W5.0.1 Additional Reading (Nested Lists & Iteration)

Acknowledgement / license

Lists with Complex Items

The lists we have seen so far have had numbers or strings as items. We've sneaked in a few more complex items, but without ever explicitly discussing what it meant to have more complex items.

In fact, the items in a list can be any type of Python object. For example, we can have a list of lists.

```
nested1 = [['a', 'b', 'c'],['d', 'e'],['f', 'g', 'h']]
print(nested1[0])
print(len(nested1))
nested1.append(['i'])
print("-------")
for L in nested1:
    print(L)
```

Line 2 prints out the first item from the list that `nested1` is bound to. That item is itself a list, so it prints out with square brackets. `nested1` has length 3, which prints out on line 3. Line 4 adds a new item to `nested1`. It is a list with one element, `'i'`, it's not just the string `'i'`.

Python Tutor gives a you a reference diagram, a visual display of the contents of `nested1`.

When you get to step 4 of the execution, take a look at the object that variable `nested1` points to. It is a list of three items, numbered 0, 1, and 2. Each item is shown in the figure as a pointer to another separate list, e.g. the item in slot 2 is the list `['f', 'g', 'h']`. When you execute line 4 you will see an extra slot appearing in `nested1` with an arrow pointing to the new 1-element list `['i']`.

With a nested list, you can make complex expressions to get or set a value in a sub-list.

```
nested1 = [['a', 'b', 'c'],['d', 'e'],['f', 'g', 'h']]
y = nested1[1]
print(y)
print(y[0])

print([10, 20, 30][1])
print(nested1[1][0])
```

Lines 1–4 above probably look pretty natural to you. Line 6 is just a reminder that you index into a literal list, one that is written out, the same way as you can index into a list referred to by a variable. `[10, 20, 30]` creates a list. `[1]` indexes into that list, pulling out the second item, 20.

Just as with a function call where the return value can be thought of as replacing the text of the function call in an expression, you can evaluate an expression like that in line 7 from left to right. Because the value of `nested1[1]` is the list `['d', 'e']`, `nested1[1][0]` is the same as `['d', 'e'][0]`. So line 7 is equivalent to lines 2 and 4; it is a simpler way of pulling out the first item from the second list.

At first, expressions like that on line 7 may look foreign. They will soon feel more natural, and you will end up using them a lot. Once you are comfortable with them, the only time you will write code like lines 2–4 is when you aren't quite sure what your data's structure is, and so you need to incrementally write and debug your code. Often, you will start by writing code like lines 2–4, then, once you're sure it's working, replace it with something like line 7.

You can change values in nested lists in the usual ways. You can even use complex expressions to change values. Consider the following

Coding Challenge 1

Below, we have provided a list of lists. Use indexing to assign the element 'horse' to the variable name `idx1` then print the value of `idx1`.

Coding Challenge 2

Using indexing, retrieve the string 'willow' from the list and assign that to the variable `plant`. Then print the `plant` variable.

Nested Iteration

When you have nested data structures, especially lists and/or dictionaries, you will frequently need nested for loops to traverse them.

```
nested1 = [['a', 'b', 'c'],['d', 'e'],['f', 'g', 'h']]
for x in nested1:
    print("level1: ")
    for y in x:
        print("     level2: " + y)
```

Line 3 executes once for each top-level list, three times in all. With each sub-list, line 5 executes once for each item in the sub-list. Try stepping through it in Python Tutor to make sure you understand what the nested iteration does.

Structuring Nested Data

When constructing your own nested data, it is a good idea to keep the structure consistent across each level. For example, if you have a list of dictionaries, then each dictionary should have the same structure, meaning the same keys and the same type of value associated with a particular key in all the dictionaries. The reason for this is because any deviation in the structure that is used will require extra code to handle those special cases. The more the structure deviates, the more you will have to use special cases.

For example, let's reconsider this nested iteration, but suppose not all the items in the outer list are lists.

```
nested1 = [1, 2, ['a', 'b', 'c'],['d', 'e'],['f', 'g', 'h']]
for x in nested1:
    print("level1: ")
    for y in x:
        print(f"     level2: {y}")
```

Now the nested iteration fails.

We can solve this with special casing, a conditional that checks the type.

```
nested1 = [1, 2, ['a', 'b', 'c'],['d', 'e'],['f', 'g', 'h']]
for x in nested1:
    print("level1: ")
    if type(x) is list:
        for y in x:
            print(f"     level2: {y}")
    else:
        print(x)
```

You can imagine how many special case if-thens we'd need, and how complicated the code would get, if we had many layers of nesting but not always a consistent structure.