Unit 2: Input, Output, Control Statements, and Strings in Python

---

2.1 Input & Output Statements

Input Statements

- **Purpose:** To take input from the user.
- **Method Used:** `input()`

Example:

```
name = input("Enter your name: ")  # Takes input from the user
print("Hello, " + name + "!")  # Prints a greeting
```

Output Statements

- **Purpose:** To display information to the user.
- **Method Used:** `print()`

Example:

```
print("Welcome to Python Programming!")  # Prints a welcome message
```

Command Line Arguments

- **Purpose:** To pass inputs to a Python script through the terminal.
- **Method Used:** `sys.argv` (from the `sys` module)

Example: Save this as `example.py`:

```
import sys

# Command-line arguments
print("Script Name:", sys.argv[0])  # First argument is always the script name
if len(sys.argv) > 1:
    print("First argument:", sys.argv[1])
```

Run it from the terminal:

```
python example.py Hello
# Output:
# Script Name: example.py
# First argument: Hello
```

2.2 Control Statements

If Statement

- Used to execute code only if a condition is `True`.

Example:

```
age = int(input("Enter your age: "))
if age >= 18:
    print("You are eligible to vote.")
```

If-Else Statement

- Used to execute one block of code if the condition is `True` and another block if it is `False`.

Example:

```
num = int(input("Enter a number: "))
if num % 2 == 0:
    print("The number is even.")
else:
    print("The number is odd.")
```

If-Elif-Else Statement

- Used to check multiple conditions.

Example:

```
marks = int(input("Enter your marks: "))
if marks >= 90:
    print("Grade: A")
elif marks >= 75:
    print("Grade: B")
elif marks >= 60:
    print("Grade: C")
else:
    print("Grade: F")
```

While Loop

- Repeats code as long as a condition is `True`.

Example:

```
count = 1
while count <= 5:
    print("Count:", count)
    count += 1
```

For Loop

- Used to iterate over a sequence (like a list, string, or range).

Example:

```
for i in range(1, 6):
    print("Number:", i)
```

Else Suite in Loops

- Executes code when the loop finishes normally (no `break`).

Example:

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

```
    print(i)
else:
    print("Loop completed successfully!")
```

Break Statement

- Exits a loop prematurely.

Example:

```
for i in range(10):
    if i == 5:
        break
    print(i)
```

Continue Statement

- Skips the current iteration and moves to the next.

Example:

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

Assert Statement

- Used for debugging; raises an error if the condition is `False`.

Example:

```
x = 5
assert x > 0, "x must be positive"
```

Return Statement

- Used in functions to return a value.

Example:

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

result = add(3, 4)
print("Sum:", result)
```

2.3 Arrays in Python - Strings and Characters

Creating Strings

- Strings are created by enclosing characters in quotes.

Example:

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

Indexing

- Accessing specific characters using their position.

Example:

```
s = "Python"
print(s[0])  # First character: P
print(s[-1]) # Last character: n
```

Slicing

- Extracting parts of a string.

Example:

```
s = "Hello, World!"
print(s[0:5])  # Output: Hello
print(s[7:])   # Output: World!
```

Repeating Strings

- Multiplying a string.

Example:

```
print("Hello " * 3)  # Output: Hello Hello Hello
```

Concatenating Strings

- Joining strings together.

Example:

```
first = "Hello"
second = "World"
print(first + ", " + second + "!")  # Output: Hello, World!
```

Comparing Strings

- Using comparison operators.

Example:

```
print("apple" == "Apple")  # Output: False
print("apple" > "banana")  # Output: False
```

2.4 String Operations

Finding and Counting Substrings

- `find()`: Returns the index of the first occurrence of a substring.
- `count()`: Returns the number of times a substring occurs.

Example:

```
s = "hello, hello, hello"
print(s.find("hello"))  # Output: 0
```

```
print(s.count("hello"))  # Output: 3
```

Replacing Strings

- `replace()`: Replaces occurrences of a substring.

Example:

```
s = "bad apple"
print(s.replace("bad", "good"))  # Output: good apple
```

Splitting Strings

- `split()`: Splits a string into a list.

Example:

```
s = "apple,banana,cherry"
print(s.split(","))  # Output: ['apple', 'banana', 'cherry']
```

Joining Strings

- `join()`: Joins a list of strings into a single string.

Example:

```
words = ["apple", "banana", "cherry"]
print(", ".join(words))  # Output: apple, banana, cherry
```

Working with Characters

- Strings can be treated as a sequence of characters.

Example:

```
s = "Python"
for char in s:
    print(char)
```

Unit 3

3.5 Functions

Functions and Methods

- Function: A self-contained block of code that encapsulates a specific task or related group of tasks.

- Method: A function that is associated with an object. In object-oriented programming, methods are defined within classes.

Defining Functions

- Syntax (Python example):

```
python
def function_name(parameters):
    code block
    return value
```

- Example:

```
python
def add(a, b):
    return a + b
```

Calling Functions

- To execute a function, use its name followed by parentheses:

```
python
result = add(5, 3)   result will be 8
```

Returning Multiple Values

- Functions can return multiple values as a tuple:

```
python
```

def calculate(a, b):

return a + b, a - b

sum_val, diff_val = calculate(10, 5) sum_val = 15, diff_val = 5

Formal and Actual Parameters

- Formal Parameters: Variables listed in the function definition.

- Actual Parameters: Values passed to the function when it is called.

- Example:

python

def greet(name): 'name' is a formal parameter

print("Hello, " + name)

greet("Alice") "Alice" is an actual parameter

3.6 Local and Global Variables

Local Variables

- Defined within a function and cannot be accessed outside it.

- Example:

python

def my_function():

local_var = "I'm local"

print(local_var)

my_function() Prints: I'm local

print(local_var) Error: NameError: name 'local_var' is not defined

Global Variables

- Defined outside any function and can be accessed anywhere in the program.
- To modify a global variable inside a function, use the global keyword:

```
python
global_var = "I'm global"

def modify_global():
    global global_var
    global_var = "I've been modified"

modify_global()
print(global_var)  Prints: I've been modified
```

Lists and Tuples

Creating Lists

- Lists are ordered collections that can hold mixed data types.
- Syntax:

```
python
my_list = [1, "two", 3.0]
```

Updating Lists

- Elements can be updated using indexing:

```
python
my_list[1] = "changed"
```

Concatenating Lists

- Use the + operator to concatenate two lists:

```
python
list_a = [1, 2]
```

list_b = [3, 4]

combined_list = list_a + list_b [1, 2, 3, 4]

Repetition of Lists

- Use the operator to repeat lists:

python

repeated_list = [1] 3 [1, 1, 1]

Aliasing and Cloning Lists

- Aliasing: Two variables refer to the same list in memory.

python

list_a = [1,2,3]

list_b = list_a Both point to the same list

- Cloning: Create a copy of the list.

python

cloned_list = list_a.copy() Creates a new list with the same elements

Sorting Lists

- Sort lists in place using .sort() or create a sorted copy with sorted().

python

my_list.sort() Sorts in place

sorted_list = sorted(my_list) Returns a new sorted list

Classes and Objects

Defining Classes and Objects

- Classes are blueprints for creating objects (instances).
- Syntax for defining a class:

```
python
class ClassName:
    def __init__(self, parameters):
        self.attribute = value

    def method_name(self):
        pass
```

Constructors

- The __init__() method initializes an object's attributes when it is created.

```
python
class Dog:
    def __init__(self, name):
        self.name = name

my_dog = Dog("Buddy")    my_dog is an instance of Dog with name "Buddy"
```

Types of Methods

1. Instance Methods: Operate on an instance of the class (using self).
2. Class Methods: Operate on the class itself (using @classmethod).
3. Static Methods: Do not operate on an instance or class (using @staticmethod).

Inner Classes

- Classes defined within another class are called inner classes.

```
python
class Outer:
    class Inner:
        pass
```

Inheritance and Polymorphism

Types of Inheritance

1. Single Inheritance: One subclass inherits from one superclass.

2. Multiple Inheritance: One subclass inherits from multiple superclasses.

3. Multilevel Inheritance: A subclass inherits from another subclass.

4. Hierarchical Inheritance: Multiple subclasses inherit from one superclass.

Using super() Method

- The super() function allows you to call methods from a parent class.

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

class Child(Parent):
    def show(self):
        super().show()    Calls Parent's show method
        print("Child")

c = Child()
c.show()    Outputs Parent followed by Child
```

Method Overloading and Overriding

- Overloading: Same method name with different parameters (not supported natively in Python but can be simulated).

- Overriding: Redefining a method in a subclass that already exists in its superclass.

python

```
class Animal:
    def sound(self):
        print("Animal sound")

class Dog(Animal):
    def sound(self):    Method overriding
        print("Bark")

d = Dog()
d.sound()    Outputs: Bark
```

Abstract Classes and Interfaces

- An abstract class cannot be instantiated and may contain abstract methods (methods without implementation).

python

```
from abc import ABC, abstractmethod

class AbstractClass(ABC):

    @abstractmethod
    def abstract_method(self):
        pass

class ConcreteClass(AbstractClass):

    def abstract_method(self):    Implementation of abstract method
        print("Implemented abstract method")
```

Unit 4

4.5 Exception Handling

Types of Exceptions:

1. **Built-in Exceptions:**
    - **SyntaxError:** Raised when there is a syntax error in the code.
    - **IndexError:** Raised when trying to access an invalid index in a list or tuple.
    - **KeyError:** Raised when a key is not found in a dictionary.
    - **TypeError:** Raised when an operation is performed on incompatible types.
    - **ValueError:** Raised when an operation receives an argument of the right type but with an invalid value.
2. **Runtime Exceptions:**
    - **ZeroDivisionError:** Raised when dividing a number by zero.
    - **FileNotFoundError:** Raised when a file is not found.
    - **AttributeError:** Raised when an invalid attribute reference occurs.
3. **User-defined Exceptions:**
    - Custom exceptions created by inheriting from the `Exception` class.

Assert Statement:

- Used for debugging and validating conditions during development.
- Syntax: `assert <condition>, <optional_message>`
- If the condition evaluates to `False`, an `AssertionError` is raised.

Except Block:

- Used to handle exceptions in a `try-except` block.
- Syntax:

```
try:
     code that may raise an exception
except ExceptionType:
     handle the exception
```

User-Defined Exceptions:

- Create custom exceptions using the `class` keyword:

```
class CustomError(Exception):
    def __init__(self, message):
        self.message = message
raise CustomError("This is a user-defined exception")
```

Logging the Exceptions:

- Use the `logging` module to log exceptions for debugging purposes.
- Example:

```
import logging

logging.basicConfig(filename='error.log', level=logging.ERROR)
```

```
try:
    result = 10 / 0
except ZeroDivisionError:
    logging.error("Attempted division by zero", exc_info=True)
```

4.6 Regular Expressions

Sequence Characters:

- Regular expressions (regex) are patterns used to match strings.
- Common sequence characters:
  - `.`: Matches any single character except newline.
  - `^`: Matches the start of the string.
  - `$`: Matches the end of the string.
  - `\d`: Matches any digit (0-9).
  - `\w`: Matches any alphanumeric character.
  - `\s`: Matches any whitespace character.
  - `+`: Matches one or more repetitions of the preceding character.
  - : Matches zero or more repetitions of the preceding character.

Creating Threads:

1. **Using `threading` Module:**

```
import threading

def task():
    print("Thread is running")

thread = threading.Thread(target=task)
thread.start()
```

2. **Extending the `Thread` Class:**

```
import threading

class MyThread(threading.Thread):
    def run(self):
        print("Thread is running")

thread = MyThread()
thread.start()
```

Thread Class Methods:

- `start()`: Starts the thread.
- `join()`: Waits for the thread to complete.
- `is_alive()`: Checks if the thread is still running.

Thread Synchronization:

1. **Locks:**
   - Ensure only one thread accesses a shared resource at a time.

```
lock = threading.Lock()
lock.acquire()
 critical section
lock.release()
```

2. **Semaphore:**

    - Allows a fixed number of threads to access a resource.

    ```
    semaphore = threading.Semaphore(2)
    semaphore.acquire()
     critical section
    semaphore.release()
    ```

Communication Between Threads:

- Use `queue.Queue` for thread-safe communication.

```
import queue
q = queue.Queue()
q.put("Data")
print(q.get())
```

Daemon Threads:

- Background threads that terminate when the main thread ends.

```
thread = threading.Thread(target=task, daemon=True)
thread.start()
```

4.7 Graphical User Interface (GUI)

Root Window:

- The main window in a GUI application created using `tkinter`.

```
import tkinter as tk
root = tk.Tk()
root.mainloop()
```

Font & Colors:

- Fonts are customized using the `font` attribute.
- Colors are set using the `bg` (background) and `fg` (foreground) attributes.

```
label = tk.Label(root, text="Hello", font=("Arial", 16), bg="yellow",
fg="blue")
```

Canvas:

- A widget used to draw shapes, lines, or images.

```
canvas = tk.Canvas(root, width=200, height=200)
canvas.create_rectangle(50, 50, 150, 150, fill="blue")
canvas.pack()
```

Frames:

- Containers used to organize other widgets.

```
frame = tk.Frame(root, bg="lightgray")
frame.pack()
```

4.8 Widgets

Common Widgets:

1. **Button:**

   ```
   button = tk.Button(root, text="Click Me", command=callback)
   ```

2. **Label:**

   ```
   label = tk.Label(root, text="This is a label")
   ```

3. **Message:**

   - Used to display multiline text.

   ```
   message = tk.Message(root, text="This is a message")
   ```

4. **Text:**

   - A multiline text editor.

   ```
   text = tk.Text(root, height=5, width=30)
   ```

5. **Scrollbar:**

   - Used to add scrolling functionality.

   ```
   scrollbar = tk.Scrollbar(root, orient="vertical", command=text.yview)
   ```

6. **Checkbutton:**

   - Allows multiple selections.

   ```
   checkbutton = tk.Checkbutton(root, text="Option 1")
   ```

7. **Radiobutton:**

   - Allows only one selection.

   ```
   radiobutton = tk.Radiobutton(root, text="Option A", value=1)
   ```

8. **Entry:**

   - A single-line text input.

   ```
   entry = tk.Entry(root)
   ```

9. **Spinbox:**

   - Used to select a value from a range.

   ```
   spinbox = tk.Spinbox(root, from_=0, to=10)
   ```

10. **Listbox:**

- Displays a list of options.

```
listbox = tk.Listbox(root)
listbox.insert(1, "Item 1")
```

11. **Menu:**

    - Creates a dropdown menu.

```
menu = tk.Menu(root)
root.config(menu=menu)
```

Creating Tables:

- Use the `Treeview` widget from the `ttk` module.

```
from tkinter import ttk
table = ttk.Treeview(root, columns=("Name", "Age"))
table.heading("Name", text="Name")
table.heading("Age", text="Age")
```

Python Exception Handling

Exception handling in Python is used to handle runtime errors (exceptions) to prevent program crashes and ensure smooth execution. Python provides a structured way to handle these errors using `try`, `except`, `finally`, and `else` blocks.

Basic Syntax

```
try:
    # Code that may raise an exception
    a = 10 / 0  # This will raise ZeroDivisionError
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

Exception Handling Blocks

1. **`try` Block**: Contains code that might raise an exception.
2. **`except` Block**: Handles exceptions if they occur in the `try` block.
3. **`finally` Block** *(Optional)*: Executes regardless of an exception (used for cleanup).
4. **`else` Block** *(Optional)*: Runs if no exception occurs.

Handling Multiple Exceptions

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ZeroDivisionError:
    print("Cannot divide by zero!")
except ValueError:
    print("Invalid input! Please enter a number.")
except Exception as e:
```

```
    print(f"An error occurred: {e}")
```

Here, we handle:

- `ZeroDivisionError` (if user enters 0)
- `ValueError` (if user enters a non-numeric value)
- A generic `Exception` (for any other unexpected errors)

Using `finally` for Cleanup

```
try:
    file = open("test.txt", "r")
    content = file.read()
except FileNotFoundError:
    print("File not found!")
finally:
    file.close()  # Ensures file is closed even if an error occurs
```

Raising Exceptions Using `raise`

You can manually raise exceptions using the `raise` keyword.

```
def check_age(age):
    if age < 18:
        raise ValueError("Age must be 18 or above")
    return "Access granted"

try:
    print(check_age(15))
except ValueError as e:
    print(f"Error: {e}")
```

Custom Exceptions

You can define custom exceptions by inheriting from Python's built-in `Exception` class.

```
class CustomError(Exception):
    pass

try:
    raise CustomError("This is a custom exception!")
except CustomError as e:
    print(e)
```

Regular Expressions (Regex) in Python

A **Regular Expression (regex)** is a powerful tool for pattern matching in strings. Python provides the `re` module for regex operations.

Basic Regex Functions

First, import the `re` module:

```
import re
```

---

1. `re.match()`

Matches a pattern only at the **beginning** of a string.

```
import re

text = "Hello World"
match = re.match(r'Hello', text)

if match:
    print("Match found!")
else:
    print("No match!")
```

✅ **Output:** `Match found!`

---

2. `re.search()`

Searches the entire string for a match.

```
text = "Python is fun!"
search = re.search(r'fun', text)

if search:
    print("Word found!")
```

✅ **Output:** `Word found!`

---

3. `re.findall()`

Finds **all occurrences** of a pattern in a string.

```
text = "cat, bat, mat, hat"
matches = re.findall(r'[cm]at', text)  # Matches 'cat' and 'mat'
print(matches)
```

✅ **Output:** `['cat', 'mat']`

---

4. `re.sub()`

Replaces matched patterns with a new string.

```
text = "I like apples"
new_text = re.sub(r'apples', 'bananas', text)
print(new_text)
```

✅ **Output:** `I like bananas`

Common Regex Patterns

Pattern	Meaning	Example
`^`	Start of string	`^Hello` (Matches "Hello World")
`$`	End of string	`world$` (Matches "Hello world")
`.`	Any character except newline	`h.t` (Matches "hat", "hit")
`\d`	Digit (0-9)	`\d{2}` (Matches "12" in "Age 12")
`\w`	Word character (A-Z, a-z, 0-9, _)	`\w+` (Matches "Python")
`\s`	Whitespace (spaces, tabs, newlines)	`\s+` (Matches spaces)
`\b`	Word boundary	`\bcat\b` (Matches "cat" but not "catch")
`[abc]`	Matches 'a', 'b', or 'c'	`[cm]at` (Matches "cat" and "mat")
`[^abc]`	Not 'a', 'b', or 'c'	`[^aeiou]` (Matches non-vowels)
`(a	b)`	Either 'a' or 'b'
`*`	0 or more occurrences	`ca*t` (Matches "ct", "cat", "caaaat")
`+`	1 or more occurrences	`ca+t` (Matches "cat", "caaaat")
`?`	0 or 1 occurrence	`colou?r` (Matches "color" or "colour")
`{n}`	Exactly n occurrences	`\d{3}` (Matches "123")
`{n,}`	At least n occurrences	`\d{2,}` (Matches "12", "1234")
`{n,m}`	Between n and m occurrences	`\d{2,4}` (Matches "12", "123", "1234")

Example: Extract Email from Text

```
import re

text = "Contact me at email@example.com for more info."
pattern = r'[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}'

email = re.findall(pattern, text)
print(email)
```

✅ **Output:** `['email@example.com']`

Example: Validate a Phone Number

```
import re

phone = "987-654-3210"
pattern = r'^\d{3}-\d{3}-\d{4}$'

if re.match(pattern, phone):
    print("Valid phone number")
else:
    print("Invalid phone number")
```

✅ **Output:** `Valid phone number`

Summary

Feature	Exception Handling	Regular Expressions
Purpose	Handle runtime errors	Pattern matching in text
Module	Built-in (`try-except`)	`re` module
Key Functions	`try-except-finally`, `raise`, `custom exceptions`	`re.match()`, `re.search()`, `re.findall()`, `re.sub()`
Use Cases	Prevent crashes, error handling	Form validation, text processing

---

Programs of GUI

Simple Login Form

This program creates a simple login form using **Tkinter**, where users can enter their username and password.

```
import tkinter as tk
from tkinter import messagebox

# Function to check the login
def check_login():
    username = entry_username.get()
    password = entry_password.get()

    if username == "admin" and password == "password123":
        messagebox.showinfo("Login Successful", "Welcome, Admin!")
    else:
        messagebox.showerror("Login Failed", "Invalid Username or Password")

# Creating the root window
root = tk.Tk()
root.title("Login Form")
root.geometry("300x200")

# Adding widgets
label_username = tk.Label(root, text="Username")
label_username.pack(pady=5)

entry_username = tk.Entry(root)
entry_username.pack(pady=5)

label_password = tk.Label(root, text="Password")
label_password.pack(pady=5)

entry_password = tk.Entry(root, show="*")
entry_password.pack(pady=5)

login_button = tk.Button(root, text="Login", command=check_login)
login_button.pack(pady=10)

root.mainloop()
```

Simple Table to Display Timetable

This program creates a simple **timetable** using the **Treeview** widget from `ttk` to display data in a tabular format.

```
import tkinter as tk
from tkinter import ttk

# Creating the root window
root = tk.Tk()
root.title("Timetable")
root.geometry("400x300")

# Creating Treeview widget for the timetable
table = ttk.Treeview(root, columns=("Day", "Subject", "Time"), show="headings")

# Defining headings
table.heading("Day", text="Day")
table.heading("Subject", text="Subject")
table.heading("Time", text="Time")

# Inserting data into the table
table.insert("", "end", values=("Monday", "Math", "9:00 AM"))
table.insert("", "end", values=("Tuesday", "Science", "10:00 AM"))
table.insert("", "end", values=("Wednesday", "History", "11:00 AM"))
table.insert("", "end", values=("Thursday", "English", "12:00 PM"))
table.insert("", "end", values=("Friday", "Physical Education", "1:00 PM"))

# Pack the table widget
table.pack(pady=20)

root.mainloop()
```

Simple Table with Buttons

This program uses a **Treeview** widget to create a table and includes a **button** that can add more rows to the table dynamically.

```
import tkinter as tk
from tkinter import ttk

# Function to add a new row to the table
def add_row():
    day = entry_day.get()
    subject = entry_subject.get()
    time = entry_time.get()
    table.insert("", "end", values=(day, subject, time))

# Creating the root window
root = tk.Tk()
root.title("Timetable with Add Row")
root.geometry("400x300")

# Creating the Treeview widget
table = ttk.Treeview(root, columns=("Day", "Subject", "Time"), show="headings")
table.heading("Day", text="Day")
table.heading("Subject", text="Subject")
table.heading("Time", text="Time")
table.pack(pady=20)
```

```
# Entry fields to add data to the table
entry_day = tk.Entry(root)
entry_day.pack(pady=5)
entry_day.insert(0, "Enter Day")

entry_subject = tk.Entry(root)
entry_subject.pack(pady=5)
entry_subject.insert(0, "Enter Subject")

entry_time = tk.Entry(root)
entry_time.pack(pady=5)
entry_time.insert(0, "Enter Time")

# Button to add a row
add_button = tk.Button(root, text="Add Row", command=add_row)
add_button.pack(pady=10)

root.mainloop()
```

Simple Table for Student Grades

This program displays a table for student grades and allows you to add more data using input fields.

```
import tkinter as tk
from tkinter import ttk

# Function to add student grade to the table
def add_grade():
    name = entry_name.get()
    subject = entry_subject.get()
    grade = entry_grade.get()
    table.insert("", "end", values=(name, subject, grade))

# Creating the root window
root = tk.Tk()
root.title("Student Grades")
root.geometry("400x300")

# Creating the Treeview widget for grades
table = ttk.Treeview(root, columns=("Name", "Subject", "Grade"),
show="headings")
table.heading("Name", text="Name")
table.heading("Subject", text="Subject")
table.heading("Grade", text="Grade")
table.pack(pady=20)

# Entry fields for user input
entry_name = tk.Entry(root)
entry_name.pack(pady=5)
entry_name.insert(0, "Enter Name")

entry_subject = tk.Entry(root)
entry_subject.pack(pady=5)
entry_subject.insert(0, "Enter Subject")

entry_grade = tk.Entry(root)
entry_grade.pack(pady=5)
entry_grade.insert(0, "Enter Grade")

# Button to add data to the table
add_button = tk.Button(root, text="Add Grade", command=add_grade)
```

```
add_button.pack(pady=10)

root.mainloop()
```

Explanation of Common Widgets:

- **Button**: Creates clickable buttons. For example, the login button in the login form.
- **Label**: Used to display text or images. For instance, the headings in the timetable.
- **Entry**: A widget for single-line text input, used for username, password, or other forms.
- **Treeview**: A table-like structure that displays data in rows and columns. Used in the timetable and grades example.
- **Button Command**: A button can call a function when clicked. For example, the "Add Row" button in the timetable adds new rows to the table.

Unit 5: Networking and Database Connectivity in Python

5.1 Networking in Python

Python has libraries like `socket`, `urllib`, and `requests` to handle networking tasks. Let's explore these with examples:

Reading Source Code of a Web Page

- **Library Used:** `urllib` or `requests`

Example (using `urllib`):

```
import urllib.request

url = "http://example.com"
response = urllib.request.urlopen(url)
webpage = response.read().decode('utf-8')  # Reading and decoding webpage
content
print(webpage)
```

Downloading Web Pages

- **Library Used:** `requests`

Example:

```
import requests

url = "http://example.com"
response = requests.get(url)

# Save the webpage content to a file
with open("webpage.html", "w") as file:
    file.write(response.text)
print("Webpage downloaded!")
```

Downloading Images

- **Library Used:** `requests`

Example:

```
import requests

url = "https://example.com/image.jpg"
response = requests.get(url)

# Save the image
with open("image.jpg", "wb") as file:
    file.write(response.content)
print("Image downloaded!")
```

5.2 Networking: TCP/IP and UDP Communication

TCP/IP Server

- **Library Used:** `socket`
- A server that listens for incoming connections.

Example:

```
import socket

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(("localhost", 12345))
server.listen(1)
print("Server is listening...")

conn, addr = server.accept()
print("Connected by", addr)
conn.sendall(b"Hello, Client!")  # Sending data
conn.close()
```

TCP/IP Client

Example:

```
import socket

client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect(("localhost", 12345))
print(client.recv(1024).decode())  # Receiving data
client.close()
```

UDP Server

Example:

```
import socket

server = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
server.bind(("localhost", 12345))
print("UDP Server is ready to receive messages...")

while True:
    data, addr = server.recvfrom(1024)
    print("Received message:", data.decode(), "from", addr)
```

UDP Client

Example:

```
import socket

client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
client.sendto(b"Hello, Server!", ("localhost", 12345))
```

File Server

- Transfers files between server and client.

Server Example:

```
import socket

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(("localhost", 12345))
server.listen(1)
print("Waiting for client...")

conn, addr = server.accept()
with open("file.txt", "r") as file:
    conn.sendall(file.read().encode())  # Sending file content
conn.close()
```

Client Example:

```
import socket

client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect(("localhost", 12345))
data = client.recv(1024).decode()
print("File content received:", data)
client.close()
```

Two-Way Communication

Example:

```
# Server
import socket

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(("localhost", 12345))
server.listen(1)
conn, addr = server.accept()

conn.sendall(b"Hello, Client!")
msg = conn.recv(1024).decode()
print("Client says:", msg)
conn.close()

# Client
import socket

client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect(("localhost", 12345))
msg = client.recv(1024).decode()
print("Server says:", msg)
client.sendall(b"Hello, Server!")
client.close()
```

Sending Simple Mail

- **Library Used:** `smtplib`

Example:

```
import smtplib

sender = "youremail@gmail.com"
receiver = "receiveremail@gmail.com"
password = "yourpassword"

message = """Subject: Test Mail
```

```
Hi, this is a test mail sent using Python."""

with smtplib.SMTP("smtp.gmail.com", 587) as server:
    server.starttls()
    server.login(sender, password)
    server.sendmail(sender, receiver, message)
    print("Email sent!")
```

5.3 Database Connectivity

Python supports multiple databases like MySQL, SQLite, PostgreSQL, and Oracle. We'll focus on **MySQL** here using the `mysql-connector` library.

Installing Required Library

```
pip install mysql-connector-python
```

Connecting to MySQL

Example:

```
import mysql.connector

db = mysql.connector.connect(
    host="localhost",
    user="root",
    password="yourpassword",
    database="testdb"
)

print("Connected to the database!")
```

Retrieving Data

Example:

```
cursor = db.cursor()
cursor.execute("SELECT * FROM students")
rows = cursor.fetchall()

for row in rows:
    print(row)
```

Inserting Data

Example:

```
cursor = db.cursor()
cursor.execute("INSERT INTO students (name, age) VALUES (%s, %s)", ("Alice",
20))
db.commit()
print("Data inserted!")
```

Updating Data

Example:

```
cursor = db.cursor()
```

```
cursor.execute("UPDATE students SET age = %s WHERE name = %s", (21, "Alice"))
db.commit()
print("Data updated!")
```

Deleting Data

Example:

```
cursor = db.cursor()
cursor.execute("DELETE FROM students WHERE name = %s", ("Alice",))
db.commit()
print("Data deleted!")
```

Creating Tables

Example:

```
cursor = db.cursor()
cursor.execute("""
CREATE TABLE IF NOT EXISTS students (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255),
    age INT
)
""")
print("Table created!")
```

5.4 Using Oracle Database from Python

Installing Required Library

```
pip install cx_Oracle
```

Connecting to Oracle Database

Example:

```
import cx_Oracle

connection = cx_Oracle.connect(
    user="yourusername",
    password="yourpassword",
    dsn="localhost/XEPDB1"
)

print("Connected to Oracle database!")
```

Executing Stored Procedures

Assume we have a stored procedure in Oracle:

```
CREATE OR REPLACE PROCEDURE greet_user(name IN VARCHAR2)
AS
BEGIN
    DBMS_OUTPUT.PUT_LINE('Hello, ' || name || '!');
END;
```

Calling it from Python:

```
cursor = connection.cursor()
cursor.callproc("greet_user", ["Alice"])
connection.close()
```

Unit 5: Networking and Database Connectivity in Python – Summary Notes

---

5.1 Networking in Python

Networking refers to the exchange of data between systems. Python provides several libraries to perform networking tasks like reading webpages, downloading files, and creating server-client systems.

Key Concepts:

1. **Reading Source Code of a Web Page:**
    - Using `urllib` or `requests`, we can retrieve the HTML source code of a webpage.
    - Useful for web scraping, analyzing content, or automation.
2. **Downloading Webpages and Images:**
    - `requests` library makes downloading files simple.
    - For images, the binary data of the image is downloaded and saved locally.

Applications:

- Automating data extraction from websites.
- Downloading files for offline use.

5.2 Networking: TCP/IP and UDP Communication

Networking models like **TCP/IP** (reliable connection) and **UDP** (faster, connectionless) are supported by Python's `socket` library.

TCP/IP Communication:

- **TCP Server:** Listens for client requests and responds. Useful for chat servers, web servers, etc.
- **TCP Client:** Connects to the server to send/receive data.

Advantages of TCP/IP:

- Reliable communication (data integrity maintained).
- Ensures proper order of data.

UDP Communication:

- **UDP Server/Client:** Used for faster communication where reliability is less critical (e.g., video streaming).
- Connectionless protocol; no acknowledgment of received data.

File Server/Client:

- Enables file sharing between systems over a network.
- Server provides the file, and the client receives it.

Two-Way Communication:

- Allows bidirectional data exchange between a server and a client (e.g., in chat applications).

Sending Simple Mail:

- Python's `smtplib` is used to send emails.
- Requires an email server (e.g., Gmail SMTP) and login credentials.

5.3 Database Connectivity in Python

Python is widely used to interact with databases like MySQL, SQLite, PostgreSQL, and Oracle. Database connectivity allows Python applications to store, retrieve, update, and delete data.

Common Steps:

1. **Connect to the Database:**
    - Use libraries like `mysql-connector` (for MySQL) or `cx_Oracle` (for Oracle).
    - Provide credentials and database details to establish a connection.
2. **Perform Operations:**
    - **Retrieve Data:** Query the database and fetch the results.
    - **Insert Data:** Add new records to the database.
    - **Update Data:** Modify existing records.
    - **Delete Data:** Remove records no longer needed.
    - **Create Tables:** Define a new table structure.
3. **Commit Changes:**
    - For operations like insert, update, and delete, changes must be saved using `commit()`.
4. **Close Connection:**
    - Always release database resources after usage.

5.4 Using Oracle Database from Python

Oracle is a powerful database system that can be used with Python through the `cx_Oracle` library.

Key Features:

- Supports advanced features like stored procedures.
- Enables high-performance queries for enterprise-scale data management.

Stored Procedures:

- These are pre-written SQL scripts stored in the database.
- Python can call these procedures to execute complex logic.

Example Use Case: A stored procedure to greet a user can be invoked from Python, reducing the amount of application-side processing.

Practical Applications of This Unit:

1. **Networking:**
    - Create chat applications using TCP/IP.
    - Build tools to download content from the internet.
    - Set up file transfer systems using Python servers.
2. **Database Connectivity:**
    - Build data-driven applications (e.g., inventory management, online stores).
    - Automate tasks like data insertion and retrieval.
    - Use Python for business analytics by connecting it to databases.
3. **Integration of Networking and Database:**
    - Develop client-server systems where data is stored in a central database.
    - Build web scrapers to extract information and store it in databases for analysis.

Conclusion:

This unit introduces Python's capabilities in networking and database connectivity. It bridges the gap between standalone Python applications and real-world systems that require internet access and database storage. With these skills, you can:

- Automate data handling tasks.
- Build scalable network applications.
- Develop data-driven solutions that interact with databases seamlessly.

These concepts are vital for projects like web scraping, email automation, chat applications, and backend systems.