6.8 Expression Trees

To wrap up our study of tree-based data structures in this course, we're going to look at one particularly rich application of trees: *representing programs*. Picture a typical Python program you've written: a few classes, more than a few functions, and dozens or even hundreds of lines of code. As humans, we read and write code as *text*, and we take for granted the fact that we can ask the computer to run our code to accomplish pretty amazing things.

But what actually happens when we "run" a program? Another program, called the *Python interpreter*, is responsible for taking our file and running it. But as you've experienced first-hand by now, writing programs that work directly with text is *hard*; reading strings of characters and extracting meaning from them requires a lot of fussing with small details. There's deeper problem with working directly with text: strings are fundamentally a *linear* structure, but programs (in Python and other programming languages) are much more complex, and in fact have a naturally *recursive* structure. For example, we can nest `for` loops and `if` statements within each other as many times as we want, in any order that we want.

So the first step that the Python interpreter takes when given a file to run is to *parse* the text from the file, and create a new representation of the program, called an *Abstract Syntax Tree (AST)*.[1]

[1] This is, in fact, a simplification: given the complex nature of parsing and Python programs, there is usually more than one kind of tree that is created during the execution of a the program, representing different "phases" of the process. You'll learn about this more in a course on programming languages or compilers.

The "Tree" part is significant: given the recursive

nature of Python programs, it is natural that we'll use a tree-based data structure to represent them!

This week, we're going to explore the basics of modeling programs using tree-based data structures. Of course, we aren't going to be able to model all of the Python language in a such short period of time, and so we'll focus on a relatively straightforward part of the language: some simple *expressions* to be evaluated.

The `Expr` class

In Python, an *expression* is a piece of code which is meant to be evaluated, returning the value of that expression.[2]

[2] This is in contrast with *statements*, which represent some kind of action like variable assignment or `return`, and with *definitions*, using keywords like `def` and `class`.

Expressions are the basic building blocks of the language, and are necessary for computing anything. But because of the immense variety of expression types in Python, we cannot use just one single class to represent all types of expressions. Instead, we'll use different classes to represent each kind of expression—but use inheritance to keep them all to the same fundamental interface. This will set our implementation of "expression trees" apart from other kinds of tree representations we have seen in the course so far.

To begin, here is an abstract class.

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

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

Notice that we haven't specified any attributes for

this class! Every type of expression will use a different set of attributes to represent the expression. Let's make this concrete by looking at two expression types.

`Num`: numeric constants

The simplest type of Python expression is a *literal constant* like `3` or `'hello'`. We'll start just by representing numeric constant (`int`s and `float`s). As you might expect, this is a pretty simple class, with just a single attribute representing the value of the constant.

```
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 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!
```

You can think of constants as being the base cases, or leaves, of an abstract syntax tree. Next, we'll look at one way of combining these constant together in larger expressions.

`BinOp`: arithmetic operations

The obvious way to combine numbers together is through the standard arithmetic operations. In

Python, an *arithmetic operation* is an expression that consists of three part: a left and right subexpression (the two *operands* of the expression), and the operator itself. We'll represent this with the following class:[3]

[3] For simplicity, we restrict the possible operations to only `+` and `*` for this example.

```
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.op == '+' or self.op == '*'
    """
    left: Expr
    op: str
    right: Expr

    def __init__(self, left: Expr, op: str, right: Expr) -> None:
        """Initialize a new binary operation expression.

        Precondition: <op> is the string '+' or '*'.
        """
        self.left = left
        self.op = op
        self.right = right
```

Note that the `BinOp` class is basically a binary tree! Its "root" value is the operator name (stored in the attribute `op`), while its left and right "subtrees" represent the two *operand subexpressions*.

For example, we could represent the simple arithmetic expression `3 + 5.5` in the following way:

```
BinOp(Num(3), '+', Num(5.5))
```

But of course, the types of the `left` and `right` attributes aren't `Num`, they're `Expr`—so either of these can be `BinOp`s as well:

```
# ((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)))
```

Now, it might seem like this representation is more complicated, and certainly more verbose. But we must be aware of our own human biases: because we're used to reading expressions like `((3 + 5.5) * (0.5 + (15.2 * -13.3)))`, we take it for granted that we can quickly *parse* this text in our heads to understand its meaning.

A computer program like the Python interpreter, on the other hand, can't do anything "in its head": a programmer needs to have written code for it every action it can take! And this is where the tree-like structure of `BinOp` really shines. To *evaluate* a binary operation, we first evaluate its left and right operands, and then combine them using the specified arithmetic operator. The code is among the simplest we've every written!

```
class BinOp:
    def evaluate(self) -> Any:
        """Return the *value* of this expression.
        """
        left_val = self.left.evaluate()
        right_val = self.right.evaluate()

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

The subtle recursive structure of expression trees

Even though the code for `BinOp.evaluate` looks simple, it actually uses recursion in a pretty subtle way. Notice that we're making pretty normal-looking recursive calls `self.left.evaluate()` and

`self.right.evaluate()`, matching the tree structure of `BinOp`. But… *where's the base case?*

This is probably the most significant difference between our expression tree representation and the tree-based classes we've studied so far in this course. Because we are using multiple subclasses of `Expr`, there are *multiple* `evaluate` methods, one in each subclass. Each time `self.left.evaluate` and `self.right.evaluate` are called, they could either refer to `BinOp.evaluate` *or* `Num.evaluate`, depending on the types of `self.left` and `self.right`.

In particular, notice that `Num.evaluate` does *not* make any subsequent calls to `evaluate`, since it just returns the object's `n` attribute. This is the true "base case" of `evaluate`, and it happens to be located in a completely different method than `BinOp.evaluate`! So fundamentally, `evaluate` is still an example of structural recursion, just one that spans multiple `Expr` subclasses.

Looking ahead

Of course, Python programs consist of much, much more than simple arithmetic expressions! In this course, we're really only scratching the surface of the full set of classes we would need to completely represent any valid Python code. But even though a complete understanding is beyond the scope of this course, the work that we're doing here is *not* merely theoretical, but is actually a concrete part of the Python language itself, and tools which operate on Python programs.

It turns out that there is a built-in Python library called `ast` (short for "abstract syntax tree") that uses the exact same approach we've covered here, but of course is comprehensive enough to cover the entire spectrum of the Python language. If you're interested in reading more about this, feel free to check out some excellent documentation at https://greentreesnakes.readthedocs.io.

CSC148 Notes Table of Contents

CSC148 Course Website