12.5 Recursive Lists

In the previous two sections, we learned a formal recursive definition of nested lists, and used this definition to design and implement recursive functions that operate on nested lists.

In this section, we'll revisit the familiar (non-nested) list data type, now applying the lens of recursion. This will be our first foray into defining recursive data types in Python, and will preview the more complex recursive data types we'll study in the next chapter.

A recursive definition of lists

All the way back in Section 1.1, we defined a list as an ordered collection of zero or more (possibly duplicated) values. So far, we have studied two ways that we can implement lists in Python: the array-based implementation of the Python `list` class, and the node-based linked list implementation from Chapter 11. Though these two implementations are quite different, they share one fundamental similarity: to perform computations over these collections, we use loops to process list elements one at a time.

Yet there is another way of both defining and operating on lists, using the recursive thinking we've developed in this chapter. To start, here is a recursive definition of a list.

- The empty list `[]` is a list.
- If `x` is a value and `r` is a list, then we can construct a new list `lst` whose first element is `x` and whose other elements are the elements of `r`.

  In this case, we call `x` the **first** element of `lst`, and `r` the **rest** of `lst`.

In other words, this definition decomposes a list into its "first" and "rest" parts, and is recursive because the "rest" part is another list. Here's an example of how you could visualize this representation in Python:

```
[1, 2, 3, 4] ==
([1] +
    ([2] +
        ([3] +
            ([4] + []))))
```

However, this is a cumbersome way of decomposing a regular Python list. So instead, we will define a new *recursive Python class* to represent this recursive definition. Here is our first attempt:

```python
from __future__ import annotations
from typing import Any


class RecursiveList:
    """A recursive implementation of the List ADT.
    """
    # Private Instance Attributes:
    #   - _first: The first item in the list.
    #   - _rest: A list containing the items that come after the first one.
    _first: Any
    _rest: RecursiveList
```

We say that this `RecursiveList` data type is recursive, because its `_rest` instance attribute refers to another instance of `RecursiveList`. However, there's a slight problem with this definition: how do we represent the empty list? After all, an empty list has no "first" or "rest" values, and so these instance attributes don't have an obvious "value" in this case.

To solve this problem we'll simply set both these attributes to `None` to represent an empty list. However, we need to be careful to document this behaviour carefully in a representation invariant, so that we keep track of exactly when we expect a `None` value.[1]

[1] As we saw with linked lists, a common programming error is using an expression without realizing its value might be `None`.

Here is our updated definition of the `RecursiveList` class, with `Optional` used for the attributes and a basic initializer:

```python
class RecursiveList:
    """A recursive implementation of the List ADT.

    Representation Invariants:
        - (self._first is None) == (self._rest is None)
    """
    # Private Instance Attributes:
    #   - _first: The first item in the list, or None if this list is empty.
    #   - _rest: A list containing the items that come after the first one,
    #            or None if this list is empty.
    _first: Optional[Any]
    _rest: Optional[RecursiveList]

    def __init__(self, first: Optional[Any], rest: Optional[RecursiveList]) -
    > None:
        """Initialize a new recursive list."""
        self._first = first
        self._rest = rest
```

And finally, here is how we could create a `RecursiveList` representing the sequence `[1, 2, 3, 4]`.[2]

[2] Compare this expression with our previous recursive decomposition of this list!

```
RecursiveList(
    1,
    RecursiveList(
        2,
        RecursiveList(
            3,
            RecursiveList(
                4,
                RecursiveList(
                    None,
                    None
                )
            )
        )
    )
)
```

From recursive definition to recursive functions

Now, so far it might seem like this definition is quite cumbersome! Let's now see some payoff: just like we did with nested lists, we can use our recursive list definition to design and implement recursive functions that operate on those lists.

Let's look at a familiar example: calculating the sum of a list of integers. We'll consider each part of our definition separately.

- Let lst be an empty list. In this case, its sum is 0.
- Let lst be a non-empty list of integers. In this case, we can decompose it into its first element, x, and the rest of its elements, r. The sum of lst is equal to x plus the sum of the rest of the elements.

  For example, if $lst = [1, 2, 3, 4]$, then $sum(lst) = 1 + sum([2, 3, 4])$.

This leads naturally to the following recursive definition for list sum:

$$sum(lst) = \begin{cases} 0, & \text{if } lst \text{ is empty} \\ (\text{first of } lst) + sum(\text{rest of } lst), & \text{if } lst \text{ is non-empty} \end{cases}$$

And as we've seen a few times this chapter, we can take this definition and translate it naturally into Python, now as a method of our new `RecursiveList` class.

```python
class RecursiveList:
    def sum(self) -> int:
        """Return the sum of the elements in this list.

        Preconditions:
            - every element in this list is an int
        """
        if self._first is None:  # Base case: this list is empty
            return 0
        else:
            return self._first + self._rest.sum()
```

You might find this implementation both surprising and unsurprising at the same time. *Unsurprising*, because it is a natural translation of the recursive mathematical function, and so if you believe that function is correct, you should believe that this Python function is correct too. But *surprising* because this function calculates the sum of an arbitrary number of elements, without using a built-in aggregation function or a loop!

To make this last point more explicit, let's compare this method against two prior "list sum" functions from earlier in this course, on a Python list and a linked list:

```python
# Built-in Python list
def sum_list(lst: list[int]) -> int:
    sum_so_far = 0
    for num in lst:
        sum_so_far += num
    return sum_so_far


# Linked list
class LinkedList:
    def sum(self) -> int:
        sum_so_far = 0
        curr = self._first

        while curr is not None:
            sum_so_far += curr.item
            curr = curr.next

        return sum_so_far


# Recursive list
class RecursiveList:
    def sum(self) -> int:
        if self._first is None:
            return 0
        else:
            return self._first + self._rest.sum()
```

With our recursive definition of `sum`, it seems like there's less "work" being done. There's no loop, or explicit traversal and access of the different list elements. By writing a recursive function, we are leaving it up to the Python interpreter to handle the recursion and keep track of the function calls for us.[3]

[3] That said, this offloading of work to the Python interpreter can lead to other problems as well, as we'll discuss later in the course. While the concept of recursion applies to all programming languages, how these languages handle recursion varies quite widely.

One last thing: nodes, revisited

You might have noticed something very familiar about the attributes of our `RecursiveList` class. Check it out:

```python
class RecursiveList:
    _first: Optional[Any]
    _rest: Optional[RecursiveList]


# From Chapter 11: Linked Lists
@dataclass
class _Node:
    item: Any
    next: Optional[_Node] = None
```

That's right! The `RecursiveList` and `_Node` classes have essentially the same structure. So in fact, `_Node` is technically a recursive class as well, even though we did not introduce them as such in the previous chapter. This may be somewhat surprising, because the code that we've written for linked lists and `RecursiveList` looks quite different.

Essentially, we can view the `_Node` and `RecursiveList` classes as two ways of looking the same data type. The code defining their attributes is virtually identical—it's how we as humans *interpret* this code that differs.

For a `_Node`, we think of it as representing a *single* list element. Its recursive attribute `next` is a "link" to another `_Node`, and we traverse these links in a loop to access each node one at a time.

On the other hand, for a `RecursiveList`, we think of it as representing an *entire* sequence of elements, not just one element. Its recursive attribute `_rest` isn't a link, it's the rest of the list itself. When computing on a `RecursiveList`, we don't try to access each item individually; instead, we make a recursive function call on the `_rest` attribute, and focus on how to use the result of that call in our computation.

This duality between a node-based view of lists and recursive view of lists is something that we'll see again when studying a new data type, the *tree*, in the next chapter.