SRM Institute of Science and Technology

Department of Computer Applications

Delhi – Meerut Road, Sikri Kalan, Ghaziabad, Uttar Pradesh – 201204

Circular – 2020-21

MCA GAI 2nd semester

Python Programming for Data Science (PG120C04J)- Lab Manual

This manual provides a structured guide for practical exercises in Python Programming for Data Science. Each lab aims to reinforce theoretical concepts through hands-on coding.

Lab 1: Simple Programs

Title

Introduction to Python: Simple Programs

Aim

To write and execute basic Python programs covering fundamental data types, variables, and simple arithmetic operations.

Procedure

1. Open a Python IDE (e.g., PyCharm, VS Code with Python extension, or a simple text editor and command prompt).
2. Create a new Python file (e.g., `lab1_program1.py`).
3. Write the source code for a simple program (e.g., adding two numbers, printing "Hello, World!").
4. Save the file.
5. Execute the program from the terminal using `python your_program_name.py` or run it directly from your IDE.
6. Observe the output and verify it matches the expected output.
7. Experiment with different inputs and operations.

Source Code

```
# Program 1.1: Hello World
print("Hello, World!")

# Program 1.2: Addition of two numbers
num1 = 10
num2 = 25
sum_result = num1 + num2
print(f"The sum of {num1} and {num2} is: {sum_result}")

# Program 1.3: Taking user input and performing a simple operation
name = input("Enter your name: ")
```

```
age = int(input("Enter your age: "))
print(f"Hello, {name}! You are {age} years old.")
print(f"Next year, you will be {age + 1} years old.")
```

Input

```
# For Program 1.3:
Enter your name: Alice
Enter your age: 30
```

Expected Output

```
# For Program 1.1:
Hello, World!

# For Program 1.2:
The sum of 10 and 25 is: 35

# For Program 1.3:
Hello, Alice! You are 30 years old.
Next year, you will be 31 years old.
```

Lab 2: Programs Using Tuples, List, Dictionary and Sets

Title

Data Structures in Python: Tuples, Lists, Dictionaries, and Sets

Aim

To understand and implement core Python data structures: tuples, lists, dictionaries, and sets, and perform common operations on them.

Procedure

1. For each data structure (tuple, list, dictionary, set), create a new Python file or section within a file.
2. Define examples of each data structure.
3. Perform operations like:
    - **Lists:** Appending, inserting, removing elements, slicing, iteration.
    - **Tuples:** Accessing elements, slicing (demonstrate immutability).
    - **Dictionaries:** Adding/accessing key-value pairs, updating values, iterating through keys/values/items.
    - **Sets:** Adding/removing elements, union, intersection, difference operations.
4. Print the results of each operation to observe the changes.

Source Code

```python
# List operations
my_list = [1, 2, 3, 4, 5]
print(f"Original List: {my_list}")
my_list.append(6)
print(f"After append(6): {my_list}")
my_list.remove(2)
print(f"After remove(2): {my_list}")
print(f"Slice [1:4]: {my_list[1:4]}")

# Tuple operations
my_tuple = (10, 20, 30, 40, 50)
print(f"\nOriginal Tuple: {my_tuple}")
print(f"Element at index 2: {my_tuple[2]}")
print(f"Slice [1:3]: {my_tuple[1:3]}")
# my_tuple[0] = 5 # Uncommenting this will raise an error (immutability)

# Dictionary operations
my_dict = {"name": "John", "age": 30, "city": "New York"}
print(f"\nOriginal Dictionary: {my_dict}")
my_dict["age"] = 31
print(f"After updating age: {my_dict}")
my_dict["occupation"] = "Engineer"
print(f"After adding occupation: {my_dict}")
print(f"Keys: {my_dict.keys()}")
print(f"Values: {my_dict.values()}")

# Set operations
set1 = {1, 2, 3, 4, 5}
set2 = {4, 5, 6, 7, 8}
print(f"\nSet 1: {set1}")
print(f"Set 2: {set2}")
print(f"Union: {set1.union(set2)}")
print(f"Intersection: {set1.intersection(set2)}")
print(f"Difference (set1 - set2): {set1.difference(set2)}")
```

Input

No specific user input is required for this program as values are hardcoded.

Expected Output

```
Original List: [1, 2, 3, 4, 5]
After append(6): [1, 2, 3, 4, 5, 6]
After remove(2): [1, 3, 4, 5, 6]
Slice [1:4]: [3, 4, 5]

Original Tuple: (10, 20, 30, 40, 50)
Element at index 2: 30
Slice [1:3]: (20, 30)

Original Dictionary: {'name': 'John', 'age': 30, 'city': 'New York'}
After updating age: {'name': 'John', 'age': 31, 'city': 'New York'}
After adding occupation: {'name': 'John', 'age': 31, 'city': 'New York',
'occupation': 'Engineer'}
Keys: dict_keys(['name', 'age', 'city', 'occupation'])
Values: dict_values(['John', 31, 'New York', 'Engineer'])

Set 1: {1, 2, 3, 4, 5}
Set 2: {4, 5, 6, 7, 8}
Union: {1, 2, 3, 4, 5, 6, 7, 8}
Intersection: {4, 5}
Difference (set1 - set2): {1, 2, 3}
```

Lab 3: Illustration on Lambda and Filters

Title

Functional Programming Concepts: Lambda Functions and Filters

Aim

To understand and apply lambda functions for creating anonymous functions and the `filter()` function for selective data processing.

Procedure

1. Define a simple lambda function that performs a basic operation (e.g., squaring a number, adding two numbers).
2. Use the `filter()` function with a lambda expression to filter elements from a list based on a specific condition (e.g., even numbers, strings starting with a specific letter).
3. Print the results to observe the output of both lambda and filter operations.

Source Code

```python
# Lambda function examples
square = lambda x: x * x
print(f"Square of 5 using lambda: {square(5)}")

add = lambda a, b: a + b
print(f"Sum of 10 and 20 using lambda: {add(10, 20)}")

# Filter function with lambda
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
# Filter even numbers
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(f"Original numbers: {numbers}")
print(f"Even numbers using filter: {even_numbers}")

# Filter strings longer than 5 characters
words = ["apple", "banana", "cat", "dog", "elephant", "frog"]
long_words = list(filter(lambda word: len(word) > 5, words))
print(f"Original words: {words}")
print(f"Words longer than 5 characters: {long_words}")
```

Input

No specific user input is required.

Expected Output

```
Square of 5 using lambda: 25
Sum of 10 and 20 using lambda: 30
Original numbers: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Even numbers using filter: [2, 4, 6, 8, 10]
Original words: ['apple', 'banana', 'cat', 'dog', 'elephant', 'frog']
Words longer than 5 characters: ['banana', 'elephant']
```

Lab 4: Implementing Inheritance

Title

Object-Oriented Programming: Implementing Inheritance

Aim

To understand and implement the concept of inheritance in Python, demonstrating how a child class can inherit properties and behaviors from a parent class.

Procedure

1. Define a parent class with some attributes and methods.
2. Define a child class that inherits from the parent class.
3. In the child class, add new attributes and methods, and optionally override parent class methods.
4. Create instances of both the parent and child classes.
5. Call methods and access attributes from both parent and child instances to demonstrate inheritance.

Source Code

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

    def make_sound(self):
        return "Generic animal sound"

    def display_info(self):
        return f"Name: {self.name}, Species: {self.species}"

# Child class inheriting from Animal
class Dog(Animal):
    def __init__(self, name, breed):
        # Call the parent class constructor
        super().__init__(name, "Dog")
        self.breed = breed

    # Override the make_sound method
    def make_sound(self):
        return "Woof! Woof!"

    # New method specific to Dog
    def fetch(self):
        return f"{self.name} is fetching the ball."

# Create instances
animal1 = Animal("Leo", "Lion")
dog1 = Dog("Buddy", "Golden Retriever")

# Demonstrate parent class functionality
print(f"Animal 1 Info: {animal1.display_info()}")
print(f"Animal 1 Sound: {animal1.make_sound()}")

# Demonstrate child class functionality (inherited and new)
print(f"\nDog 1 Info: {dog1.display_info()}") # Inherited method
print(f"Dog 1 Sound: {dog1.make_sound()}")   # Overridden method
print(f"Dog 1 Action: {dog1.fetch()}")       # New method
print(f"Dog 1 Breed: {dog1.breed}")          # New attribute
```

Input

No specific user input is required.

Expected Output

```
Animal 1 Info: Name: Leo, Species: Lion
Animal 1 Sound: Generic animal sound

Dog 1 Info: Name: Buddy, Species: Dog
Dog 1 Sound: Woof! Woof!
Dog 1 Action: Buddy is fetching the ball.
Dog 1 Breed: Golden Retriever
```

Lab 5: Implementing Method Overloading

Title

Object-Oriented Programming: Implementing Method Overloading (Pythonic Way)

Aim

To understand the concept of method overloading and implement it in Python using default arguments or variable-length arguments, as Python does not support traditional method overloading.

Procedure

1. Define a class with a method that needs to perform different actions based on the number or type of arguments.
2. Implement "method overloading" using:
   - **Default arguments:** Provide default values for parameters.
   - **Variable-length arguments (`*args`, `**kwargs`):** Handle an arbitrary number of positional or keyword arguments.
   - **Type checking (if necessary):** Use `isinstance()` to check argument types within the method.
3. Call the method with different argument combinations to demonstrate its varied behavior.

Source Code

```
class Calculator:
    def add(self, a, b=None, c=None):
        """
        Simulates method overloading for addition.
        Can add two or three numbers.
        """
        if b is None and c is None:
            return a
        elif c is None:
            return a + b
        else:
            return a + b + c

    def display_message(self, message, prefix="Info", suffix="."):
        """
        Demonstrates method overloading using default arguments.
        """
        return f"[{prefix}] {message}{suffix}"

    def process_data(self, *args):
        """
        Demonstrates method overloading using *args (variable positional
arguments).
        Can sum numbers or concatenate strings.
        """
        if all(isinstance(arg, (int, float)) for arg in args):
            return sum(args)
        elif all(isinstance(arg, str) for arg in args):
            return " ".join(args)
        else:
            return "Unsupported data types for processing."


calc = Calculator()
```

```
# Demonstrate add method
print(f"Add (one arg): {calc.add(5)}")
print(f"Add (two args): {calc.add(5, 10)}")
print(f"Add (three args): {calc.add(5, 10, 15)}")

# Demonstrate display_message method
print(f"\nMessage 1: {calc.display_message('Operation completed')}")
print(f"Message 2: {calc.display_message('Error occurred', prefix='ERROR',
suffix='!')}")

# Demonstrate process_data method
print(f"\nProcess Data (numbers): {calc.process_data(1, 2, 3, 4)}")
print(f"Process Data (strings): {calc.process_data('Hello', 'world', 'this',
'is', 'Python')}")
print(f"Process Data (mixed types): {calc.process_data(1, 'apple', 3)}")
```

Input

No specific user input is required.

Expected Output

```
Add (one arg): 5
Add (two args): 15
Add (three args): 30

Message 1: [Info] Operation completed.
Message 2: [ERROR] Error occurred!

Process Data (numbers): 10
Process Data (strings): Hello world this is Python
Process Data (mixed types): Unsupported data types for processing.
```

Lab 6: Illustration on How to Raise an Exception

Title

Error Handling: Raising and Handling Exceptions

Aim

To understand how to explicitly raise exceptions in Python and handle them gracefully using `try`, `except`, `else`, and `finally` blocks.

Procedure

1. Write a function that performs an operation that might lead to an error (e.g., division by zero, accessing an invalid index, or custom validation).
2. Inside the function, use the `raise` keyword to explicitly raise a built-in exception (e.g., `ValueError`, `ZeroDivisionError`) or a custom exception if a certain condition is not met.
3. Call this function within a `try-except` block to catch the raised exception.
4. Demonstrate the use of `else` (code to run if no exception occurs) and `finally` (code to run regardless of exception).

Source Code

```
def divide_numbers(numerator, denominator):
    """
    Divides two numbers and raises a ValueError if denominator is zero
    or if inputs are not numbers.
    """
    if not isinstance(numerator, (int, float)) or not isinstance(denominator, 
(int, float)):
        raise TypeError("Both numerator and denominator must be numbers.")
    if denominator == 0:
        raise ValueError("Denominator cannot be zero.")
    return numerator / denominator

def process_age(age):
    """
    Processes an age value, raising an exception if it's invalid.
    """
    if not isinstance(age, int):
        raise TypeError("Age must be an integer.")
    if age < 0 or age > 120:
        raise ValueError("Age must be between 0 and 120.")
    return f"Processing age: {age} years."

# Example 1: Handling division by zero
print("--- Division Example ---")
try:
    result = divide_numbers(10, 2)
    print(f"Result of division: {result}")
except ValueError as e:
    print(f"Error: {e}")
except TypeError as e:
    print(f"Error: {e}")
else:
    print("Division successful, no exception occurred.")
finally:
    print("Division attempt complete.\n")
```

```
# Example 2: Handling invalid age
print("--- Age Processing Example ---")
try:
    message = process_age(25)
    print(message)
except ValueError as e:
    print(f"Error: {e}")
except TypeError as e:
    print(f"Error: {e}")
else:
    print("Age processed successfully.")
finally:
    print("Age processing attempt complete.\n")

# Example 3: Triggering an error
print("--- Triggering Errors ---")
try:
    divide_numbers(10, 0)
except ValueError as e:
    print(f"Caught expected error: {e}")
finally:
    print("Attempted division by zero.\n")

try:
    process_age("thirty")
except TypeError as e:
    print(f"Caught expected error: {e}")
finally:
    print("Attempted processing non-integer age.\n")
```

Input

No specific user input is required.

Expected Output

```
--- Division Example ---
Result of division: 5.0
Division successful, no exception occurred.
Division attempt complete.

--- Age Processing Example ---
Processing age: 25 years.
Age processed successfully.
Age processing attempt complete.

--- Triggering Errors ---
Caught expected error: Denominator cannot be zero.
Attempted division by zero.

Caught expected error: Age must be an integer.
Attempted processing non-integer age.
```

Lab 7: Implementing Modules

Title

Code Organization: Implementing and Using Python Modules

Aim

To understand how to create and use Python modules to organize code into reusable units.

Procedure

1. Create a separate Python file (e.g., `my_math_module.py`) that acts as a module.
2. Inside this module, define several functions (e.g., `add`, `subtract`, `multiply`).
3. In another Python file (e.g., `main_program.py`) in the same directory, import the created module.
4. Call the functions from the imported module and print their results.
5. Demonstrate different ways of importing (e.g., `import module_name`, `from module_name import function_name`, `import module_name as alias`).

Source Code

File: `my_math_module.py`

```
# my_math_module.py

PI = 3.14159

def add(x, y):
    """Returns the sum of two numbers."""
    return x + y

def subtract(x, y):
    """Returns the difference of two numbers."""
    return x - y

def multiply(x, y):
    """Returns the product of two numbers."""
    return x * y

def divide(x, y):
    """Returns the division of two numbers. Handles division by zero."""
    if y == 0:
        return "Error: Cannot divide by zero!"
    return x / y

def circle_area(radius):
    """Calculates the area of a circle."""
    return PI * radius * radius

if __name__ == "__main__":
    # This block runs only when my_math_module.py is executed directly
    print("This is a test of my_math_module.")
    print(f"5 + 3 = {add(5, 3)}")
    print(f"Area of circle with radius 5: {circle_area(5)}")
```

File: `main_program.py`

```
# main_program.py

# Method 1: Import the entire module
import my_math_module
print("--- Using 'import my_math_module' ---")
print(f"10 + 5 = {my_math_module.add(10, 5)}")
print(f"20 - 7 = {my_math_module.subtract(20, 7)}")
print(f"Value of PI: {my_math_module.PI}")

# Method 2: Import specific functions
from my_math_module import multiply, divide
print("\n--- Using 'from my_math_module import ...' ---")
print(f"6 * 4 = {multiply(6, 4)}")
print(f"10 / 2 = {divide(10, 2)}")
print(f"10 / 0 = {divide(10, 0)}")

# Method 3: Import with an alias
import my_math_module as mm
print("\n--- Using 'import my_math_module as mm' ---")
print(f"Area of circle with radius 7: {mm.circle_area(7)}")
```

Input

No specific user input is required.

Expected Output

```
--- Using 'import my_math_module' ---
10 + 5 = 15
20 - 7 = 13
Value of PI: 3.14159

--- Using 'from my_math_module import ...' ---
6 * 4 = 24
10 / 2 = 5.0
10 / 0 = Error: Cannot divide by zero!

--- Using 'import my_math_module as mm' ---
Area of circle with radius 7: 153.9371
```

Lab 8: Implementing Threads

Title

Concurrency: Implementing Threads in Python

Aim

To understand and implement multi-threading in Python using the `threading` module to perform concurrent tasks.

Procedure

1. Import the `threading` and `time` modules.
2. Define a function that simulates a task (e.g., printing messages with a delay).
3. Create multiple `Thread` objects, each targeting the defined function.
4. Start each thread using the `start()` method.
5. Use the `join()` method to wait for threads to complete before the main program exits.
6. Observe how tasks run concurrently.

Source Code

```
import threading
import time

def task(name, duration):
    """
    A function that simulates a task with a given duration.
    """
    print(f"Thread {name}: Starting task...")
    time.sleep(duration) # Simulate work
    print(f"Thread {name}: Task finished after {duration} seconds.")

def main():
    print("Main program: Starting threads...")

    # Create threads
    thread1 = threading.Thread(target=task, args=("One", 3))
    thread2 = threading.Thread(target=task, args=("Two", 2))
    thread3 = threading.Thread(target=task, args=("Three", 4))

    # Start threads
    thread1.start()
    thread2.start()
    thread3.start()

    # Wait for all threads to complete
    thread1.join()
    thread2.join()
    thread3.join()

    print("Main program: All threads have finished.")

if __name__ == "__main__":
    main()
```

Input

No specific user input is required.

Expected Output

The exact order of "Starting task..." and "Task finished..." messages might vary slightly due to thread scheduling, but the overall sequence will demonstrate concurrency.

```
Main program: Starting threads...
Thread One: Starting task...
Thread Two: Starting task...
Thread Three: Starting task...
Thread Two: Task finished after 2 seconds.
Thread One: Task finished after 3 seconds.
Thread Three: Task finished after 4 seconds.
Main program: All threads have finished.
```

Lab 9: Illustration on Command Line Arguments and Regular Expressions

Title

Scripting Utilities: Command Line Arguments and Regular Expressions

Aim

To learn how to accept and process command-line arguments in Python scripts and to use regular expressions for pattern matching and text manipulation.

Procedure

Part 1: Command Line Arguments

1. Import the `sys` module.
2. Access command-line arguments using `sys.argv`.
3. Write a script that takes arguments and performs an action (e.g., summing numbers, greeting a user).

Part 2: Regular Expressions

1. Import the `re` module.
2. Use `re.search()`, `re.findall()`, `re.match()`, and `re.sub()` with various regular expression patterns.
3. Experiment with different patterns for tasks like email validation, phone number extraction, or finding specific words.

Source Code

File: `cli_example.py`

```
# cli_example.py
import sys

def process_arguments():
    if len(sys.argv) < 2:
        print("Usage: python cli_example.py <command> [args...]")
        print("Commands: greet <name>, sum <num1> <num2>...")
        return

    command = sys.argv[1]
    args = sys.argv[2:]

    if command == "greet":
        if len(args) > 0:
            name = " ".join(args)
            print(f"Hello, {name}!")
        else:
            print("Please provide a name for 'greet'.")
    elif command == "sum":
        if len(args) > 0:
            try:
                numbers = [float(arg) for arg in args]
                total = sum(numbers)
                print(f"The sum of {', '.join(args)} is: {total}")
            except ValueError:
                print("All arguments for 'sum' must be numbers.")
        else:
            print("Please provide numbers for 'sum'.")
    else:
        print(f"Unknown command: {command}")

if __name__ == "__main__":
    process_arguments()
```

File: `regex_example.py`

```
# regex_example.py
import re

def demonstrate_regex():
    text = "My phone number is 123-456-7890. Call me at 987.654.3210 or email
me at test@example.com."

    # 1. re.search(): Find the first occurrence
    phone_pattern = r'\d{3}[-.]\d{3}[-.]\d{4}'
    match = re.search(phone_pattern, text)
    if match:
        print(f"First phone number found: {match.group()}")

    # 2. re.findall(): Find all occurrences
    all_phones = re.findall(phone_pattern, text)
    print(f"All phone numbers found: {all_phones}")

    # 3. Email validation
    email_pattern = r'[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}'
    email_match = re.search(email_pattern, text)
    if email_match:
        print(f"Email found: {email_match.group()}")

    # 4. re.sub(): Replace occurrences
    new_text = re.sub(r'phone number', 'contact number', text)
    print(f"Text after substitution: {new_text}")

    # 5. Using re.match() - matches only at the beginning of the string
    print("\n--- re.match() example ---")
    if re.match(r'My', text):
        print("'My' matches at the beginning.")
    else:
        print("'My' does not match at the beginning.")

    if re.match(r'phone', text):
        print("'phone' matches at the beginning.")
    else:
        print("'phone' does not match at the beginning.")

if __name__ == "__main__":
    demonstrate_regex()
```

Input

For `cli_example.py` (run from terminal):

```
python cli_example.py greet John Doe
python cli_example.py sum 10 20 5.5
python cli_example.py unknown_command
python cli_example.py sum one two
```

For `regex_example.py`: No specific user input is required, the text is hardcoded.

Expected Output

For `cli_example.py`:

```
Hello, John Doe!
The sum of 10, 20, 5.5 is: 35.5
Unknown command: unknown_command
All arguments for 'sum' must be numbers.
```

For `regex_example.py`:

```
First phone number found: 123-456-7890
All phone numbers found: ['123-456-7890', '987.654.3210']
Email found: test@example.com
Text after substitution: My contact number is 123-456-7890. Call me at
987.654.3210 or email me at test@example.com.

--- re.match() example ---
'My' matches at the beginning.
'phone' does not match at the beginning.
```

Lab 10: Descriptive Statistics Using NumPy

Title

Numerical Computing: Descriptive Statistics with NumPy

Aim

To use the NumPy library to perform basic descriptive statistical operations on numerical data, such as mean, median, standard deviation, variance, minimum, and maximum.

Procedure

1. Import the `numpy` library.
2. Create a NumPy array (1D or 2D) with numerical data.
3. Apply various NumPy statistical functions to the array:
   - `np.mean()`
   - `np.median()`
   - `np.std()` (standard deviation)
   - `np.var()` (variance)
   - `np.min()`
   - `np.max()`
   - `np.sum()`
4. Print the results of each statistical calculation.
5. Experiment with calculating statistics along different axes for 2D arrays.

Source Code

```
import numpy as np

# Create a 1D NumPy array
data_1d = np.array([12, 15, 18, 20, 22, 25, 28, 30, 32, 35])
print(f"1D Data: {data_1d}")

print("\n--- Descriptive Statistics for 1D Data ---")
print(f"Mean: {np.mean(data_1d)}")
print(f"Median: {np.median(data_1d)}")
print(f"Standard Deviation: {np.std(data_1d)}")
print(f"Variance: {np.var(data_1d)}")
print(f"Minimum: {np.min(data_1d)}")
print(f"Maximum: {np.max(data_1d)}")
print(f"Sum: {np.sum(data_1d)}")

# Create a 2D NumPy array (matrix)
data_2d = np.array([
    [10, 15, 20],
    [25, 30, 35],
    [40, 45, 50]
])
print(f"\n2D Data:\n{data_2d}")

print("\n--- Descriptive Statistics for 2D Data ---")
print(f"Mean of all elements: {np.mean(data_2d)}")
print(f"Mean along columns (axis=0): {np.mean(data_2d, axis=0)}")
print(f"Mean along rows (axis=1): {np.mean(data_2d, axis=1)}")

print(f"Standard Deviation along columns (axis=0): {np.std(data_2d,
axis=0)}")
print(f"Maximum along rows (axis=1): {np.max(data_2d, axis=1)}")
```

Input

No specific user input is required.

Expected Output

```
1D Data: [12 15 18 20 22 25 28 30 32 35]

--- Descriptive Statistics for 1D Data ---
Mean: 23.7
Median: 23.5
Standard Deviation: 7.03917594951474
Variance: 49.559999999999995
Minimum: 12
Maximum: 35
Sum: 237

2D Data:
[[10 15 20]
 [25 30 35]
 [40 45 50]]

--- Descriptive Statistics for 2D Data ---
Mean of all elements: 30.0
Mean along columns (axis=0): [25. 30. 35.]
Mean along rows (axis=1): [15. 30. 45.]
Standard Deviation along columns (axis=0): [12.47219129 12.47219129
12.47219129]
Maximum along rows (axis=1): [20 35 50]
```

Lab 11: Illustrate Indexing Operations in Data Frame

Title

Data Manipulation: Indexing Operations in Pandas DataFrames

Aim

To understand and apply various indexing and selection techniques on Pandas DataFrames using `loc`, `iloc`, and boolean indexing.

Procedure

1. Import the `pandas` library.
2. Create a Pandas DataFrame from a dictionary or list of lists.
3. Perform the following indexing operations:
    - **Label-based indexing (`.loc`):** Select rows by label, columns by label, and specific cells.
    - **Integer-location based indexing (`.iloc`):** Select rows by integer position, columns by integer position, and specific cells.
    - **Boolean indexing:** Select rows based on a condition applied to one or more columns.
4. Print the DataFrame subsets obtained from each indexing operation.

Source Code

```
import pandas as pd

# Create a sample DataFrame
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'],
    'Age': [24, 27, 22, 32, 29],
    'City': ['New York', 'London', 'Paris', 'New York', 'London'],
    'Salary': [70000, 85000, 60000, 95000, 78000]
}
df = pd.DataFrame(data, index=['A', 'B', 'C', 'D', 'E'])
print("Original DataFrame:")
print(df)

print("\n--- .loc (Label-based Indexing) ---")
# Select row(s) by label
print("\nSelect row 'C':")
print(df.loc['C'])

print("\nSelect rows 'A' to 'D':")
print(df.loc['A':'D'])

# Select column(s) by label
print("\nSelect 'Age' column:")
print(df.loc[:, 'Age'])

# Select specific cell(s)
print("\nSelect cell at row 'B', column 'City':")
print(df.loc['B', 'City'])

print("\nSelect 'Name' and 'Salary' for rows 'A' and 'E':")
print(df.loc[['A', 'E'], ['Name', 'Salary']])
print("\n--- .iloc (Integer-location based Indexing) ---")
# Select row(s) by integer position
print("\nSelect row at index 2 (Charlie):")
print(df.iloc[2])

print("\nSelect rows from index 0 to 3 (exclusive of 4):")
print(df.iloc[0:4])

# Select column(s) by integer position
print("\nSelect column at index 1 (Age):")
print(df.iloc[:, 1])

# Select specific cell(s)
print("\nSelect cell at row index 1, column index 2 (London):")
print(df.iloc[1, 2])

print("\nSelect first two rows and first three columns:")
print(df.iloc[0:2, 0:3])


print("\n--- Boolean Indexing ---")
# Select rows where Age > 25
print("\nRows where Age > 25:")
print(df[df['Age'] > 25])

# Select rows where City is 'New York'
print("\nRows where City is 'New York':")
print(df[df['City'] == 'New York'])

# Select rows where Age > 25 AND City is 'London'
print("\nRows where Age > 25 AND City is 'London':")
print(df[(df['Age'] > 25) & (df['City'] == 'London')])

# Select 'Name' and 'Salary' for rows where Salary > 75000
print("\nName and Salary for rows where Salary > 75000:")
print(df.loc[df['Salary'] > 75000, ['Name', 'Salary']])
```

Input

No specific user input is required.

Expected Output

```
Original DataFrame:
      Name  Age      City  Salary
A    Alice   24  New York   70000
B      Bob   27    London   85000
C  Charlie   22     Paris   60000
D    David   32  New York   95000
E      Eve   29    London   78000

--- .loc (Label-based Indexing) ---

Select row 'C':
Name      Charlie
Age            22
City        Paris
Salary      60000
Name: C, dtype: object

Select rows 'A' to 'D':
      Name  Age      City  Salary
A    Alice   24  New York   70000
B      Bob   27    London   85000
```

```
C  Charlie   22     Paris   60000
D    David   32  New York   95000

Select 'Age' column:
A    24
B    27
C    22
D    32
E    29
Name: Age, dtype: int64

Select cell at row 'B', column 'City':
London

Select 'Name' and 'Salary' for rows 'A' and 'E':
    Name  Salary
A  Alice   70000
E    Eve   78000

--- .iloc (Integer-location based Indexing) ---

Select row at index 2 (Charlie):
Name    Charlie
Age          22
City      Paris
Salary    60000
Name: C, dtype: object

Select rows from index 0 to 3 (exclusive of 4):
      Name  Age      City  Salary
A    Alice   24  New York   70000
B      Bob   27    London   85000
C  Charlie   22     Paris   60000
D    David   32  New York   95000

Select column at index 1 (Age):
A    24
B    27
C    22
D    32
E    29
Name: Age, dtype: int64

Select cell at row index 1, column index 2 (London):
London

Select first two rows and first three columns:
    Name  Age      City
A  Alice   24  New York
B    Bob   27    London

--- Boolean Indexing ---

Rows where Age > 25:
    Name  Age      City  Salary
B    Bob   27    London   85000
D  David   32  New York   95000
E    Eve   29    London   78000

Rows where City is 'New York':
     Name  Age      City  Salary
A   Alice   24  New York   70000
D   David   32  New York   95000

Rows where Age > 25 AND City is 'London':
   Name  Age    City  Salary
```

```
B   Bob   27  London   85000
E   Eve   29  London   78000

Name and Salary for rows where Salary > 75000:
    Name  Salary
B    Bob   85000
D  David   95000
E    Eve   78000
```

Lab 12: Illustrate Various Plots Using Pandas and Matplotlib

Title

Data Visualization: Various Plots with Pandas and Matplotlib

Aim

To create various types of data visualizations (e.g., line plots, bar plots, histograms, scatter plots) using Pandas' built-in plotting capabilities and the Matplotlib library.

Procedure

1. Import `pandas` and `matplotlib.pyplot`.
2. Create a sample DataFrame with numerical and categorical data.
3. Generate the following plots:
   - **Line Plot:** To show trends over time or ordered categories.
   - **Bar Plot:** To compare discrete categories.
   - **Histogram:** To show the distribution of a single numerical variable.
   - **Scatter Plot:** To visualize the relationship between two numerical variables.
   - **Box Plot:** To show the distribution and outliers for categorical groups.
4. Add titles, labels, and legends to the plots for clarity.
5. Use `plt.show()` to display each plot.

Source Code

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

# Set a style for better aesthetics
plt.style.use('seaborn-v0_8-darkgrid')

# Create a sample DataFrame
data = {
    'Month': ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep',
'Oct', 'Nov', 'Dec'],
    'Sales': [150, 180, 220, 200, 250, 280, 300, 270, 240, 210, 190, 160],
    'Expenses': [100, 120, 150, 130, 170, 190, 200, 180, 160, 140, 130, 110],
    'City_A_Temp': [5, 7, 12, 18, 22, 25, 28, 27, 22, 16, 10, 6],
    'City_B_Temp': [8, 10, 15, 20, 24, 27, 30, 29, 24, 18, 12, 9],
    'Category': ['A', 'B', 'A', 'C', 'B', 'A', 'C', 'B', 'A', 'C', 'B', 'A'],
    'Values': np.random.randint(10, 100, 12) # Random values for distribution
}
df = pd.DataFrame(data)

# --- 1. Line Plot (Sales over Months) ---
plt.figure(figsize=(10, 6))
df.plot(x='Month', y=['Sales', 'Expenses'], kind='line', marker='o',
ax=plt.gca())
plt.title('Monthly Sales and Expenses Trend')
plt.xlabel('Month')
plt.ylabel('Amount ($)')
plt.legend(['Sales', 'Expenses'])
plt.grid(True)
plt.tight_layout()
plt.show()

# --- 2. Bar Plot (Average Sales per Category) ---
plt.figure(figsize=(8, 5))
```

```
df.groupby('Category')['Sales'].mean().plot(kind='bar', color=['skyblue',
'lightcoral', 'lightgreen'], ax=plt.gca())
plt.title('Average Sales per Category')
plt.xlabel('Category')
plt.ylabel('Average Sales ($)')
plt.xticks(rotation=0) # Keep labels horizontal
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()

# --- 3. Histogram (Distribution of Values) ---
plt.figure(figsize=(8, 5))
df['Values'].plot(kind='hist', bins=7, edgecolor='black', alpha=0.7,
color='purple', ax=plt.gca())
plt.title('Distribution of Random Values')
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()

# --- 4. Scatter Plot (City A vs City B Temperature) ---
plt.figure(figsize=(8, 6))
df.plot(x='City_A_Temp', y='City_B_Temp', kind='scatter', s=df['Sales']/5,
alpha=0.7, color='red', ax=plt.gca())
plt.title('Temperature Comparison: City A vs City B (Size by Sales)')
plt.xlabel('City A Temperature (°C)')
plt.ylabel('City B Temperature (°C)')
plt.grid(True)
plt.tight_layout()
plt.show()

# --- 5. Box Plot (Sales distribution by Category) ---
plt.figure(figsize=(8, 6))
df.boxplot(column='Sales', by='Category', grid=True, ax=plt.gca())
plt.title('Sales Distribution by Category')
plt.suptitle('') # Suppress the default suptitle created by boxplot
plt.xlabel('Category')
plt.ylabel('Sales ($)')
plt.tight_layout()
plt.show()
```

Input

No specific user input is required.

Expected Output

(Graphical output - plots will be displayed in separate windows or inline if using a Jupyter Notebook/similar environment). The output will be five distinct plots:

1. A line plot showing 'Sales' and 'Expenses' trends over 'Months'.
2. A bar plot showing the average 'Sales' for each 'Category'.
3. A histogram showing the frequency distribution of 'Values'.
4. A scatter plot showing the relationship between 'City_A_Temp' and 'City_B_Temp', with point size representing 'Sales'.
5. A box plot showing the distribution of 'Sales' for each 'Category'.

Lab 13: Building GUI Application with Tkinter

Title

Graphical User Interface: Building Applications with Tkinter

Aim

To develop a simple graphical user interface (GUI) application using Python's built-in Tkinter library, demonstrating widgets like labels, buttons, and entry fields.

Procedure

1. Import the `tkinter` module.
2. Create the main application window (`Tk()`).
3. Add various widgets:
   - `Label`: To display static text.
   - `Entry`: To get user input.
   - `Button`: To trigger actions.
4. Define functions to be executed when buttons are clicked.
5. Use layout managers (e.g., `pack()`, `grid()`) to arrange widgets.
6. Start the Tkinter event loop using `mainloop()`.

Source Code

```
import tkinter as tk
from tkinter import messagebox

def greet_user():
    """Function to be called when the 'Greet' button is clicked."""
    user_name = name_entry.get()
    if user_name:
        messagebox.showinfo("Greeting", f"Hello, {user_name}!")
    else:
        messagebox.showwarning("Input Error", "Please enter your name!")

def clear_input():
    """Function to clear the input field."""
    name_entry.delete(0, tk.END)

# Create the main application window
root = tk.Tk()
root.title("Simple Tkinter GUI App")
root.geometry("400x200") # Set window size
root.resizable(False, False) # Make window non-resizable

# Create a Label widget
label = tk.Label(root, text="Enter your name:", font=('Arial', 12))
label.pack(pady=10) # Add some padding

# Create an Entry widget (text input field)
name_entry = tk.Entry(root, width=30, font=('Arial', 12))
name_entry.pack(pady=5)

# Create a Greet Button
greet_button = tk.Button(root, text="Greet", command=greet_user,
                         bg='lightblue', fg='black', font=('Arial', 12,
'bold'))
greet_button.pack(pady=5)
```

```
# Create a Clear Button
clear_button = tk.Button(root, text="Clear", command=clear_input,
                         bg='lightgray', fg='black', font=('Arial', 12))
clear_button.pack(pady=5)

# Start the Tkinter event loop
root.mainloop()
```

Input

(User interaction with the GUI)

1. Type a name into the text field.
2. Click the "Greet" button.
3. Click the "Clear" button.

Expected Output

(A graphical window will appear)

1. **Initial Window:** A window titled "Simple Tkinter GUI App" with a label "Enter your name:", an empty text entry field, a "Greet" button, and a "Clear" button.
2. **After typing "Alice" and clicking "Greet":** A small pop-up message box will appear with the title "Greeting" and the message "Hello, Alice!".
3. **After clicking "Clear":** The text entry field will become empty.
4. **After clicking "Greet" with an empty field:** A pop-up message box will appear with the title "Input Error" and the message "Please enter your name!".

Lab 14: Creating Tables Using SQLite

Title

Database Management: Creating Tables with SQLite

Aim

To understand how to interact with SQLite databases in Python, specifically focusing on creating a new database file and defining tables within it.

Procedure

1. Import the `sqlite3` module.
2. Establish a connection to an SQLite database file. If the file doesn't exist, it will be created.
3. Create a cursor object to execute SQL commands.
4. Write a `CREATE TABLE` SQL statement to define the schema of a new table.
5. Execute the SQL statement using the cursor.
6. Commit the changes to the database.
7. Close the database connection.
8. (Optional) Verify table creation by connecting again and querying for table names.

Source Code

```
import sqlite3
import os

# Define the database file name
DB_FILE = 'mydatabase.db'

def create_database_and_table():
    conn = None # Initialize connection to None
    try:
        # Connect to SQLite database (creates it if it doesn't exist)
        conn = sqlite3.connect(DB_FILE)
        cursor = conn.cursor()
        print(f"Connected to database: {DB_FILE}")

        # SQL statement to create a table named 'students'
        # IF NOT EXISTS prevents error if table already exists
        create_table_sql = """
        CREATE TABLE IF NOT EXISTS students (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            name TEXT NOT NULL,
            age INTEGER,
            major TEXT
        );
        """
        # Execute the CREATE TABLE statement
        cursor.execute(create_table_sql)
        print("Table 'students' created successfully (or already exists).")

        # Commit the changes
        conn.commit()
        print("Changes committed to database.")

    except sqlite3.Error as e:
        print(f"SQLite error: {e}")
    finally:
        # Close the connection
        if conn:
            conn.close()
            print("Database connection closed.")

def verify_table_exists():
    conn = None
    try:
        conn = sqlite3.connect(DB_FILE)
        cursor = conn.cursor()
        cursor.execute("SELECT name FROM sqlite_master WHERE type='table' AND
name='students';")
        if cursor.fetchone():
            print(f"\nVerification: Table 'students' exists in {DB_FILE}.")
        else:
            print(f"\nVerification: Table 'students' DOES NOT exist in
{DB_FILE}.")
    except sqlite3.Error as e:
        print(f"SQLite verification error: {e}")
    finally:
        if conn:
            conn.close()

if __name__ == "__main__":
    # Clean up previous database file for a fresh start (optional)
    if os.path.exists(DB_FILE):
        os.remove(DB_FILE)
        print(f"Removed existing database file: {DB_FILE}")

    create_database_and_table()
    verify_table_exists()
```

Input

No specific user input is required.

Expected Output

```
Removed existing database file: mydatabase.db
Connected to database: mydatabase.db
Table 'students' created successfully (or already exists).
Changes committed to database.
Database connection closed.

Verification: Table 'students' exists in mydatabase.db.
```

(A file named `mydatabase.db` will be created in the same directory as the Python script.)

Lab 15: Illustration on Database Connectivity

Title

Database Integration: Illustrating Database Connectivity (CRUD Operations)

Aim

To demonstrate full database connectivity by performing basic CRUD (Create, Read, Update, Delete) operations on an SQLite database table using Python.

Procedure

1. Import the `sqlite3` module.
2. Connect to an existing SQLite database (or create one if it doesn't exist, and create a table as in Lab 14).
3. Implement functions for:
    - **Inserting** new records into the table.
    - **Retrieving** all records or specific records from the table.
    - **Updating** existing records.
    - **Deleting** records.
4. Execute these functions sequentially to demonstrate each operation.
5. Commit changes after insert, update, or delete operations.
6. Close the database connection.

Source Code

```
import sqlite3
import os

DB_FILE = 'university.db'

def setup_database():
    """Connects to DB, creates 'students' table if not exists."""
    conn = sqlite3.connect(DB_FILE)
    cursor = conn.cursor()
    cursor.execute("""
        CREATE TABLE IF NOT EXISTS students (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            name TEXT NOT NULL,
            age INTEGER,
            major TEXT
        );
    """)
    conn.commit()
    conn.close()
    print(f"Database '{DB_FILE}' and table 'students' ensured to exist.")

def insert_student(name, age, major):
    """Inserts a new student record."""
    conn = sqlite3.connect(DB_FILE)
    cursor = conn.cursor()
    cursor.execute("INSERT INTO students (name, age, major) VALUES (?, ?,
?)", (name, age, major))
    conn.commit()
    conn.close()
    print(f"Inserted: {name}")

def select_all_students():
    """Retrieves and prints all student records."""
    conn = sqlite3.connect(DB_FILE)
    cursor = conn.cursor()
    cursor.execute("SELECT * FROM students")
    rows = cursor.fetchall()
    conn.close()
    print("\n--- All Students ---")
    if not rows:
        print("No students found.")
        return
    for row in rows:
        print(row)
    return rows

def update_student_major(student_id, new_major):
    """Updates the major of a student by ID."""
    conn = sqlite3.connect(DB_FILE)
    cursor = conn.cursor()
    cursor.execute("UPDATE students SET major = ? WHERE id = ?", (new_major,
student_id))
    conn.commit()
    conn.close()
    print(f"Updated student ID {student_id}'s major to {new_major}.")

def delete_student(student_id):
    """Deletes a student record by ID."""
    conn = sqlite3.connect(DB_FILE)
    cursor = conn.cursor()
    cursor.execute("DELETE FROM students WHERE id = ?", (student_id,))
    conn.commit()
    conn.close()
    print(f"Deleted student ID {student_id}.")

if __name__ == "__main__":
    # Clean up previous database file for a fresh start
    if os.path.exists(DB_FILE):
        os.remove(DB_FILE)
        print(f"Removed existing database file: {DB_FILE}")

    setup_database()

    # --- CREATE (Insert) ---
    print("\n--- Inserting Records ---")
    insert_student("Alice Smith", 20, "Computer Science")
    insert_student("Bob Johnson", 22, "Physics")
    insert_student("Charlie Brown", 21, "Mathematics")

    # --- READ (Select All) ---
    select_all_students()

    # --- UPDATE ---
    print("\n--- Updating Record ---")
    # Assuming Alice is ID 1 (due to AUTOINCREMENT starting from 1)
    update_student_major(1, "Data Science")
    select_all_students() # Show updated record

    # --- DELETE ---
    print("\n--- Deleting Record ---")
    # Assuming Bob is ID 2
    delete_student(2)
    select_all_students() # Show remaining records

    print("\nDatabase operations complete.")
```

Input

No specific user input is required.

Expected Output

```
Removed existing database file: university.db
Database 'university.db' and table 'students' ensured to exist.

--- Inserting Records ---
Inserted: Alice Smith
Inserted: Bob Johnson
Inserted: Charlie Brown

--- All Students ---
(1, 'Alice Smith', 20, 'Computer Science')
(2, 'Bob Johnson', 22, 'Physics')
(3, 'Charlie Brown', 21, 'Mathematics')

--- Updating Record ---
Updated student ID 1's major to Data Science.

--- All Students ---
(1, 'Alice Smith', 20, 'Data Science')
(2, 'Bob Johnson', 22, 'Physics')
(3, 'Charlie Brown', 21, 'Mathematics')

--- Deleting Record ---
Deleted student ID 2.

--- All Students ---
(1, 'Alice Smith', 20, 'Data Science')
(3, 'Charlie Brown', 21, 'Mathematics')

Database operations complete.
```

(A file named `university.db` will be created/modified in the same directory.)