaces.
- :>5 → right-align marks in 5 spaces.

Topic 9: Collections Data Types – Tuples, Lists, Sets, Dictionaries (4/9/25)

9.1 Introduction

- **Collections** in Python store **multiple values** in a single variable.
- Python provides **four main collection types**:
    1. Tuple
    2. List
    3. Set
    4. Dictionary
- Each collection has **unique properties** and is used in **different scenarios**.

9.2 Tuple

9.2.1 Definition

- Ordered collection of elements enclosed in **parentheses ()**.
- **Immutable** – cannot be changed after creation.

```
tpl = (1, 2, 3, "Python")
print(tpl)
```

9.2.2 Accessing Elements

```
print(tpl[0])   # 1
print(tpl[-1])  # Python
```

9.2.3 Operations

```
# Concatenation
tpl2 = tpl + (4, 5)
print(tpl2)  # (1, 2, 3, 'Python', 4, 5)

# Repetition
print(tpl * 2)  # (1, 2, 3, 'Python', 1, 2, 3, 'Python')
```

9.3 List

9.3.1 Definition

- Ordered collection enclosed in **square brackets []**.
- **Mutable** – elements can be added, modified, or removed.

```
lst = [1, 2, 3, "Python"]
```

9.3.2 Access and Modification

```
print(lst[0])   # 1
lst[1] = 20
print(lst)      # [1, 20, 3, 'Python']
```

9.3.3 Common Methods

Method	Description
.append(x)	Add element at end
.insert(i,x)	Add element at index i
.remove(x)	Remove first occurrence of x
.pop()	Remove last element (or index)
.sort()	Sort the list
.reverse()	Reverse the list
.copy()	Copy the list

Example:

```
lst.append(50)
lst.insert(1, 15)
lst.remove(3)
print(lst)
```

9.4 Set

9.4.1 Definition

- Unordered collection enclosed in **curly braces {}**.
- Stores **unique elements**.
- **Mutable**, but no indexing or slicing.

```
st = {1, 2, 3, 2, 3}
print(st)  # {1, 2, 3}  (duplicates removed)
```

9.4.2 Common Methods

Method	Description
.add(x)	Add element
.remove(x)	Remove element (error if not found)
.discard(x)	Remove element (no error if not found)
.union()	Union of two sets
.intersection()	Intersection of two sets

Example:

```
st.add(4)
st.discard(2)
print(st)  # {1, 3, 4}
```

9.5 Dictionary

9.5.1 Definition

- Collection of **key-value pairs** enclosed in **curly braces {}**.
- Keys must be **unique and immutable**.
- Values can be **any type**.

```
dct = {"name": "Jyothi", "age": 22, "course": "Python"}
```

9.5.2 Access and Modification

```
print(dct["name"])  # Jyothi
dct["age"] = 23
dct["grade"] = "A"
```

9.5.3 Common Methods

Method	Description
.keys()	Returns keys
.values()	Returns values
.items()	Returns key-value pairs
.get(key)	Returns value for key
.pop(key)	Removes key-value pair

Example:

```
print(dct.keys())    # dict_keys(['name', 'age', 'course', 'grade'])
print(dct.values())  # dict_values(['Jyothi', 23, 'Python', 'A'])
```

9.6 Summary Table of Collections

Type	Ordered	Mutable	Duplicates	Example
Tuple	Yes	No	Yes	(1, 2, 3)
List	Yes	Yes	Yes	[1, 2, 3]
Set	No	Yes	No	{1, 2, 3}
Dictionary	No	Yes	Keys No	{"a":1,"b":2}

9.7 Example Program

Problem: Store and display student info using different collections

```
# Tuple: Student ID
student_id = (101, 102, 103)
print("Student IDs:", student_id)

# List: Student Names
student_names = ["Jyothi", "Anil", "Maya"]
student_names.append("Zara")
print("Student Names:", student_names)

# Set: Unique Grades
grades = {"A", "B", "A", "C"}
print("Unique Grades:", grades)

# Dictionary: Student Details
student_info = {"name": "Jyothi", "age": 22, "course": "Python"}
print("Student Info:", student_info)
```

9.8 Summary

- **Tuple:** Immutable, ordered collection.
- **List:** Mutable, ordered collection.
- **Set:** Mutable, unordered, unique elements.
- **Dictionary:** Key-value pairs, mutable, keys are unique.

- Choosing the right collection depends on **ordering, mutability, and uniqueness requirements**.

Topic 10: Iterating and Copying Collections (5/9/25)

10.1 Introduction

- **Iteration**: Access each element of a collection **one by one**.
- **Copying**: Create a **duplicate of a collection** to avoid modifying the original.
- Essential for **processing data** and **manipulating collections safely**.

10.2 Iterating Collections

- Python provides **loops** for iterating collections:

10.2.1 Using for loop

List Example:

```
fruits = ["Apple", "Banana", "Mango"]
for fruit in fruits:
   print(fruit)
```

Tuple Example:

```
numbers = (1, 2, 3)
for num in numbers:
   print(num)
```

Set Example (unordered):

```
unique_grades = {"A", "B", "C"}
for grade in unique_grades:
   print(grade)
```

Dictionary Example:

```
student = {"name": "Jyothi", "age": 22}
for key in student:
   print(key, ":", student[key])

# OR using .items()
```

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

10.2.2 Using while loop

- Iterate using **index** for ordered collections.

```
fruits = ["Apple", "Banana", "Mango"]
i = 0
while i < len(fruits):
   print(fruits[i])
   i += 1
```

10.3 Copying Collections

- **Assignment (=)** does **not create a new copy**, it references the same object.
- To copy, use **methods or slicing**.

10.3.1 List Copy

```
lst1 = [1, 2, 3]
lst2 = lst1       # references same list
lst3 = lst1.copy() # creates new copy
lst4 = lst1[:]    # slicing copy

lst1.append(4)
print(lst1)  # [1,2,3,4]
print(lst2)  # [1,2,3,4] (same reference)
print(lst3)  # [1,2,3]   (independent copy)
```

10.3.2 Dictionary Copy

```
d1 = {"a":1, "b":2}
d2 = d1         # same reference
d3 = d1.copy()   # new copy

d1["c"] = 3
print(d1)  # {'a':1,'b':2,'c':3}
print(d2)  # {'a':1,'b':2,'c':3}
print(d3)  # {'a':1,'b':2}
```

10.3.3 Set Copy

```
s1 = {1,2,3}
s2 = s1.copy()
s1.add(4)
print(s1)  # {1,2,3,4}
print(s2)  # {1,2,3}
```

10.4 Nested Collections

- For **nested lists or dictionaries**, use **copy.deepcopy()** for a full copy.

```
import copy

nested_list = [[1,2],[3,4]]
shallow_copy = nested_list.copy()
deep_copy = copy.deepcopy(nested_list)

nested_list[0][0] = 99
print(nested_list)    # [[99,2],[3,4]]
print(shallow_copy)   # [[99,2],[3,4]]  (shared inner list)
print(deep_copy)      # [[1,2],[3,4]]  (independent copy)
```

10.5 Example Program

Problem: Iterate a list of students and copy it for backup

```
students = ["Jyothi", "Anil", "Maya"]

# Iterate
for student in students:
   print("Student:", student)

# Copy
backup_students = students.copy()
students.append("Zara")

print("Original:", students)
print("Backup:", backup_students)
```

Output:

```
Student: Jyothi
Student: Anil
```

Student: Maya
Original: ['Jyothi', 'Anil', 'Maya', 'Zara']
Backup: ['Jyothi', 'Anil', 'Maya']

10.6 Summary

- **Iteration**: Access each element using for or while.
- **Copying**: Use .copy(), slicing, or deepcopy() for nested collections.
- Assignment (=) creates a **reference**, not a copy.
- Safe copying prevents **unintended modification** of original data.

Topic 11: Introduction to PIP (8/9/25)

11.1 Introduction

- **PIP** stands for **"Python Package Installer"**.
- It is a **tool to install and manage Python libraries and packages**.
- PIP makes it easy to **extend Python functionality** by adding third-party modules.

11.2 Why Use PIP?

- Python's **standard library** has many built-in modules, but sometimes we need **external libraries** like:
    - numpy for numerical computations
    - pandas for data analysis
    - matplotlib for plotting
    - requests for HTTP requests
- PIP allows you to **install, update, and uninstall these packages** easily.

11.3 Checking PIP Installation

- PIP comes **pre-installed with Python 3.4+**.
- Check version in command prompt/terminal:

pip --version

Example Output:

pip 23.2.1 from C:\Python39\lib\site-packages\pip (python 3.9)

11.4 Installing Packages

- **Syntax:**

pip install package_name

Example:

pip install numpy

- Installs numpy library and all its dependencies.

11.5 Upgrading Packages

- Keep packages up-to-date using:

pip install --upgrade package_name

Example:

pip install --upgrade pandas

11.6 Uninstalling Packages

- Remove unwanted packages:

pip uninstall package_name

Example:

pip uninstall matplotlib

11.7 Listing Installed Packages

- List all installed packages:

pip list

- Show outdated packages:

```
pip list --outdated
```

11.8 Using Packages in Python

- Once installed, **import the package** in Python scripts:

```
import numpy as np
arr = np.array([1, 2, 3])
print(arr)
```

11.9 Virtual Environments (Optional but Important)

- Virtual environments help **manage separate package versions** for different projects.
- Create a virtual environment:

```
python -m venv myenv
```

- Activate it:
  - Windows: myenv\Scripts\activate
  - Linux/Mac: source myenv/bin/activate
- Install packages **inside the virtual environment** using PIP.

11.10 Example Workflow

1. Check if PIP is installed:

```
pip --version
```

2. Install a package:

```
pip install requests
```

3. Use it in Python:

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

4. Upgrade package if needed:

```
pip install --upgrade requests
```

5. Uninstall package:

```
pip uninstall requests
```

11.11 Summary

- **PIP** is a **package manager** for Python.
- Allows **installing, upgrading, uninstalling, and listing packages**.
- Works seamlessly with **virtual environments** to avoid version conflicts.
- Essential for **Python development and project management**.