Advanced Data Collections (n-dimensional collections)

Introduction

Welcome to the final lecture in QuickPass: Advanced Data Collections! In this lecture, we are going to explore multi-dimensional data collections, mainly n-dimensional lists and how to access and manipulate the data they contain

Table of Contents

Data Collection Review (Lists)

- A list is a collection of values that can be of different types (integers, strings, etc.).
- Lists are ordered and indexed, which means each element has a position and can be accessed with bracket notation `[]`
- Indexes start at `[0]` when going forward in a list, and start at `[-1]` when going backwards.

In [ ]:
```
my_list = [1,2,3,4,5]
print(my_list[0]) # Returns 1
print(my_list[-1]) # Returns 5
```

```
1
5
```

Multi-Dimensional Data Structures

2-Dimensional Lists

A 2-dimensional list is a list of lists. It's like having rows and columns of data. Nested lists still use bracket notation to access data, but we have to add more indexes to access single data points. We still use bracket notation, but just more of them to go through the dimensions. Take a look at the example below:

In [ ]:
```
# neatly formatted for a visual to see rows and columns
my_matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
```

In `my_matrix`, a single bracket index will return an entire list.

In [ ]:
```
my_matrix[-1] # returns [7, 8, 9]
```

Out[ ]:
```
[7, 8, 9]
```

To access a single data point, we need to add another bracket with the index of the data point we desire (again, starting at `0` for forward indexes).

To access the `8`, we would need to enter `my_matrix[2][1]`

In [ ]:
```
my_matrix[2][1]
```

Out[ ]:
```
8
```

Also, because lists in Python are mutable, we can change values in-place using the assignment operator, adding brackets as we increase dimensions. Lets change the `5` to a `500`.

In [ ]:
```
my_matrix[1][1] = 500
my_matrix
```

Out[ ]:
```
[[1, 2, 3], [4, 500, 6], [7, 8, 9]]
```

As we know, lists are iterable which means that we can loop over them. How can we loop over a 2-dimensional list? Just like with indexing, we can add a layer to our loops by *nesting* a `for` loop inside of a `for` loop.

In [ ]:
```
for row in my_matrix:
    for item in row:
        print(item)
```

```
1
2
3
4
500
6
7
8
9
```

3-Dimensional Lists

Going up in dimensions, we just add another layer to our data. At this point, keeping track of the dimensions in this format can be a bit daunting. See the example below:

In [ ]:
```
my_three_dim_list = [[[1,2,3], [4,5,6], [7,8,9]], [[10,11,12], [13,14,15], [16,17,18]], [[19,20,21], [22,23,24], [25,26,27]]]  # this example is limited to one line and
```

How would you return the value `24`? In this case, we would just need to add more brackets to our bracket notation.

In [ ]:
```
my_three_dim_list[2][1][2]
```

Out[ ]:
```
24
```

How could we loop through this data and print out what we need? We just need to add another layer to our nesting!

Think of the structure as a Rubik's Cube. There are 3 layers, each containing rows and columns.

1. The outer loop is going to iterate over the layers one by one. It will run 3 times.
2. For each iteration of the outer loop, the middle loop iterates over every row of the layer. It will run a total of 9 times
3. The innermost loop iterates over every item for each iteration of the middle loop. It will run a total of 27 times

In [ ]:
```
for layer in my_three_dim_list:
    for row in layer:
        for item in row:
            print(item)
```

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
```

Adding and Removing Elements

You can add elements using `append()`, `extend()`, or `insert()`. You can remove elements using `remove()`, `pop()`, or `del`.

In [ ]:
```
my_matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
```

In [ ]:
```
my_matrix.append([400, 500, 600])
my_matrix
```

Out[ ]:
```
[[1, 2, 3], [4, 5, 6], [7, 8, 9], [400, 500, 600]]
```

Replacing Rows and Columns

To replace a row, simply assign a new list to the row index. To replace a column, use a `for` loop to iterate through rows and change elements.

In [ ]:
```
my_matrix[-1] = [10, 11, 12]
my_matrix
```

Out[ ]:
```
[[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]
```

In [ ]:
```
for i in range(len(my_matrix)):
    for index, item in enumerate(my_matrix[i]):
        if index == 1:
            my_matrix[i][index] = item * 100
my_matrix
```

Out[ ]:
```
[[1, 200, 3], [4, 500, 6], [7, 800, 9], [10, 1100, 12]]
```

Other Nested Data Structures

While we typically represent n-dimensional data with nested lists (also called nested arrays), we can nest other data structures into each other depending on the use case. Would it be useful to nest a list as a value in a dictionary?

In [ ]:
```
english_french_dictionary = {"hello": "bonjour"}

english_french_dictionary['hello'] = ['bonjour', 'salut']
english_french_dictionary
```

Out[ ]:
```
{'hello': ['bonjour', 'salut']}
```

The school has a data structure that contains all personnel associated with students (Teachers, parents, and students). Having access to only the parent info, how could you find out their childrens' names?

In [ ]:
```
school_dictionary = {
    "person_12345" : {
        'first_name': "Jim",
        'last_name': 'Smith',
        'suffix': 'Sr.',
        'student': False,
        'teacher': False,
        'email': 'jsmith@gmail.com',
        'class':[],
        'children': ['person_94523', 'person_23456', 'person_34567'],
        'phone': '452-672-6777',
        'guardian': True,
    },
    'person_94523': {
        'first_name': "Jimmy",
        'last_name': 'Smith',
        'suffix': 'Jr.',
        'student': True,
        'teacher': False,
        'email': 'N/A',
        'class':[],
        'children': 'N/A',
        'phone': 'N/A',
        'guardian': False,
    },
    'person_23456': {
        'first_name': "Kelly",
        'last_name': 'Smith',
        'suffix': 'N/A',
        'student': True,
        'teacher': False,
        'email': 'N/A',
        'class':[],
        'children': 'N/A',
        'phone': 'N/A',
        'guardian': False,
    },
    'person_87554': {
        'first_name': "Jorge",
        'last_name': 'Gonzalez',
        'suffix': 'N/A',
        'student': True,
        'teacher': False,
        'email': 'N/A',
        'class':[],
        'children': 'N/A',
        'phone': 'N/A',
        'guardian': False,
    },
    'person_34567': {
        'first_name': "Robert",
        'last_name': 'Smith',
        'suffix': 'N/A',
        'student': True,
        'teacher': "person_56566",
        'email': 'N/A',
        'class':[],
        'children': 'N/A',
        'phone': 'N/A',
        'guardian': False,
    },
    'person_56566': {
        'first_name': "Gertrude",
        'last_name': 'Brown',
        'suffix': '',
        'student': False,
        'teacher': True,
        'email': 'N/A',
        'class':['person_34567', 'person_95959', 'person_67234', 'person_94111', 'person_87554'],
        'children': 'N/A',
        'phone': '567-654-6543',
        'guardian': False,
    }
}
```

In [ ]:
```
names = []

for child in school_dictionary['person_12345']['children']:
    names.append(f"{school_dictionary[child]['first_name']} {school_dictionary[child]['last_name']}")

print(names)
```

```
['Jimmy Smith', 'Kelly Smith', 'Robert Smith']
```

Code Examples and Practice

Modifying a Seating Chart

Let's say you have a seating chart for a theater, and you need to update a reserved seat.

```
seating_chart = [
    ["A", "B", "C", "D"],
    ["E", "F", "G", "H"],
    ["I", "J", "K", "L"]
]
```

```
# Someone reserved seat "F" (row 2, seat 2). Let's update it.
seating_chart[1][1] = "X"
```

Adding Rows and Columns

You have a sales dataset, and you want to add a new salesperson's data.

In [ ]:
```
sales_data = [
    ["Name", "Jan", "Feb", "Mar"],
    ["Alice", 100, 120, 90],
    ["Bob", 80, 85, 90]
]

# Adding a new row for "Charlie."
new_salesperson = ["Charlie", 110, 95, 105]
sales_data.append(new_salesperson)

# Adding a new column for "Apr."
for i in range(len(sales_data)):
    sales_data[i].append(0)  # Initial value for April.

sales_data
```

Out[ ]:
```
[['Name', 'Jan', 'Feb', 'Mar', 0],
 ['Alice', 100, 120, 90, 0],
 ['Bob', 80, 85, 90, 0],
 ['Charlie', 110, 95, 105, 0]]
```

Practical Applications of n-Dimensional Lists

- 2-dimensional lists are essential for organizing data tables, images, games, and more.
- 3-dimensional data structures are used in several geospatial, data science, and machine-learning applications

Understanding how to work with nested data is an important skill to develop as you learn programming fundamentals. Luckily, there are python libraries that are designed to take the complications of n-dimensional data structures and provide more efficient ways to manipulate the data and save processing energy. A couple examples of those libraries are:

- Numpy - Specialized data analysis library for working with n-dimensional data
- Pandas - Python library used for working with data sets.

Conclusion

In this lecture, you've explored the fundamentals of working with multi-dimensional data structures in Python. Practice is the key to mastering this skill, so continue to work with these and all other python data structures to grow your knowledge.