P.W. SKILLS- Python Basics Assignment

1. Explain the key features of Python that make it a popular choice for programming.

Ans: Python is a high-level, interpreted, dynamic programming language that supports object-oriented and procedural programming.

- KEY FEATURES OF PYTHON ARE:

1. Readability and Simplicity

- **Clear Syntax**: Python's syntax is designed to be intuitive and easy to read, resembling plain English. This makes it accessible for beginners and helps in maintaining and understanding code.
- **Minimalistic Design**: Python emphasizes simplicity and reduces the clutter often seen in other programming languages, allowing developers to focus on solving problems.

2. Versatile and Multiparadigm

- **Multi-Paradigm**: Python supports multiple programming paradigms, including procedural, object-oriented, and functional programming, allowing developers to choose the best approach for their project.
- **Wide Applicability**: It can be used for web development, data analysis, artificial intelligence, scientific computing, automation, and more, making it suitable for a wide range of applications.

3. Extensive Standard Library

- **Rich Libraries and Frameworks**: Python has a vast standard library and a thriving ecosystem of third-party packages (e.g., NumPy, pandas, Django, Flask), which simplifies tasks such as web development, data manipulation, and machine learning.
- **Built-in Functions**: The standard library includes many modules and functions for common tasks, reducing the need for writing code from scratch.

4. Community and Support

- **Large Community**: Python has a massive and active community that contributes to a wealth of resources, tutorials, and documentation. This community support is invaluable for both beginners and experienced developers.
- **Open Source**: Being open-source, Python is freely available, allowing users to contribute to its development and share their libraries and tools with the community.

5. Cross-Platform Compatibility

- **Platform Independence**: Python can run on various operating systems (Windows, macOS, Linux) without requiring changes to the code, facilitating easy deployment and collaboration across different environments.

6. Strong Support for Integration

- **Interoperability**: Python easily integrates with other languages (e.g., C, C++, Java) and technologies, making it a good choice for projects that require combining different systems.
- **APIs and Web Services**: Python can efficiently handle APIs and web services, which is essential for modern software development.

7. Rapid Development and Prototyping

- **Fast Development Cycle**: Python's simplicity and readability allow for rapid development and prototyping, enabling developers to build applications quickly and iterate on them efficiently.
- **Dynamic Typing**: Python is dynamically typed, allowing developers to write code without explicitly declaring variable types, which can speed up development but may require more careful testing.

8. Strong Data Handling and Visualization

- **Data Analysis Libraries**: Python has powerful libraries like pandas, NumPy, and Matplotlib, making it a preferred choice for data analysis, manipulation, and visualization.
- **Machine Learning and AI**: Libraries such as TensorFlow, Keras, and Scikit-learn have made Python a go-to language for machine learning and artificial intelligence applications.

9. Educational Use

- **Popular in Academia**: Python is often used as a teaching language in computer science and programming courses due to its simplicity, encouraging new programmers to learn coding concepts without getting bogged down by complex syntax.

---

2. Describe the role of predefined keywords in Python and provide examples of how they are used in a program.

Ans: Predefined keywords in Python, often referred to as **reserved words**, play a critical role in the structure and functionality of the language. These keywords have special meanings and cannot be used for any other purpose, such as naming variables or functions. They are essential for defining the syntax and behaviour of the Python programming language.

- Role of Predefined Keywords in Python:

1. **Control Flow**: Keywords such as if, elif, else, for, and while are used to control the flow of the program, allowing for conditional execution and iteration.
2. **Function Definitions**: The def keyword is used to define functions, making it clear where a function starts and how it behaves.
3. **Data Types**: Keywords like True, False, and None represent fundamental data types that are integral to Python's functionality.
4. **Exception Handling**: Keywords such as try, except, finally, and raise are used to handle errors and exceptions in a controlled manner.
5. **Class Definitions**: The class keyword is used to define new classes, supporting object-oriented programming.
6. **Context Managers**: The with keyword is used to manage resources effectively, ensuring that they are properly cleaned up after use.

List of Common Python Keywords

- Control Flow: if, else, elif, for, while, break, continue
- Function and Class Definitions: def, class, return
- Importing Modules: import, from, as
- Error Handling: try, except, finally, raise
- Boolean Logic: and, or, not
- Miscellaneous: True, False, None, in, is, with, async, await

- **Examples of How Keywords Are Used in a Program:**

```
# Function definition using the 'def' keyword
def greet(name):
return f"Hello, {name}!"
# Main block of code
if __name__ == "__main__":  # Check if the script is being run directly
names = ["Jack", "Marie", "Stephen"]  # List of names
# Loop through the list using 'for'
for name in names:
message = greet(name)  # Call the greet function
print(message)  # Output the greeting
# Using 'try' and 'except' for error handling
try:
# Attempting to convert a string to an integer
number = int(input("Enter a number: "))
print(f"You entered: {number}")
except ValueError:  # Handle the case where conversion fails
print("That's not a valid number!")
finally:
print("Program has finished.")  # This block always runs
```

---

3. Compare and contrast mutable and immutable objects in Python with examples.

Ans: In Python, objects can be classified as either **mutable** or **immutable**, depending on whether their state or value can be changed after they are created. Understanding the differences between these two types of objects is crucial for writing effective Python code.

1. Mutable objects are objects whose state or value can be changed after they are created. This means you can modify the content of these objects without creating a new object.

- Common Mutable Objects:
- Lists
- Dictionaries
- Sets

2. Immutable objects are objects whose state or value cannot be changed after they are created. Any modification to an immutable object results in the creation of a new object.

- Common Immutable Objects:
- Strings
- Tuples
- Frozen sets

- Contrast between Mutable and Immutable Objects:

FEATURES	MUTABLE OBJECTS	IMMUTABLE OBJECTS
DEFINITION	Can be changed after creation.	Cannot be changed after creation.
EXAMPLES	Lists, dictionaries, sets.	Strings, tuples, frozen sets.
MEMORY BEHAVIOUR	Modifications affect the same object in memory.	Modifications create a new object in memory.
PERFORMANCE	Generally faster for updates due to in-place changes.	May involve more overhead due to new object creation.
USE CASES	Ideal for collections of items that may change.	Ideal for fixed data, keys in dictionaries, and function arguments

EXAMPLES OF MUTABLE OBJECTS:

1. Lists

```
my_list = [1, 2, 3]
```

```
my_list[0] = 10        # Changing an element
my_list.append(4)      # Adding an element
my_list.remove(2)      # Removing an element
print(my_list)         # Output: [10, 3, 4]
```

2. Dictionaries

```
my_dict = {'a': 1, 'b': 2}
my_dict['a'] = 10      # Changing a value
my_dict['c'] = 3       # Adding a new key-value pair
del my_dict['b']       # Removing a key-value pair
print(my_dict)         # Output: {'a': 10, 'c': 3}
```

3. Sets

```
my_set = {1, 2, 3}
my_set.add(4)          # Adding an element
my_set.remove(2)       # Removing an element
print(my_set)          # Output: {1, 3, 4}
```

EXAMPLES OF IMMUTABLE OBJECTS:

1. Tuples

```
my_tuple = (1, 2, 3)
# my_tuple[0] = 10    # This will raise a TypeError
print(my_tuple)        # Output: (1, 2, 3)
```

2. Strings

```
my_string = "hello"
# my_string[0] = 'H'  # This will raise a TypeError
new_string = my_string.upper()  # Creates a new string
print(new_string)      # Output: "HELLO"
```

3. Frozensets

```
my_frozenset = frozenset([1, 2, 3])

# my_frozenset.add(4)  # This will raise an AttributeError

print(my_frozenset)    # Output: frozenset({1, 2, 3})
```

---

4. Discuss the different types of operators in Python and provide examples of how they are used.

Ans: Python divides the operators in the following groups:

1. Arithmetic operators
2. Comparison operators
3. Logical operators
4. Assignment operators
5. Bitwise operators
6. Identity operators
7. Membership operators

1. Arithmetic Operators

Arithmetic operators are used to perform mathematical calculations which include addition, subtraction, multiplication, division, floor division, modulus (remainder), and exponentiation.

- **EXAMPLE:**

```
a = 10
b = 3
print("Addition:", a + b)              # Output: 13
print("Subtraction:", a - b)           # Output: 7
print("Multiplication:", a * b)        # Output: 30
print("Division:", a / b)              # Output: 3.3333
print("Floor Division:", a // b)       # Output: 3
print("Modulus:", a % b)               # Output: 1
print("Exponentiation:", a ** b)       # Output: 1000
```

2. Comparison Operators

Comparison operators are used to compare two values and return a Boolean result (True or False). This includes operators like- equal to, not equal to, greater than, less than, greater than or equal to, and less than or equal to.

- **EXAMPLE:**

```
x = 10
y = 20
print("Equal to:", x == y)                          # Output: False
print("Not equal to:", x != y)                      # Output: True
print("Greater than:", x > y)                       # Output: False
print("Less than:", x < y)                          # Output: True
print("Greater than or equal to:", x >= y)          # Output: False
print("Less than or equal to:", x <= y)             # Output: True
```

3. Logical Operators

Logical operators are used to combine conditional statements. This includes 3 operators –

and - Returns True if both operands are true

or - Returns True if at least one operand is true

not - Returns True if the operand is false

- **EXAMPLE:**

```
a = True
b = False

print("AND Operator:", a and b)  # Output: False
print("OR Operator:", a or b)    # Output: True
print("NOT Operator:", not a)    # Output: False
```

4. Assignment Operators

Assignment operators are used to assign values to variables.

- **EXAMPLE:**

```
x = 10
x += 5  # x = x + 5
print("Add and Assign:", x)  # Output: 15
x *= 2  # x = x * 2
print("Multiply and Assign:", x)  # Output: 3
```

5. Bitwise Operators

Bitwise operators are used to perform operations on binary representations of integers.

- **EXAMPLE:**

```
a = 5  # binary: 0101
b = 3  # binary: 0011

print("Bitwise AND:", a & b)  # Output: 1 (binary: 0001)
print("Bitwise OR:", a | b)   # Output: 7 (binary: 0111)
```

```
print("Bitwise XOR:", a ^ b)  # Output: 6 (binary: 0110)
print("Bitwise NOT:", ~a)      # Output: -6 (inverts bits)
print("Left Shift:", a << 1)   # Output: 10 (binary: 1010)
print("Right Shift:", a >> 1)  # Output: 2 (binary: 0010)
```

6. Identity Operators

Identity operators are used to compare the memory locations of two objects.

- **EXAMPLE:**

```
a = [1, 2, 3]
b = a  # b points to the same list as a
c = list(a)  # c is a new list that is a copy of a

print("a is b:", a is b)      # Output: True
print("a is c:", a is c)      # Output: False
print("a is not c:", a is not c)  # Output: True
```

7. Membership Operators

Membership operators are used to test if a value or variable is found in a sequence (like a list, tuple, or string).

- **EXAMPLE:**

```
my_list = [1, 2, 3, 4, 5]
print("Membership Operator (in):", 3 in my_list)  # Output: True
print("Membership Operator (not in):", 6 not in my_list)  # Output: True
```

---

5. Explain the concept of type casting in Python with examples.

Ans: Type casting in Python refers to the conversion of one data type into another. This is a common operation when you need to perform operations that require compatible data types, or when you want to ensure that values are treated in a specific way. Python provides several built-in functions for type casting, making it easy to convert between types such as integers, floats, strings, lists, and tuples.

Here is the most commonly used type casting functions in Python:

1. **int()**: Converts a value to an integer.
2. **float()**: Converts a value to a float.
3. **str()**: Converts a value to a string.
4. **list()**: Converts a value to a list.
5. **tuple()**: Converts a value to a tuple.
6. **set()**: Converts a value to a set.
7. **dict()**: Converts a sequence of key-value pairs to a dictionary.

- **Examples of Type Casting:**

1. Converting to Integer

The int() function can convert a float or a string representation of a number into an integer. If the float has a decimal part, it will be truncated (not rounded).

```
# Converting a float to an integer
num_float = 5.9
num_int = int(num_float)
print("Float to Integer:", num_int)  # Output: 5

# Converting a string to an integer
num_str = "42"
num_int_from_str = int(num_str)
print("String to Integer:", num_int_from_str)  # Output: 42
```

2. Converting to Float

The float() function converts integers and string representations of numbers into floats.

```
# Converting an integer to a float
num_int = 5
num_float = float(num_int)
print("Integer to Float:", num_float)  # Output: 5.0

# Converting a string to a float
num_str = "3.14"
num_float_from_str = float(num_str)
print("String to Float:", num_float_from_str)  # Output: 3.14
```

3. Converting to String

You can convert integers, floats, and other types into strings using the str() function.

```
# Converting an integer to a string
num_int = 100
num_str = str(num_int)
print("Integer to String:", num_str)  # Output: '100'

# Converting a float to a string
num_float = 3.14
num_str_from_float = str(num_float)
print("Float to String:", num_str_from_float)  # Output: '3.14'
```

4. Converting to List

The list() function can convert strings, tuples, and other iterable types into lists.

```
# Converting a string to a list of characters
string_value = "hello"
list_from_string = list(string_value)
print("String to List:", list_from_string)  # Output: ['h', 'e', 'l', 'l', 'o']

# Converting a tuple to a list
tuple_value = (1, 2, 3)
list_from_tuple = list(tuple_value)
print("Tuple to List:", list_from_tuple)  # Output: [1, 2, 3]
```

5. Converting to Tuple

The tuple() function can convert lists and strings into tuples.

```
# Converting a list to a tuple
list_value = [1, 2, 3]
tuple_from_list = tuple(list_value)
print("List to Tuple:", tuple_from_list)  # Output: (1, 2, 3)

# Converting a string to a tuple
string_value = "abc"
tuple_from_string = tuple(string_value)
print("String to Tuple:", tuple_from_string)  # Output: ('a', 'b', 'c')
```

6. Converting to Set

The set() function can convert lists, tuples, and strings into sets, which are unordered collections of unique elements.

```
# Converting a list to a set
list_value = [1, 2, 2, 3, 4]
set_from_list = set(list_value)
print("List to Set:", set_from_list)  # Output: {1, 2, 3, 4}

# Converting a string to a set of unique characters
string_value = "hello"
set_from_string = set(string_value)
print("String to Set:", set_from_string)  # Output: {'h', 'e', 'l', 'o'}
```

---

6. How do conditional statements work in Python? Illustrate with examples.

Ans: Conditional statements in Python are used to execute specific blocks of code based on whether a condition is true or false. They allow your program to make decisions and perform different actions depending on the values of variables or the results of expressions. The main conditional statements in Python include if, elif, and else.

- Basic Structure of Conditional Statements:

1. **if** statement: This is the primary conditional statement. If the condition evaluates to True, the code block inside the if statement is executed.
2. **elif** statement: Short for "else if," this statement allows you to check multiple conditions. If the preceding if condition is False, the elif condition is evaluated next.
3. **else** statement: This is executed if none of the previous conditions (if or elif) are True.

- **Examples**

Example 1: Basic if Statement

```
age = 18

if age >= 18:
    print("You are an adult.")
```

Example 2: Using if and else

```
age = 16
if age >= 18:
    print("You are an adult.")
else:
    print("You are not an adult yet.")
```

Example 3: Using if, elif, and else

```
marks = 75
if marks >= 90:
    print("Grade: A")
elif marks >= 75:
    print("Grade: B")
elif marks >= 60:
    print("Grade: C")
else:
  print("Grade: D")
```

Example 4: Nested Conditional Statements

```
temperature = 30

if temperature > 25:
    print("It's a hot day.")
    if temperature > 35:
        print("It's extremely hot!")
else:
```

```
print("It's a cool day.")
```

---

7. Describe the different types of loops in Python and their use cases with examples.

Ans: In Python, loops are used to execute a block of code repeatedly as long as a specified condition is true or for a certain number of iterations. The two primary types of loops in Python are the **for loop** and the **while loop**. Each type has its use cases and can be controlled using loop control statements such as break and continue. Below is a description of each loop type along with examples.

1. The for Loop

The for loop is used to iterate over a sequence (like a list, tuple, string, or range). It allows you to execute a block of code for each item in the sequence.

- Use Cases:
- Iterating through elements in a list or tuple.
- Processing characters in a string.
- Executing a block of code a specific number of times using the range() function.

Example 1: Iterating Over a List

```
fruits = ['apple', 'banana', 'cherry']

for fruit in fruits:
    print(fruit)
```

Example 2: Using range()

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

2. The while Loop

The while loop repeatedly executes a block of code as long as a specified condition is true. It is useful when the number of iterations is not known in advance.

- Use Cases:
- Repeating a task until a certain condition is met.
- Continuously asking for user input until valid data is provided.

Example 1: Basic while Loop

```
count = 0
```

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

Example 2: User Input Validation

```
password = ""
while password != "secret":
    password = input("Enter password: ")

print("Access granted!")
```

3. Loop Control Statements

Both for and while loops can be controlled using the following statements:

- **break**: Exits the loop immediately.
- **continue**: Skips the current iteration and continues with the next iteration of the loop.
- **else**: A loop can have an else block that executes when the loop completes normally (not terminated by a break).

Example 1: Using break

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

Example 2: Using continue

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

Example 3: Using else with a Loop

```
for i in range(3):
    print(i)
else:
    print("Loop completed.")
```

4. Nested Loops

You can place one loop inside another loop. This can be useful for working with multi-dimensional data structures.

Use Cases:

- When working with matrices or lists of lists.
- When you need to compare elements of two sequences.

```
# Example: Nested for loops to create a multiplication table

for i in range(1, 4):  # outer loop

    for j in range(1, 4):  # inner loop

        print(i * j, end=' ')

    print()  # for a new line after each row
```
