CSC148 - Expression Trees: Control Flow Structures

Now we're going to extend our classes to represent two basic control flow structures in Python: `if` statements and `for` loops (limited to being over a range of numbers). First, make sure you understand the `Module` class and the implementation of its `evaluate` method. You'll need this same idea here, since both `if` statements and `for` loops contain blocks of code.

1. First, read through the following class.

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

    === 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]
```

   To make sure you understand this class, answer the following questions.

   (a) Write down an expression that represents the following Python statement (we asked you to do something similar on this week's prep quiz for arithmetic expressions).

```
if False:
    x = 3
    y = 4
else:
    y = 5
```

   (b) In Python, the `else` part is optional. How could we represent an `if` but no `else` block using the `If` class above?

2. Now, implement the `If.evaluate` method. Note that you can use `if` statements in your implementation!

```
def evaluate(self, env: Dict[str, Any]) -> Optional[Any]:
    """Evaluate this statement.

    >>> stmt = If(Bool(True),
    ...           [Assign('x', Num(1))],
    ...           [Assign('y', Num(0))])
    ...
    >>> env = {}
    >>> stmt.evaluate(env)
    >>> env
    {'x': 1}
    """
```

3. A `for` loop repeats the same block of code once for each value in a given iterable, like a list or range of numbers. To keep things simple, for this worksheet we're only going to consider `for` loops over a fixed range of integers.

   Read through the following class, and then implement its `evaluate` method. Think carefully about how you make the `for` loop variable accessible when you evaluate the statements in the loop body. Note that you can use `for` loops in your implementation!

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

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

    === 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]

    def evaluate(self, env: Dict[str, Any]) -> Optional[Any]:
        """Evaluate this statement.

        Raise a TypeError if the start or stop expression does *not*
        evaluate to integers. (This is technically a bit stricter than real Python.)

        >>> statement = ForRange('x', Num(1), BinOp(Num(2), '+', Num(3)),
        ...                      [Print(Name('x'))])
        >>> statement.evaluate({})
        1
        2
        3
        4
        """
```