1. Compare and contrast the float and Decimal classes' benefits and drawbacks.

Sol:-

The float and Decimal classes in Python represent numbers with floating-point precision, but they have some differences in terms of benefits and drawbacks:

Float:

Benefits:

Efficient: float is implemented using the native floating-point support of the underlying hardware, making it efficient for arithmetic operations.

Wide range: float can represent a wide range of numbers, both large and small.

Built-in math functions: The float class provides built-in math functions like sin, cos, sqrt, etc., and supports mathematical operations such as addition, subtraction, multiplication, and division.

Drawbacks:

Limited precision: Floating-point numbers have limited precision due to the nature of their representation, which can lead to small rounding errors.

Inexact representation: Some decimal numbers cannot be represented exactly as float, leading to potential inaccuracies in calculations.

Loss of precision in arithmetic: Repeated arithmetic operations on float numbers can accumulate small errors, known as floating-point errors.

Decimal:

Benefits:

Arbitrary precision: The Decimal class offers arbitrary precision decimal arithmetic, allowing for precise and accurate decimal calculations without rounding errors.

Control over precision: You can specify the desired precision and rounding behavior for Decimal objects.

Accurate representation of decimal numbers: Decimal can represent decimal numbers exactly, making it suitable for financial calculations and applications that require high precision.

Drawbacks:

Slower performance: The Decimal class performs decimal arithmetic using software-based algorithms, which can be slower compared to hardware-based floating-point operations.

Larger memory footprint: Decimal objects require more memory compared to float objects due to the additional precision and decimal representation.

Limited range: The range of Decimal numbers is limited by the available memory, and extremely large or small numbers may not be representable.

2. Decimal('1.200') and Decimal('1.2') are two objects to consider. In what sense are these the same object? Are these just two ways of representing the exact same value, or do they correspond to different internal states?

Sol:-

Decimal('1.200') and Decimal('1.2') represent the same mathematical value, but they have different internal states because they have different string representations. The trailing zero in '1.200' is significant and indicates that it is explicitly representing the value with three decimal places, whereas '1.2' represents the same value with only one decimal place.

3. What happens if the equality of Decimal('1.200') and Decimal('1.2') is checked?

Sol:-

If you were to compare these two objects using the == operator, Python would consider them unequal because they have different internal states. However, if you perform mathematical operations or comparisons between them, their mathematical values will be considered the same, and the internal representation differences will not affect the result.

4. Why is it preferable to start a Decimal object with a string rather than a floating-point value?

Sol:-

It is preferable to start a Decimal object with a string rather than a floating-point value to avoid potential precision loss and rounding errors associated with floating-point representations.

5. In an arithmetic phrase, how simple is it to combine Decimal objects with integers?

Sol:-

Combining Decimal objects with integers in arithmetic operations is straightforward and simple. The Decimal class in Python supports arithmetic operations with various numeric types, including integers.

from decimal import Decimal

decimal\_num = Decimal('3.14')

integer\_num = 5

# Addition

result = decimal\_num + integer\_num

print(result) # Output: 8.14

# Subtraction

result = decimal\_num - integer\_num

print(result) # Output: -1.86

# Multiplication

result = decimal\_num \* integer\_num

print(result) # Output: 15.7

# Division

result = decimal\_num / integer\_num

print(result) # Output: 0.628

6. Can Decimal objects and floating-point values be combined easily?

Sol:-

Combining Decimal objects and floating-point values requires some consideration to ensure accurate and expected results. While it is possible to combine Decimal objects with floating-point values, there are some considerations to keep in mind due to the inherent imprecision of floating-point arithmetic.

When a Decimal object is combined with a floating-point value, the floating-point value will be implicitly converted to a Decimal object before the operation takes place. However, because floating-point values have limited precision and may suffer from rounding errors, the result of the operation may not be exact.

7. Using the Fraction class but not the Decimal class, give an example of a quantity that can be expressed with absolute precision.

Sol:-

from fractions import Fraction

fraction\_num = Fraction(3, 5)

print(fraction\_num) # Output: 3/5

8. Describe a quantity that can be accurately expressed by the Decimal or Fraction classes but not by a floating-point value.

Sol:-

A quantity that can be accurately expressed by the Decimal or Fraction classes but not by a floating-point value is a repeating or non-terminating decimal. Floating-point values are stored in a binary format and have finite precision, which means they can only approximate certain decimal values.

from decimal import Decimal

from fractions import Fraction

decimal\_num = Decimal('0.333')

fraction\_num = Fraction(1, 3)

print(decimal\_num) # Output: 0.333

print(fraction\_num) # Output: 1/3

Q9.Consider the following two fraction objects: Fraction(1, 2) and Fraction(1, 2). (5, 10). Is the internal state of these two objects the same? Why do you think that is?

Sol:-

No, the internal state of the two fraction objects Fraction(1, 2) and Fraction(1, 2). (5, 10) is not the same. The first fraction Fraction(1, 2) represents the value 1/2, which is equivalent to the fraction Fraction(5, 10). However, they are not the same object.

The Fraction class in Python simplifies fractions by dividing both the numerator and denominator by their greatest common divisor (GCD) to obtain the simplest form. In this case, both Fraction(1, 2) and Fraction(5, 10) simplify to the same value, which is Fraction(1, 2). However, the internal state of the two objects may still be different.

When fractions are created, they are typically normalized by dividing the numerator and denominator by their GCD. This normalization process ensures that fractions are in their simplest form. As a result, even if two fractions have different representations initially, they will be reduced to the same internal state if they represent the same value.

Q10. How do the Fraction class and the integer type (int) relate to each other? Containment or inheritance?

Sol:-

The Fraction class and the integer type (int) in Python do not have a direct relationship of containment or inheritance. They are distinct data types that serve different purposes.

The Fraction class is used to represent rational numbers as fractions, allowing precise and exact representation of fractional values. It provides functionality for performing arithmetic operations on fractions, simplifying them, and converting them to different representations.

On the other hand, the integer type (int) represents whole numbers without fractional parts. Integers are used for performing mathematical operations involving whole numbers, such as addition, subtraction, multiplication, and division.

While both the Fraction class and the integer type deal with numbers, they have different internal implementations and serve different purposes. The Fraction class is designed specifically for handling rational numbers, while integers are a fundamental data type in Python.