Changmiao Yu 24/2129

CSC148 - Partial Tracing Recursive Functions

On this worksheet, you'll practice the *partial tracing* technique for recursive functions covered in this week's prep.

1. Here is a partial implementation of a nested list function that returns a brand-new list.

```
def flatten(obj: int | list) -> list[int]:
    """Return a (non-nested) list of the integers in nested list <obj>.

    The integers are returned in the left-to-right order they appear
    in <obj>.

    >>> flatten(6)
    [6]
    >>> flatten([1, [-2, 3], -4])
    [1, -2, 3, -4]
    >>> flatten([[[[[0]]], -1])
    [0, -1]
    """
    if isinstance(obj, int):
        # Base case omitted  [obj]   return [obj].
    else:
        result = []
        for sublist in obj:
            result.extend(flatten(sublist))   a recursive call (to the same function)
        return result
```

Our goal is to determine whether the recursive step 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 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!

Only use the docstring, no need to read code

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

result.

`sublist`	`flatten(sublist)`	Value of `s` at the *end* of the iteration
N/A	N/A	`[]` (initial value of `s`)
`[0, -1]`	`[0, -1]`	`[]+[0,-1]` = `[0, -1]`
`-2`	`[-2]`	`[0, -1, -2]`
`[[-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 code. Remember that every time you reach a recursive call, don't trace into it—use the value you calculated in the second column!

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

Of course it does

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

```
return [obj]
```

[0,-1] [0,-1] [0,-1]

=2 [-2] [0,-1,-2]

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

return [obj]

def nested-sum(obj: list | int) → int:

>>> nest_s([[1,2,3], 4, [5]])

15.

if isinstance(obj, int):
    return obj

else:
    s = 0
    for sublist in obj:
        s += nested_sum(sublist)
    return s

"confidence Table" for function sum

depth	example	returned answer	correct?	confident.
0	6	6	✓	☺
1	[3,5,9]	17	✓	☺
1	[]	0	✓	
2	[148, [1,2], [], [0]] (148, 3, 0, 10)	161	✓	

The sublists all have lower depth, all lower depth are in lower depth

(~~induction~~ induction)

CSC148 - Writing Recursive Functions

To review, here is the basic recursive code template for nested lists, which you got lots of practice on in this week's prep:

```
def f(obj: int | list) -> ...:
    if isinstance(obj, int):
        ...
    else:
        ...
        for sublist in obj:
            ... f(sublist) ...
```

Recall that we 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 list `[10, [[20]], [30, 40]]`:

1 3 2 2

- the depth of 10 is 1
- the depth of 20 is 3
- the depths of 30 and 40 are 2

For a nested list that is a single integer, e.g. 100, the depth of the integer is 0.
Your goal for this worksheet is to implement the following function:

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

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

1. *Base case example and implementation.*

depth

(a) Write *two* doctest examples where `obj` is an `int`, covering different possibilities for `d` that result in different return values. Make sure you understand why the return values differ in your two doctest examples before moving on.

```
>>> items_at_depth(47, 1)
[]
>>> items_at_depth(47, 0)
[47]
```

Base case:
>>> id(5, 0)
[5] ← integer 5 has depth 0
>>> id(5, 1]
[] ← input is integer which has depth 0, d>0 (max d)
So output is []

(b) Implement the base case of `items_at_depth`. It's okay to have a nested if statement.

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

~~if int==~~
if d==0:
return [obj]
else:
return []

Recursive Design Recipe.

1. Identify the recursive structure
   We have nested list, it has recursive structure.
   We'll "recurse over" it.

2. Base case
   a) Identify base case & write doctest
   >>> nested_list_contains(10, 10)
   True
   >>> nested_list_contains(0, 5)
   False.
   b) implement base case

3. Recursive case
   a) example.
   >>> nested_l([1,2,3],4,[[[5]]],11], 11)
   True
   b) write down recursive calls need to be made.
   nlc([1,2,3],5) => False
   nlc(4,5) => False
   nlc([[[5]]],5) => True, we can stop
   nlc(11,5)

```
def nested_list_contains(obj, item: int) -> bool:
    if isinstance(obj, int):
        return obj == item
    else:
        for sublist in obj:
            if nested_list_contains(sublist, item):
                return True
        return False
```

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

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

```
>>> items_at_depth(obj, 0)
[]
>>> items_at_depth(obj, 3)
[20, 30]
```

(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 (similar to what we did for `nested_list_contains`). Complete the table below.

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

Problem! The results of these recursive calls don't actually help calculate the correct return value of `items_at_depth(obj, 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, d-1)`
`[]`	`10`	`[]`
`[20]`	`[[20]]`	`[20]`
`[30]`	`[[30], 40]`	`[30]`

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

```
def items_at_depth(obj: int | list, d: int) -> list[int]:
    if isinstance(obj, int):
        ... # From above
    else:
        if d == 0:
            return []
        answer = []
        for sublist in obj:
            answer.extend(items_at_depth(sublist, d-1))
        return answer
```

prefer to not think in else, my the way.

```
if isinstance(obj, int):
    if int == 0:
        return [obj]
    else:
        return []
else:
    list_depth_d = []
    for sublist in obj:
        list_depth_d.extend(iad(sublist, d-1))
    return list_depth_d
```

4. If you followed our nested list recursive code template, you likely 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. If you don't have time to do this in class, try doing it for homework!