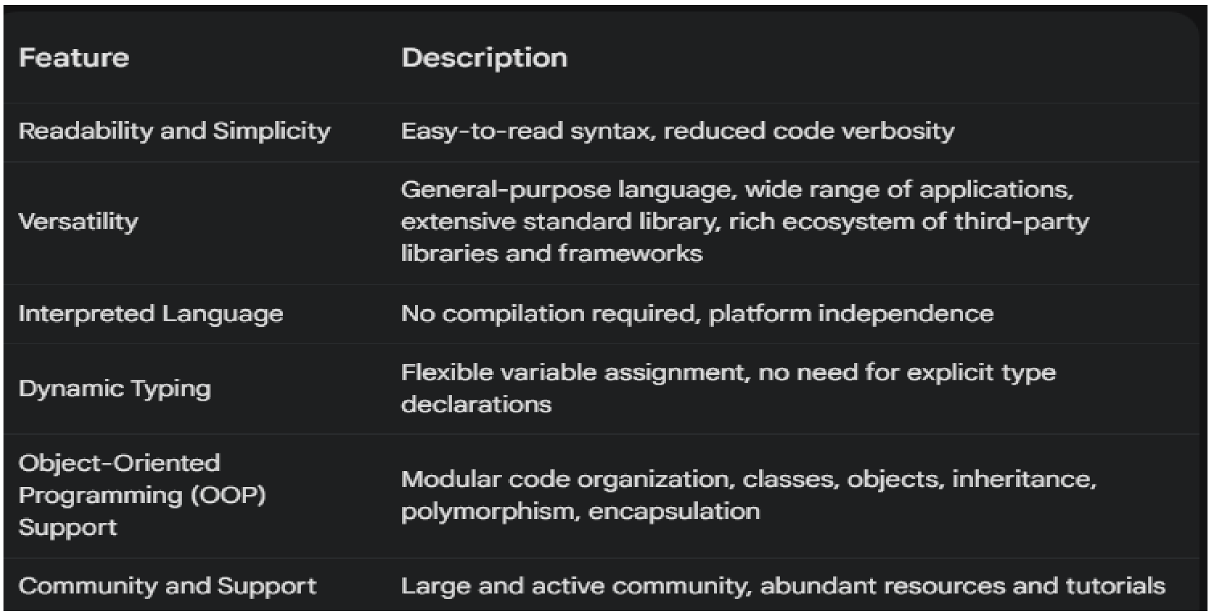
Assignment 1

Python

Ques1: Explain the key features of Python that make it a popular choice for programming

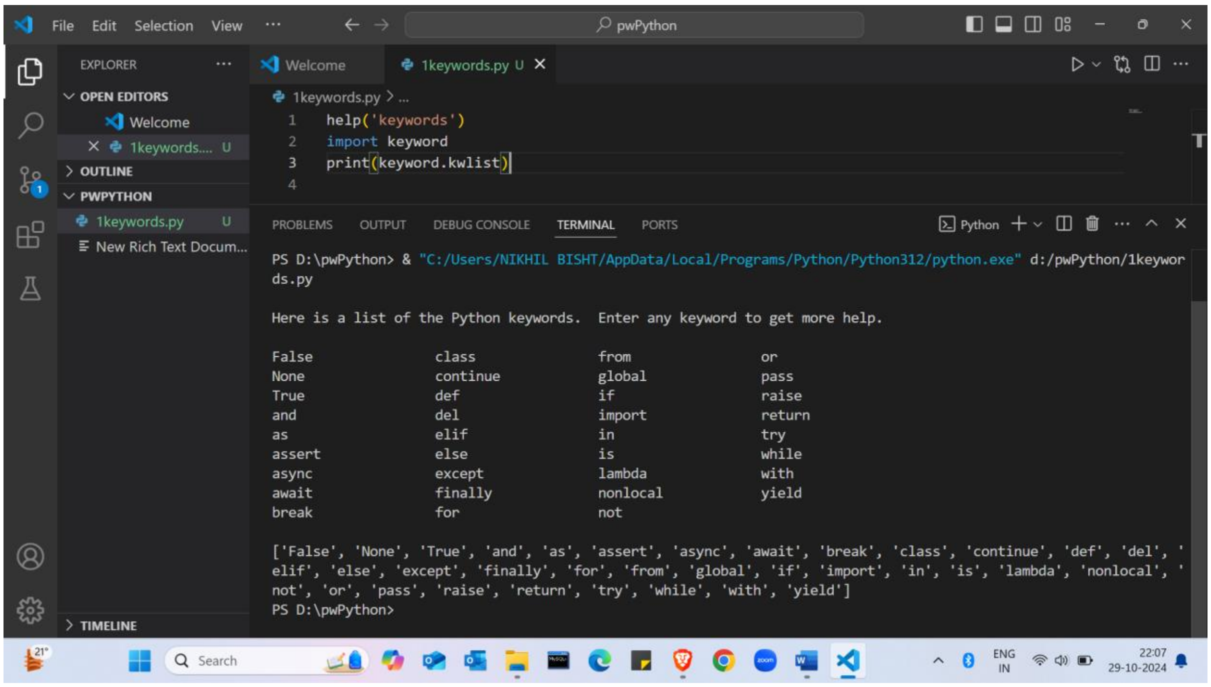
Ans1: Python's popularity stems from its readability, versatility, and extensive support. Its simple syntax, vast libraries, and cross-platform compatibility make it a powerful tool for various applications, from web development to data science and machine learning.



Feature	Description
Readability and Simplicity	Easy-to-read syntax, reduced code verbosity
Versatility	General-purpose language, wide range of applications, extensive standard library, rich ecosystem of third-party libraries and frameworks
Interpreted Language	No compilation required, platform independence
Dynamic Typing	Flexible variable assignment, no need for explicit type declarations
Object-Oriented Programming (OOP) Support	Modular code organization, classes, objects, inheritance, polymorphism, encapsulation
Community and Support	Large and active community, abundant resources and tutorials

Ques2: Describe the role of predefined keywords in Python and provide examples of how they are used in a program.

Ans2: Predefined keywords are reserved words in Python that have special meanings and are used to define the structure and control flow of a program. They cannot be used as variable names or identifiers.



```python
help('keywords')
import keyword
print(keyword.kwlist)
```

```
PS D:\pwPython> & "C:/Users/NIKHIL BISHT/AppData/Local/Programs/Python/Python312/python.exe" d:/pwPython/1keywor
ds.py

Here is a list of the Python keywords.  Enter any keyword to get more help.

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

['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']
PS D:\pwPython>
```

Ques3: Compare and contrast mutable and immutable objects in Python with examples.

Ans3: In Python, objects are classified as either mutable or immutable, depending on whether their values can be changed after they are created.

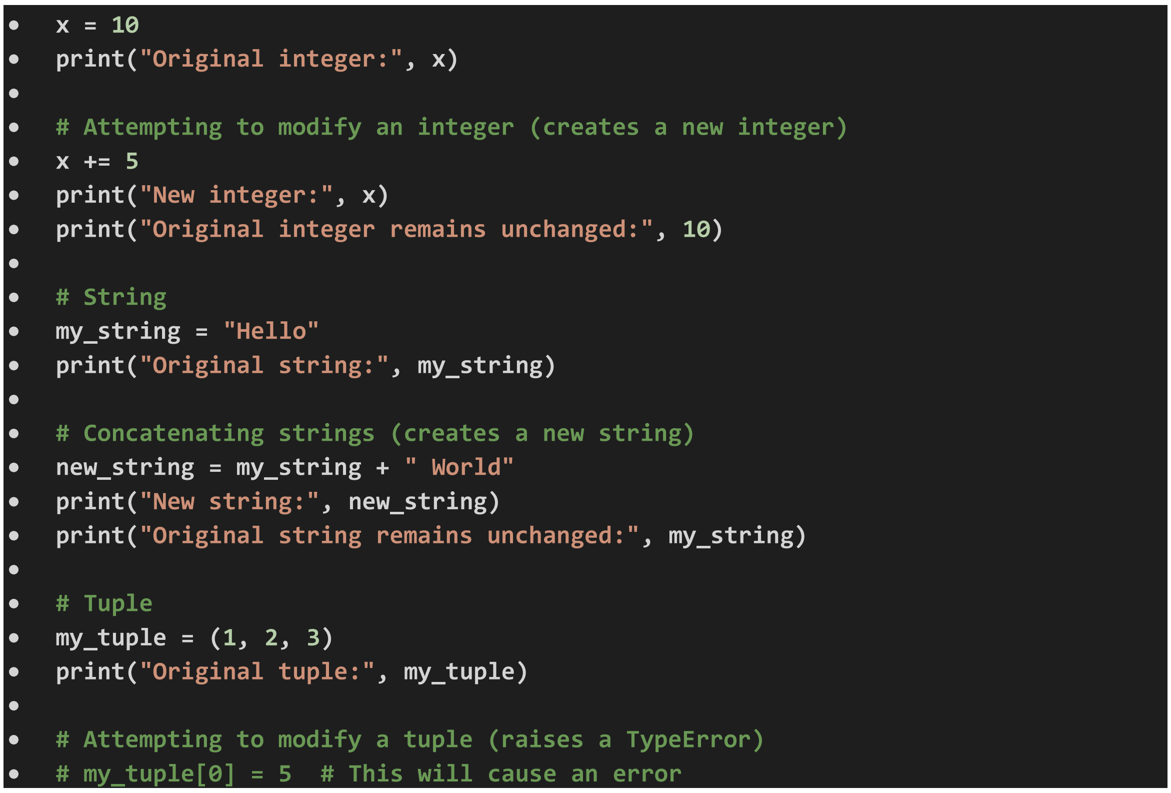
Mutable Objects:

- **Definition:** Mutable objects can be modified after they are created.
- **Examples:** Lists, dictionaries, sets
- **Behavior:**
    - Changes made to a mutable object affect the original object.
    - Appending, removing, or modifying elements directly changes the object.

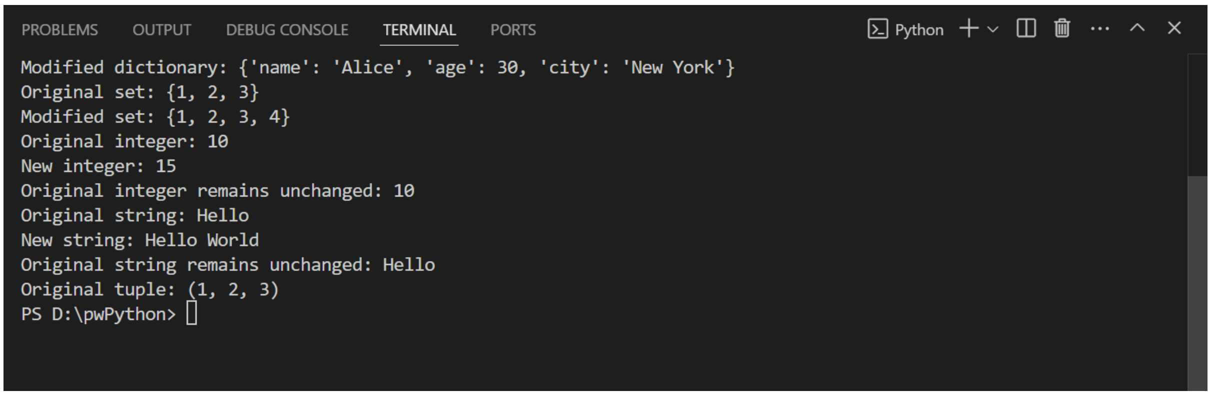
Immutable Objects:

- **Definition:** Immutable objects cannot be changed after they are created.
- **Examples:** Numbers (integers, floats), strings, tuples
- **Behavior:**
    - Any operation that appears to modify an immutable object actually creates a new object.
    - The original object remains unchanged.

```
# Mutable Objects

# List
my_list = [1, 2, 3]
print("Original list:", my_list)

# Modify the list
my_list.append(4)
print("Modified list:", my_list)

# Dictionary
my_dict = {"name": "Alice", "age": 30}
print("Original dictionary:", my_dict)

# Modify the dictionary
my_dict["city"] = "New York"
print("Modified dictionary:", my_dict)

# Set
my_set = {1, 2, 3}
print("Original set:", my_set)

# Add an element to the set
my_set.add(4)
print("Modified set:", my_set)

# Immutable Objects

# Integer
```

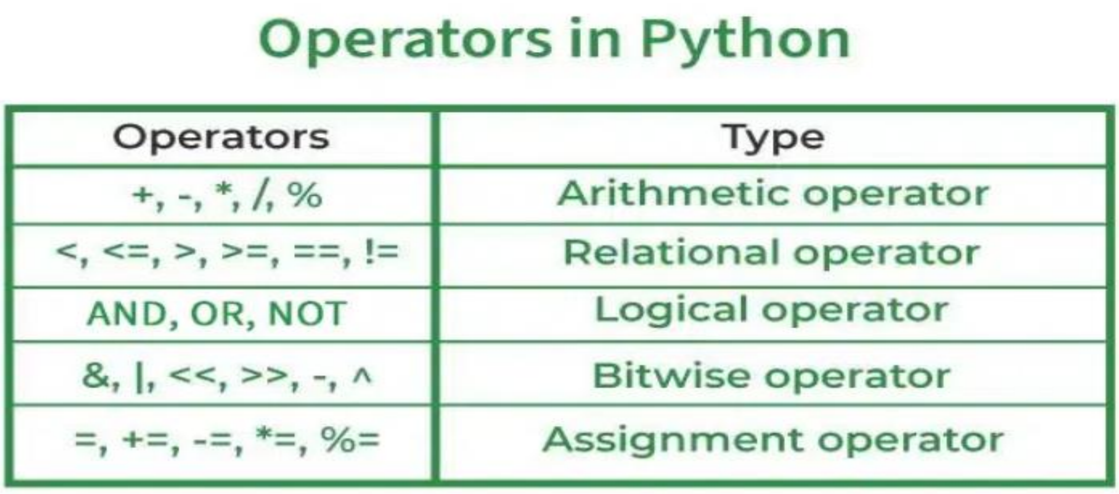


```python
x = 10
print("Original integer:", x)

# Attempting to modify an integer (creates a new integer)
x += 5
print("New integer:", x)
print("Original integer remains unchanged:", 10)

# String
my_string = "Hello"
print("Original string:", my_string)

# Concatenating strings (creates a new string)
new_string = my_string + " World"
print("New string:", new_string)
print("Original string remains unchanged:", my_string)

# Tuple
my_tuple = (1, 2, 3)
print("Original tuple:", my_tuple)

# Attempting to modify a tuple (raises a TypeError)
# my_tuple[0] = 5  # This will cause an error
```



```
Modified dictionary: {'name': 'Alice', 'age': 30, 'city': 'New York'}
Original set: {1, 2, 3}
Modified set: {1, 2, 3, 4}
Original integer: 10
New integer: 15
Original integer remains unchanged: 10
Original string: Hello
New string: Hello World
Original string remains unchanged: Hello
Original tuple: (1, 2, 3)
PS D:\pwPython> 
```

Ques4: Discuss the different types of operators in Python and provide examples of how they are used.

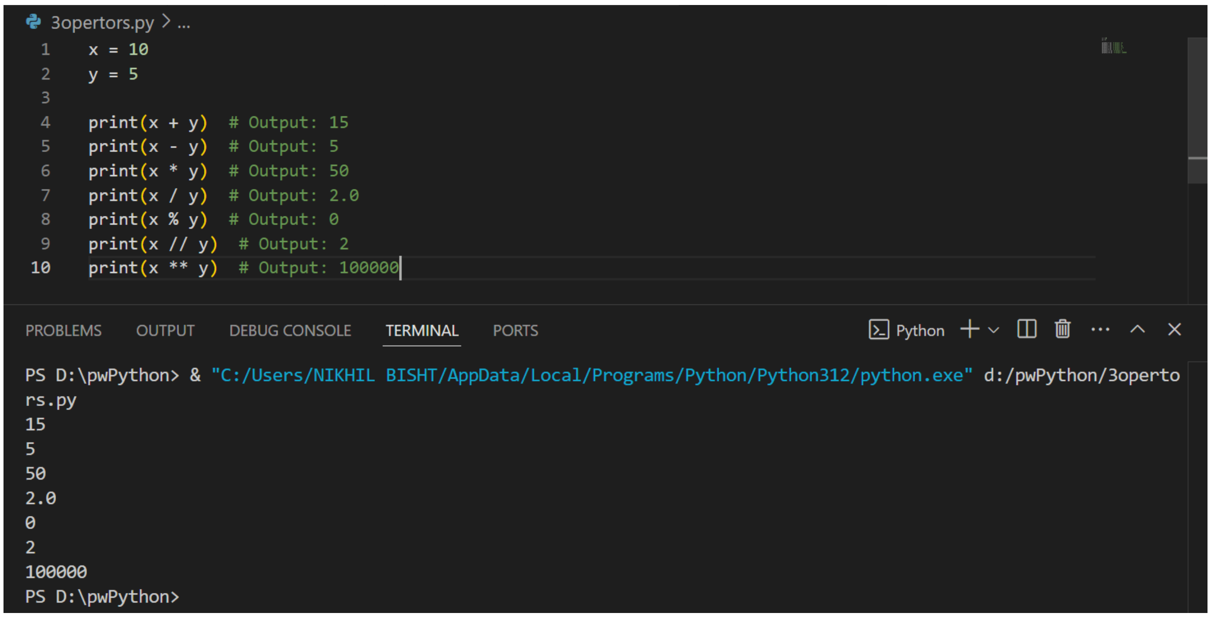


Operators in Python

Operators	Type
+, -, *, /, %	Arithmetic operator
<, <=, >, >=, ==, !=	Relational operator
AND, OR, NOT	Logical operator
&, |, <<, >>, -, ^	Bitwise operator
=, +=, -=, *=, %=	Assignment operator

1. Arithmetic Operators

- Used for basic mathematical operations:
  - +: Addition
  - -: Subtraction
  - *: Multiplication
  - /: Division
  - %: Modulus (remainder)
  - //: Floor division (integer division)
  - **: Exponentiation

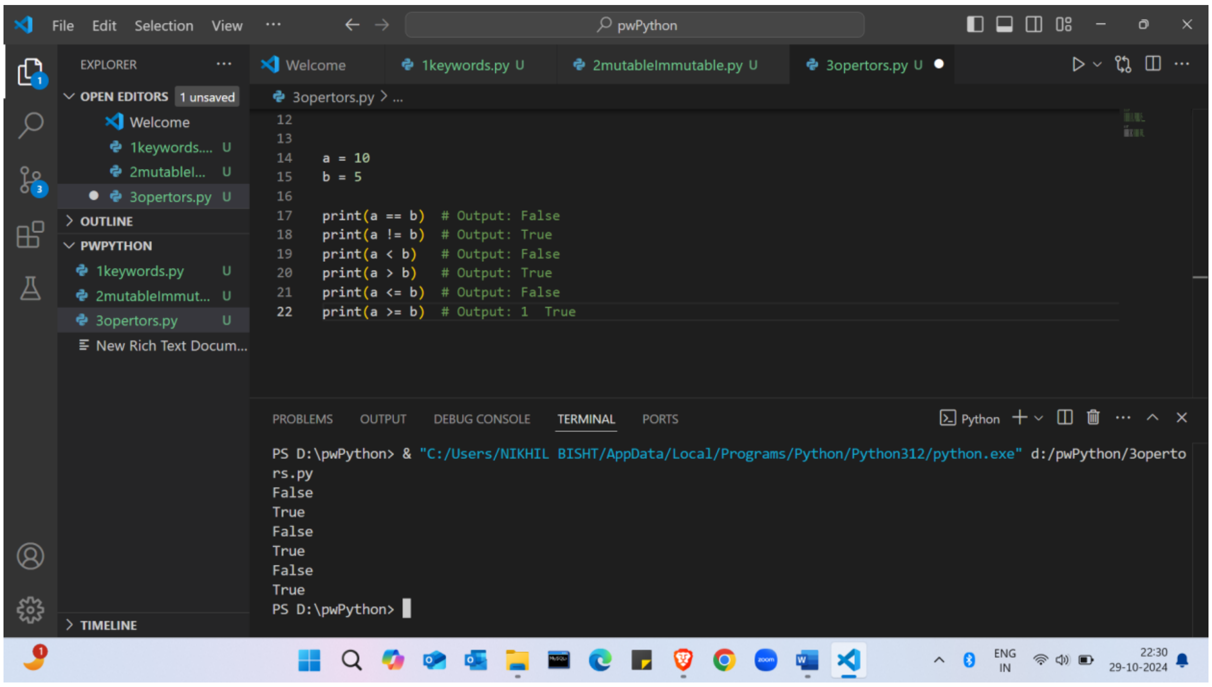


```python
x = 10
y = 5

print(x + y)  # Output: 15
print(x - y)  # Output: 5
print(x * y)  # Output: 50
print(x / y)  # Output: 2.0
print(x % y)  # Output: 0
print(x // y)  # Output: 2
print(x ** y)  # Output: 100000
```

```
PS D:\pwPython> & "C:/Users/NIKHIL BISHT/AppData/Local/Programs/Python/Python312/python.exe" d:/pwPython/3operto
rs.py
15
5
50
2.0
0
2
100000
PS D:\pwPython>
```

2. Comparison Operators

- Used to compare values:
  - ==: Equal to
  - !=: Not equal to
  - <: Less than
  - >: Greater than
  - <=: Less than or equal to
  - >=: Greater than or equal to

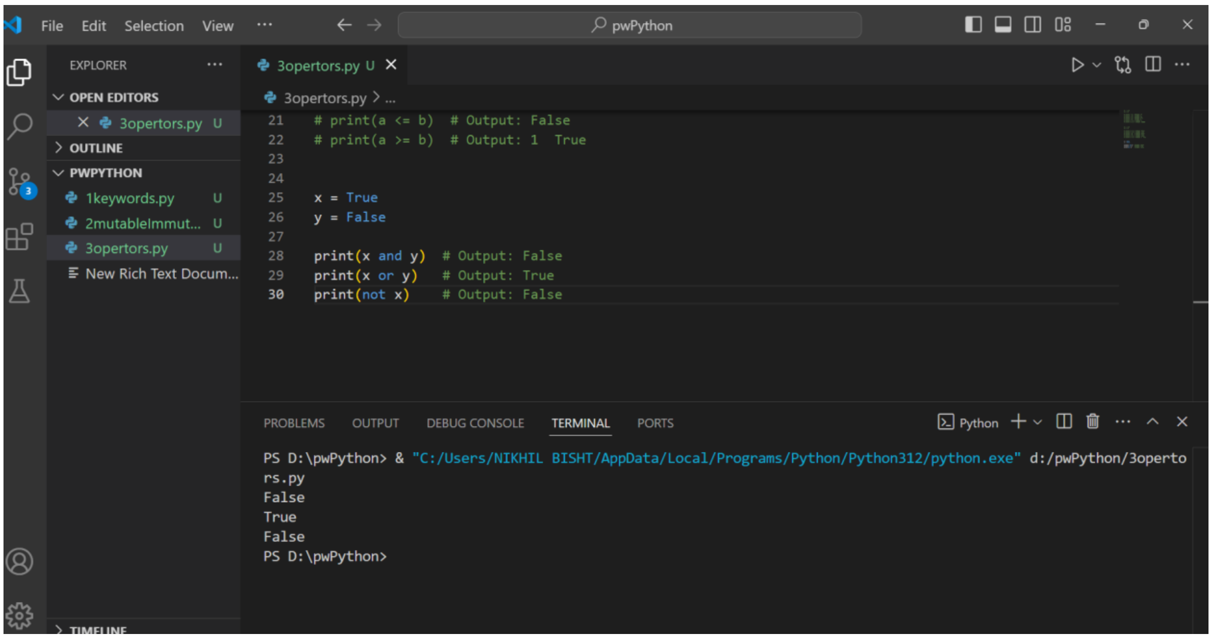


```
a = 10
b = 5

print(a == b)  # Output: False
print(a != b)  # Output: True
print(a < b)   # Output: False
print(a > b)   # Output: True
print(a <= b)  # Output: False
print(a >= b)  # Output: 1  True
```

```
PS D:\pwPython> & "C:/Users/NIKHIL BISHT/AppData/Local/Programs/Python/Python312/python.exe" d:/pwPython/3operto
rs.py
False
True
False
True
False
True
PS D:\pwPython> 
```

3. Logical Operators

- Used to combine conditional statements:
  - and: Logical AND
  - or: Logical OR
  - not: Logical NOT

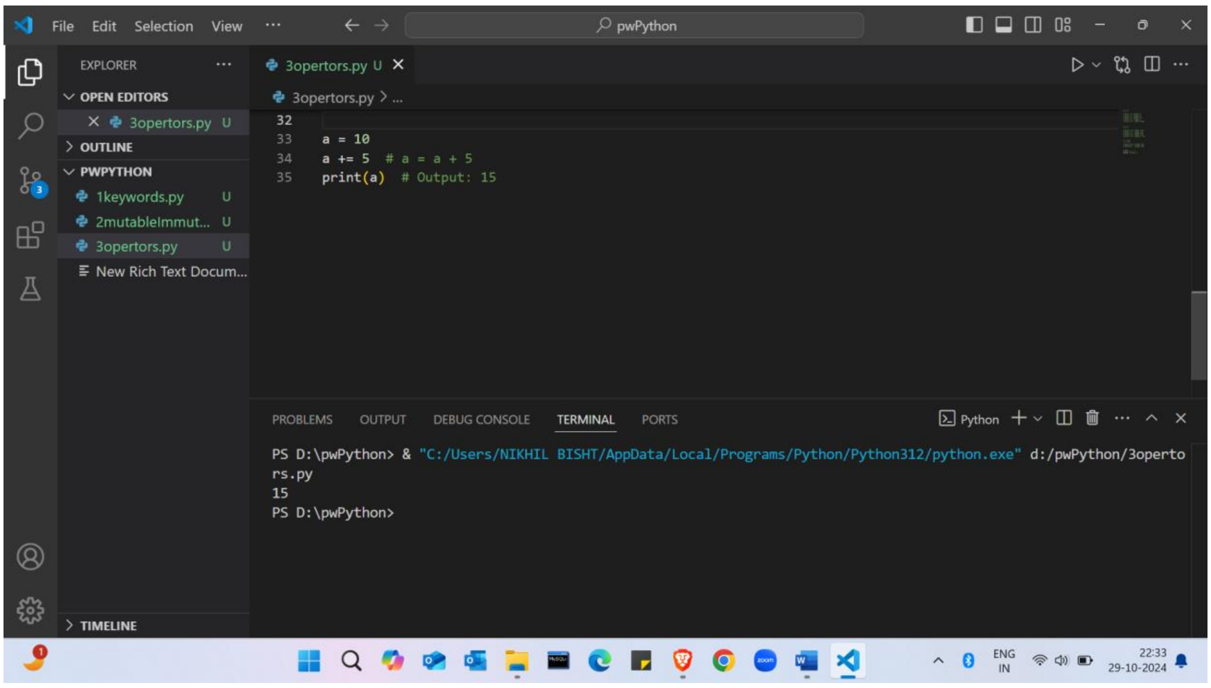


```
# print(a <= b)  # Output: False
# print(a >= b)  # Output: 1  True


x = True
y = False

print(x and y)  # Output: False
print(x or y)   # Output: True
print(not x)    # Output: False
```

```
PS D:\pwPython> & "C:/Users/NIKHIL BISHT/AppData/Local/Programs/Python/Python312/python.exe" d:/pwPython/3operto
rs.py
False
True
False
PS D:\pwPython>
```

4. Assignment Operators

- Used to assign values to variables:
    - =: Simple assignment
    - +=: Add and assign
    - -=: Subtract and assign
    - *=: Multiply and assign
    - /=: Divide and assign
    - %=: Modulus and assign
    - //=: Floor division and assign
    - **=: Exponentiation and assign

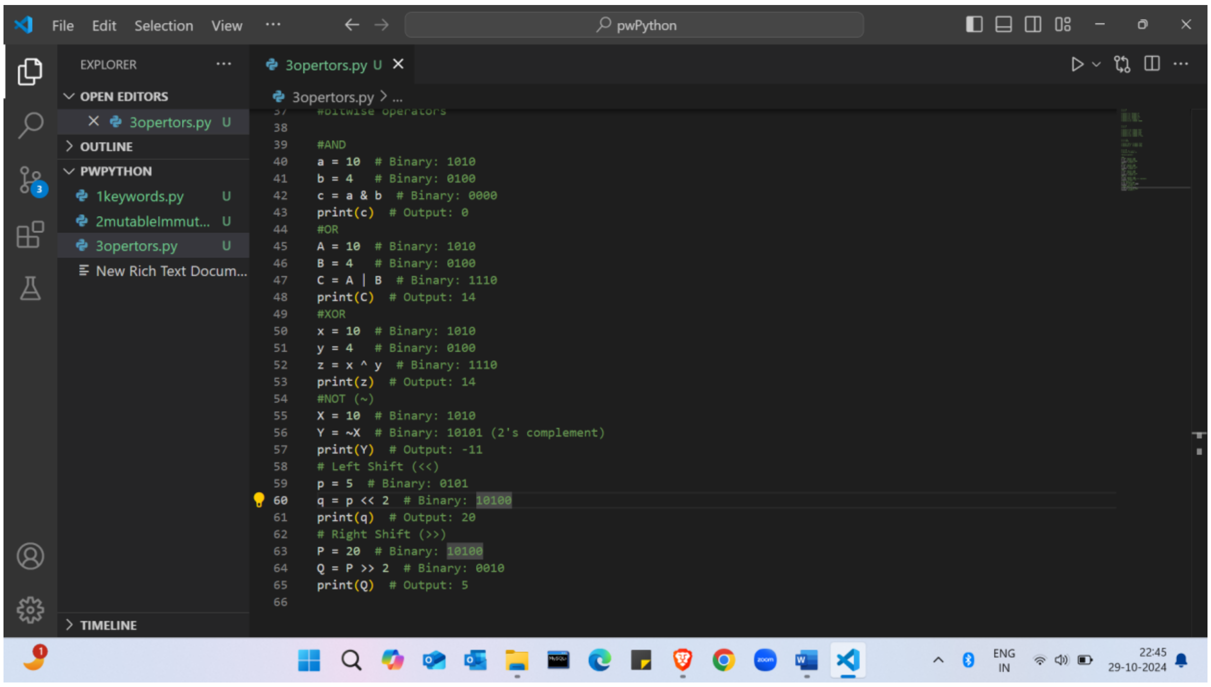


```python

a = 10
a += 5  # a = a + 5
print(a)  # Output: 15
```

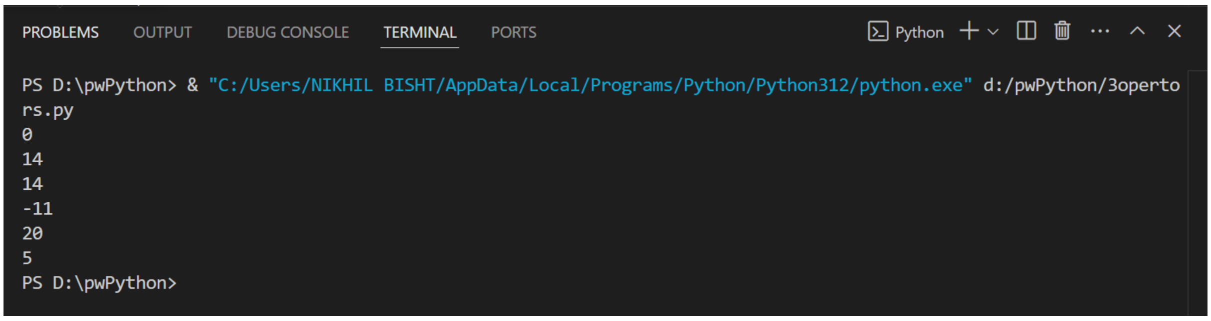
```
PS D:\pwPython> & "C:/Users/NIKHIL BISHT/AppData/Local/Programs/Python/Python312/python.exe" d:/pwPython/3operto
rs.py
15
PS D:\pwPython>
```

5. Bitwise Operators

- Used to perform bitwise operations on integers:
    - &: Bitwise AND
    - |: Bitwise OR
    - ^: Bitwise XOR
    - ~: Bitwise NOT
    - <<: Left shift
    - >>: Right shift



```python
#bitwise operators

#AND
a = 10  # Binary: 1010
b = 4   # Binary: 0100
c = a & b  # Binary: 0000
print(c)  # Output: 0
#OR
A = 10  # Binary: 1010
B = 4   # Binary: 0100
C = A | B  # Binary: 1110
print(C)  # Output: 14
#XOR
x = 10  # Binary: 1010
y = 4   # Binary: 0100
z = x ^ y  # Binary: 1110
print(z)  # Output: 14
#NOT (~)
X = 10  # Binary: 1010
Y = ~X  # Binary: 10101 (2's complement)
print(Y)  # Output: -11
# Left Shift (<<)
p = 5  # Binary: 0101
q = p << 2  # Binary: 10100
print(q)  # Output: 20
# Right Shift (>>)
P = 20  # Binary: 10100
Q = P >> 2  # Binary: 0010
print(Q)  # Output: 5

```



```
PS D:\pwPython> & "C:/Users/NIKHIL BISHT/AppData/Local/Programs/Python/Python312/python.exe" d:/pwPython/3operto
rs.py
0
14
14
-11
20
5
PS D:\pwPython>
```

6. Identity Operators

- Used to compare objects' identities:
    - is: Checks if two objects are the same object
    - is not: Checks if two objects are not the same object

7. Membership Operators

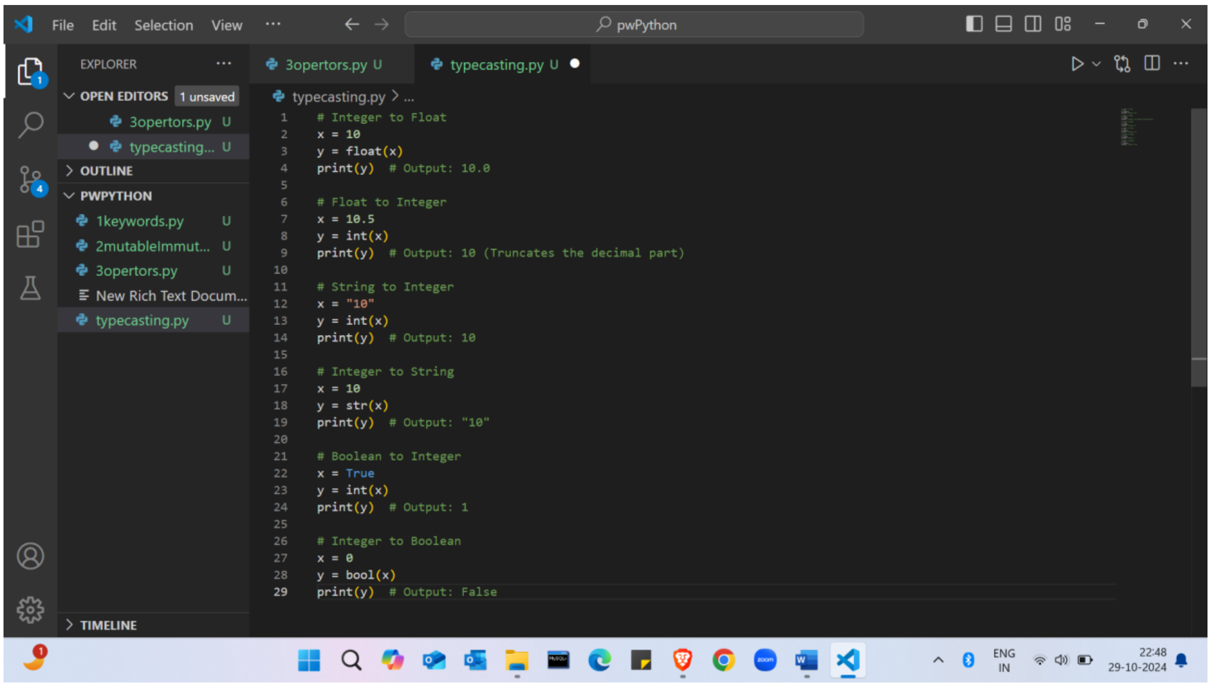
- Used to test membership in sequences:
    - in: Checks if a value is present in a sequence
    - not in: Checks if a value is not present in a sequence

Ques5: Explain the concept of type casting in Python with examples.

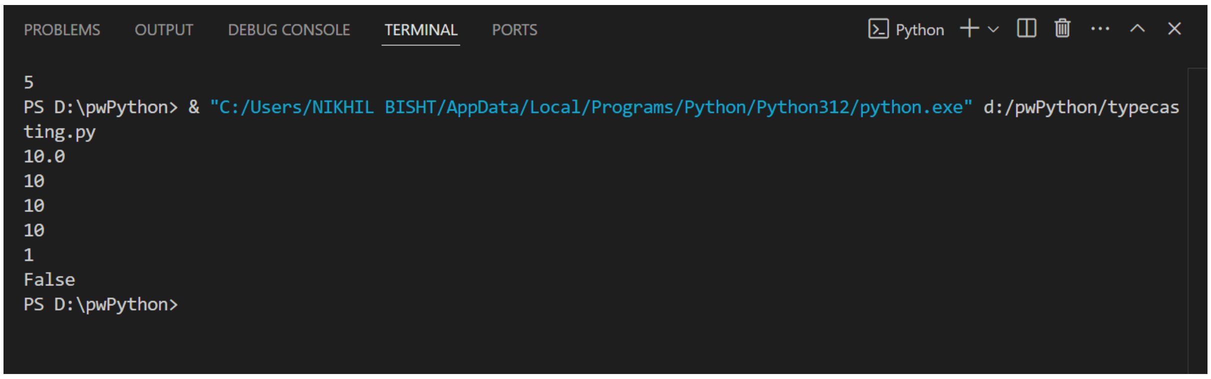
Ans5: Type casting, also known as type conversion, is the process of converting one data type to another. Python provides several built-in functions to perform type casting:

1. **int(): Converts a value to an integer.**
2. **float(): Converts a value to a floating-point number.**

3. **str(): Converts a value to a string.**
4. **bool(): Converts a value to a Boolean (True or False)**



```python
# Integer to Float
x = 10
y = float(x)
print(y)  # Output: 10.0

# Float to Integer
x = 10.5
y = int(x)
print(y)  # Output: 10 (Truncates the decimal part)

# String to Integer
x = "10"
y = int(x)
print(y)  # Output: 10

# Integer to String
x = 10
y = str(x)
print(y)  # Output: "10"

# Boolean to Integer
x = True
y = int(x)
print(y)  # Output: 1

# Integer to Boolean
x = 0
y = bool(x)
print(y)  # Output: False
```



```
5
PS D:\pwPython> & "C:/Users/NIKHIL BISHT/AppData/Local/Programs/Python/Python312/python.exe" d:/pwPython/typecas
ting.py
10.0
10
10
10
1
False
PS D:\pwPython>
```

Implicit Type Conversion: Python often performs implicit type conversion when it's safe to do so. For example, when you add an integer and a float, Python automatically converts the integer to a float before performing the addition

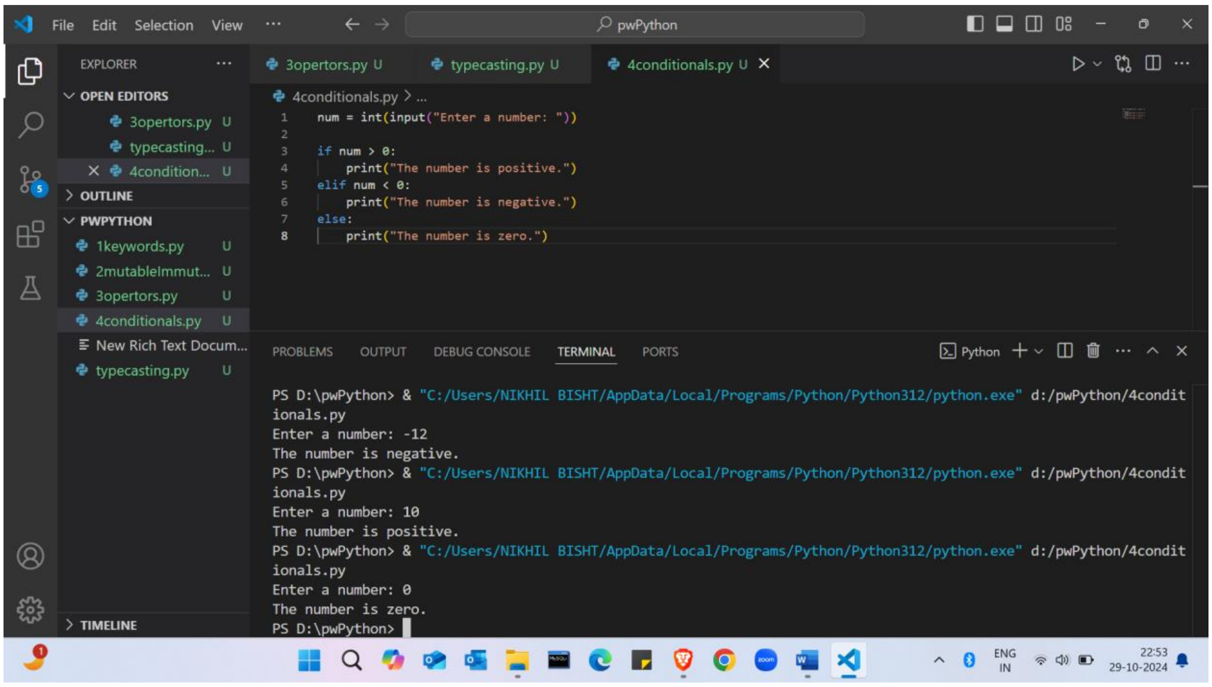
Type Errors: If you attempt to convert a value to an incompatible type, Python will raise a TypeError. For instance, converting the string "hello" to an integer would result in a ValueError

Data Loss: Be cautious when converting between data types, as you may lose information. For example, converting a float to an integer truncates the decimal part.

Ques6: How do conditional statements work in Python? Illustrate with examples.

Ans6: Conditional statements allow you to control the flow of your program based on specific conditions. Python uses the if, elif, and else keywords to create conditional blocks.

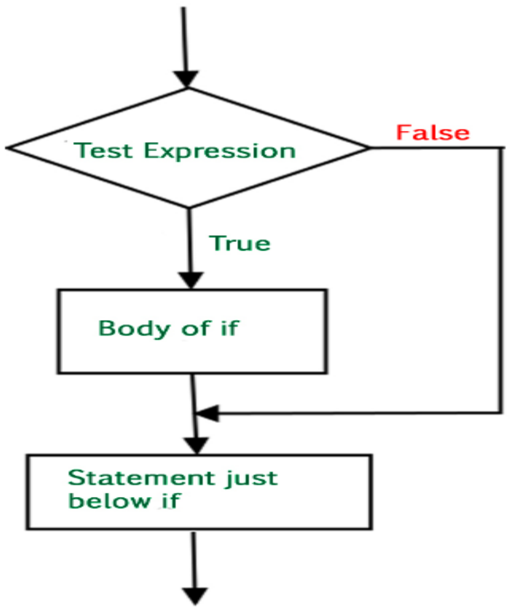
```
if condition:
    # Code to be executed if the condition is True
elif condition2:
    # Code to be executed if the first condition is False and the second
else:
    # Code to be executed if all previous conditions are False
```



```
num = int(input("Enter a number: "))

if num > 0:
    print("The number is positive.")
elif num < 0:
    print("The number is negative.")
else:
    print("The number is zero.")
```

```
PS D:\pwPython> & "C:/Users/NIKHIL BISHT/AppData/Local/Programs/Python/Python312/python.exe" d:/pwPython/4condit
ionals.py
Enter a number: -12
The number is negative.
PS D:\pwPython> & "C:/Users/NIKHIL BISHT/AppData/Local/Programs/Python/Python312/python.exe" d:/pwPython/4condit
ionals.py
Enter a number: 10
The number is positive.
PS D:\pwPython> & "C:/Users/NIKHIL BISHT/AppData/Local/Programs/Python/Python312/python.exe" d:/pwPython/4condit
ionals.py
Enter a number: 0
The number is zero.
PS D:\pwPython> 
```

flow chart of conditional statements



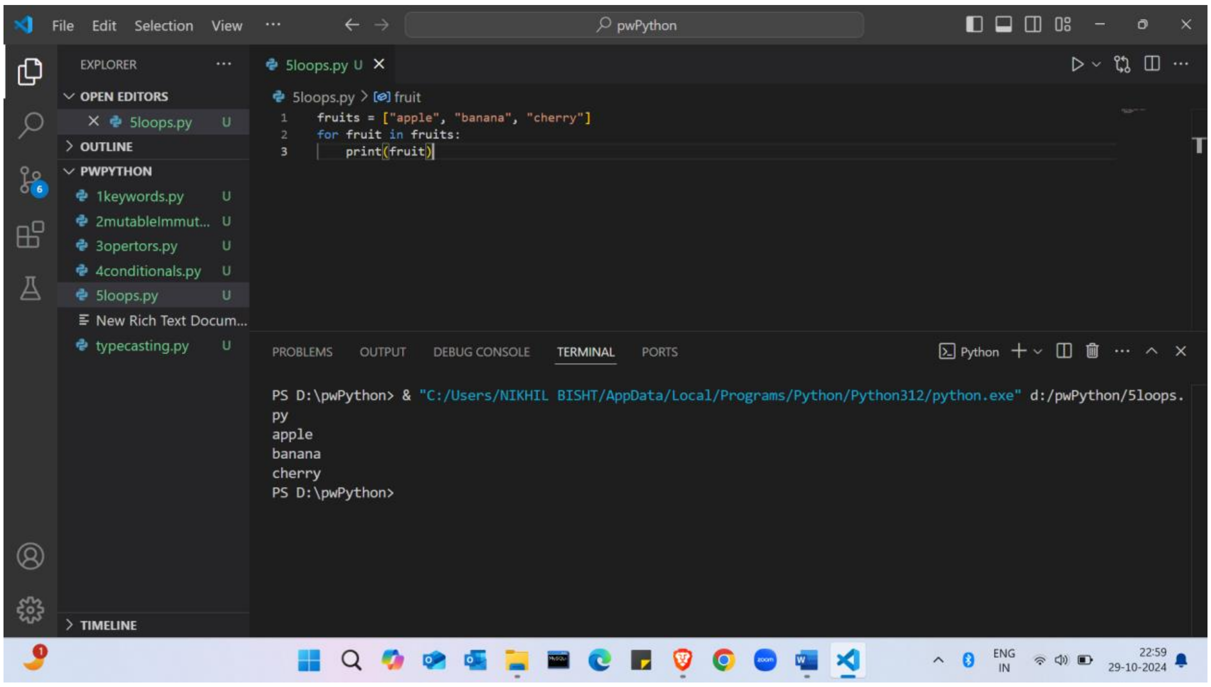
Ques7: Describe the different types of loops in Python and their use cases with examples.

Ans7: Python primarily employs two types of loops: for and while.

1. For Loop:

- **Use Case:** Iterating over a sequence (like a list, tuple, string, or range).
- **Syntax:**

```
for item in sequence:
    # code to be executed
```



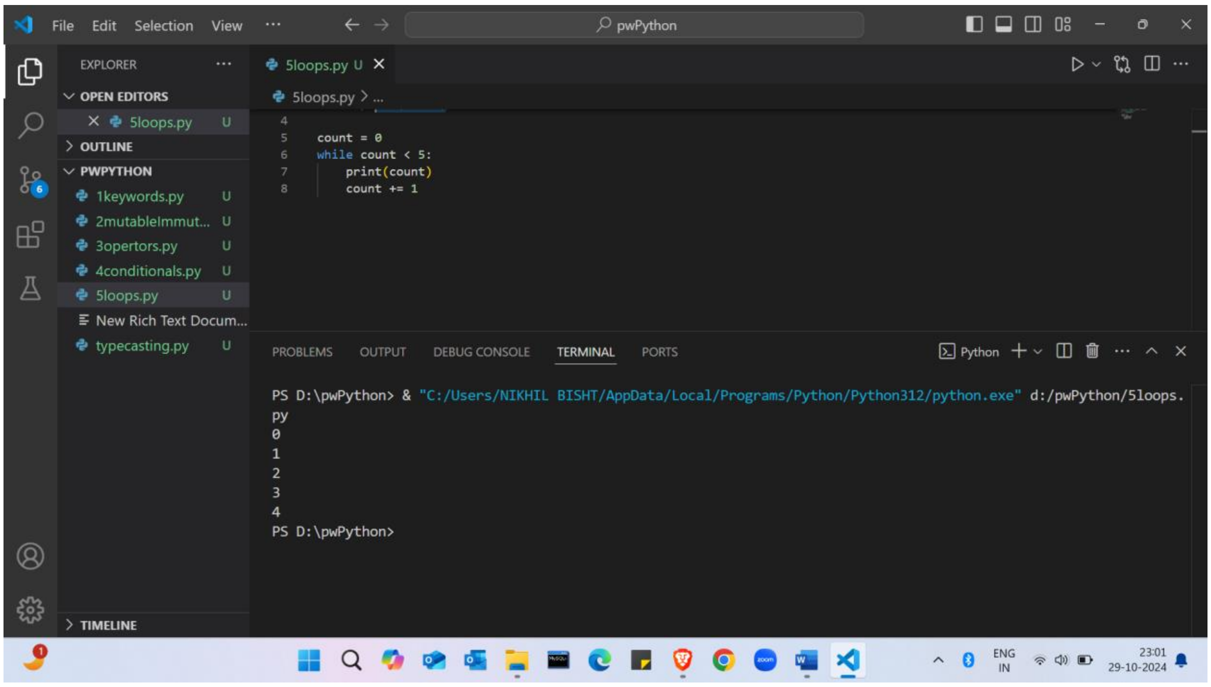
```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

```
PS D:\pwPython> & "C:/Users/NIKHIL BISHT/AppData/Local/Programs/Python/Python312/python.exe" d:/pwPython/5loops.
py
apple
banana
cherry
PS D:\pwPython>
```

2. While Loop:

- **Use Case:** Repeating a block of code as long as a certain condition is true.
- **Syntax:**

```
while condition:
    # code to be executed
```

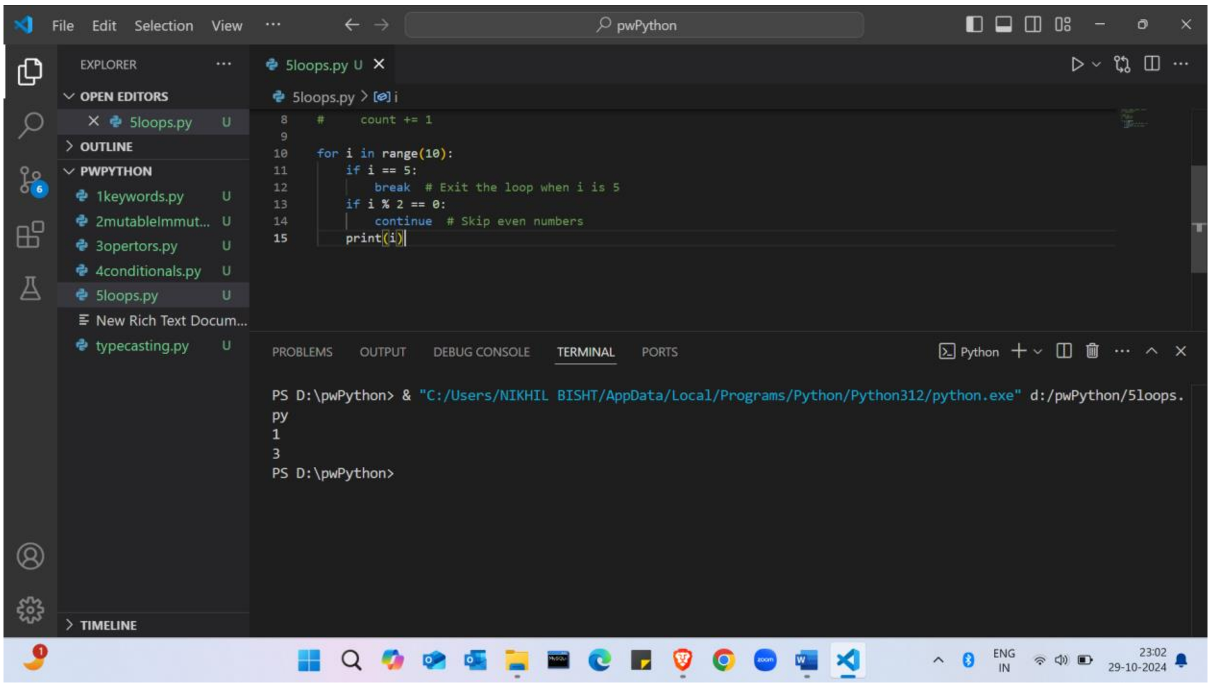


```python
count = 0
while count < 5:
    print(count)
    count += 1
```

```
PS D:\pwPython> & "C:/Users/NIKHIL BISHT/AppData/Local/Programs/Python/Python312/python.exe" d:/pwPython/5loops.
py
0
1
2
3
4
PS D:\pwPython>
```

dditional Considerations:

- **break statement:** Exits the loop immediately.
- **continue statement:** Skips the current iteration and moves to the next one.



```python
#    count += 1

for i in range(10):
    if i == 5:
        break  # Exit the loop when i is 5
    if i % 2 == 0:
        continue  # Skip even numbers
    print(i)
```

```
PS D:\pwPython> & "C:/Users/NIKHIL BISHT/AppData/Local/Programs/Python/Python312/python.exe" d:/pwPython/5loops.
py
1
3
PS D:\pwPython>
```

By effectively using these loops, you can automate repetitive tasks and control the flow of your Python programs.