Name: - Girish Shenoy

Assignment 1

Python Basics Questions

1. What is Python, and why is it popular?

Sol: -

Python is simple and interpreted programming language.

Python is popular for the following reasons:-

i) Easy to understand: - python's syntax is simple and easy to understand.

ii) Object Oriented Programming:- Python is an object oriented programming language.

iii) Interpreted: - Python executes the code line by line which is very useful for begineers

iv) Platform Independent: - Python code written once can run anywhere.

2. What is an interpreter in Python?

Sol:-

An interpreter is a computer program that directly executes instructions written in a programming or scripting language, without requiring them previously to have been compiled into a machine-language program. Python is an interpreted language, which means the code is executed line by line by the interpreter.

3. What are pre-defined keywords in Python?

Sol:-

Pre-defined keywords in Python are special reserved words that have built-in meanings and can't be used as variable names.

Some of the keywords are if, else, for, while, def, return, and, or, not, is, True, False, None, try, except, raise and many more

🔍 To see all keywords: import keyword print(keyword.kwlist)

4. Can keywords be used as variable names?

Sol:-

Pre-defined keywords in Python — also called reserved keywords — are words that are built into the language syntax. You can't use them as variable names, function names, or identifiers because they serve a specific purpose in Python's grammar.

If any pre-defined keywords are used as variable names, then it will show Syntax Error.

5. What is mutability in Python?

Sol:-

Mutability in Python means whether an object can be changed after creation.

✅ Mutable: Can change — list, dict, set

❌ Immutable: Can't change — int, str, tuple, bool

Example:

x = [1, 2, 3]

x[0] = 9 # mutable list

y = "hello"

y[0] = "H" # error, strings are immutable

6. Why are lists mutable, but tuples are immutable?

Sol:-

🔁 Lists are mutable because:

- They are designed for dynamic data.
- You can add, remove, or change items freely.
- Python stores list elements in memory in a way that supports modification.
- For Example: -

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

my_list[0] = 10 # Works
```

🔒 Tuples are immutable because:

- They are meant for fixed, constant data.
- Immutability makes them **faster**, **hashable**, and safe for use as dictionary keys.
- Python stores them in a way that prevents changes.
- For Example: -

```
my_tuple = (1, 2, 3)

my_tuple[0] = 10  # Error
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-2-a4102b41870d> in <cell line: 0>()
      1 my_tuple = (1, 2, 3)
      2
----> 3 my_tuple[0] = 10  # Error

TypeError: 'tuple' object does not support item assignment
```

7. What is the difference between “==” and “is” operators in Python?

Sol:-

== (Equality)	is (Identity)
It compares Values (content)	It comapres Memory location (object identity)
It returns True if values are equal; otherwise False	It returns True if both reference the exact same object; otherwise False
It check if two variables have the same data	It check if two variables point to the same object
It is overloadable (custom classes can define **eq**)	It cannot be overridden
Always safe for Immutable; compares value equality	It may return True for small immutable literals due to interning, but should not be relied upon for value checks
Example: - [1, 2] == [1, 2] → True	[1, 2] is [1, 2] → False

8. What are logical operators in Python?

Sol:-

Logical operators are used to combine conditional statements.

Operator	Meaning	Example
and	True if both are true	x > 5 and x < 10
or	True if at least one is true	x > 5 or x < 2
not	Reverses the condition	not(x > 5) → True if x ≤ 5

These are often used in if statements to control program flow.

- For Example:

```
x = 7

# Using 'and'
print(x > 5 and x < 10)  # True

# Using 'or'
print(x > 10 or x == 7)  # True

# Using 'not'
print(not(x > 10))       # True
```

```
True
True
True
```

9. What is type casting in Python?

Sol:-

- Type Casting is the conversion of a one datatype into another is called as type casting.
- It has 2 types: - Implicit And Explicit Type Casting
- In Implicit Type Casting, one small datatype is converted into bigger datatypes and Python auto-converts during operations.
- For Example: -

```
x = 5 + 2.0  # x becomes 7.0 (int → float)
```

10. What is the difference between implicit and explicit type casting?

Sol:-

Implicit Type Casting	Explicit Type Casting
It is automatically done by Python	It is done by Programmer manually
It is done silently during operations	It uses functions like int(), float()
It has less control	It has full control
It has lower risk of data loss	It has higher risk of data loss if not handled properly
For Example: - x = 5 + 2.0 → x becomes 7.0	For Example: - x = int("10") → x becomes 10

```
# Code Examples

# Implicit Casting
x = 5      # int
y = 2.0    # float
z = x + y  # z becomes 7.0 (float)

# Explicit Casting
a = "10"
b = int(a)  # b becomes 10 (int)
```

11. What is the purpose of conditional statements in Python?

Sol:-

Conditional statements let your Python programs make decisions and execute different code paths based on whether given conditions evaluate to True or False.

- Control Flow Directs the program to run certain blocks only if specific criteria are met.
- Decision Making Enables logic like "if this, do that; otherwise, do something else."
- Readability & Maintainability Structures complex logic clearly with if / elif / else branches.

```
age = 18

if age >= 18:
    print("You can vote.")
elif age > 13:
    print("You are a teenager.")
else:
    print("You are a child.")
```

```
You can vote.
```

Here, the if/elif/else statements control which message is printed based on age.

12. How does the elif statement work?

Sol:-

The elif keyword in Python stands for else if and lets you test additional conditions only if the preceding if (or elif) was False.

```
x = 15

if x < 10:
    print("x is less than 10")
elif x < 20:
    print("x is between 10 and 19")
elif x < 30:
    print("x is between 20 and 29")
else:
    print("x is 30 or more")
```

How it works

1. Evaluate the first if condition (x < 10).
2. If True, run its block and skip all following elif/else.
3. If False, check the next elif condition (x < 20).
4. Repeat for each elif.
5. If none are True, run the final else block (if provided).

This chaining lets you handle multiple, mutually exclusive cases in one structured block.

13. What is the difference between for and while loops?

Sol:-

for Loop	while Loop
It iterates over each item in a sequence (list, tuple, string, etc.)	It repeats as long as a condition remains True
It stops when it reaches the end of the sequence	It stops when the condition evaluates to False
The iteration count is determined by sequence length	The iteration count depends on runtime logic
It is used for processing or transforming each element	It is used for waiting for or polling on a dynamic condition
Low Infinite Loop Risk due to finite sequence	Higher Infinite Loop Risk as it must ensure condition will eventually fail
Control Keywords include break continue	Control Keywords include break, continue

14. Describe a scenario where a while loop is more suitable than a for loop.

Sol:-

- A while loop shines when you need to repeat some action until an unpredictable condition becomes false—especially when you don't know up front how many iterations you'll need.

Scenario: Waiting for Valid User Input

- Imagine you're writing a login prompt that keeps asking for a password until the user gets it right. You don't know in advance how many attempts they'll take, so a for loop (which iterates a fixed number of times) isn't ideal.

```
correct_password = "open_sesame"
user_input = ""

# Keep asking until the password matches
while user_input != correct_password:
    user_input = input("Enter password: ")
    if user_input != correct_password:
        print("Incorrect—try again.")

print("Access granted!")
```

- Why while is better here:

Dynamic exit condition: Continues until user_input == correct_password.

- Unknown iteration count:

You can't predict how many tries the user needs.

- Simplicity:

The loop's continuation condition directly mirrors the real-world requirement.

- Other common uses of while loops:

Reading sensor data until a stop signal arrives.

Polling an API until a specific status is returned.

Game loops that run until the player quits.

Practical Questions

1. Write a Python program to print "Hello, World!"

Sol:-

```
print("Hello World!")
```

```
Hello World!
```

2. Write a Python program that displays your name and age.

Sol:-

```
name = "Ram"
age = 30

print("Name:", name)
print("Age:", age)
```

```
Name: Ram
Age: 30
```

3. Write code to print all the pre-defined keywords in Python using the keyword library.

Sol:-

```
import keyword

keyword.kwlist
```

```
['False',
 'None',
 'True',
 'and',
 'as',
 'assert',
 'async',
 'await',
 'break',
 'class',
 'continue',
 'def',
 'del',
 'elif',
 'else',
 'except',
 'finally',
 'for',
 'from',
 'global',
 'if',
 'import',
 'in',
 'is',
 'lambda',
 'nonlocal',
 'not',
 'or',
 'pass',
 'raise',
 'return',
 'try',
 'while',
 'with',
 'yield']
```

4. Write a program that checks if a given word is a Python keyword.

Sol:-

```
word_to_check = "is not"

if word_to_check in keyword.kwlist:
  print(f"'{word_to_check}' is a Python keyword.")
```

```
else:
  print(f"'{word_to_check}' is not a Python keyword.")
```

```
'is not' is not a Python keyword.
```

5. Create a list and tuple in Python, and demonstrate how attempting to change an element works differently for each

Sol:-

```
# Creating a list

my_list = [1, 2, 3, 4, 5]
print("Original list:", my_list)

# Attempt to change an element in the list
my_list[0] = 10
print("List after changing element:", my_list)

# Create a tuple
my_tuple = (1, 2, 3, 4, 5)
print("\nOriginal tuple:", my_tuple)

# Attempt to change an element in the tuple
try:
  my_tuple[0] = 10
except TypeError as e:
  print("Attempting to change an element in the tuple resulted in an error:", e)
```

```
Original list: [1, 2, 3, 4, 5]
List after changing element: [10, 2, 3, 4, 5]

Original tuple: (1, 2, 3, 4, 5)
Attempting to change an element in the tuple resulted in an error: 'tuple' object does not support item assignment
```

6. Write a function to demonstrate the behavior of mutable and immutable arguments

Sol:-

```
def demonstrate_mutability(mutable_list, immutable_string):
  """
  Args:
    mutable_list: A list (mutable).
    immutable_string: A string (immutable).
  """

  print(f"Inside function: Initial mutable list: {mutable_list}")
  print(f"Inside function: Initial immutable string: {immutable_string}")

  # Modify the mutable argument
  mutable_list.append(4)
  print(f"Inside function: Modified mutable list: {mutable_list}")

  # Attempt to modify the immutable argument (creates a new string object)
  immutable_string = immutable_string + " World!"
  print(f"Inside function: Modified immutable string: {immutable_string}")


# Demonstrate with a list (mutable)
my_list = [1, 2, 3]
print(f"\nBefore function call: my_list: {my_list}")

demonstrate_mutability(my_list, "Hello")

# The list is modified
print(f"After function call: my_list: {my_list}")

# Demonstrate with a string (immutable)
my_string = "Hello"
print(f"\nBefore function call: my_string: {my_string}")

demonstrate_mutability([10, 20], my_string)

# The string is not modified
print(f"After function call: my_string: {my_string}")
```

```

Before function call: my_list: [1, 2, 3]
Inside function: Initial mutable list: [1, 2, 3]
Inside function: Initial immutable string: Hello
Inside function: Modified mutable list: [1, 2, 3, 4]
Inside function: Modified immutable string: Hello World!
After function call: my_list: [1, 2, 3, 4]

Before function call: my_string: Hello
Inside function: Initial mutable list: [10, 20]
Inside function: Initial immutable string: Hello
Inside function: Modified mutable list: [10, 20, 4]
Inside function: Modified immutable string: Hello World!
After function call: my_string: Hello
```

7. Write a program that performs basic arithmetic operations on two user-input numbers

Sol:-

```
# Get input from the user
num1 = float(input("Enter the first number: "))
num2 = float(input("Enter the second number: "))

# Perform basic arithmetic operations
addition = num1 + num2
subtract = num1 - num2
multiply = num1 * num2


# Check for division by zero before performing division
if num2 != 0:
  divide = num1 / num2
else:
  print("Division by zero is not allowed.")

print(f"Sum: {addition}")
print(f"Difference: {subtract}")
print(f"Product: {multiply}")
print(f"Division: {divide}")
```

```
Enter the first number: 25
Enter the second number: 41
Sum: 66.0
Difference: -16.0
Product: 1025.0
Division: 0.6097560975609756
```

8. Write a program to demonstrate the use of logical operators.

Sol:-

```
# Define some variables
a = True
b = False
c = True

# Using AND operator

# Output: False
print("a and b:", a and b)

# Output: True
print("a and c:", a and c)

# Using OR operator

# Output: True
print("a or b:", a or b)

# Output: False
print("b or b:", b or b)

# Using NOT operator

# Output: False
print("not a:", not a)

# Output: True
print("not b:", not b)

# Combining and and or logical operators

# Output: True
```

```
print("(a and b) or c:", (a and b) or c)

# Output: True
print("a and (b or c):", a and (b or c))
```

```
a and b: False
a and c: True
a or b: True
b or b: False
not a: False
not b: True
(a and b) or c: True
a and (b or c): True
```

9. Write a Python program to convert user input from string to integer, float, and boolean types

Sol:-

```
user_input_str = input("Enter a value: ")

# Converting to integer
try:
  user_input_int = int(user_input_str)

  print(f"Converted to integer: {user_input_int} (Type: {type(user_input_int)})")
except ValueError:
  print(f"Could not convert '{user_input_str}' to integer.")

# Converting to float
try:
  user_input_float = float(user_input_str)

  print(f"Converted to float: {user_input_float} (Type: {type(user_input_float)})")
except ValueError:
  print(f"Could not convert '{user_input_str}' to float.")

# Converting to boolean (simple examples)
# Consider common string representations of True/False
if user_input_str.lower() == 'true':
  user_input_bool = True

  print(f"Converted to boolean: {user_input_bool} (Type: {type(user_input_bool)})")
elif user_input_str.lower() == 'false':
  user_input_bool = False

  print(f"Converted to boolean: {user_input_bool} (Type: {type(user_input_bool)})")
else:

  user_input_bool = bool(user_input_str)
  print(f"Converted to boolean (using bool()): {user_input_bool} (Type: {type(user_input_bool)})")
```

```
Enter a value: 25
Converted to integer: 25 (Type: <class 'int'>)
Converted to float: 25.0 (Type: <class 'float'>)
Converted to boolean (using bool()): True (Type: <class 'bool'>)
```

10. Write code to demonstrate type casting with list elements.

Sol:-

```
my_list = ["1", "2", "3.5", "True", "False"]

# Type casting elements to integer (will raise ValueError for "3.5", "True", "False")
print("Converting to integers:")
try:
  int_list = [int(item) for item in my_list]
  print(int_list)
except ValueError as e:
  print(f"Error during integer conversion: {e}")

# Type casting elements to float
print("\nConverting to floats:")
try:
  float_list = [float(item) for item in my_list]
  print(float_list)
except ValueError as e:
  print(f"Error during float conversion: {e}")

# Type casting elements to boolean
```

```
print("\nConverting to booleans:")

# Using a simple logic for string to boolean conversion
bool_list = []
for item in my_list:
  if item.lower() == 'true':
    bool_list.append(True)
  elif item.lower() == 'false':
    bool_list.append(False)
  else:
    bool_list.append(bool(item))

bool_list
```

```
Converting to integers:
Error during integer conversion: invalid literal for int() with base 10: '3.5'

Converting to floats:
Error during float conversion: could not convert string to float: 'True'

Converting to booleans:
[True, True, True, True, False]
```

11. Write a program that checks if a number is positive, negative, or zero

Sol:-

```
number = int(input("Enter a number to check "))

if number > 0:
  print(f"{number} is a positive number.")
elif number < 0:
  print(f"{number} is a negative number.")
else:
  print(f"{number} is zero.")
```

```
Enter a number to check -2
-2 is a negative number.
```

12. Write a for loop to print numbers from 1 to 10

```
for i in range(1, 11):
  print(i)
```

```
1
2
3
4
5
6
7
8
9
10
```

13. Write a Python program to find the sum of all even numbers between 1 and 50

Sol:-

```
sum = 0

for i in range(1, 51):
  if i % 2 == 0:
    sum += i

print(f"The sum of all even numbers between 1 and 50 is {sum}.")
```

```
The sum of all even numbers between 1 and 50 is 650.
```

14. Write a program to reverse a string using a while loop

Sol:-

```
input_string = "hello world"
reversed_string = ""
index = len(input_string) - 1
```

```
while index >= 0:
  reversed_string += input_string[index]
  index -= 1

print(f"The reversed string is: {reversed_string}")
```

```
The reversed string is: dlrow olleh
```

15. Write a Python program to calculate the factorial of a number provided by the user using a while loop

Sol:-

```
num = int(input("Enter a non-negative integer: "))

factorial = 1

if num < 0:
  print("Factorial is not defined for negative numbers.")
elif num == 0:
  print("The factorial of 0 is 1")
else:
  i = 1
  while i <= num:
    factorial = factorial * i
    i = i + 1

  print("The factorial of", num, "is", factorial)
```

```
Enter a non-negative integer: 5
The factorial of 5 is 120
```