Note: Operators that appear between operands, like 1 + rate, are infix operators. Operator overloading is necessary to support infix operator notation with user-defined or extension types, such as NumPy arrays

Note: Operator overloading allows user-defined objects to interoperate with infix operators such as + and |, or unary operators like - and ~.(()), attribute access (.), and item access/slicing ([]) are also operators in Python, but this chapter covers unary and infix operators.

1. Unary operators are designed to work on single operand(like ++, --) but the infix operator comes straight between 2 operands.

2. There are 3 unary operators with their related special method in python.
   a. ~ Bitwise invert , implemented by __invert__ : ~x = -(x + 1)
   b. -Arithmetic negation,implemented by __neg__ : x= -20 so -x=20
   c. + Arithmetic unary plus :implemented by __pos__ : x == +x

3. Special methods implementing unary or infix operators should never change the value of the operands. Expressions with such operators are expected to produce results by creating new objects. Only augmented assignment operators may change the first operand (self),

4. Prominent Note: if an operator special method cannot return a valid result because of type incompatibility, it should return NotImplemented and not raise TypeError. By

returning NotImplemented, you leave the door open for the implementer of the other operand type to perform the operation when Python tries the reversed method call.

5. To support operations involving objects of different types, Python implements a spe- cial dispatching mechanism for the infix operator special methods. Given an expres- sion a + b, the interpreter will perform these steps:
   a. If a has __add__, call a.__add__(b) and return result unless it's NotImplemented.
   b. If a doesn't have __add__, or calling it returns NotImplemented, check if b has __radd__, then call b.__radd__(a) and return result unless it's NotImplemented
   c. If b doesn't have __radd__, or calling it returns NotImplemented, raise TypeError with an unsupported operand types message

Note: In Python, we can change the way operators work for user-defined types.
For example, the + operator will perform arithmetic addition on two numbers, merge two lists, or concatenate two strings. This feature in Python that allows the same operator to have different meaning according to the context is called operator overloading.

6. __matmul__ and __rmatmul__ are being used as @ infix operator special methods (for multiplying 2 same-length vectors).

7. Rich Comparison Operators: The handling of the rich comparison operators ==, !=, >, =, and <= by the Python interpreter is a bit different.

   a. The same set of methods is used in forward and reverse operator calls. For example, in the case of ==, both the forward and reverse calls invoke __eq__, only swapping arguments; and a forward call to __gt__ is followed by a reverse call to __lt__ with the arguments swapped.

8. If a class does not implement the in-place operators listed in Table 16-1, the augmen- ted assignment operators work as syntactic sugar: a += b is evaluated exactly as a = a + b. That's the expected behavior for immutable types, and if you have __add__, then += will work with no additional code.

9. The in-place special methods should never be implemented for immutable types like our Vector class. This is fairly obvious, but worth stating anyway.