MES Abasaheb Garware College (Autonomous), Karve Road, Pune

F.Y.B.Sc. (Data Science) Semester-I

DS-112-PR

Practical course in Python Programming

Work Book

To be implemented from Academic Year 2025–2026

Name: _____

College Name:_____

Roll No.: _____

Division: _____

Academic Year: _____

Introduction

1. About the work book:

This workbook is intended to be used by **F.Y.B.Sc. (Data Science)** students for the Principals of Programming Languages Assignments in **Semester–I**. This workbook is designed by considering all the practical concepts / topics mentioned in syllabus.

2. The objectives of this workbook are:

1) Defining the scope of the course.
2) To bring the uniformity in the practical conduction and implementation in all colleges affiliated to SPPU.
3) To have continuous assessment of the course and students.
4) Providing ready reference for the students during practical implementation.
5) Provide more options to students so that they can have good practice before facing the examination.
6) Catering to the demand of slow and fast learners and accordingly providing the practice assignments to them.

3. How to use this workbook:

1) The workbook is divided into two sections. Section-I is related to Python assignments and Section-II is related to Principles of Programming languages.
2) The **Section-I (Python)** is divided into **03 assignments.** Each Python assignment has three SETs. It is mandatory for students to complete the SET A and SET B in given slot.
3) The **Section-II (Principles of Programming Languages)** is divided into **03 assignments**.

4. Instructions to the students

1) Students are expected to carry this book every time they come to the lab for practice.
2) Students should prepare oneself beforehand for the Assignment by reading the relevant material.
3) Instructor will specify which problems to solve in the lab during the allotted slot and student should complete them and get verified by the instructor. However student should spend additional hours in Lab and at home to cover as many problems as possible given in this work book.
4) Students will be assessed for each exercise on a scale from 0 to 5.

\Not Done	**0**
Incomplete	**1**
Late Complete	**2**
Needs Improvement	**3**
Complete	**4**
Well Done	**5**

5. Guidelines for Instructors

1) Explain the assignment and related concepts in a round ten minutes if required or by demonstrating the software.
2) You should evaluate each assignment carried out by a student on a scale of 5 as specified above by ticking appropriate box.
3) The value should also be entered on assignment completion page of the respective Lab course.

6. Guidelines for Lab administrator

You have to ensure appropriate hardware and software is made available to each student.

Assignment Completion Sheet

Sr. No.	Assignment Name	Marks (out of 5)	Signature
1	Basic Python		
2	Python Strings		
3	Python Tuple		
4	Python Set		
5	Python Dictionary		
6	Functions in Python		
7	Object Oriented Programming in Python		
	Total (out of 35)		
	Total (out of 15)		

CERTIFICATE

This is to certify that Mr./Ms._____ has successfully completed **F.Y.B.Sc. (Data Science) DS-112-PR Practical course in Python Programming Semester I** course in year _____and his/her seat no. is _____.

He / She have scored _____ marks out of 15.

Instructor **Internal Examiner** **External Examiner**

ASSIGNMENT NO.1: BASIC PYTHON

Introduction to Python

Python is a general purpose, dynamic, high level and interpreted programming language. It supports Object Oriented programming approach to develop applications. It is simple and easy to learn. It also provides high level data structures.

The implementation of Python was started in December 1989 by Guido Van Rossum at CWI in Netherland.

Starting the Interpreter

- After installation, the python interpreter lives in the installed directory. By default, it is /usr/local/bin/pythonX.X in Linux/Unix and C:\PythonXX in Windows, where the 'X' denotes the version number.
- To run Python from the command prompt, we need to add this location to the search path.
- Search path is a list of directories (locations) where the operating system searches for executables.

Use of Python

Python is used by hundreds of thousands of programmers worldwide. Python has many standard libraries which come with Python when it is installed. On the Internet there are many other libraries available that make Python even more powerful.

Some things that Python is often used for are:

- Web development
- AI & Machine learning
- Game programming
- Desktop GUI Applications
- Software Development
- Business Applications
- 3D CAD Applications
- Scientific programming
- Network programming

First Python Program

This is a small example of a Python program. It shows "Hello World!" on the screen.

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

Save this as `helloWorld.py` and run it using:

```
python helloWorld.py
```

Output:

```
Hello, world!
```

In this program we have used the built-in function `print()` to print out a string to the screen.

Python Programming Modes

1. Immediate/Interactive Mode

Typing `python` in the command line will invoke the interpreter in immediate mode. We can directly type in Python expressions and press enter to get the output.

python

```
>>> 1 + 1
2
>>> print("Hello World")
Hello World
```

The `>>>` is the Python prompt. It tells us that the interpreter is ready for our input. To exit this mode type `exit()` or `quit()` and press enter.

2. Script Mode Programming

Write Python code in a file with `.py` extension and run it using the interpreter.

Example: Create a file `test.py`

python

```
print("Hello Python!")
```

Run it:

```
$ python test.py
```

Output:

```
Hello Python!
```

3. Integrated Development Environment (IDE)

We can use any text editing software to write a Python script file. But using an IDE can make our life a lot easier. IDE provides useful features like code hinting, syntax highlighting and checking, file explorers etc.

Popular Python IDEs:

- IDLE (comes with Python)

- PyCharm
- VS Code
- Jupyter Notebook
- Spyder

Python Comments

Comments are used to explain code and make it more readable.

Single-line comment:

Comments are created by beginning a line with the hash (#) character.

python

```
# This is a single line comment
print("Hello")  # This is also a comment
```

Multi-line comment:

Comments that span multiple lines are created by using triple quotes (""" or ''').

python

```
"""
This is a multiline comment
It can span several lines
and describes your code in detail
"""
```

Indentation

Python uses indentation to define blocks of code. The enforcement of indentation makes the code look neat and clean.

Correct Example:

python

```
if True:
    print('Hello')
    a = 5
```

Incorrect indentation will result in IndentationError.

Python Data Types

Atomic Data Types

Python has following atomic data types:

- **Integers (int)** - Whole numbers like 5, 100, -50
- **Floating point (float)** - Decimal numbers like 3.14, -0.5
- **Strings (str)** - Text like "Hello", 'Python'
- **Boolean (bool)** - True or False

Python Numbers

Python supports different numerical types:

1. int (signed integers)

python

```
a = 10
b = -50
c = 0
```

2. float (floating point real values)

python

```
x = 3.14
y = -2.5
z = 0.0
```

3. complex (complex numbers)

python

```
c = 3 + 4j
```

Python Strings

Strings in Python are identified as a contiguous set of characters represented in quotation marks. Python allows for either pairs of single or double quotes.

python

```
str1 = 'Hello all'
str2 = "Python Programming"
str3 = '''Multi-line
string'''
```

Python Lists

Lists are the most versatile compound data types. A list contains items of different data types separated by commas and enclosed within square brackets ([]).

python

```
list_obj = ['table', 59, 2.69, "chair"]
fruits = ['apple', 'banana', 'orange']
numbers = [1, 2, 3, 4, 5]
```

Python Tuples

A tuple is an immutable sequence data type similar to the list. A tuple consists of values separated by commas and enclosed in parentheses (()).

python

```
tuple_obj = (786, 2.23, "college")
coordinates = (10, 20)
```

Python Dictionary

Python dictionaries are key-value pairs. They work like hash tables and are enclosed by curly braces ({}).

python

```
dict_obj = {'roll_no': 15, 'name': 'xyz', 'per': 69.88}
student = {'name': 'John', 'age': 20, 'grade': 'A'}
```

Python Operators

i. Arithmetic Operators

Operator	Description	Example
+	Addition	5 + 3 = 8
-	Subtraction	5 - 3 = 2
*	Multiplication	5 * 3 = 15
/	Division	5 / 2 = 2.5
%	Modulus	5 % 2 = 1
**	Exponent	2 ** 3 = 8
//	Floor Division	9 // 2 = 4

Floor Division Examples:

python

```
9 // 2 = 4
9.0 // 2.0 = 4.0
-11 // 3 = -4
```

ii. Logical Operators

Operator	Description	Example
and	True if both operands are true	x and y
or	True if either operand is true	x or y
not	True if operand is false	not x

iii. Relational/Comparison Operators

Operator	Description	Example
==	Equal to	a == b
!=	Not equal to	a != b
<	Less than	a < b
<=	Less than or equal to	a <= b
>	Greater than	a > b
>=	Greater than or equal to	a >= b

iv. Assignment Operators

python

```
=    # Simple assignment
+=   # Add and assign
-=   # Subtract and assign
*=   # Multiply and assign
/=   # Divide and assign
%=   # Modulus and assign
**=  # Exponent and assign
//=  # Floor division and assign
```

v. Bitwise Operators

Operator	Description
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
~	Bitwise NOT
<<	Left shift
>>	Right shift

vi. Membership Operators

Operator	Description	Example
in	True if value found in sequence	x in list
not in	True if value not found	x not in list

vii. Identity Operators

Operator	Description	Example
is	True if operands are identical	x is y
is not	True if operands not identical	x is not y

Decision Making Statements

i. If Statement

python

```
if expression:
   statement(s)
```

Example:

python

```
age = 18
if age >= 18:
   print("You are eligible to vote")
```

ii. If...Else Statements

python

```
if expression:
   statement(s)
else:
   statement(s)
```

Example:

python

```
number = 10
if number % 2 == 0:
   print("Even number")
else:
   print("Odd number")
```

iii. If...Elif...Else Statements

python

```
if expression1:
   statement(s)
elif expression2:
```

```
    statement(s)
else:
    statement(s)
```

Example:

python

```
marks = 75
if marks >= 90:
    print("Grade A")
elif marks >= 80:
    print("Grade B")
elif marks >= 70:
    print("Grade C")
else:
    print("Grade D")
```

iv. Nested If Statements

You can have an if...elif...else construct inside another if...elif...else construct.

python

```
if expression1:
    statement(s)
    if expression2:
        statement(s)
    elif expression3:
        statement(s)
    else:
        statement(s)
elif expression4:
    statement(s)
else:
    statement(s)
```

Python Loops

i. While Loop

A while loop repeatedly executes statements as long as a given condition is true.

Syntax:

python

```
while condition:
    statement(s)
```

Example:

python

```
count = 0
while count < 3:
    print('The count is:', count)
    count = count + 1
```

Output:

```
The count is: 0
The count is: 1
The count is: 2
```

ii. For Loop

A for loop has the ability to iterate over the items of any sequence, such as a list or a string.

Syntax:

python

```
for variable in sequence:
    statement(s)
```

Example:

python

```
fruits = ['apple', 'banana', 'orange']
for fruit in fruits:
    print(fruit)
```

Output:

```
apple
banana
orange
```

Using range():

python

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

Output:

```
0
1
2
3
4
```

Question Set A

1) Write a Python Program to Calculate the Average of Numbers in a Given List.

2) Write a program which accepts 6 integer values and prints "DUPLICATES" if any of the values entered are duplicates otherwise it prints "ALL UNIQUE".

Example: Let 6 integers are (32, 10, 45, 90, 45, 6) then output "DUPLICATES" to be printed.

3) Write a program which accepts an integer value as input and print "Ok" if value is between 1 to 50 (both inclusive) otherwise it prints "Out of range".

4) Write a program which finds sum of digits of a number.

Example: n=130 then output is 4 (1+3+0).

5) Write a program which prints Fibonacci series of a number.

Question Set B

1) Write a program which accept an integer value 'n' and display all prime numbers till 'n'.

2) Write a program that accept two integer values and if both are equal then prints "SAME identity" otherwise prints "DIFFERENT identity".

3) Write a program to display following pattern:

```
1 2 3 4
1 2 3
1 2
1
```

4) Write a program to reverse a given number.

Question Set C

1) Write a Sequential search function which searches an item in a sorted list. The function should return the index of element to be searched in the list.

Assignment Evaluation

0: Not Done [ ]
1: Incomplete [ ]
2: Late Complete [ ]
3: Needs Improvement [ ]
4: Complete [ ]
5: Well Done [ ]

Signature of Instructor: _____

ASSIGNMENT NO.2: PYTHON STRINGS

Introduction to Strings

Strings in Python are sequences of characters enclosed in quotes. Strings are immutable, which means once created, they cannot be changed. Python provides many built-in methods to work with strings.

Creating Strings

Strings can be created using single quotes, double quotes, or triple quotes.

python

```
# Single quotes
str1 = 'Hello World'

# Double quotes
str2 = "Python Programming"

# Triple quotes (for multi-line strings)
str3 = '''This is a
multi-line
string'''

str4 = """Another way to
create multi-line
string"""
```

Accessing Characters in String

Individual characters in a string can be accessed using indexing. Python uses zero-based indexing.

python

```
message = "Python"

# Accessing characters
print(message[0])  # Output: P
print(message[1])  # Output: y
print(message[5])  # Output: n
```

```
# Negative indexing (from end)
print(message[-1])  # Output: n
print(message[-2])  # Output: o
```

String Slicing

Slicing allows you to extract a portion of a string.

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

python

```
text = "Python Programming"

# Basic slicing
print(text[0:6])   # Output: Python
print(text[7:])    # Output: Programming
print(text[:6])    # Output: Python

# Negative slicing
print(text[-11:])  # Output: Programming

# Step slicing
print(text[::2])   # Output: Pto rgamn
print(text[::-1])  # Output: gnimmargorP nohtyP (reverse)
```

String Concatenation

Strings can be joined together using the + operator.

python

```
first_name = "John"
last_name = "Doe"

# Concatenation
full_name = first_name + " " + last_name
print(full_name)  # Output: John Doe

# Using * for repetition
message = "Hello! "
print(message * 3)  # Output: Hello! Hello! Hello!
```

String Methods

Python provides many built-in string methods. Here are some commonly used ones:

Case Conversion Methods

python

```
text = "Python Programming"

print(text.upper())      # PYTHON PROGRAMMING
print(text.lower())      # python programming
print(text.capitalize()) # Python programming
print(text.title())      # Python Programming
print(text.swapcase())   # pYTHON pROGRAMMING
```

Search and Replace Methods

python

```
sentence = "Python is easy to learn"

# find() - returns index of substring, -1 if not found
print(sentence.find("easy"))    # Output: 10
print(sentence.find("hard"))    # Output: -1

# index() - returns index, raises error if not found
print(sentence.index("easy"))   # Output: 10

# count() - counts occurrences
print(sentence.count("e"))      # Output: 2

# replace() - replaces substring
new_sentence = sentence.replace("easy", "fun")
print(new_sentence)  # Python is fun to learn
```

Checking Methods

python

```
# isalpha() - checks if all characters are alphabets
print("Python".isalpha())    # True
print("Python3".isalpha())   # False
```

```
# isdigit() - checks if all characters are digits
print("123".isdigit())      # True
print("12a".isdigit())      # False

# isalnum() - checks if alphanumeric
print("Python3".isalnum())  # True
print("Python 3".isalnum())  # False

# isspace() - checks if all are whitespace
print("   ".isspace())      # True
print(" a ".isspace())      # False

# startswith() and endswith()
text = "Python Programming"
print(text.startswith("Python"))  # True
print(text.endswith("ing"))       # True
```

Trimming Methods

python

```
text = "   Python Programming   "

# strip() - removes leading and trailing whitespace
print(text.strip())      # "Python Programming"

# lstrip() - removes leading whitespace
print(text.lstrip())     # "Python Programming   "

# rstrip() - removes trailing whitespace
print(text.rstrip())     # "   Python Programming"
```

Splitting and Joining

python

```
# split() - splits string into list
sentence = "Python is easy to learn"
words = sentence.split()
print(words)  # ['Python', 'is', 'easy', 'to', 'learn']

csv_data = "apple,banana,orange"
```

```
fruits = csv_data.split(",")
print(fruits)  # ['apple', 'banana', 'orange']
# join() - joins list into string
words = ['Python', 'is', 'fun']
sentence = " ".join(words)
print(sentence)  # "Python is fun"

# Joining with different separator
print("-".join(words))  # "Python-is-fun"
```

String Formatting

1. Using % operator (old style)

python

```
name = "John"
age = 25
message = "My name is %s and I am %d years old" % (name, age)
print(message)
# Output: My name is John and I am 25 years old
```

2. Using format() method

python

```
name = "John"
age = 25
message = "My name is {} and I am {} years old".format(name, age)
print(message)

# With index
message = "I am {1} years old. My name is {0}".format(name, age)
print(message)

# With named arguments
message = "My name is {n} and I am {a} years old".format(n=name, a=age)
print(message)
```

3. Using f-strings (Python 3.6+)

python

```python
name = "John"
age = 25

message = f"My name is {name} and I am {age} years old"
print(message)

# With expressions
print(f"Next year I will be {age + 1} years old")
```

String Length

Use the `len()` function to get the length of a string.

python

```python
text = "Python Programming"
print(len(text))  # Output: 18
```

Iterating Through Strings

python

```python
# Using for loop
text = "Python"
for char in text:
    print(char)

# With index
for i in range(len(text)):
    print(f"Character at index {i}: {text[i]}")
```

Escape Characters

Special characters in strings are represented using escape sequences.

python

```python
# Common escape characters
print("Hello\nWorld")      # New line
print("Hello\tWorld")      # Tab
print("Hello\\World")      # Backslash
print("He said, \"Hi\"")   # Double quote
print('It\'s Python')      # Single quote
```

String Comparison

Strings can be compared using comparison operators.

python

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

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

Checking Substring

python

```python
text = "Python Programming"

# Using 'in' operator
print("Python" in text)     # True
print("Java" in text)       # False

# Using 'not in' operator
print("Java" not in text)   # True
```

Question Set A

1) Write a program to count the number of vowels in a given string.

2) Write a program to check if a string is a palindrome (reads the same forwards and backwards).

Example: "madam" is a palindrome, "python" is not.

3) Write a program to reverse a string without using built-in reverse functions.

4) Write a program to count the frequency of each character in a string.

Example: For "hello", output should be h:1, e:1, l:2, o:1

5) Write a program to remove all whitespace from a string.

Question Set B

1) Write a program to find the first non-repeating character in a string.

2) Write a program to check if two strings are anagrams (contain same characters in different order).

Example: "listen" and "silent" are anagrams.

3) Write a program to convert a string to title case without using the title() method.

Example: "python programming" should become "Python Programming"

4) Write a program to replace all occurrences of a substring with another substring in a given string.

Question Set C

1) Write a program to find the longest word in a sentence.

2) Write a program to check if a string contains only digits, only alphabets, or both.

Assignment Evaluation

0: Not Done [ ]
1: Incomplete [ ]
2: Late Complete [ ]
3: Needs Improvement [ ]
4: Complete [ ]
5: Well Done [ ]

Signature of Instructor: _____

ASSIGNMENT NO.3: PYTHON TUPLE

Introduction to Tuples

A tuple is a collection data type in Python that is ordered and immutable. Tuples are similar to lists, but unlike lists, tuples cannot be changed after creation. Tuples are written with round brackets ().

Creating Tuples

python

```
# Empty tuple
empty_tuple = ()

# Tuple with integers
numbers = (1, 2, 3, 4, 5)

# Tuple with mixed data types
mixed = (1, "Hello", 3.14, True)

# Tuple with one element (note the comma)
single = (5,)

# Without parentheses (tuple packing)
coordinates = 10, 20, 30

# Using tuple() constructor
my_tuple = tuple([1, 2, 3])
```

Note: To create a tuple with a single element, you must include a trailing comma. Without the comma, Python will not recognize it as a tuple.

python

```
not_a_tuple = (5)     # This is just an integer
actual_tuple = (5,)   # This is a tuple
```

Accessing Tuple Elements

Indexing

Tuples use zero-based indexing like lists.

python

```python
fruits = ("apple", "banana", "orange", "mango")

# Positive indexing
print(fruits[0])  # apple
print(fruits[1])  # banana
print(fruits[3])  # mango

# Negative indexing
print(fruits[-1])  # mango
print(fruits[-2])  # orange
```

Slicing

You can extract portions of a tuple using slicing.

python

```python
numbers = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

print(numbers[2:5])    # (2, 3, 4)
print(numbers[:4])     # (0, 1, 2, 3)
print(numbers[5:])     # (5, 6, 7, 8, 9)
print(numbers[-3:])    # (7, 8, 9)
print(numbers[::2])    # (0, 2, 4, 6, 8)
print(numbers[::-1])   # (9, 8, 7, 6, 5, 4, 3, 2, 1, 0)
```

Tuple Operations

Concatenation

python

```python
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)

result = tuple1 + tuple2
print(result)  # (1, 2, 3, 4, 5, 6)
```

Repetition

python

```python
numbers = (1, 2)
```

```
repeated = numbers * 3
print(repeated)  # (1, 2, 1, 2, 1, 2)
```

Membership Testing

python

```
fruits = ("apple", "banana", "orange")

print("apple" in fruits)     # True
print("mango" in fruits)     # False
print("grape" not in fruits) # True
```

Tuple Methods

Tuples have only two built-in methods because they are immutable.

1. count()

Returns the number of times a value appears in the tuple.

python

```
numbers = (1, 2, 3, 2, 4, 2, 5)
count = numbers.count(2)
print(count)  # 3
```

2. index()

Returns the index of the first occurrence of a value.

python

```
fruits = ("apple", "banana", "orange", "banana")
index = fruits.index("banana")
print(index)  # 1

# If element not found, it raises ValueError
# print(fruits.index("mango"))  # ValueError
```

Tuple Length

Use `len()` function to get the number of elements in a tuple.

python

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

```
print(len(numbers))  # 5
```

Iterating Through Tuples

Using for loop

```
fruits = ("apple", "banana", "orange")
for fruit in fruits:
    print(fruit)
```

With index

```
fruits = ("apple", "banana", "orange")
for i in range(len(fruits)):
    print(f"Index {i}: {fruits[i]}")
```

Using enumerate()

python

```
fruits = ("apple", "banana", "orange")
for index, fruit in enumerate(fruits):
    print(f"{index}: {fruit}")
```

Tuple Unpacking

Tuple unpacking allows you to assign tuple elements to multiple variables.

```
point = (10, 20)
x, y = point
print(x)  # 10
print(y)  # 20


# Unpacking with multiple values
person = ("John", 25, "Engineer")
name, age, profession = person
print(name)      # John
print(age)       # 25
print(profession)  # Engineer
# Using * for remaining values
numbers = (1, 2, 3, 4, 5)
first, *middle, last = numbers
print(first)   # 1
```

```
print(middle)  # [2, 3, 4]
print(last)    # 5
```

Nested Tuples

Tuples can contain other tuples.

```
nested = ((1, 2), (3, 4), (5, 6))
# Accessing nested elements
print(nested[0])      # (1, 2)
print(nested[0][0])   # 1
print(nested[1][1])   # 4
```

Converting Between Tuples and Lists

```
# List to tuple
my_list = [1, 2, 3, 4]
my_tuple = tuple(my_list)
print(my_tuple)  # (1, 2, 3, 4)


# Tuple to list
my_tuple = (1, 2, 3, 4)
my_list = list(my_tuple)
print(my_list)  # [1, 2, 3, 4]
```

Immutability of Tuples

Tuples cannot be modified after creation. You cannot add, remove, or change elements.

```
numbers = (1, 2, 3, 4)
# These operations will raise errors:
# numbers[0] = 10       # TypeError
# numbers.append(5)     # AttributeError
# del numbers[0]        # TypeError
```

However, if a tuple contains mutable objects like lists, those objects can be modified:

```
my_tuple = (1, 2, [3, 4])
my_tuple[2][0] = 99
print(my_tuple)  # (1, 2, [99, 4])
```

Advantages of Tuples

1. **Immutable**: Once created, cannot be changed (data protection)
2. **Faster**: Tuples are faster than lists
3. **Can be used as dictionary keys**: Unlike lists
4. **Safe**: Data cannot be accidentally modified

Common Tuple Functions

python

```python
numbers = (5, 2, 8, 1, 9, 3)

# max() - returns maximum value
print(max(numbers))  # 9

# min() - returns minimum value
print(min(numbers))  # 1

# sum() - returns sum of all elements
print(sum(numbers))  # 28

# sorted() - returns sorted list (not tuple)
print(sorted(numbers))  # [1, 2, 3, 5, 8, 9]
```

Tuple Comparison

Tuples can be compared using comparison operators. Python compares tuples element by element.

```python
tuple1 = (1, 2, 3)
tuple2 = (1, 2, 4)
tuple3 = (1, 2, 3)
print(tuple1 == tuple3)  # True
print(tuple1 < tuple2)   # True
print(tuple1 > tuple2)   # False
```

Question Set A

1) Write a program to find the maximum and minimum element in a tuple without using max() and min() functions.

2) Write a program to count the occurrence of each element in a tuple.

Example: For tuple (1, 2, 3, 2, 1, 3, 2), output should show 1:2, 2:3, 3:2

3) Write a program to convert a tuple of strings into a single string.

Example: ("Hello", "World", "Python") should become "HelloWorldPython"

4) Write a program to find the sum of all even numbers in a tuple.

5) Write a program to remove duplicates from a tuple (return a new tuple with unique elements).

Question Set B

1) Write a program to find the second largest element in a tuple.

2) Write a program to sort a tuple of tuples based on the second element.

Example: ((1, 5), (3, 2), (2, 8)) should become ((3, 2), (1, 5), (2, 8))

3) Write a program to check if all elements in a tuple are unique.

4) Write a program to swap the first and last element of a tuple.

Question Set C

1) Write a program to find common elements between two tuples.

2) Write a program to create a tuple of squares of numbers from 1 to n.

Example: If n=5, output should be (1, 4, 9, 16, 25)

Assignment Evaluation

0: Not Done [ ]
1: Incomplete [ ]
2: Late Complete [ ]
3: Needs Improvement [ ]
4: Complete [ ]
5: Well Done [ ]

Signature of Instructor: _____

ASSIGNMENT NO.4: PYTHON SET

Introduction to Sets

A set is an unordered collection of unique elements in Python. Sets are mutable (can be modified), but they do not allow duplicate values. Sets are defined using curly braces {} or the set() function.

Key Features of Sets

- **Unordered**: Elements have no specific order
- **Unique**: No duplicate elements allowed
- **Mutable**: Can add or remove elements
- **Unindexed**: Cannot access elements by index

Creating Sets

python

```python
# Empty set
empty_set = set()  # Note: {} creates an empty dictionary

# Set with integers
numbers = {1, 2, 3, 4, 5}

# Set with mixed data types
mixed = {1, "Hello", 3.14, True}

# Set from a list (duplicates removed automatically)
my_list = [1, 2, 2, 3, 3, 3, 4]
my_set = set(my_list)
print(my_set)  # {1, 2, 3, 4}

# Set from a string
char_set = set("hello")
print(char_set)  # {'h', 'e', 'l', 'o'}
```

Note: Empty curly braces {} create an empty dictionary, not a set. Use `set()` to create an empty set.

Accessing Set Elements

Since sets are unordered and unindexed, you cannot access elements using index numbers. However, you can loop through the set or check if an element exists.

python

```python
fruits = {"apple", "banana", "orange"}

# Looping through set
for fruit in fruits:
    print(fruit)

# Checking membership
print("apple" in fruits)     # True
print("mango" in fruits)     # False
print("grape" not in fruits) # True
```

Adding Elements to a Set

add()

Adds a single element to the set.

python

```python
fruits = {"apple", "banana"}
fruits.add("orange")
print(fruits) # {'apple', 'banana', 'orange'}

# Adding duplicate (no effect)
fruits.add("apple")
print(fruits) # {'apple', 'banana', 'orange'}
```

update()

Adds multiple elements from any iterable (list, tuple, set, etc.).

python

```python
fruits = {"apple", "banana"}
fruits.update(["orange", "mango", "grape"])
print(fruits) # {'apple', 'banana', 'orange', 'mango', 'grape'}

# Update with multiple iterables
fruits.update(["kiwi"], ("pear",))
print(fruits)
```

Removing Elements from a Set

remove()

Removes the specified element. Raises KeyError if element doesn't exist.

python

```python
fruits = {"apple", "banana", "orange"}
fruits.remove("banana")
print(fruits)  # {'apple', 'orange'}

# fruits.remove("mango")  # KeyError
```

discard()

Removes the specified element. Does NOT raise error if element doesn't exist.

python

```python
fruits = {"apple", "banana", "orange"}
fruits.discard("banana")
print(fruits)  # {'apple', 'orange'}

fruits.discard("mango")  # No error
print(fruits)
```

pop()

Removes and returns a random element (since sets are unordered).

python

```python
fruits = {"apple", "banana", "orange"}
removed = fruits.pop()
print(f"Removed: {removed}")
print(fruits)
```

clear()

Removes all elements from the set.

python

```python
fruits = {"apple", "banana", "orange"}
fruits.clear()
print(fruits)  # set()
```

Set Operations

Union

Combines all elements from two or more sets (removes duplicates).

python

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}

# Using | operator
result = set1 | set2
print(result)  # {1, 2, 3, 4, 5}

# Using union() method
result = set1.union(set2)
print(result)  # {1, 2, 3, 4, 5}

# Union of multiple sets
set3 = {5, 6, 7}
result = set1.union(set2, set3)
print(result)  # {1, 2, 3, 4, 5, 6, 7}
```

Intersection

Returns elements common to all sets.

python

```
set1 = {1, 2, 3, 4}
set2 = {3, 4, 5, 6}

# Using & operator
result = set1 & set2
print(result)  # {3, 4}

# Using intersection() method
result = set1.intersection(set2)
print(result)  # {3, 4}
```

Difference

Returns elements in first set but not in other sets.

python

```python
set1 = {1, 2, 3, 4}
set2 = {3, 4, 5, 6}

# Using - operator
result = set1 - set2
print(result)  # {1, 2}

# Using difference() method
result = set1.difference(set2)
print(result)  # {1, 2}
```

Symmetric Difference

Returns elements in either set, but not in both.

python

```python
set1 = {1, 2, 3, 4}
set2 = {3, 4, 5, 6}

# Using ^ operator
result = set1 ^ set2
print(result)  # {1, 2, 5, 6}

# Using symmetric_difference() method
result = set1.symmetric_difference(set2)
print(result)  # {1, 2, 5, 6}
```

Set Comparison Methods

issubset()

Checks if all elements of one set are in another.

python

```python
set1 = {1, 2}
set2 = {1, 2, 3, 4}

print(set1.issubset(set2))  # True
```

```
print(set2.issubset(set1))  # False


# Using <= operator
print(set1 <= set2)  # True
```

issuperset()

Checks if one set contains all elements of another.

python

```
set1 = {1, 2, 3, 4}
set2 = {1, 2}

print(set1.issuperset(set2))  # True
print(set2.issuperset(set1))  # False

# Using >= operator
print(set1 >= set2)  # True
```

isdisjoint()

Checks if two sets have no common elements.

python

```
set1 = {1, 2, 3}
set2 = {4, 5, 6}
set3 = {3, 4, 5}

print(set1.isdisjoint(set2))  # True
print(set1.isdisjoint(set3))  # False
```

Set Methods Summary

Method	Description
add()	Adds an element to the set
clear()	Removes all elements
copy()	Returns a shallow copy
difference()	Returns difference of sets
discard()	Removes specified element
intersection()	Returns intersection of sets
isdisjoint()	Checks if sets have no common elements
issubset()	Checks if set is subset
issuperset()	Checks if set is superset

Method	Description
pop()	Removes random element
remove()	Removes specified element
symmetric_difference()	Returns symmetric difference
union()	Returns union of sets
update()	Updates set with another set

Frozen Sets

Frozen sets are immutable versions of sets. Once created, they cannot be modified.

python

```python
# Creating frozen set
frozen = frozenset([1, 2, 3, 4])
print(frozen)  # frozenset({1, 2, 3, 4})

# Frozen sets can be used as dictionary keys
my_dict = {frozen: "value"}

# Operations that work
print(len(frozen))
print(2 in frozen)

# Operations that don't work
# frozen.add(5)      # AttributeError
# frozen.remove(1)   # AttributeError
```

Set Comprehension

Similar to list comprehension, sets can be created using set comprehension.

python

```python
# Squares of numbers
squares = {x**2 for x in range(1, 6)}
print(squares)  # {1, 4, 9, 16, 25}

# Even numbers
evens = {x for x in range(10) if x % 2 == 0}
print(evens)  # {0, 2, 4, 6, 8}

# Characters from string
```

```
chars = {char.upper() for char in "hello"}
print(chars)  # {'H', 'E', 'L', 'O'}
```

Common Set Functions

```
numbers = {5, 2, 8, 1, 9}
# len() - number of elements
print(len(numbers))  # 5
# max() - maximum element
print(max(numbers))  # 9
# min() - minimum element
print(min(numbers))  # 1
# sum() - sum of all elements
print(sum(numbers))  # 25
# sorted() - returns sorted list
print(sorted(numbers))  # [1, 2, 5, 8, 9]
```

Practical Applications of Sets

1. Remove Duplicates

python

```
numbers = [1, 2, 2, 3, 3, 3, 4, 4, 5]
unique = list(set(numbers))
print(unique)  # [1, 2, 3, 4, 5]
```

2. Finding Common Elements

python

```
list1 = [1, 2, 3, 4, 5]
list2 = [4, 5, 6, 7, 8]
common = set(list1) & set(list2)
print(common)  # {4, 5}
```

3. Finding Unique Elements

python

```
list1 = [1, 2, 3, 4, 5]
list2 = [4, 5, 6, 7, 8]
unique_to_first = set(list1) - set(list2)
print(unique_to_first)  # {1, 2, 3}
```

Question Set A

1) Write a program to find the union, intersection, and difference of two sets entered by the user.

2) Write a program to check if a set is a subset of another set.

3) Write a program to remove duplicates from a list using sets.

4) Write a program to find elements that are present in the first set but not in the second set.

5) Write a program to check if two sets are disjoint (have no common elements).

Question Set B

1) Write a program to find symmetric difference between two sets.

2) Write a program to add multiple elements to a set at once from user input.

3) Write a program to find the length of a set and check if a specific element exists in it.

Question Set C

1) Write a program that takes two sentences as input and finds:

- Common words in both sentences
- Words unique to the first sentence
- Words unique to the second sentence

2) Write a program to perform all set operations (union, intersection, difference, symmetric difference) on three sets.

Assignment Evaluation

0: Not Done [ ]
1: Incomplete [ ]
2: Late Complete [ ]
3: Needs Improvement [ ]
4: Complete [ ]
5: Well Done [ ]

Signature of Instructor: _____

ASSIGNMENT NO.5: PYTHON DICTIONARY

Introduction to Dictionaries

A dictionary in Python is an unordered collection of key-value pairs. Each key is unique and is used to access its corresponding value. Dictionaries are mutable, meaning they can be modified after creation.

Key Features of Dictionaries

- **Key-Value Pairs**: Data stored as pairs
- **Unordered**: No specific order (in Python 3.7+, insertion order is maintained)
- **Mutable**: Can be modified after creation
- **Unique Keys**: Each key must be unique
- **Keys must be immutable**: Strings, numbers, tuples can be keys

Creating Dictionaries

python

```python
# Empty dictionary
empty_dict = {}
# or
empty_dict = dict()

# Dictionary with data
student = {
    "name": "John",
    "age": 20,
    "grade": "A"
}

# Dictionary with different data types
mixed_dict = {
    "name": "Alice",
    "marks": [85, 90, 88],
    "passed": True,
    1: "First"
}

# Using dict() constructor
person = dict(name="Bob", age=25, city="Mumbai")
```

```
print(person)  # {'name': 'Bob', 'age': 25, 'city': 'Mumbai'}

# Dictionary from list of tuples
pairs = [("a", 1), ("b", 2), ("c", 3)]
my_dict = dict(pairs)
print(my_dict)  # {'a': 1, 'b': 2, 'c': 3}
```

Accessing Dictionary Elements

Using Keys

python

```
student = {
   "name": "John",
   "age": 20,
   "grade": "A"
}

# Direct access
print(student["name"])  # John
print(student["age"])   # 20

# Using get() method (safer - doesn't raise error)
print(student.get("name"))   # John
print(student.get("city"))   # None (no error)
print(student.get("city", "Not Found"))  # Not Found (default value)
```

Note: Using `dict[key]` raises KeyError if key doesn't exist. Using `get()` returns None or a default value instead.

Adding and Modifying Elements

python

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

# Adding new key-value pair
student["grade"] = "A"
print(student)  # {'name': 'John', 'age': 20, 'grade': 'A'}

# Modifying existing value
student["age"] = 21
```

```python
print(student)  # {'name': 'John', 'age': 21, 'grade': 'A'}

# Using update() to add/modify multiple items
student.update({"city": "Mumbai", "grade": "A+"})
print(student)
```

Removing Elements

del statement

python

```python
student = {"name": "John", "age": 20, "grade": "A"}

# Remove specific key
del student["age"]
print(student)  # {'name': 'John', 'grade': 'A'}

# Delete entire dictionary
# del student
```

pop()

Removes and returns the value of specified key.

python

```python
student = {"name": "John", "age": 20, "grade": "A"}

age = student.pop("age")
print(age)     # 20
print(student)  # {'name': 'John', 'grade': 'A'}

# With default value
city = student.pop("city", "Not Found")
print(city)  # Not Found
```

popitem()

Removes and returns the last inserted key-value pair (in Python 3.7+).

python

```python
student = {"name": "John", "age": 20, "grade": "A"}
```

```
item = student.popitem()
print(item)    # ('grade', 'A')
print(student)  # {'name': 'John', 'age': 20}
```

clear()

Removes all items from dictionary.

python

```
student = {"name": "John", "age": 20}
student.clear()
print(student)  # {}
```

Dictionary Methods

keys()

Returns all keys in the dictionary.

python

```
student = {"name": "John", "age": 20, "grade": "A"}

keys = student.keys()
print(keys)  # dict_keys(['name', 'age', 'grade'])

# Convert to list
keys_list = list(student.keys())
print(keys_list)  # ['name', 'age', 'grade']
```

values()

Returns all values in the dictionary.

python

```
student = {"name": "John", "age": 20, "grade": "A"}

values = student.values()
print(values)  # dict_values(['John', 20, 'A'])

# Convert to list
values_list = list(student.values())
print(values_list)  # ['John', 20, 'A']
```

items()

Returns all key-value pairs as tuples.

python

```python
student = {"name": "John", "age": 20, "grade": "A"}

items = student.items()
print(items)  # dict_items([('name', 'John'), ('age', 20), ('grade', 'A')])

# Convert to list
items_list = list(student.items())
print(items_list)
```

copy()

Creates a shallow copy of the dictionary.

python

```python
original = {"name": "John", "age": 20}
duplicate = original.copy()

duplicate["name"] = "Alice"
print(original)   # {'name': 'John', 'age': 20}
print(duplicate)  # {'name': 'Alice', 'age': 20}
```

Iterating Through Dictionaries

Iterate through keys

python

```python
student = {"name": "John", "age": 20, "grade": "A"}

# Method 1
for key in student:
    print(key, student[key])

# Method 2
for key in student.keys():
    print(key, student[key])
```

Iterate through values

python

```python
for value in student.values():
    print(value)
```

Iterate through key-value pairs

python

```python
for key, value in student.items():
    print(f"{key}: {value}")
```

Nested Dictionaries

Dictionaries can contain other dictionaries.

python

```python
students = {
    "student1": {
        "name": "John",
        "age": 20,
        "grade": "A"
    },
    "student2": {
        "name": "Alice",
        "age": 21,
        "grade": "B"
    }
}

# Accessing nested values
print(students["student1"]["name"])  # John
print(students["student2"]["grade"])  # B

# Iterating through nested dictionary
for student_id, info in students.items():
    print(f"\n{student_id}:")
    for key, value in info.items():
        print(f"  {key}: {value}")
```

Dictionary Comprehension

Create dictionaries using a concise syntax.

python

```python
# Squares dictionary
squares = {x: x**2 for x in range(1, 6)}
print(squares)  # {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

# With condition
even_squares = {x: x**2 for x in range(10) if x % 2 == 0}
print(even_squares)  # {0: 0, 2: 4, 4: 16, 6: 36, 8: 64}

# From two lists
keys = ['a', 'b', 'c']
values = [1, 2, 3]
my_dict = {k: v for k, v in zip(keys, values)}
print(my_dict)  # {'a': 1, 'b': 2, 'c': 3}
```

Checking for Keys

python

```python
student = {"name": "John", "age": 20, "grade": "A"}

# Using 'in' operator
if "name" in student:
    print("Name exists")

if "city" not in student:
    print("City doesn't exist")

# Using get() with condition
age = student.get("age")
if age:
    print(f"Age is {age}")
```

Dictionary Functions

python

```python
marks = {"Math": 85, "Science": 90, "English": 88}
```

```python
# len() - number of key-value pairs
print(len(marks))  # 3

# max() - maximum key
print(max(marks))  # Science (alphabetically)

# min() - minimum key
print(min(marks))  # English

# max value
print(max(marks.values()))  # 90

# sorted() - sorted keys
print(sorted(marks))  # ['English', 'Math', 'Science']
```

Dictionary Methods Summary

Method	Description
clear()	Removes all items
copy()	Returns shallow copy
get(key, default)	Returns value for key
items()	Returns key-value pairs
keys()	Returns all keys
pop(key)	Removes and returns value
popitem()	Removes last inserted pair
setdefault(key, default)	Returns value, sets if not exists
update(dict)	Updates dictionary
values()	Returns all values

fromkeys() Method

Creates a new dictionary with specified keys and values.

python

```python
# With same value for all keys
keys = ["name", "age", "grade"]
default_dict = dict.fromkeys(keys, "Unknown")
print(default_dict)
# {'name': 'Unknown', 'age': 'Unknown', 'grade': 'Unknown'}
```

```python
# Without value (defaults to None)
keys = ["a", "b", "c"]
my_dict = dict.fromkeys(keys)
print(my_dict)  # {'a': None, 'b': None, 'c': None}
```

setdefault() Method

Returns value if key exists, otherwise inserts key with default value.

python

```python
student = {"name": "John", "age": 20}
# Key exists
name = student.setdefault("name", "Unknown")
print(name)  # John
# Key doesn't exist
grade = student.setdefault("grade", "A")
print(grade)  # A
print(student)  # {'name': 'John', 'age': 20, 'grade': 'A'}
```

Merging Dictionaries

Using update()

python

```python
dict1 = {"a": 1, "b": 2}
dict2 = {"c": 3, "d": 4}

dict1.update(dict2)
print(dict1)  # {'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

Using | operator (Python 3.9+)

python

```python
dict1 = {"a": 1, "b": 2}
dict2 = {"c": 3, "d": 4}
merged = dict1 | dict2
print(merged)  # {'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

Using unpacking

```python
dict1 = {"a": 1, "b": 2}
dict2 = {"c": 3, "d": 4}
```

```
merged = {**dict1, **dict2}
print(merged)  # {'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

Question Set A

1) Write a program to create a dictionary with roll number as key and name as value for 5 students. Display all key-value pairs.

2) Write a program to count the frequency of each word in a given sentence using a dictionary.

Example: "hello world hello" should give {'hello': 2, 'world': 1}

3) Write a program to merge two dictionaries entered by the user.

4) Write a program to find the key with the maximum value in a dictionary.

5) Write a program to check if a given key exists in a dictionary.

Question Set B

1) Write a program to remove a key from a dictionary and handle the case when key doesn't exist.

2) Write a program to sort a dictionary by its values in descending order.

3) Write a program to create a dictionary from two lists - one containing keys and another containing values.

Question Set C

1) Write a program to find all keys with the same value in a dictionary.

2) Write a program to create a dictionary where keys are numbers from 1 to n and values are their cubes.

Assignment Evaluation

0: Not Done [ ]
1: Incomplete [ ]
2: Late Complete [ ]
3: Needs Improvement [ ]
4: Complete [ ]
5: Well Done [ ]

Signature of Instructor: _____

ASSIGNMENT NO.6: FUNCTIONS IN PYTHON

Introduction to Functions

A function is a block of organized, reusable code that performs a specific task. Functions help in breaking our program into smaller and modular chunks. As our program grows larger, functions make it more organized and manageable.

Benefits of Functions

- **Code Reusability**: Write once, use many times
- **Modularity**: Break complex problems into smaller parts
- **Readability**: Makes code easier to understand
- **Maintainability**: Easier to fix bugs and add features
- **Avoid Repetition**: Follow DRY (Don't Repeat Yourself) principle

Defining a Function

Functions are defined using the `def` keyword.

Syntax:

python

```
def function_name(parameters):
    """docstring"""
    statement(s)
    return value
```

Example:

python

```
def greet():
    """This function prints a greeting message"""
    print("Hello! Welcome to Python")


# Calling the function
greet()
```

Function with Parameters

Parameters allow you to pass data to functions.

python

```
def greet(name):
```

```
    """Function with one parameter"""
    print(f"Hello, {name}!")

greet("John")   # Output: Hello, John!
greet("Alice")  # Output: Hello, Alice!
```

Multiple Parameters

python

```
def add_numbers(a, b):
    """Function to add two numbers"""
    result = a + b
    print(f"Sum of {a} and {b} is {result}")

add_numbers(5, 3)  # Output: Sum of 5 and 3 is 8
add_numbers(10, 20)  # Output: Sum of 10 and 20 is 30
```

Return Statement

The `return` statement is used to return a value from a function.

python

```
def add(a, b):
    """Returns sum of two numbers"""
    return a + b

result = add(5, 3)
print(result)  # Output: 8

# Multiple return values (as tuple)
def get_student_info():
    name = "John"
    age = 20
    grade = "A"
    return name, age, grade

name, age, grade = get_student_info()
print(f"{name}, {age}, {grade}")
```

Types of Arguments

1. Positional Arguments

Arguments passed in the correct positional order.

python

```python
def describe_person(name, age, city):
    print(f"{name} is {age} years old and lives in {city}")

describe_person("John", 25, "Mumbai")
# Output: John is 25 years old and lives in Mumbai
```

2. Keyword Arguments

Arguments passed with parameter names.

python

```python
def describe_person(name, age, city):
    print(f"{name} is {age} years old and lives in {city}")

describe_person(age=25, city="Mumbai", name="John")
# Output: John is 25 years old and lives in Mumbai

# Mix of positional and keyword
describe_person("John", city="Mumbai", age=25)
```

3. Default Arguments

Parameters with default values.

python

```python
def greet(name, message="Hello"):
    print(f"{message}, {name}!")

greet("John")          # Output: Hello, John!
greet("Alice", "Hi")   # Output: Hi, Alice!
greet("Bob", message="Hey")  # Output: Hey, Bob!
```

4. Variable-length Arguments

*args (Non-keyword Variable Arguments)

Allows passing any number of positional arguments.

python

```python
def sum_all(*numbers):
    """Sum all numbers passed"""
    total = 0
    for num in numbers:
        total += num
    return total

print(sum_all(1, 2, 3))        # 6
print(sum_all(1, 2, 3, 4, 5))  # 15
print(sum_all(10, 20))         # 30
```

**kwargs (Keyword Variable Arguments)

Allows passing any number of keyword arguments.

python

```python
def display_info(**info):
    """Display all information passed"""
    for key, value in info.items():
        print(f"{key}: {value}")

display_info(name="John", age=25, city="Mumbai")
# Output:
# name: John
# age: 25
# city: Mumbai

display_info(roll=101, grade="A", marks=85)
```

Combining All Types

python

```python
def my_function(a, b, *args, default=10, **kwargs):
    print(f"a: {a}")
    print(f"b: {b}")
    print(f"args: {args}")
    print(f"default: {default}")
    print(f"kwargs: {kwargs}")
```

```
my_function(1, 2, 3, 4, 5, default=20, x=100, y=200)
```

Scope of Variables

Local Variables

Variables defined inside a function.

python

```
def my_function():
    x = 10  # Local variable
    print(x)

my_function()  # Output: 10
# print(x)  # Error: x is not defined outside function
```

Global Variables

Variables defined outside all functions.

python

```
x = 10  # Global variable

def my_function():
    print(x)  # Can access global variable

my_function()  # Output: 10
print(x)      # Output: 10
```

Using global Keyword

To modify global variables inside a function.

python

```
x = 10

def modify():
    global x
    x = 20
    print(f"Inside function: {x}")

print(f"Before: {x}")  # 10
```

```
modify()          # Inside function: 20
print(f"After: {x}")  # 20
```

Lambda Functions (Anonymous Functions)

Small anonymous functions defined using `lambda` keyword.

Syntax: `lambda arguments: expression`

python

```
# Regular function
def square(x):
    return x ** 2

# Equivalent lambda function
square = lambda x: x ** 2

print(square(5))  # Output: 25

# Lambda with multiple arguments
add = lambda a, b: a + b
print(add(3, 5))  # Output: 8

# Lambda with multiple parameters
multiply = lambda x, y, z: x * y * z
print(multiply(2, 3, 4))  # Output: 24
```

Lambda with map()

python

```
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x**2, numbers))
print(squared)  # [1, 4, 9, 16, 25]
```

Lambda with filter()

python

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
evens = list(filter(lambda x: x % 2 == 0, numbers))
print(evens)  # [2, 4, 6, 8, 10]
```

Recursive Functions

A function that calls itself.

python

```python
def factorial(n):
    """Calculate factorial recursively"""
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n - 1)

print(factorial(5))  # Output: 120

# Fibonacci using recursion
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(7))  # Output: 13
```

Docstrings

Documentation strings that explain what the function does.

python

```python
def calculate_area(length, width):
    """
    Calculate area of a rectangle.

    Parameters:
    length (float): Length of rectangle
    width (float): Width of rectangle

    Returns:
    float: Area of rectangle
    """
    return length * width
```

```
# Access docstring
print(calculate_area.__doc__)
```

Built-in Functions

Python provides many built-in functions:

python

```
# Mathematical functions
print(abs(-5))      # 5
print(pow(2, 3))    # 8
print(round(3.14159, 2))  # 3.14

# Type conversion
print(int("10"))    # 10
print(float("3.14"))  # 3.14
print(str(100))     # "100"

# Sequence functions
numbers = [1, 2, 3, 4, 5]
print(len(numbers))  # 5
print(max(numbers))  # 5
print(min(numbers))  # 1
print(sum(numbers))  # 15

# Input/Output
name = input("Enter name: ")
print(f"Hello, {name}")
```

Function Best Practices

1. **Use descriptive names**: `calculate_total()` instead of `calc()`
2. **One task per function**: Each function should do one thing well
3. **Keep it short**: Ideally less than 20-30 lines
4. **Add docstrings**: Document what your function does
5. **Avoid global variables**: Pass data through parameters
6. **Return values**: Instead of printing, return values when possible

Question Set A

1) Write a function to find the maximum of three numbers.

2) Write a function to check if a number is prime or not.

3) Write a function to calculate the factorial of a number (both iterative and recursive).

4) Write a function that accepts a list and returns a new list with unique elements.

5) Write a function to calculate the sum of all even numbers in a list.

Question Set B

1) Write a function to convert temperature from Celsius to Fahrenheit and vice versa.

2) Write a function that accepts a string and returns the count of vowels and consonants.

3) Write a function that accepts variable number of arguments and returns their average.

4) Write a lambda function to find the maximum of three numbers.

Question Set C

1) Write a recursive function to calculate the sum of digits of a number.

2) Write a function that accepts a list of numbers and returns a dictionary with count of even and odd numbers.

Example: [1,2,3,4,5] should return {'even': 2, 'odd': 3}

Assignment Evaluation

0: Not Done [ ]
1: Incomplete [ ]
2: Late Complete [ ]
3: Needs Improvement [ ]
4: Complete [ ]
5: Well Done [ ]

Signature of Instructor: _____

ASSIGNMENT NO.7: OBJECT ORIENTED PROGRAMMING IN PYTHON

Introduction to OOP

Object-Oriented Programming (OOP) is a programming approach based on objects and classes. An object is simply a collection of data (variables) and methods (functions) that act on that data. A class is a blueprint for the object.

Key Concepts of OOP

1. **Class**: Blueprint or template for creating objects
2. **Object**: Instance of a class
3. **Encapsulation**: Bundling data and methods together
4. **Inheritance**: Creating new classes from existing ones
5. **Polymorphism**: Same interface for different data types
6. **Abstraction**: Hiding complex implementation details

Classes and Objects

Defining a Class

python

```python
class Student:
    """A simple Student class"""
    pass


# Creating an object
student1 = Student()
print(student1)  # <__main__.Student object at 0x...>
```

Class with Attributes

python

```python
class Student:
    # Class attribute (shared by all instances)
    school = "ABC School"

    def __init__(self, name, roll_no):
        # Instance attributes (unique to each instance)
        self.name = name
        self.roll_no = roll_no
```

```
# Creating objects
student1 = Student("John", 101)
student2 = Student("Alice", 102)

print(student1.name)     # John
print(student2.name)     # Alice
print(student1.school)   # ABC School
```

The init Method (Constructor)

The `__init__` method is called automatically when an object is created. It is used to initialize the object's attributes.

python

```
class Car:
    def __init__(self, brand, model, year):
        self.brand = brand
        self.model = model
        self.year = year
        print(f"New car created: {brand} {model}")

car1 = Car("Toyota", "Camry", 2020)
# Output: New car created: Toyota Camry

print(car1.brand)  # Toyota
print(car1.model)  # Camry
print(car1.year)   # 2020
```

Instance Methods

Methods that work on instance attributes.

python

```
class Student:
    def __init__(self, name, marks):
        self.name = name
        self.marks = marks

    def display_info(self):
        """Display student information"""
```

```
        print(f"Name: {self.name}")
        print(f"Marks: {self.marks}")

    def calculate_grade(self):
        """Calculate and return grade"""
        if self.marks >= 90:
            return "A"
        elif self.marks >= 80:
            return "B"
        elif self.marks >= 70:
            return "C"
        else:
            return "D"

student = Student("John", 85)
student.display_info()
# Output:
# Name: John
# Marks: 85

grade = student.calculate_grade()
print(f"Grade: {grade}")  # Grade: B
```

Class Methods

Methods that work on class attributes. Defined using `@classmethod` decorator.

python

```
class Student:
    school = "ABC School"  # Class attribute
    student_count = 0

    def __init__(self, name):
        self.name = name
        Student.student_count += 1

    @classmethod
    def get_school(cls):
        return cls.school
```

```
    @classmethod
    def get_student_count(cls):
        return cls.student_count

# Using class method
print(Student.get_school())  # ABC School

student1 = Student("John")
student2 = Student("Alice")
print(Student.get_student_count())  # 2
```

Static Methods

Methods that don't use instance or class attributes. Defined using `@staticmethod` decorator.

python

```
class MathOperations:
    @staticmethod
    def add(a, b):
        return a + b

    @staticmethod
    def multiply(a, b):
        return a * b

# Using static method (no need to create object)
print(MathOperations.add(5, 3))      # 8
print(MathOperations.multiply(4, 2))  # 8
```

Encapsulation

Restricting direct access to some of an object's components.

Public, Protected, and Private Members

python

```
class BankAccount:
    def __init__(self, account_number, balance):
        self.account_number = account_number  # Public
        self._balance = balance               # Protected (convention)
```

```
        self.__pin = 1234               # Private (name mangling)

    def get_balance(self):
        """Public method to access protected attribute"""
        return self._balance

    def __validate_pin(self, pin):
        """Private method"""
        return pin == self.__pin

    def withdraw(self, amount, pin):
        if self.__validate_pin(pin):
            if amount <= self._balance:
                self._balance -= amount
                return True
        return False

account = BankAccount("12345", 1000)
print(account.account_number)   # 12345 (accessible)
print(account.get_balance())    # 1000 (through method)
# print(account.__pin)          # Error: AttributeError
```

Inheritance

Creating a new class from an existing class.

Single Inheritance

python

```
# Parent class
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        print(f"{self.name} makes a sound")

# Child class
class Dog(Animal):
    def speak(self):
```

```
        print(f"{self.name} barks")

class Cat(Animal):
    def speak(self):
        print(f"{self.name} meows")

# Using inheritance
dog = Dog("Tommy")
dog.speak()  # Tommy barks

cat = Cat("Kitty")
cat.speak()  # Kitty meows
```

Using super()

Access parent class methods and attributes.

python

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def display(self):
        print(f"Name: {self.name}, Age: {self.age}")

class Student(Person):
    def __init__(self, name, age, roll_no):
        super().__init__(name, age)  # Call parent constructor
        self.roll_no = roll_no

    def display(self):
        super().display()  # Call parent method
        print(f"Roll No: {self.roll_no}")

student = Student("John", 20, 101)
student.display()
# Output:
# Name: John, Age: 20
```

```
# Roll No: 101
```

Multiple Inheritance

A class can inherit from multiple parent classes.

python

```
class Father:
    def skills(self):
        print("Programming, Gardening")

class Mother:
    def skills(self):
        print("Cooking, Art")

class Child(Father, Mother):
    def skills(self):
        print("My skills:")
        Father.skills(self)
        Mother.skills(self)

child = Child()
child.skills()
# Output:
# My skills:
# Programming, Gardening
# Cooking, Art
```

Multilevel Inheritance

Chain of inheritance.

python

```
class GrandParent:
    def display(self):
        print("GrandParent class")

class Parent(GrandParent):
    def show(self):
        print("Parent class")
```

```python
class Child(Parent):
    def view(self):
        print("Child class")


child = Child()
child.display()  # From GrandParent
child.show()     # From Parent
child.view()     # From Child
```

Polymorphism

Method Overriding

Child class provides specific implementation of parent class method.

python

```python
class Shape:
    def area(self):
        pass


class Rectangle(Shape):
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width


class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius ** 2

# Polymorphism in action
shapes = [Rectangle(5, 3), Circle(4)]

for shape in shapes:
    print(f"Area: {shape.area()}")
```

```
# Output:
# Area: 15
# Area: 50.24
```

Operator Overloading

Define custom behavior for operators.

python

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        """Overload + operator"""
        return Point(self.x + other.x, self.y + other.y)

    def __str__(self):
        """Overload str() function"""
        return f"({self.x}, {self.y})"

p1 = Point(1, 2)
p2 = Point(3, 4)
p3 = p1 + p2  # Uses __add__ method

print(p3)  # (4, 6) - Uses __str__ method
```

Special Methods (Magic Methods)

Methods with double underscores (dunder methods).

python

```
class Book:
    def __init__(self, title, pages):
        self.title = title
        self.pages = pages

    def __str__(self):
        """String representation"""
        return f"{self.title} ({self.pages} pages)"
```

```
    def __len__(self):
        """Length of the book"""
        return self.pages

    def __eq__(self, other):
        """Equality comparison"""
        return self.pages == other.pages

    def __lt__(self, other):
        """Less than comparison"""
        return self.pages < other.pages

book1 = Book("Python Basics", 200)
book2 = Book("Advanced Python", 300)

print(book1)        # Python Basics (200 pages)
print(len(book1))   # 200
print(book1 == book2)  # False
print(book1 < book2)   # True
```

Property Decorators

Getters and setters using @property decorator.

python

```
class Temperature:
    def __init__(self):
        self._celsius = 0

    @property
    def celsius(self):
        """Getter for celsius"""
        return self._celsius

    @celsius.setter
    def celsius(self, value):
        """Setter for celsius"""
        if value < -273.15:
```

```
            raise ValueError("Temperature below absolute zero")
        self._celsius = value

    @property
    def fahrenheit(self):
        """Convert celsius to fahrenheit"""
        return (self._celsius * 9/5) + 32

temp = Temperature()
temp.celsius = 25
print(temp.celsius)     # 25
print(temp.fahrenheit)  # 77.0

# temp.celsius = -300   # ValueError
```

Abstract Classes

Classes that cannot be instantiated and require subclasses to implement methods.

python

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def sound(self):
        pass

    @abstractmethod
    def move(self):
        pass

class Dog(Animal):
    def sound(self):
        return "Bark"

    def move(self):
        return "Run"

# animal = Animal()  # Error: Cannot instantiate abstract class
```

```
dog = Dog()
print(dog.sound())  # Bark
print(dog.move())   # Run
```

Class Attributes vs Instance Attributes

python

```
class Student:
    school = "ABC School"  # Class attribute

    def __init__(self, name):
        self.name = name   # Instance attribute

student1 = Student("John")
student2 = Student("Alice")

# Instance attributes are unique
print(student1.name)  # John
print(student2.name)  # Alice

# Class attribute is shared
print(student1.school)  # ABC School
print(student2.school)  # ABC School

# Changing class attribute
Student.school = "XYZ School"
print(student1.school)  # XYZ School
print(student2.school)  # XYZ School
```

Question Set A

1) Create a class called `Rectangle` with attributes length and width. Add methods to calculate area and perimeter.

2) Create a class `BankAccount` with methods to deposit, withdraw, and display balance.

3) Create a parent class `Vehicle` and child classes `Car` and `Bike` that inherit from it. Override a method in child classes.

4) Create a class `Student` with private attributes and public methods to access them.

5) Create a class `Circle` with method to calculate area. Overload the `+` operator to add two circles (add their radii).

Question Set B

1) Create a class hierarchy: `Animal` -> `Mammal` -> `Dog`. Demonstrate multilevel inheritance.

2) Create a class `Employee` with class method to keep count of total employees.

3) Create a class `Book` with special methods `__str__`, `__len__`, and `__eq__`.

4) Create a class `Temperature` with property decorators for Celsius and Fahrenheit conversion.

Question Set C

1) Create an abstract class `Shape` with abstract method `area()`. Create subclasses `Rectangle`, `Circle`, and `Triangle` that implement the area method.

2) Create a class `Library` that manages a collection of books. Implement methods to add books, remove books, search books, and display all books.

Assignment Evaluation

0: Not Done [ ]
1: Incomplete [ ]
2: Late Complete [ ]
3: Needs Improvement [ ]
4: Complete [ ]
5: Well Done [ ]

Signature of Instructor: _____

End of Lab Workbook

Note for Students:

1. Complete all question sets for each assignment
2. Test your programs thoroughly with different inputs
3. Follow Python naming conventions and coding standards
4. Add comments to explain your code
5. Submit assignments on time