**Experiment Number – 15**

**Title - Program to demonstrate binary operator overloading**

**Theory -**

Python operators work for predefined data types like int, str, list, etc, but we can change the way an operator works depending on the types of operands that we use. **Operator Overloading** means giving extended meaning beyond their predefined operational meaning. We may use any inbuilt or user-defined operand. This is the feature of operator overloading in Python that allows the same built-in operator to behave differently according to the context of the implementation of a problem.

Operator overloading in Python provides the ability to override the functionality of a built-in operator in user-defined classes.

For example, the “\*” operator can be overloaded not only as a multiplier for numbers but also as a repetition operator for lists or strings. The operator + is used to add two integers as well as join two strings and merge two lists. It is achievable because ‘+’ operator is overloaded by int class and str class.

Operator overloading is also known as Operator Ad-hoc Polymorphism. Operator overloading allows operators to have user-defined meanings on user-defined types (classes). It is used to customize the definition of Python operators for a user-defined class.

**Python Special Functions**

In Python we have certain functions which are used to perform some special tasks, these functions are known as **special functions**. These functions begin and end with a “\_\_” (double underscore).

In Python, we have certain special functions also known as magic methods. These special functions are invoked automatically by Python in certain special situations, or we may also invoke them ourselves.

These special functions can be used to overload the built-in operators for our custom classes. For example, the \_\_add\_\_() special function is used to overload the “+” operator for our custom classes.

## Python magic functions used for operator overloading:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Expression** | **Internally** |
| Addition | p1 + p2 | p1.\_\_add\_\_(p2) |
| Subtraction | p1 - p2 | p1.\_\_sub\_\_(p2) |
| Multiplication | p1 \* p2 | p1.\_\_mul\_\_(p2) |
| Power | p1 \*\* p2 | p1.\_\_pow\_\_(p2) |
| Division | p1 / p2 | p1.\_\_truediv\_\_(p2) |
| Floor Division | p1 // p2 | p1.\_\_floordiv\_\_(p2) |
| Remainder (modulo) | p1 % p2 | p1.\_\_mod\_\_(p2) |
| Bitwise Left Shift | p1 << p2 | p1.\_\_lshift\_\_(p2) |
| Bitwise Right Shift | p1 >> p2 | p1.\_\_rshift\_\_(p2) |
| Bitwise AND | p1 & p2 | p1.\_\_and\_\_(p2) |
| Bitwise OR | p1 | p2 | p1.\_\_or\_\_(p2) |
| Bitwise XOR | p1 ^ p2 | p1.\_\_xor\_\_(p2) |
| Bitwise NOT | ~p1 | p1.\_\_invert\_\_(self) |
| Unary Plus | +p1 | p1.\_\_pos\_\_(self) |
| Unary Minus | -p1 | p1.\_\_neg\_\_(self) |
| Less than | p1 < p2 | p1.\_\_lt\_\_(p2) |
| Less than or equal to | p1 <= p2 | p1.\_\_le\_\_(p2) |
| Equal to | p1 == p2 | p1.\_\_eq\_\_(p2) |
| Not equal to | p1 != p2 | p1.\_\_ne\_\_(p2) |
| Greater than | p1 > p2 | p1.\_\_gt\_\_(p2) |
| Greater than or equal to | p1 >= p2 | p1.\_\_ge\_\_(p2) |

## Advantages of Operator Overloading

* **Code readability**: Operator overloading allows developers to use familiar operators to perform operations on user-defined objects. This makes code more intuitive and easier to read, especially when working with complex data types.
* **Simplicity**: Overloading operators allows developers to create simpler, more elegant code that is easier to understand and maintain. It can reduce the amount of code needed to perform certain operations, making programs more concise and efficient.
* **Flexibility**: By overloading operators, developers can define custom behavior for built-in operators to work with their own classes. This provides a great deal of flexibility in how classes are used, and can be especially useful in libraries and APIs.
* **Consistency**: Operator overloading can help ensure consistency in how objects behave, making it easier to reason about code and debug errors.
* **Ease of use**: Overloading operators makes it easier for developers to perform common operations, since they can use familiar operators to perform them. This can improve the developer experience and speed up the development process.
* **Customization**: Overloading operators allows developers to create custom operations for their classes, making them more powerful and flexible. This can be especially useful for numerical and scientific computing, where custom operations are often needed.
* **Integration with built-in functions**: Python's built-in functions and modules often rely on operator overloading, making it an important tool for integrating user-defined objects with the rest of the language. This can improve the performance and functionality of Python programs.

Exercise –

1. Write a program to overload unary – operator
2. Write a program to overload multiplication operator
3. Write a program to overload equal to operator
4. Write a program to overload division operator
5. Write a program to overload remainder operator