14.3 From Expressions to Statements

In the previous two sections, we looked at how to represent Python expressions using abstract syntax trees. However, expressions are only one subset of possible Python code, and so in this section we'll turn our attention to representing and evaluating Python statements as well.

The `Statement` abstract class

As we learned all the way back in Section 1.4, all expressions are statements, but not all statements are expressions. When we evaluate an expression, we expect to get a value returned. But when we evaluate an statement, we often do not get back a value, but instead some other effect, like recording a variable assignment, manipulating control flow or returning from a function.

So how we'll represent this in code is by creating a new abstract class `Statement`, that is a parent of `Expr`.

```
class Statement:
    """An abstract class representing a Python statement.

    We think of a Python statement as being a more general piece of code than
      a
    single expression, and that can have some kind of "effect".
    """
    def evaluate(self, env: dict[str, Any]) -> Optional[Any]:
        """Evaluate this statement with the given environment.

        This should have the same effect as evaluating the statement by the
        real Python interpreter.

        Note that the return type here is Optional[Any]: evaluating a
         statement
        could produce a value (this is true for all expressions), but it
         might
        only have a *side effect* like mutating `env` or printing something.
        """
        raise NotImplementedError


def Expr(Statement):
    """An abstract class representing a Python expression.

    We've now modified this class to be a subclass of Statement.
    """
```

You might wonder, why bother with an `Expr` class at all? As we'll see in this section, `Statement` may have many different subclasses beyond `Expr`, representing non-expression statements like assignment statements. Keeping `Expr` is useful for distinguishing between expression types and non-expression types. For example, a `BinOp`'s `left` and `right` attributes must be of type `Expr`, not `Statement`.[1]

[1] Consider, for example, the *invalid* Python code `(x = 3) + (y = 5)`. Assignment statements can't be added together!

`Assign`: *an assignment statement*

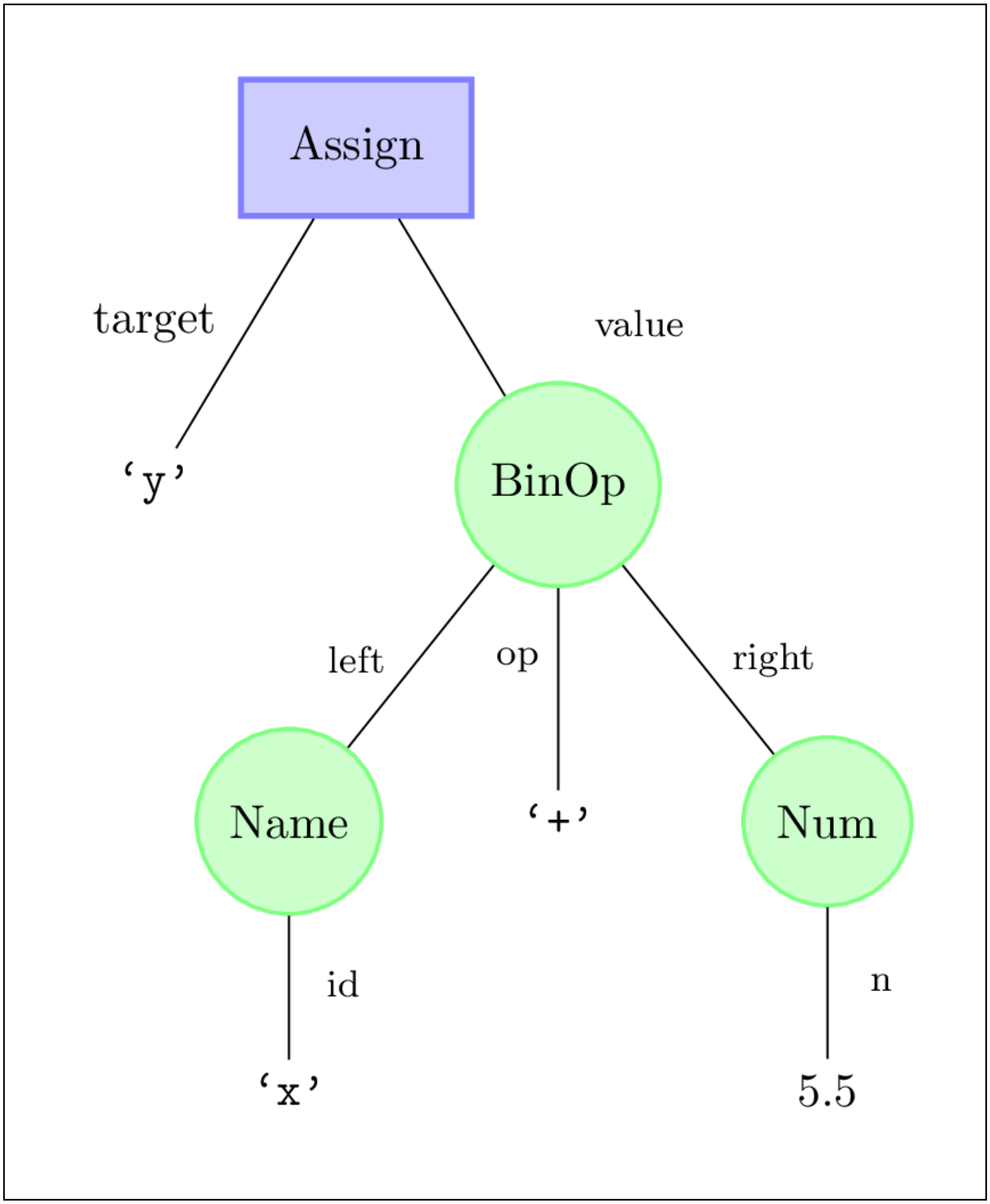
Let's start by resolving an issue we encountered at the end of the previous section: how do we actually build up the variable environment? To do this, we'll need to represent assignment statements as a new data type that we'll call `Assign`:

```python
class Assign(Statement):
    """An assignment statement (with a single target).

    Instance Attributes:
        - target: the variable name on the left-hand side of the equals sign
        - value: the expression on the right-hand side of the equals sign
    """
    target: str
    value: Expr

    def __init__(self, target: str, value: Expr) -> None:
        """Initialize a new Assign node."""
        self.target = target
        self.value = value

    def evaluate(self, env: dict[str, Any]) -> ...:
        """Evaluate this statement with the given environment.
        """
```

For example, here's how we could represent the statement `y = x + 5.5`:

```python
Assign('y', BinOp(Name('x'), '+', Num(5.5)))
```

Let's consider how to implement `Assign.evaluate`. Intuitively, we know what needs to happen: we need to evaluate its `value`, and the nassign that value to the variable `target`. But what does "assigning" a variable actually mean in this context? This is exactly the purpose of `env`: it stores the current variable bindings! So to assign a variable, we need to mutate `env`.[2]



[2] Note the return type below: since this method only performs a mutation operation, the return type is `None`.

```
class Assign:
    def evaluate(self, env: dict[str, Any]) -> None:
        """Evaluate this statement with the given environment.
        """
        env[self.target] = self.value.evaluate(env)
```

`Print`: *displaying text to the user*

The second `Statement` subclass we'll consider is `Print`, which represents a call to the function `print`.[3] We'll use these to actually display values to the user.

[3] In earlier versions of Python, `print` was actually a keyword (like `def` or `return`), and so really did require its own statement type. However, in our current version of Python, `print` is a function, and so this statement type is really a special case of a more general "function call" expression.

```
class Print(Statement):
    """A statement representing a call to the `print` function.

    Instance Attributes:
        - argument: The argument expression to the `print` function.
    """
    argument: Expr

    def __init__(self, argument: Expr) -> None:
        """Initialize a new Print node."""
        self.argument = argument

    def evaluate(self, env: dict[str, Any]) -> None:
        """Evaluate this statement.

        This evaluates the argument of the print call, and then actually
        prints it. Note that it doesn't return anything, since `print`
         doesn't
        return anything.
        """
        print(self.argument.evaluate(env))
```

`Module`: a sequence of statements

Now that we have two different `Statement` subclasses, let's talk about putting statements together. For this purpose we'll define a new class called `Module`, which represents a full Python program, consisting of a sequence of statements.

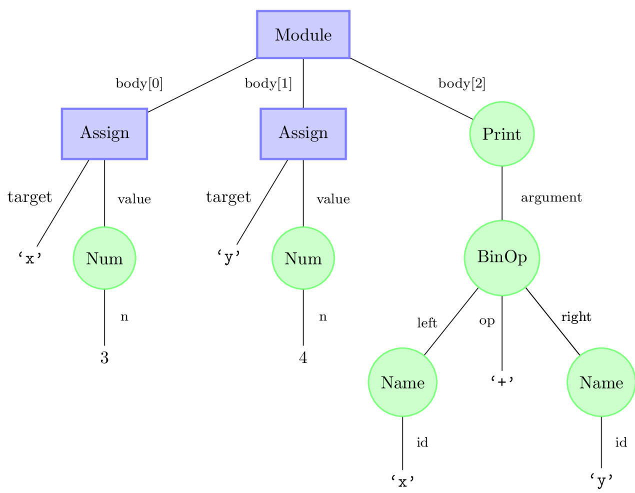
```
class Module:
    """A class representing a full Python program.

    Instance Attributes:
        - body: A sequence of statements.
    """
    body: list[Statement]

    def __init__(self, body: list[Statement]) -> None:
        """Initialize a new module with the given body."""
        self.body = body
```

For example, consider the following short Python program:

```
x = 3
y = 4
print(x + y)
```

We can represent this as follows:

```
Module([
    Assign('x', Num(3)),
    Assign('y', Num(4)),
    Print(BinOp(Name('x'), '+', Name('y')))
])
```



Note that `Module` itself is *not* a subclass of `Statement`, as a `Module` can't be nested recursively within other `Module`s. However, we can think of a `Module` as being the *root* of a complete abstract syntax tree, as visualized by the above diagram.

Evaluating modules

To evaluate a module, we do two things: first, initialize an empty dictionary to represent the environment (starting with no variable bindings), and then iterate over each statement the module body and evaluate it.

```
class Module:
    def evaluate(self) -> None:
        """Evaluate this statement with the given environment.
        """
        env = {}
        for statement in self.body:
            statement.evaluate(env)
```

Let's revisit our above example:

```python
x = 3
y = 4
print(x + y)

# Or, as an abstract syntax tree:
Module([
    Assign('x', Num(3)),
    Assign('y', Num(4)),
    Print(BinOp(Name('x'), '+', Name('y')))
])
```

In this case, the module body has three statements. The first two `Assign` statements mutate `env` when we call their `evaluate` methods, so that `env` becomes the dictionary `{'x': 3, 'y': 4}`. Then in the final `Print` statement, this `env` is passed recursively through the calls to `evaluate`, so that the vales of `x` and `y` can be looked up.

Control flow statements

Finally, we'll briefly mention that we can represent compound statements (i.e., statements that consist of other statements) using the same idea as a `Module`. For example, here is one way we could define a restricted form of an if statement that has just two branches, an if and else branch:[4]

[4] Try combining this with the `Expr` classes you defined in the prep exercises for this chapter!

```python
class If(Statement):
    """An if statement.

    This is a statement of the form:

        if <test>:
            <body>
        else:
            <orelse>

    Instance Attributes:
        - test: The condition expression of this if statement.
        - body: A sequence of statements to evaluate if the condition is
         True.
        - orelse: A sequence of statements to evaluate if the condition is
         False.
                  (This would be empty in the case that there is no `else`
         block.)
    """
```

```
    test: Expr
    body: list[Statement]
    orelse: list[Statement]
```

And similarly, we can represent a for loop over a range of numbers:

```
class ForRange(Statement):
    """A for loop that loops over a range of numbers.

        for <target> in range(<start>, <stop>):
            <body>

    Instance Attributes:
        - target: The loop variable.
        - start: The start for the range (inclusive).
        - stop: The end of the range (this is *exclusive*, so <stop> is not
     included
              in the loop).
        - body: The statements to execute in the loop body.
    """
    target: str
    start: Expr
    stop: Expr
    body: list[Statement]
```

We encourage you to try implementing these two classes as exercises!