6.8 Expression Trees

- Amazing application of trees: *representing programs*

PYTHON INTERPRETER:
- Taking our python file and running it
- Strings -> linear structure
- Programs -> recursive structures
- When given a file:
    - Parse text from file to Abstract Syntax Tree (ABT)
    - The Abstract Syntax Tree can be modelled using tree-based data structures

THE EXPR CLASS:
- **Expression :** a piece of code which is meant to be evaluated, returns value of that expression.
- Expressions are building blocks of the language (necessary for computations)
- There are diff types of expressions but kept within the same interface through inheritance

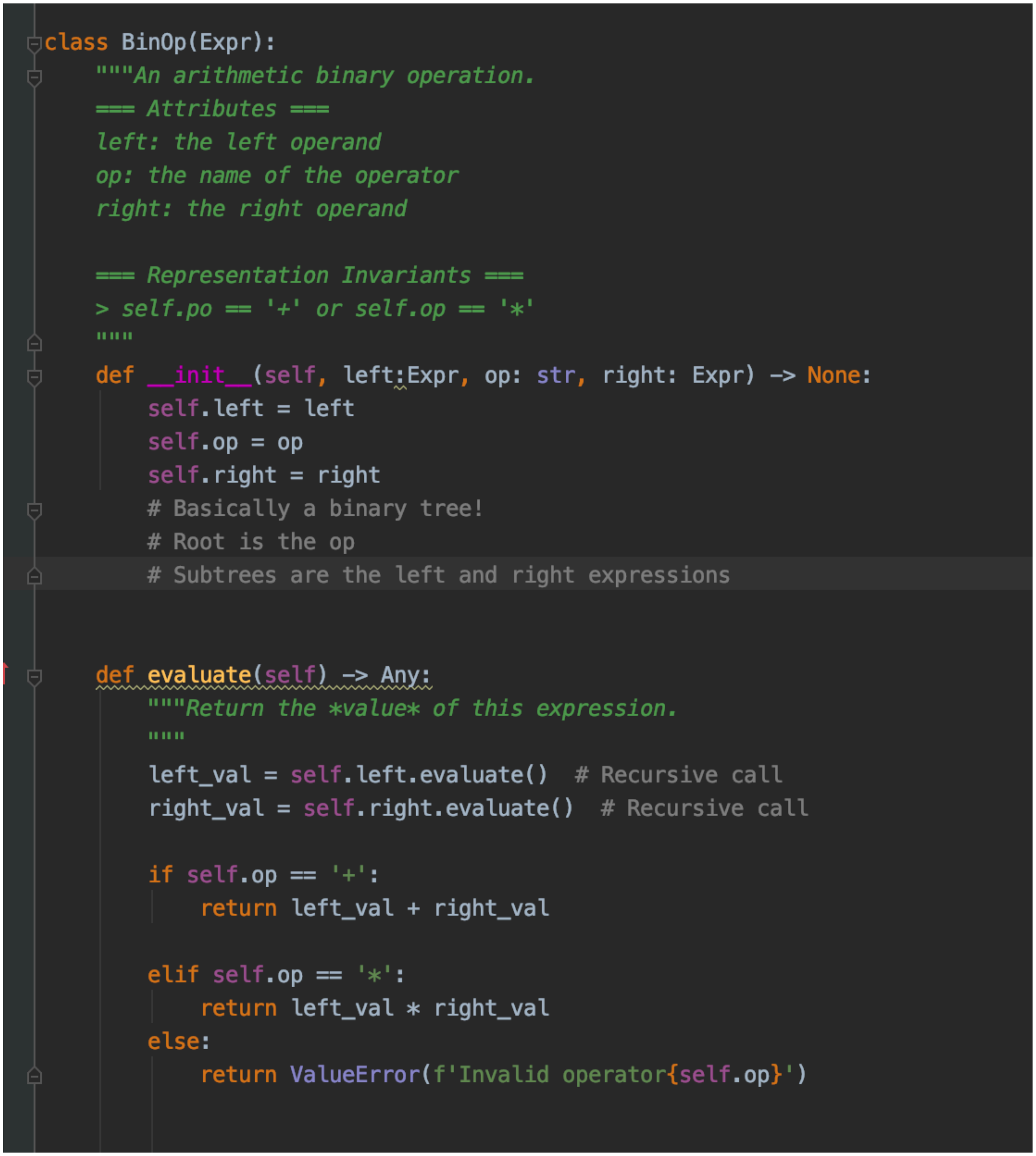
```
class Expr:
    """An abstract class representing a Python expression.
    """
    def evaluate(self) -> Any:
        """Return the *value* of this expression.

        The returned value should be the result of how this expression would be
        evaluated by the Python interpreter.
        """
        raise NotImplementedError
```

NUMERIC CONSTANTS:
- Ints and floats
- The value of the constant (what the class does)
- The base cases/ leaves of an abstract syntax tree

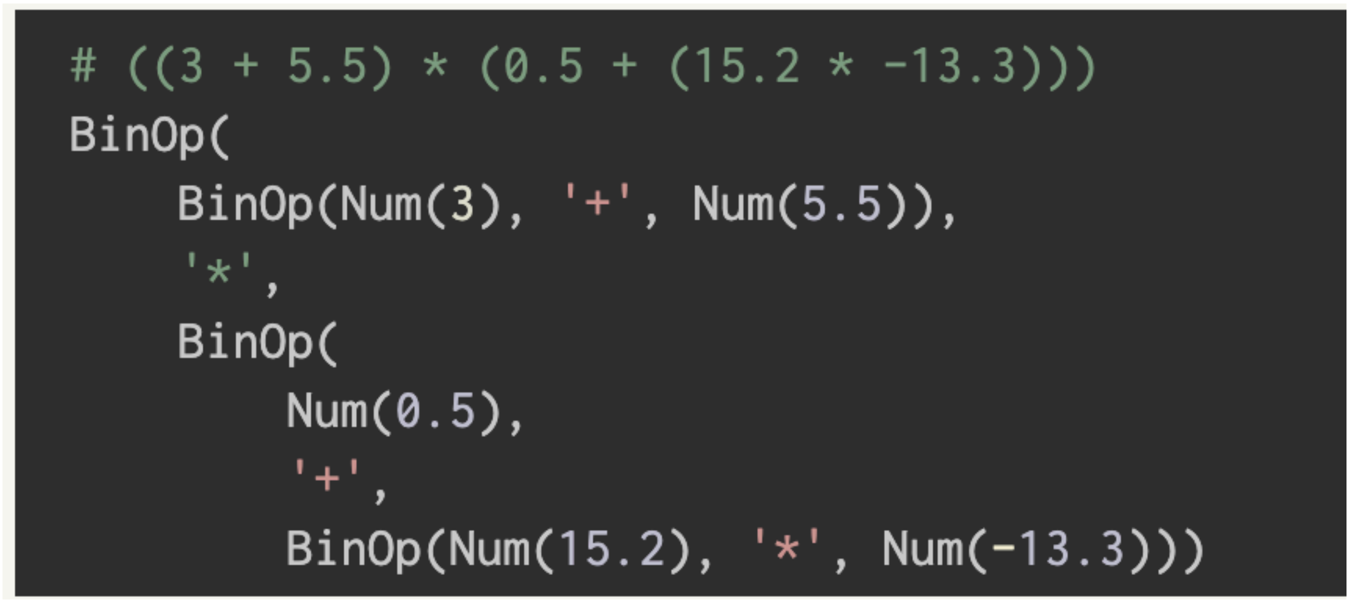
```
class Num(Expr):
    """An numeric constant literal.

    === Attributes ===
    n: the value of the constant
    """
    n: Union[int, float]

    def __init__(self, number: Union[int, float]) -> None:
        """Initialize a new numeric constant."""
        self.n = number

    def evaluate(self) -> Any:
        """Return the *value* of this expression.

        The returned value should be the result of how this expression would be
        evaluated by the Python interpreter.

        >>> number = Num(10.5)
        >>> number.evaluate()
        10.5
        """
        return self.n  # Simply return the value itself!
```

`BinOp` **ARITHMETIC OPERATIONS:**

- Arithmetic operation: an expression of three parts (left and right subexpressions & operator itself)
- Basically a binary tree:
    - Root: the operand:
    - Subtrees: left and right subexpressions



```python
class BinOp(Expr):
    """An arithmetic binary operation.
    === Attributes ===
    left: the left operand
    op: the name of the operator
    right: the right operand

    === Representation Invariants ===
    > self.po == '+' or self.op == '*'
    """
    def __init__(self, left:Expr, op: str, right: Expr) -> None:
        self.left = left
        self.op = op
        self.right = right
        # Basically a binary tree!
        # Root is the op
        # Subtrees are the left and right expressions


    def evaluate(self) -> Any:
        """Return the *value* of this expression.
        """
        left_val = self.left.evaluate()  # Recursive call
        right_val = self.right.evaluate()  # Recursive call

        if self.op == '+':
            return left_val + right_val

        elif self.op == '*':
            return left_val * right_val
        else:
            return ValueError(f'Invalid operator{self.op}')
```



```
# ((3 + 5.5) * (0.5 + (15.2 * -13.3)))
BinOp(
    BinOp(Num(3), '+', Num(5.5)),
    '*',
    BinOp(
        Num(0.5),
        '+',
        BinOp(Num(15.2), '*', Num(-13.3)))
```

- Although there are recursive calls in the methods... the base cases themselves are the child classes.
- **In this case**:
  - Base Case is class Num
  - BinOp Class makes the recursive call and adds or multiplies the numbers accordingly
-