1. Explain the difference between '/' and '//' operators in Python with respect to their output types and behavior with negative numbers.

Ans:

`/` (True Division)

- Always returns a **float**
- Performs exact mathematical division

Example

```
5 / 2      # 2.5
-5 / 2     # -2.5
```

`//` (Floor Division)

- Returns **floor value** of division
- Output type: `int` (if operands are int)
- Rounds down toward negative infinity

Example

```
5 // 2      # 2
-5 // 2     # -3   (important difference)
```

Key Difference with Negative Numbers

- `/` keeps exact value
- `//` moves toward $-\infty$, not toward zero

1. What is the difference between bitwise and logical operators? When would you use each?

Ans:

Logical Operators

- Work on **boolean values**
- Used in **conditions**

Operator	Meaning
`and`	True if both true
`or`	True if any true
`not`	Negates value

```
x = 5
y = 3
x > 2 and y < 4   # True
```

Bitwise Operators

- Work on **binary representation of integers**
- Used in **low-level operations**

Operator	Meaning
&	AND
|	OR
^	XOR
~	NOT
<<	Left shift
>>	Right shift

```
5 & 3   # 1
```

When to Use

- **Logical** → conditions, decision making
- **Bitwise** → flags, masking, hardware, performance-critical code

1. Determine the output of these expressions and explain the order of operations:

   x = 5

   y = 3

   z = 2

   - Expression 1

     result1 = x + y * z ** 2 - (x + y) / z

   - Expression 2

     result2 = x > y and y > z or not (x + y == 8)

   - Expression 3

     result3 = x & y | ~z + 1

   - Expression 4

     result4 = x * y // z + 3 ** 2 %

Ans:

Given

```
x = 5
y = 3
z = 2
```

Expression 1

```
result1 = x + y * z ** 2 - (x + y) / z
```

Order

1. `z ** 2 = 4`
2. `y * 4 = 12`
3. `(x + y) = 8`
4. `8 / 2 = 4.0`
5. `5 + 12 - 4.0 = 13.0`

Output

```
13.0
```

Expression 2

```
result2 = x > y and y > z or not (x + y == 8)
```

Evaluation

1. `x > y → True`
2. `y > z → True`
3. `True and True → True`
4. `(x + y == 8) → True`
5. `not True → False`
6. `True or False → True`

Output

```
True
```

Expression 3

```
result3 = x & y | ~z + 1
```

Steps

- `~z = ~2 = -3`
- `-3 + 1 = -2`
- `x & y = 5 & 3 = 1`
- `1 | -2 = -1`

Output

```
-1
```

Expression 4

```
result4 = x * y // z + 3 ** 2 %
```

 Invalid Expression

Reason

`%` operator requires **two operands**

```
x = 5
y = 3
z = 2

# Expression 1

result1 = x + y * z ** 2 - (x + y) / z
print("1.", result1)

# Expression 2

result2 = x > y and y > z or not (x + y == 8)
print("2.", result2)

# Expression 3

result3 = x & y | ~z + 1
print("3.", result3)
```

```
1. 13.0
2. True
3. -1
```

```
# Expression 4

result4 = x * y // z + 3 ** 2 %
print("4.", result4)
```

```
  Cell In[6], line 3
    result4 = x * y // z + 3 ** 2 %
                                  ^
SyntaxError: invalid syntax
```

1. Write a program that takes two numbers as input and swaps their values without using a temporary variable. Use both the traditional method and Python's multiple assignment method. Compare the two approaches.

Ans: Traditional (Arithmetic) Method

```
a = 10
b = 5

a = a + b
b = a - b
a = a - b
```

Pythonic Method (Multiple Assignment)

```
a, b = b, a
```

Method	Safe	Readable
Arithmetic	⚠ (overflow risk)	✅
Pythonic	✅	✅

1. Given the string "Python Programming", write code to:
   - Print every alternate character
   - Print the string in reverse without using the reverse function
   - Count the number of vowels and consonants

```
s = "Python Programming"

print(s[::2])
print(s[::-1])

vowels = ['a', 'e', 'i', 'o', 'u']
num_of_vowels = 0
num_of_consonants = 0
for i in s:
  if i.lower() in vowels:
    num_of_vowels += 1
  else:
    num_of_consonants += 1

print(f"""
Number of Vowels: {num_of_vowels}
Number of Consonants: {num_of_consonants}
""")
```

```
Pto rgamn
gnimmargorP nohtyP

Number of Vowels: 4
Number of Consonants: 14
```

1. Create a program that generates a multiplication table for numbers 1 to 10, but skip multiples of 3 using the continue statement. Format the output in a neat table structure.

```
for i in range(1, 11):
  for j in range(1, 11):
    if (i * j) % 3 == 0:
      continue
    else:
      print(f"{i} x {j} = {i*j}")
  print()
```

```
1 x 1 = 1
1 x 2 = 2
1 x 4 = 4
1 x 5 = 5
```

```
1 x 7 = 7
1 x 8 = 8
1 x 10 = 10

2 x 1 = 2
2 x 2 = 4
2 x 4 = 8
2 x 5 = 10
2 x 7 = 14
2 x 8 = 16
2 x 10 = 20


4 x 1 = 4
4 x 2 = 8
4 x 4 = 16
4 x 5 = 20
4 x 7 = 28
4 x 8 = 32
4 x 10 = 40

5 x 1 = 5
5 x 2 = 10
5 x 4 = 20
5 x 5 = 25
5 x 7 = 35
5 x 8 = 40
5 x 10 = 50


7 x 1 = 7
7 x 2 = 14
7 x 4 = 28
7 x 5 = 35
7 x 7 = 49
7 x 8 = 56
7 x 10 = 70

8 x 1 = 8
8 x 2 = 16
8 x 4 = 32
8 x 5 = 40
8 x 7 = 56
8 x 8 = 64
8 x 10 = 80


10 x 1 = 10
10 x 2 = 20
10 x 4 = 40
10 x 5 = 50
10 x 7 = 70
10 x 8 = 80
```

```
10 x 10 = 100
```

1. Explain **, //, is and in operators of python.

Ans: | Operator | Meaning | Example | | -------- | ---------- | ---------- | | `**` | Power | `2 ** 3 = 8` | | `//` | Floor division | `5 // 2 = 2` | | `is` | Identity check | `a is b` | | `in` | Membership | `'a' in 'cat'` |

1. Explain any five string methods of python with an example.
    a. `upper()`

    Purpose: Converts all characters in the string to uppercase.

    b. `lower()`

    Purpose: Converts all characters in the string to lowercase.

    c. `strip()`

    Purpose: Removes leading and trailing whitespace from the string.

    d. `replace(old, new)`

    Purpose: Replaces all occurrences of a substring with another substring.

    e. `split()`

    Purpose: Splits a string into a list of substrings based on a delimiter.

```
s = "python"
print("1.", s.upper())

s = "PROGRAMMING"
print("2.", s.lower())

s = "  hello world  "
print("3.", s.strip())

s = "I like Java"
print("4.", s.replace("Java", "Python"))

s = "Python is easy"
print("5.", s.split())
```

```
1. PYTHON
2. programming
3. hello world
4. I like Python
5. ['Python', 'is', 'easy']
```

1. Write down similarities and differences between lists and tuples with an example.

Ans:

Similarities between Lists and Tuples

1. Ordered collection
   - Elements are stored in a specific order.
2. Indexed
   - Elements can be accessed using index values (starting from 0).
3. Allow duplicate elements
   - Same value can appear multiple times.
4. Can store heterogeneous data
   - Different data types can be stored together.
5. Iterable
   - Can be traversed using loops.

Example:

```
lst = [10, "Python", 3.5, 10]
tup = (10, "Python", 3.5, 10)
```

Differences between Lists and Tuples

Feature	List	Tuple
Mutability	Mutable (can be changed)	Immutable (cannot be changed)
Syntax	Uses `[ ]`	Uses `( )`
Speed	Slower	Faster
Memory Usage	More memory	Less memory
Built-in Methods	Many (`append`, `remove`)	Few (`count`, `index`)
Use Case	Dynamic data	Fixed data

Example Demonstrating Difference

List (Mutable)

```
lst = [1, 2, 3]
lst[0] = 10
print(lst)
```

```
[10, 2, 3]
```

Tuple (Immutable)

```
tup = (1, 2, 3)
tup[0] = 10
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent
call last)
/tmp/ipython-input-1722513740.py in <cell line: 0>()
```

```
      1 tup = (1, 2, 3)
----> 2 tup[0] = 10

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

1. What is the usage of tuples in python? List out operations of tuples. Explain at least two operations of tuples.

Ans:

Usage of Tuples in Python

A tuple is an ordered, immutable collection of elements in Python.

Why tuples are used

1. Data safety – values cannot be modified accidentally
2. Faster than lists – better performance for fixed data
3. Used as dictionary keys – because they are immutable
4. Return multiple values from functions
5. Represent fixed records (e.g., coordinates, RGB values)

Example:

```
point = (10, 20)   # x, y coordinate
```

Operations of Tuples in Python

Common tuple operations include:

1. Indexing
2. Slicing
3. Concatenation
4. Repetition
5. Membership (in, not in)
6. Iteration
7. Length (len())
8. Count (count())
9. Index (index())

Explanation of Two Tuple Operations

1. **Indexing**

   Used to access a specific element of a tuple using its index.

```
t = (10, 20, 30, 40)
print(t[1])

20
```

1. **Slicing**

Used to extract a portion of a tuple.

```
t = (10, 20, 30, 40, 50)
print(t[1:4])

(20, 30, 40)
```

Additional Examples of Tuple Operations

Concatenation

Membership Test

```
# Concatenation
t1 = (1, 2)
t2 = (3, 4)
print("Concatenation:", t1 + t2)

# Membership Test
t = (5, 10, 15)
print("Membership Test:", 10 in t)

Concatenation: (1, 2, 3, 4)
Membership Test: True
```

1. Write a program to take a string and a character as input and replace all occurrences of the character with a '*' in the string.

```
user_string = input("Enter String: ")
ch = input("Enter Character to Replace with '*': ")

user_string = user_string.replace(ch, '*')

print(user_string)

Enter String: Hii my name is Vatsal
Enter Character to Replace with '*': i
H** my name *s Vatsal
```

1. Write a program to find all prime numbers between two given numbers using functions and list comprehension.

```
def isPrime(n):
  if n < 2:
    return False
  for i in range(2, int(n**0.5) + 1):
    if n % i == 0:
      return False
  return True

def prime_between_two_numbers(start, end):
  if start > end or end < 2:
    return []
  prime = []
  for i in range(start, end + 1):
```

```
    if isPrime(i):
      prime.append(i)

  return prime

print(prime_between_two_numbers(2, 11))
```

```
[2, 3, 5, 7, 11]
```

1. Using regular expression, check entered mobile number is valid or not.

```
import re

mobile = input("Enter mobile number: ")

pattern = r'^[6-9]\d{9}$'

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

```
Enter mobile number: 9629262972
Valid
```

1. Explain concept of lambda function with usage of filter(), map(), and reduce() function.

Ans:

Lambda Function in Python

A lambda function is a small, anonymous function defined using the lambda keyword.

Characteristics

- No function name
- Can have multiple arguments
- Contains only one expression
- Returns value automatically
- Used for short, temporary operations

Syntax

```
lambda arguments : expression
```

Example

```
square = lambda x: x * x
print(square(5))
```

```
25
```

1. `map()` Function

Concept

`map()` applies a function to each element of an iterable and returns a map object.

Syntax

```
map(function, iterable)
```

Usage

- Transform data
- Apply same operation to all elements

Example Using Lambda

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

```
[2, 4, 6, 8]
```

1. `filter()` Function

Concept

`filter()` selects elements from an iterable based on a condition.

Syntax

```
filter(function, iterable)
```

Usage

- Remove unwanted data
- Conditional selection

Example Using Lambda

```
numbers = [1, 2, 3, 4, 5, 6]
result = filter(lambda x: x % 2 == 0, numbers)
print(list(result))
```

```
[2, 4, 6]
```

1. `reduce()` Function

Concept

`reduce()` applies a function cumulatively to the elements and reduces them to a single value.

`reduce()` is in the functools module.

Syntax

```
reduce(function, iterable)
```

Usage

- Aggregation (sum, product, max)
- Cumulative computation

Example Using Lambda

```
from functools import reduce

numbers = [1, 2, 3, 4]
result = reduce(lambda a, b: a + b, numbers)
print(result)
```

```
10
```

Comparison of `map()`, `filter()`, `reduce()` | Function | Purpose | Output | | ---------- | ---------- | ---------- | | `map()` | Transform elements | Iterable | | `filter()` | Select elements | Iterable | | `reduce()` | Combine elements | Single value |