Week 5 - Recursion

Updated Oct 8, 2:30pm

This week, we're going to learn about a powerful idea called **recursion**, which we'll be using in various ways for the rest of the course. However, recursion is much more than just a programming technique: it is really a way thinking about solving problems. Many students when learning recursion for the first time get hung up on tracing through how recursive code works, and completely miss the bigger picture. If you can't **think recursively**, there's no way you'll be successful in writing recursive code!

The key idea of recursion is this: identify how an object or problem can be broken down into *smaller instances with the same structure*. We saw one example of this with linked lists last week: a linked list can be divided into two parts, the first element in the list and the rest of the list, where the rest of the list is another linked list. Another simple example of *recursive structure* is a family tree, which begins with some parents, and then the family trees of their children, which containg the family trees of *their* children, etc.

In programming, we exploit the recursive structure of objects and problems by making use of **recursive functions**, which are functions that call themselves as helper functions. You've actually seen one instance of this already, when you implemented the `PeopleChain` method `get_nth` with a call to `get_nth`.

Recursive Linked Lists

For today's lecture, we're going to make explicit the recursive structure of linked lists, and see how we can implement the methods from last week in a recursive rather than iterative way.

The key idea here is for you to get used to thinking recursively so that you can write and understand recursive functions with no confusion! (Recursion is often one of the trickiest parts of this course, which is why we're going to go slow here.)

Note that there's only one class, `LinkedListRec`, that has two attributes: `first`, representing the first item in the list, and `rest`, a `LinkedListRec` object representing all the other elements in the list. As a first step, let's take a look at the constructor:

```
class LinkedListRec:

    def __init__(self, items):
        """ (LinkedListRec, list) -> NoneType

        Create a new linked list containing the elements in items.
        If items is empty, self.first initialized to EmptyValue.
        """
        if len(items) == 0:
            self.first = None
            self.rest = None
        else:
            self.first = items[0]
```

```
            self.rest = LinkedListRec(items[1:])
```

It shouldn't come as a surprise that even the constructor is recursive: when using recursive implementations like this one, you expect (almost) all of its key functions to be implemented recursively. This is usually the simplest or most elegant way of doing it, as you'll see!

This constructor follows the basic recursive pattern: there is a base case when `len(items) == 0`, which creates an empty list, and then a recursive case which initializes the first item in the list, and then **creates a new list with the other items.** This last part, where we use the constructor of the `LinkedListRec` class again, is the key point here. This should make sense intuitively - the rest of the list should be a `LinkedListRec` object containing all of the other values in `items`!

Aside: List of None?

If the `items` argument is empty, we want to create an empty linked list, and we did so by setting `self.first = None`. There is one slight annoyance with this, though: what if we want to create a list whose first value actually *is* `None`? Doing so might run the risk of the list being interpreted as an empty list, if we check for emptiness by comparing `self.first` to `None`.

We can get around this problem simply by creating a dummy class (much like our exception classes), whose only purpose is to signify something special - in this case, that the list is empty. We do that as follows here:

```
class EmptyValue:
    pass

class LinkedListRec:
    def __init__(self, items):
        """ (LinkedListRec, list) -> NoneType

        Create a new linked list containing the elements in items.
        If list is empty, self.first initialized to EmptyValue.
        """
        if len(items) == 0:
            self.first = EmptyValue
            self.rest = None
        else:
            self.first = items[0]
            self.rest = LinkedListRec(items[1:])

    def is_empty(self):
        return self.first is EmptyValue
```

Recursive Traversal

Let's write `__getitem__`, which of course is analogous to `get_nth` from `PeopleChain`.

We know that we can identify the recursive structure of a list by separating it into its first item and the *rest* of the list, something we've made quite explicit in our class.

With that in mind, here's the key question that prompts our recursive thinking: *how is finding an item at position `index` in the list related to finding an item at a certain position*

in the rest of the list?

If you missed the lecture or have forgotten, take a few minutes to think about this question before moving on. Say we have the list `[1,2,3,4,5]` and want to access the item at position `2`; how is this related to accessing an item from the rest of the list, `[2,3,4,5]`?

Main idea (the recursive thought)

The items of the rest of the list are in the same order as the original list, except their indices have been shifted down by 1. So the item at position 2 in the original list is the same as the item at position 1 in the rest of the list!

Generalizing this a little yields the following code:

```
def __getitem__(self, index):
    return self.rest.__getitem__(index - 1)
    # Or equivalently, self.rest[index - 1]
```

Unfortunately, this behaviour doesn't quite work when `index = 0`, i.e., when we're accessing the first element of the list. (Why?) In this case, we need to do something different, but luckily one of our attributes stores exactly the data we need in this case:

```
def __getitem__(self, index):
    if index == 0:
        return self.first
    else:
        return self.rest.__getitem__(index - 1)
        # Or equivalently, self.rest[index - 1]
```

Finally, we should probably handle the case where `index` is out of bounds.

It is extremely important to note that our **recursive thinking** correctly handles this case; if, for example, `self` is a list containing 5 items, and `index = 9`, then we expect an error to be raised. The recursive call we make, `self.rest.__getitem__(index - 1)`, operates on a list of size 4, with `index = 8`, and this should *also raise an error*.

Aside: the sticking point

But you might stop here and say, "but the problem is it *doesn't* raise an error here!" This is factually true, but distracting. If we **assume** the recursive call works properly, then our code should be correct. This means that the problem with our code does *not* lie in our recursive thinking, but instead in some other part.

Let me repeat: **Our recursive thinking is correct, despite our function not handling index errors correctly. The problem lies elsewhere.**

The fix: a missing "base case"

If our code is incorrect but we are confident in our recursive thinking, then it must be the case that we've missed something much simpler, some particular form of input that could cause an error.

In this case, we've forgotten about the empty list. Aha! An error should be raised whenever we try to get an item from an empty list; the index will *always* be out of bounds. Fixing this solves our problem:

```
def __getitem__(self, index):
    if self.is_empty():
        raise IndexError
    elif index == 0:
        return self.first
    else:
        return self.rest.__getitem__(index - 1)
        # Or equivalently, self.rest[index - 1]
```

In today's lecture, we're going to look at some simple examples of recursion based on some mathematics.

Thinking Recursively

There are two main ways to think about recursive functions. One is the *low-level approach*: tracing through how a computer handles recursive functions by tracking the call stack, the local variables, and the objects stored in the heap at any point in time; this is what you'll see in the Python visualiser.

However, I've seen enough students crash and burn with this approach that I **strongly discourage using this thought process to understand recursive functions right now.** That's not to say that understading how a computer handles recursion isn't interesting or important; but you'll get plenty more experience seeing how the computer works in general later in this course and beyond, and so for this introduction to recursion, I think it's much more important to be able to write correct recursive programs at a high-level.

It might seem strange to suggest that you use recursion without understanding how Python handles it, but this is exactly what you do for every helper function you've ever called! In fact, this is one of the lynchpins of modular design and an interface, that separates what a function does with how it works. And this is the key idea in the *high-level approach* to thinking about recursive functions: every time you make a recursive call, rather than tracing through the call, *immediately* determine what happens based on the function docstring. That is, **you don't need to trace through the recursive call to figure out what it does; what's what the docstring is for!**

David Liu (liudavid at cs dot toronto dot edu)
Come find me in BA4260
Site design by Deyu Wang (dyw999 at gmail dot com)