CSC148 Worksheet 16 Solution

Hyungmo Gu

April 26, 2020

Question 1

a. The doctests for the base case is

```
    """
    >>> nested_list_contains(1,1)
    True
    >>> nested_list_contains(1,2)
    False
    """

```

Using this fact, we can write

```
    from typing import Union, List

    def nested_list_contains(obj: Union[int, List], item: int) -> bool:
        """Return whether the given item appears in <obj>.
        Note that if <obj> is an integer, this function checks whether
        <item> is equal to <obj>.

        >>> nested_list_contains(1,1)
        True
        >>> nested_list_contains(1,2)
        False
        """

        if isinstance(self, int):
            return obj == item

```

Listing 1: worksheet_16_q1a_solution

b. Consider the following doctest

```
    """
    >>> nested_list_contains([4,2,2,[6,5,7,[8]]],8)
    True
    """

```

Using the base case from question 1.a, and the basic recursive design recipe, we can conclude the algorithm will behave as follows

1) $4 \rightarrow 4 == item? \rightarrow$ False

2) $2 \rightarrow 2 == item? \rightarrow$ False

3) $2 \rightarrow$ False

4)[6,5,7,[8]] $\rightarrow$ Recursion

    5) $6 \rightarrow 6 == item? \rightarrow$ False

    6) $5 \rightarrow 5 == item? \rightarrow$ False

    7) $7 \rightarrow 7 == item? \rightarrow$ False

    8) [8] $\rightarrow$ Recursion

        9) $8 \rightarrow 8 == item? \rightarrow$ True (function terminates)

11) Function Terminates until the end of recursion

Now, no new parameters other than *obj* and *item* are required, since

1. for the traversing and checking of elements, they are done using the two parameters.
2. for bringing the value 'True' to user, it is done by repeatedly ending the recursive function call early with the value
3. for brining the value 'False' to user, it is done by returning False at the end.

```
c1  from typing import Union, List
2
3
4   def nested_list_contains(obj: Union[int, List], item: int) -> bool:
5       """Return whether the given item appears in <obj>.
6       Note that if <obj> is an integer, this function checks whether
7       <item> is equal to <obj>.
8
9       >>> nested_list_contains([4,2,2,[6,5,7,[8]]],8)
10      True
11      >>> nested_list_contains([4,2,2,[6,5,7,[8]]],9)
12      False
13      """
14
15      if isinstance(self, int):
16          return obj == item
17      else:
18          # ==================== (Solution) ======================
19          for sublist in obj:
20              result = nested_list_contains(sublist, item)
21
22              if result:
```

```
                    return result

            return False
            # ========================================================
```

Listing 2: worksheet_16_q1c_solution

Question 2

a. The doctests for the base case is

```
    """
    >>> first_at_depth(1,2)
    None
    """
```

Using this fact, we can write

```
    from typing import Union, List, Optional


    def first_at_depth(obj: Union[int, List], d: int) -> Optional[int]:
        """Return the first (leftmost) item in <obj> at depth <d>.
        Return None if there is no item at depth <d>.
        Precondition: d >= 0.

        >>> first_at_depth(1,2)
        None
        """

        if isinstance(obj, int):
            return None
```

Listing 3: worksheet_16_q2a_solution

b. First, we need to write doctests for the function call on input of some complexity.

Consider the following doctests.

```
    """
    >>> first_at_depth([1,2,[3,4,[5,6]]],1)
    1
    >>> first_at_depth([1,2,[3,4,[5,6]]],3)
    5
    >>> first_at_depth([1,2,[3,4,[5,6]]],4)
    None
    >>> first_at_depth([[1,2,[3]],4,[[5],6]],3)
    3
    """
```

Second, we need to write down the relevant recursive calls for each sub-nested-list of input.

The recursive calls for the example *first_at_depth([1,2,[3,4,[5,6]]],3)* is as follows

1) 1 → is current_depth == 3? → False → return None
2) 2 → is current_depth == 3? → False → return None
3) [3,4,[5,6]] → Recursion

    4) 3 → is current_depth == 3? → False → return None
    5) 4 → is current_depth == 3? → False → return None
    6) [5,6] → Recursion

        7) 5 → is current_depth == 3? → True → return 5

8) Function Terminates with return value 5 until the end of recursion

Finally, we need to think about the extra parameters for each function.

It follows from above that the extra parameter required for this function is *current_depth*. To add this parameter, we need to change function *first_at_depth* from

```
    first_at_depth(obj: Union[int, List], d: int) -> Optional[int]:
```

to

```
    first_at_depth(obj: Union[int, List], d: int, current_depth: int) ->
        Optional[int]:
```

Rough Work:

1. Write a doctest for the function call on input of some complexity

   First, we need to write a doctest for the function call on input of some complexity

First, we need to write doctests for the function call on input of some complexity.

Consider the following doctests.

```
"""
>>> first_at_depth([1,2,[3,4,[5,6]]],1)
1
>>> first_at_depth([1,2,[3,4,[5,6]]],3)
5
>>> first_at_depth([1,2,[3,4,[5,6]]],4)
None
>>> first_at_depth([[1,2,[3]],4,[[5],6]],3)
3
"""

```

2. Write down the relevant recursive calls for each sub-nested-list of input.

   Second, we need to write down the relevant recursive calls for each sub-nested-list of input.

Second, we need to write down the relevant recursive calls for each sub-nested-list of input.

The recursive calls for the example *first_at_depth([1,2,[3,4,[5,6]]],3)* is as follows

1) 1 → is current_depth == 3? → False → return None
2) 2 → is current_depth == 3? → False → return None
3) [3,4,[5,6]] → Recursion

    4) 3 → is current_depth == 3? → False → return None
    5) 4 → is current_depth == 3? → False → return None
    6) [5,6] → Recursion

        7) 5 → is current_depth == 3? → True → return 5

8) Function Terminates with return value 5 until the end of recursion

3. Think about the extra parameters for each function.

   Finally, we need to think about the extra parameters for each function.

Finally, we need to think about the extra parameters for each function.

It follows from above that the extra parameter required for this function is *current_depth*. To add this parameter, we need to change function *first_at_depth* from

```
first_at_depth(obj: Union[int, List], d: int) ->
    Optional[int]:
```

to

```
first_at_depth(obj: Union[int, List], d: int,
    current_depth: int) -> Optional[int]:
```

First, we need to write doctests for the function call on input of some complexity.

Consider the following doctests.

```
        """
        >>> first_at_depth([1,2,[3,4,[5,6]]],1)
        1
        >>> first_at_depth([1,2,[3,4,[5,6]]],3)
        5
        >>> first_at_depth([1,2,[3,4,[5,6]]],4)
        None
        >>> first_at_depth([[1,2,[3]],4,[[5],6]],3)
        3
        """

```

Second, we need to write down the relevant recursive calls for each sub-nested-list of input.

The recursive calls for the example *first_at_depth([1,2,[3,4,[5,6]]],3)* is as follows

1) 1 → is current_depth == 3? → False → return None
2) 2 → is current_depth == 3? → False → return None
3) [3,4,[5,6]] → Recursion

    4) 3 → is current_depth == 3? → False → return None
    5) 4 → is current_depth == 3? → False → return None
    6) [5,6] → Recursion

        7) 5 → is current_depth == 3? → True → return 5

8) Function Terminates with return value 5 until the end of recursion

Finally, we need to think about the extra parameters for each function.

It follows from above that the extra parameter required for this function is *current_depth*. To add this parameter, we need to change function *first_at_depth* from

```
        first_at_depth(obj: Union[int, List], d: int) -> Optional
            [int]:
```

to

```
        first_at_depth(obj: Union[int, List], d: int,
            current_depth: int) -> Optional[int]:
```