CSC111 Lecture 6: More with Nested Lists

Hisbaan Noorani

January 27, 2021

Contents

1 Exercise 1: Inductive Reasoning for Recursive Functions

This exercise is designed to give you practice *inductive reasoning* (aka *partial tracing*) when working with recursive functions. **Do this exercise on paper, not in PyCharm.** You're developing a new skill here: reasoning recursive code by using your brain, not relying on a computer!

1. Here is a partial implementation of a nested list function that returns a brand-new list. We have deliberately omitted the base case, since it is *not relevant* for reasoning about the recursive step.

```python
def flatten(nested_list: Union[int, list]) -> list[int]:
    """Return a (non-nested) list of the integers in nested_list.

    The integers are returned in the left-to-right order they appear in
    nested_list.
    """
    if isinstance(nested_list, int):
        # Base case omitted

    else:
        result_so_far = []
        for sublist in nested_list:
            result_so_far.extend(flatten(sublist))
        return result_so_far
```

Our goal is to determine whether the recursive case is correct *without* fully tracing (or running) this code. Consider the function call `flatten([[0, -1], -2, [[-3, [-5], -7]]])`.

(a) What should `flatten([[0, -1], -2, [[-3, [-5], -7]]])` return, according to its docstring?

`[0, -1, -2, -3, -5, -7]`

(b) We'll use the loop accumulation table below to partially trace the call `flatten([[0, -1], -2, [[-3, [-5], -7]]])`. Complete the **first two columns** of this table, assuming that `flatten` works properly on each recursive call. Remember that filling out these two columns can be done *just* using the argument value and `flatten`'s docstring; you don't need to worry about the code at all!

Note: the nested list `[[0, -1], -2, [[-3, [-5], -7]]]` has just *three* sublists.

`sublist`	`flatten(sublist)`	`result_so_far`
`N/A`	`N/A`	`[]`
`[0, -1]`	`[0, -1]`	`[0, -1]`
`-2`	`[-2]`	`[0, -1, -2]`
`list([-3, [-5], -7])`	`[-3, -5, -7]`	`[0, -1, -2, -3, -5, -7]`

(c) Use the third column of the table to complete the partial trace of the recursive step. Remember that every time you reach a recursive call, don't trace into it—use the value you calculated in the second column!

Done in table

(d) Compare the final value of `result_so_far` with the expected return value of `flatten`. Does this match?

Yes, they match.

(e) Finally, write down an implementation of the `base case` of `flatten` directly on the code above.

`return[nested_list]`

2 Exercise 2: Designing recursive functions (1)

In this exercise, you'll get some practice implenting a new recursive function that operates on a nested list. We've broken down this exercise into various steps, following the *Recursive Function Design Recipe* we covered in lecture.

For your reference, here is the *nested list function code template*:

```python
def f(nested_list: Union[int, list]) -> ...:
    if isinstance(nested_list, int):
        ...
    else:
        # With a loop
        ...
        for sublist in nested_list:
            ... f(sublist) ...

        # Or with a comprehension
        ... [f(sublist) for sublist in nested_list] ...
```

Here is the function we'll implement; read through its docstring carefully first.

```python
def nested_list_contains(nested_list: Union[int, list], item: int) -> bool:
    """Return whether the given item appears in nested_list.

    If nested_list is an integer, return whether it is equal to item.
    """
```

1. *Base case example and implementation.*

   (a) Write a doctest example for this function where the argument `nested_list` is an `int`.

```
>>> nested_list_contains(3, 3)
True
>>> nested_list_contains(2, 3)
False
```

   (b) Implement the base case of this function.

```python
def nested_list_contains(nested_list: Union[int, list], item: int) -> bool:
    if isinstance(nested_list, int):
        return nested_list == item
    else:
        ...
```

2. *Recursive step examples.*

   (a) Consider the following doctest example:

```
>>> nested_list_contains([ [1, [30], 40], [], 77], 50)
False
```

Complete the following table, showing each of the sublists of `[[1, [30], 40], [], 77]` in this example, as well as what each recursive call would return. Note that we've set the `item` argument to `50` in the recursive calls, since we need to search for `50` in each sublist.

sublist	nested_list_contains(sublist, 50)
[1, [30], 40]	False
[]	False
77	False

   (b) Repeat, but with the following doctest example instead:

```
>>> nested_list_contains([[1, [30], 40], [], 77], 40)
True
```

sublist	nested_list_contains(sublist, 40)
[1, [30], 40]	True
[]	False (Doesn't run)
77	False (Doesn't run)

3. *Generalize examples and implement the recursive step.*

   Implement the recursive step for this function. Use your answers to the previous question to determine how to *aggregate* the results of the recursive call. You can use a loop or a built-in aggregation function and comprehension.

```python
def nested_list_contains(nested_list: Union[int, list], item: int) -> bool:
    if isinstance(nested_list, int):
        ...
    else:
        # Version 1, comprehension
        # Note, this only works if you have a suitable built in agregation function
        # Dropping the square brackets, we allow early return therefore > efficiency
        return any(nested_list_contains(sublist, item) for sublist in nested_list)

        # Version 2, for loop
        for sublist in nested_list:
            if nested_list_contains(sublist, item):
                return True

        retturn False
```

3 Exercise 3: Designing recursive functions (2)

Now for something a bit more complex. We previously defined the *depth* of a nested list as the maximum number of times a list is nested within another one in the nested list. Analogously, we can define the **depth of an integer in a nested list** to be the number of nested lists enclosing the object.

For example, in the nested list `[10, [[20]], [[30], 40]]`:

- The depth of `10` is 1
- The depths of `20` and `30` are 3
- The depth of `40` is 2.

For a nested list that is a single integer x, the depth of x is 0.

Your goal is to implement the following function:

```python
def items_at_depth(nested_list: Union[int, list], d: int) -> list[int]:
    """Return the list of all integers in nested_list that have depth d.

    Preconditions:
        - d >= 0
    """
```

1. *Base case example and implementation.*

(a) Write *two* doctests where `nested_list` is an `int`, covering different possibilities for `d` that lead to different return values. Make sure you understand how the return value differs in your two doctests before moving on.

```
>>> items_at_depth(10, 0)
[10]
>>> items_at_depth(10, 1)
[]
```

(b) Implement the base case. It's okay to have a nested if statement at this point.

```
def items_at_depth(nested_list: Union[int, list], d: int) -> list[int]:
    if isinstance(nested_list, int):
        if d == 0:
            return [nested_list]
        else:
            return []
    else:
        ...
```

2. *Recursive step examples.* Now let's consider the recursive step. Let `nested_list = [10, [[20]], [[30], 40]]`.

(a) Write *two* doctests with this nested list, one where `d = 0` and one where `d = 3`. Make sure you understand these two cases before moving on.

(b) Now let's study the `d = 3` case in more detail. Here is the start of a recursive call table for this example, by passing `d = 3` into each recursive call. Complete the table.

`sublist`	`items_at_depth(sublist, 3)`
`10`	`[]`
`[[20]]`	`[]`
`[[30], 40]`	`[]`

Problem! It looks like the results of the recursive calls don't actually help calculate the expected return value of `items_at_depth(nested_list, 3)`.

We need to change the depth argument we're passing in.

(c) If we want the items at depth `3` in our original nested list, what depth value should we pass into the recursive calls? Complete the table below.

`sublist`	`items_at_depth(sublist, 2)`
`10`	`[]`
`[[20]]`	`[20]`
`[[30], 40]`	`[30]`

3. *Generalize examples and implement the recursive step.* Finally, implement `items_at_depth`. Once again, it's okay to have a nested if statement.

```python
def items_at_depth(nested_list: Union[int, list], d: int) -> list[int]:
    if isinstance(nested_list, int):
        ...
    else:
        if d == 0:
            return []
        else:
            result_so_far = []
            for sublist in nested_list:
                result_so_far.extend(items_at_depth(sublist, d - 1))

            return results_so_far
```

4. If you followed our nested list template in this exercise, you would have ended up with nested if statements. Try rewriting your code so that there are no nested if statements, instead using `elif` branches for the different cases.

```python
def items_at_depth(nested_list: Union[int, list], d: int) -> list[int]:
    if d == 0 and isinstance(nested_list, int):
        return [nested_list]
    elif d == 0:
        return []
    elif isinstance(nested_list, int):
        return []
    else:
        results_so_far = []
        for sublist in nested_list:
            result_so_far.extend(items_at_depth(sublist, d - 1))

        return results_so_far
```

4 Additional exercises

4.1 Debugging with partial tracing

Now consider the following function and partial implementation:

```python
def uniques(nested_list: Union[int, list]) -> list[int]:
    """Return a (non-nested) list of the integers in nested_list, with no duplicates.
    """
    if isinstance(obj, int):
        # Base case omitted
    else:
        result_so_far = []
        for sublist in obj:
            result_so_far.extend(uniques(sublist))
        return result_so_far
```

It turns out that there is a problem with the recursive step in this function, and it has the insidious feature of being *sometimes correct, and sometimes incorrect.*

1. To make sure you understand this, find *two* example inputs to `uniques`: one in which partial tracing would lead to us thinking there's no error, and one in which partial tracing would lead us to find an error. Each input should have **three** sublists.

   Input that does NOT reveal an error:

   Expected output:

   Loop acculation table (fill it in column by column, and verify that the final accumulator value matches the expected output)

`sublist`	`uniques(sublist)`	`result_so_far`
N/A	N/A	`[]`

   Input that DOES reveal an error:

   Expected output:

   Loop acculation table (fill it in column by column, and verify that the final accumulator value doesn't match the expected output)

`sublist`	`uniques(sublist)`	`result_so_far`
N/A	N/A	`[]`

2. What you've provided above is a *counterexample* that shows that this recursive step is incorrect. This is a good start, but we'd like to go deeper. Describe in English *why* the recursive step is incorrect, i.e., what the problem with the code is.

   (Comment: we aren't asking you to fix the implementation of this method here, but you can do so as a good homework exercise.)

4.2 Mutating a nested list

In this exercise, we're going to look at a more complex form of recursion on nested lists involving *mutation.*

The running example we'll use for this worksheet is the following function:

```python
def add_one(nested_list: Union[int, list]) -> None:
    """Add one to every number stored in nested_list.

    Do nothing if nested_list is an int.

    If nested_list is a list, *mutate* it to change the numbers stored.

    >>> lst0 = 1
    >>> add_one(lst0)
    >>> lst0
    1
    >>> lst1 = []
    >>> add_one(lst1)
    >>> lst1
    []
    >>> lst2 = [1, [2, 3], [[[5]]]]
    >>> add_one(lst2)
    >>> lst2
    [2, [3, 4], [[[6]]]]
    """
    # if isinstance(nested_list, int):
    #     ...
    # else:
    #     for sublist in nested_list:
    #         ... add_one(sublist) ...
```

1. To start, think about the *base case* for this function. Implement it in the space below.

   Hint: read the docstring carefully—it tells you exactly what to do.

```
if isinstance(nested_list, int):
```

2. Now for the recursive step. The limitation of the standard `for sublist in nested_list` loop is that while it can mutate individual sublists of `nested_list` that are lists, it can't modify any *integer* element of `nested_list` directly.

   To make sure you understand this, suppose `nested_list = [1, 2, 3]`, and we run the following code on it:

```
else:
    # nested_list = [1, 2, 3]
    for sublist in nested_list:
        sublist = sublist + 1
```

   What would be the result of executing this code?

   Hint: draw a memory model diagram to trace the function.

3. In general, if we want to mutate elements of a list, we loop over the *indexes* of the list rather than its elements directly:

```
for i in range(0, len(nested_list)):
```

Using this idea, implement add_one in the space below.

Hint: you'll need different cases in your loop for when obj[i] is an integer or a list.

```python
def add_one(nested_list: Union[int, list]) -> None:
    """Add one to every number stored in nested_list.

    Do nothing if nested_list is an int.

    If nested_list is a list, *mutate* it to change the numbers stored.
    """
    if isinstance(nested_list, int):
        # Write your answer to Question 1 here.
    else:
        for i in range(0, len(nested_list)):
```