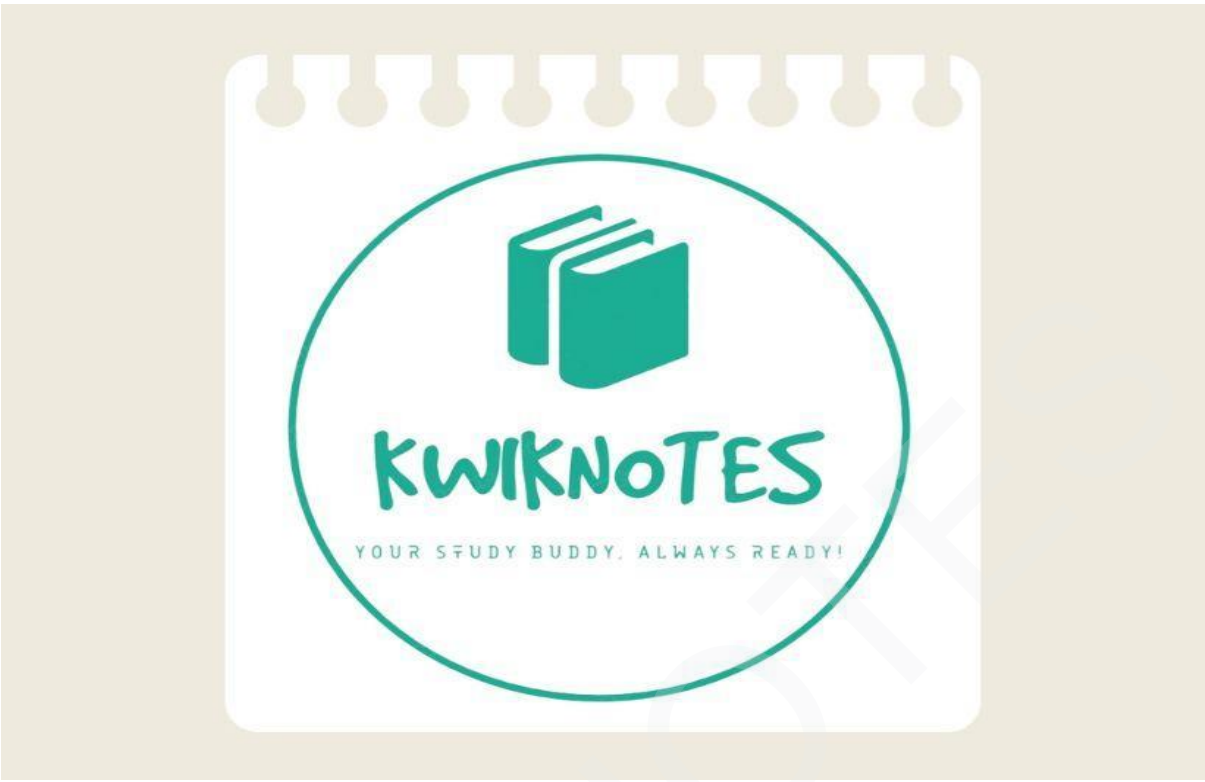

Python Programming

Introduction to Python:

Python is a high-level programming language known for its simplicity and readability.

It supports both structured and object-oriented programming.

Python has a large standard library, making it suitable for a wide range of applications.

Algorithm :

- An algorithm is a sequence of instructions designed to obtain desired results when executed in a specific sequence.

- It should have a clear sequence of instructions, finite steps, and at least one input and output.

Properties of an Algorithm:

- Written in simple English.

- Each step is unique and self-explanatory.

- Must have at least one input and output.

- Has a finite number of steps.

Building Blocks of an Algorithm:

- Sequence: Ordered actions executed in a predetermined order.

- Selection (Decision): Based on a question, the program takes different actions.

- Iteration (Repetition): Repeating a set of actions.

Flowchart:

- A pictorial representation of an algorithm.

- Uses symbols to represent instructions and statements.

- Helps visualize the program logic.

Pseudo Code:

- An informal way to plan program logic.

- Uses simple English and can represent variables, functions, and control structures.

- Easily written and modified.

Advantages of Using a Flowchart:

- Effective communication.
- Proper documentation.
- Efficient coding and debugging.
- Efficient program maintenance.

Disadvantages of Flowcharts:

- Not a visual representation.
- Lacks a complete design picture.

Data Types and Variables

Data Types in Python:

- Numbers (int, float, complex)
- Strings
- Lists
- Tuples
- Dictionaries

Variables:

- A name that refers to a value.
- Assignment statement assigns a value to a variable.

Rules for Variable Names:

- Can contain letters (lowercase/uppercase), digits, and underscores.
- Must start with a letter or underscore.
- Case-sensitive.
- Avoid using special symbols.
- Can be of any length.

Expressions and Statements

Expressions:

- Combinations of values, variables, operators, and function calls.
- Evaluate to a single value.

Statements:

- Instructions that Python can execute.
- Include assignment statements, control flow statements, etc.
- Don't produce a result.

i) Assignment Statements:

- Assign a value to a variable.
- e.g., `a = 5`

ii) Multiline Statements:

- Use the line continuation character (`\`) to continue a statement over multiple lines.

Keywords:

- Reserved words in Python.
- Can't be used as variable names.

eg: if,else,while,def

Operators

1. Arithmetic Operators:

- Perform mathematical operations.
- e.g., `+, -, *, /, %, **`

2. Comparison Operators:

- Compare values.
- e.g., `>, <, >=, <=, ==, !=`

3. Assignment Operators:

- Assign values.
- e.g., `=, +=, -=`

4. Logical Operators:

- Perform logical operations.
- e.g., `and, or, not`

5. Bitwise Operators:

- Operate on binary digits.
- e.g., `&, |, ^`

6. Membership Operators:

- Test for membership in a sequence.

- e.g., `in, not in`

7. Identity Operators:

- Test for object identity.

- e.g., `is, is not`

Control Flow

1. `if` Statement:

- Used to test a condition.

- Executes a block of code if the condition is true.

```
condition = True

if condition:

  print("Condition is true")
```

2. Chained Conditional (`elif`) Statement:

- Used to check multiple conditions in sequence.

```
condition1 = False

condition2 = True

if condition1:

  print("Condition 1 is true")

elif condition2:

  print("Condition 2 is true")
```

3. `for` Loop:

- Iterates over a sequence of elements.

- Uses a variable to represent each element.

```
sequence = [1, 2, 3, 4, 5]

for element in sequence:
```

```
  print(element)
```

4. `while` Loop:

- Repeatedly executes a block of code as long as a condition is true.

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

5.`break` Statement:

- Exits a loop prematurely.

```
for i in range(5):
  if i == 3:
   break
  print(i)
```

6. `continue` Statement:

- Skips the rest of the current iteration in a loop.

```
for i in range(5):
  if i == 3:
   continue
  print(i)
```

7. `range()` Function:

- Generates a sequence of numbers for iteration.

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

8. Pass Statement:

- The `pass` statement is used when you don't want any code to execute.

- It essentially acts as a placeholder and does nothing.

- Syntax: `pass`

```
for i in range(5):
    if i == 3:
        pass  # Placeholder
    else:
        print(i)
```

9. Return Statement in Python:

- The `return` statement is used in functions to specify what value the function should give back when it's called.

- When the `return` statement is executed, the function exits immediately.

- It can be used to send a result, variable, or any expression back to the caller.

- You can have multiple `return` statements in a function, but only one will be executed.

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


result = add(3, 4)  # Returns 7
```

Example:

In this example, the `add` function returns the sum of `a` and `b`. When `add(3, 4)` is called, it returns `7`.

Sure, here are detailed notes with short example code for each of the topics you mentioned:

Functions:

1. Built-in Functions:

- Python provides a rich library of built-in functions, which are ready-to-use and perform various tasks.

- Examples:

```
print("Hello, World!")  # print() displays text on the console.
```

```
length = len("Python")  # len() returns the length of a string.
```

2. User-Defined Functions:

- You can create your own functions using the `def` keyword.

- Example:

```
def greet(name):
    print("Hello, " + name)
greet("Alice")  # Call the greet function
```

3. Anonymous Functions (Lambda Functions):

- Lambda functions are small, anonymous functions defined with the `lambda` keyword.

- Example:

```
double = lambda x: x * 2
result = double(5)
print(result)  # Output: 10
```

Elements of User-Defined Functions:

- User-defined functions consist of a name, parameters, a block of code, and a return statement.

```
def add(a, b):
    result = a + b
    return result
sum = add(3, 4)
```

Arguments and Return Values:

- Functions can accept arguments and return values.

```
def square(x):
    return x * x
result = square(5)
```

Formal vs. Actual Arguments:

- Formal arguments are parameters defined in the function, while actual arguments are values passed when calling the function.

- Example:

```
def greet(name, message):
    print(message + ", " + name)
greet("Bob", "Good morning")  # "name" and "message" are formal arguments
```

Scope and Lifetime:

- Variables in functions have local scope, meaning they're only accessible within the function.

- Example:

```
def my_function():
    x = 10  # x has a local scope within my_function
    print(x)
my_function()
```

Positional, Keyword Arguments & Default Arguments:

- Functions can accept arguments by position or by explicitly naming them.

- Default arguments have preset values.

- Example:

```
def describe_pet(animal, name, color="brown"):
    print("I have a " + color + " " + animal + " named " + name)
describe_pet("dog", "Fido")
describe_pet(name="Whiskers", color="gray", animal="cat")
```

Nested Functions:

- You can define functions within other functions. Inner functions have access to the outer function's variables.

```
def outer_function():
    x = 10
    def inner_function():
        print("Value of x from outer function:", x)
    inner_function()
outer_function()
```

Using Lambdas with filter(), map(), and reduce() functions:

- Lambdas are often used with functions like `filter()`, `map()`, and `reduce()` for quick, inline operations on iterables.

- Example:

```
numbers = [1, 2, 3, 4, 5]
squared_numbers = list(map(lambda x: x ** 2, numbers))
```

Decorators:

- Decorators are functions that modify the behavior of other functions. They are often used for code optimization or to add functionality.

- Example:

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper
@my_decorator
def say_hello():
```

```
    print("Hello!")
say_hello()
```

Iterators:

- Iterators are objects used to loop through containers like lists, tuples, and dictionaries.

- Example:

```
my_list = [1, 2, 3]
my_iterator = iter(my_list)
```

Generators:

- Generators are a type of iterable, but they generate values one at a time when needed, saving memory.

- Example:

```
def my_generator():
    yield 1
    yield 2
    yield 3
gen = my_generator()
value = next(gen)
```

Recursion:

- A function can call itself in a process known as recursion. It's used for solving problems that can be broken down into smaller, similar subproblems.

- Example (Factorial calculation):

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

Modules:

Importing Modules:

- Modules are files containing Python code that can be imported to reuse functions and variables.

- Example:

```
import math

result = math.sqrt(16)
```

Standard Library Modules:

- Python has a rich standard library with modules like `math` (for mathematical functions) and `random` (for random number generation).

- Example:

```
import random

random_number = random.randint(1, 100)
```

Custom Modules:

- You can create your own modules by organizing Python code into separate `.py` files and importing them into your programs.

- Example (Assuming a custom module `my_module.py` exists):

```
import my_module

result = my_module.my_function()
```

Arrays (NumPy)

Single-dimensional Arrays:

- NumPy provides arrays that can hold elements of the same data type.

- Example:

```
import numpy as np
```

```
my_array = np.array([1, 2, 3, 4, 5])
```

Multi-dimensional Arrays (Up to Three Dimensions):

- NumPy allows you to create multi-dimensional arrays, including 2D and 3D arrays.

- Example:

```
import numpy as np

matrix = np.array([[1, 2, 3], [4, 5, 6]])
```

Array Creation using `array`, `linspace`, `logspace`, `arange`, `zeros`, `ones`:

- NumPy provides various functions to create arrays with specific properties:

  - `array()`: Convert lists or tuples to arrays.

  - `linspace()`: Create an array with equally spaced values over a specified range.

  - `logspace()`: Create an array with values that are logarithmically spaced.

  - `arange()`: Create an array with values within a specified range.

  - `zeros()`: Create an array filled with zeros.

  - `ones()`: Create an array filled with ones.

Operations on Arrays:

- NumPy provides various operations like addition, subtraction, element-wise multiplication, and more for arrays.

- Example:

```
import numpy as np

array1 = np.array([1, 2, 3])

array2 = np.array([4, 5, 6])

result = array1 + array2
```

Strings:

- Strings are sequences of characters enclosed in single or double quotes.

- Example:

```
my_string = "Hello, Python"
```

Immutability:

- Strings are immutable, meaning you cannot change the characters in an existing string.

- Example:

```
my_string = "Hello"
my_string[0] = 'J'  # This will result in an error
```

String Creation:

- Strings can be created using single quotes, double quotes, or triple quotes for multi-line strings.

- Example:

```
single_quoted = 'Hello'
double_quoted = "World"
multi_line = '''This is a
multi-line string'''
```

String Indexing and Slicing:

- Strings support indexing and slicing to access individual characters or substrings.

- Example:

```
my_string = "Python"
char = my_string[0]  # Access the first character 'P'
substring = my_string[1:4]  # Slice 'yth' (characters 1, 2, and 3)
```

String Manipulation:

- Python provides many built-in string manipulation methods like `upper()`, `lower()`, `strip()`, `replace()`, and more.

- Example:

```
my_string = "   Hello, Python   "
upper_case = my_string.upper()
stripped = my_string.strip()
replaced = my_string.replace("Python", "World")
```

The Subscript Operator:

- The subscript operator (`[]`) is used for accessing characters in a string using their index.
- Example:

```
my_string = "Python"
first_char = my_string[0]  # Access the first character 'P'
```

Searching Substrings:

- You can check if a substring exists within a string using the `in` operator.
- Example:

```
my_string = "Hello, Python"
contains_hello = "Hello" in my_string  # True
contains_java = "Java" in my_string  # False
```

Certainly, here are detailed notes with short example code for each of the topics you mentioned:

File Handling: Text and Binary Files

Writing and Reading Operations:

- Python allows you to work with files using built-in functions like `open()`, `read()`, `write()`, `close()`, etc.
- Example of writing to a text file:

```
with open("example.txt", "w") as file:
    file.write("Hello, Python!")
```

- Example of reading from a text file:

```
with open("example.txt", "r") as file:
    content = file.read()
```

Random Access to Files:

- You can move the file pointer to a specific location within a file to read or write at that position.

- Example:

```
with open("example.txt", "r") as file:
    file.seek(5)  # Move to the 6th character
    content = file.read(5)  # Read the next 5 characters
```

The `with` Statement:

- The `with` statement is used for proper file handling, ensuring files are automatically closed when done.

- Example:

```
with open("example.txt", "r") as file:
    content = file.read()
```

Pickle in Python:

- Pickle is a module in Python for serializing and deserializing Python objects, allowing you to save and load data structures.

- Example of writing a Python object to a file:

```
import pickle
data = {"name": "Alice", "age": 30}
with open("data.pkl", "wb") as file:
    pickle.dump(data, file)
```

Manipulating Files and Directories:

- Python provides modules like `os` and `shutil` for working with files and directories.

- Example of renaming a file:

```
import os
os.rename("old_file.txt", "new_file.txt")
```

Closing Files:

- Always close files after working with them to ensure data integrity.
- Example:

```
file = open("example.txt", "w")
file.write("Hello, Python!")
file.close()
```

Introduction to Object-Oriented Programming

Features:

- Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects."
- Key features include encapsulation, inheritance, and polymorphism.

Classes & Objects:

- A class is a blueprint for creating objects, and an object is an instance of a class.
- Example of defining a class and creating objects:

```
class Dog:
    def __init__(self, name):
        self.name = name
dog1 = Dog("Buddy")
dog2 = Dog("Max")
```

Immutable vs. Mutable Objects:

- Immutable objects cannot be changed after creation (e.g., strings).
- Mutable objects can be modified after creation (e.g., lists).

- Example:

```
name = "Alice"  # Immutable
name = "Bob"  # Creates a new string
numbers = [1, 2, 3]  # Mutable
numbers.append(4)  # Modifies the list
```

Access Modifiers:

- Access modifiers like public, private, and protected control the visibility of class members.
- Example:

```
class MyClass:
    def __init__(self):
        self.public_var = 42
        self._protected_var = "hidden"
        self.__private_var = "secret"
```

Attributes and Methods:

- Attributes are variables that store data within a class.
- Methods are functions defined within a class.
- Example:

```
class Circle:
    def __init__(self, radius):
        self.radius = radius
    def area(self):
        return 3.14 * self.radius**2
```

Data Hiding:

- Data hiding is achieved using double underscores to make attributes and methods private.
- Example:

```
class MyData:
    def __init__(self):
        self.__private_data = "hidden"
    def get_data(self):
        return self.__private_data
```

The `self` Variable:

- In Python, `self` refers to the instance of the class.
- It is the first parameter in all instance methods.
- Example:

```
class MyClass:
    def __init__(self, data):
        self.data = data
    def display(self):
        print(self.data)
```

Constructor:

- A constructor is a special method used to initialize objects.
- In Python, the constructor is `__init__()`.
- Example:

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

Instance Variables and Class or Static Variables:

- Instance variables belong to an instance of a class.
- Class variables are shared among all instances of a class.

- Example:

```
class Employee:
    company = "ABC Inc."  # Class variable
    def __init__(self, name, salary):
        self.name = name  # Instance variable
        self.salary = salary
```

Inner Classes:

- A class defined inside another class is an inner class.
- Example:

```
class Outer:
    def __init__(self):
        self.outer_data = 42
    class Inner:
        def __init__(self):
            self.inner_data = "Hello"
```

Passing Members of One Class to Another Class:

- You can pass objects of one class as arguments to methods of another class.
- Example:

```
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age
class School:
    def admit_student(self, student):
        print(f"Admitted student: {student.name}")
```

Exception Handling

Error, Exception: Preliminaries and Exception Class Hierarchy:

- Errors are problems in a program that cause it to terminate.
- Exceptions are errors that can be handled during program execution.
- Example:

```
try:
    result = 1 / 0
except ZeroDivisionError as e:
    print(f"An error occurred: {e}")
```

Handling Exceptions using `try`, `except`, and `finally` Clauses:

- Use `try` to enclose code that may raise an exception, and `except` to handle it.
- The `finally` block is executed regardless of whether an exception occurred.
- Example:

```
try:
    result = 1 / 0
except ZeroDivisionError as e:
    print(f"An error occurred: {e}")
finally:
    print("Execution complete")
```

Raising Exceptions:

- You can raise exceptions using the `raise` statement.
- Example:

```
def check_age(age):
    if age < 0:
        raise ValueError("Age cannot be negative")
```

Assertions:

- Assertions are used to check conditions that must be true for the program to continue.
- Example:

```
def divide(a, b):
    assert b != 0, "Cannot divide by zero"
    return a / b
```

User-Defined Exceptions:

- You can define custom exceptions by creating a new class.
- Example:

```
class MyError(Exception):
    def __init__(self, message):
        self.message = message
```

Exception Logging:

- Use the `logging` module to log exception details.
- Example:

```
import logging
try:
    result = 1 / 0
except ZeroDivisionError as e:
    logging.exception("An error occurred")
```

GUI Programming with Tkinter

Creating User Interface:

- Tkinter is a standard GUI library for Python. To create a GUI, you first need to create a main application window.

- Example of creating a basic window:

```
import tkinter as tk
window = tk.Tk()
window.title("My GUI")
window.geometry("400x300")
window.mainloop()
```

GUI Widgets:

- Widgets are the building blocks of a GUI. Common widgets include buttons, labels, entry fields, etc.

- Example of creating a button:

```
button = tk.Button(window, text="Click Me")
button.pack()
```

Creating Layouts:

- You can use geometry managers like `pack`, `grid`, and `place` to arrange widgets in the window.

- Example using `pack`:

```
button1 = tk.Button(window, text="Button 1")
button2 = tk.Button(window, text="Button 2")
button1.pack(side="left")
button2.pack(side="left")
```

Check Box:

- A checkbox is a widget that allows the user to select options.

- Example of creating a checkbox:

```
checkbox = tk.Checkbutton(window, text="Check Me")
checkbox.pack()
```

Radio Buttons:

- Radio buttons allow the user to select a single option from a group.
- Example of creating radio buttons:

```
radio_var = tk.IntVar()
radio1 = tk.Radiobutton(window, text="Option 1", variable=radio_var, value=1)
radio2 = tk.Radiobutton(window, text="Option 2", variable=radio_var, value=2)
radio1.pack()
radio2.pack()
```

List Box:

- A list box is a widget for displaying a list of items from which the user can select one or more.
- Example of creating a list box:

```
listbox = tk.Listbox(window)
listbox.insert(1, "Item 1")
listbox.insert(2, "Item 2")
listbox.pack()
```

Menus:

- Menus provide a way to create dropdown menus in your application.
- Example of creating a menu bar:

```
menu_bar = tk.Menu(window)
file_menu = tk.Menu(menu_bar, tearoff=0)
file_menu.add_command(label="New")
file_menu.add_command(label="Open")
file_menu.add_separator()
```

```
file_menu.add_command(label="Exit", command=window.quit)

menu_bar.add_cascade(label="File", menu=file_menu)

window.config(menu=menu_bar)
```

Dialog Boxes:

- Dialog boxes are used for user interactions like file open or save dialogs.

- Example of a file dialog:

```
from tkinter import filedialog

file_path = filedialog.askopenfilename()
```

Tables:

- Tkinter itself does not provide table widgets. You can create tables using frames and labels or consider using external libraries like `tkintertable`.

Network Programming

Basics of Sockets:

- Sockets are endpoints for sending or receiving data across a computer network.

- Python provides the `socket` library for socket programming.

Socket Methods:

- Common socket methods include `socket()`, `bind()`, `listen()`, `accept()`, `connect()`, `send()`, and `recv()`.

TCP and UDP Sockets:

- TCP (Transmission Control Protocol) provides reliable, connection-oriented communication.

- UDP (User Datagram Protocol) offers connectionless, low-latency communication.

Two-Way Client-Server Communication:

- You can create client-server applications where the server listens for connections and the client connects to the server.

Sending Email:

- Python's `smtplib` library is used for sending emails through SMTP (Simple Mail Transfer Protocol).

Database Access

Advantages of a DBMS over Files:

- Database Management Systems (DBMS) offer data integrity, security, and efficient data retrieval compared to flat files.

Database Connectivity Operations:

- Operations include creating, inserting, selecting, deleting, dropping, updating, and performing joins in a database.

- Example using SQLite for database operations:

```
import sqlite3

connection = sqlite3.connect("mydb.db")

cursor = connection.cursor()

cursor.execute("CREATE TABLE IF NOT EXISTS students (id INTEGER PRIMARY
KEY, name TEXT, age INTEGER)")

cursor.execute("INSERT INTO students (name, age) VALUES (?, ?)",
("Alice", 25))

cursor.execute("SELECT * FROM students WHERE age > 20")

results = cursor.fetchall()

connection.commit()

connection.close()
```