Async Presentation Notes Unit 3

Geon's Content (3.1.1 - 3.5.1):

3.2.1 Sequences

- Last week we learned about ints floats and booleans. Now we learn about combining these all together into something called "composite types."
- Sequence = ordered arrangement of items one after another. List, tuples, arrays, etc. Can be strings, integers, floats, etc.
- Lists are the most common
    - Can create a list by using square brackets with elements separated by commas [1, 2, 3, 4]
    - You can save it to a variable new_list = [1,2,3,4] second_list = [5, 6, 7]
    - First_list + second_list adds the list together [1, 2, 3, 4, 5, 6, 7]
    - Remember that python starts base 0
    - Remember slice command third_list[1::3] grabs every 3 starting from second
    - Commands
        - 3 in new_list -> true
        - len(third_list) = number of lists
        - max(new_list) = largest number
        - Min = lowest number
        - my_list.index(3) = shows where first occurrence of "3" is
        - My_list.count(4) = how many times "4" shows up

3.2.2 Sequences Drill

Opportunity to try out your skills!

3.3.1 Lists

- List commands
    - x.append(5) = adds a 5 to the end of a list
    - X[0] = 10 = swaps the value of the index
    - x.extend([12,13]) = extends a list with multiple elements
    - Del x[0] = deletes that index
    - x.clear() = deletes the entire list
    - x.insert(0,12) = inserts a element to a list and then shifts the rest to the right
    - Last_value = x.pop() = Takes value (default to last without a index) from a list
    - x.remove(500) = removes the first occurrence of the typed in element
    - x.sort() x.reverse() = sorts ascending and descending

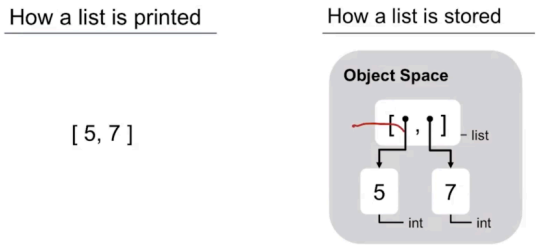
3.3.2 Lists Drill

Opportunity try out your skills!

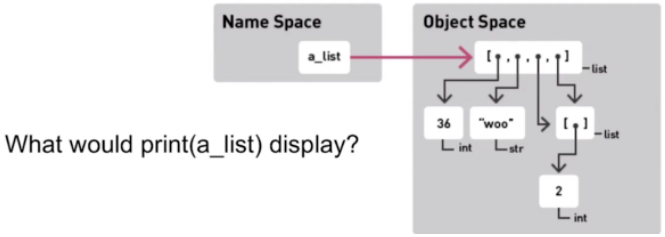
3.4.1 Lists and Mutability

- Two main ideas

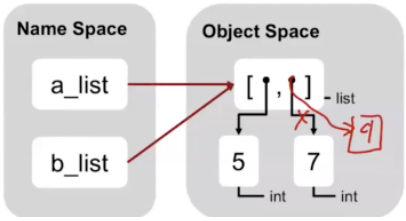
- **Objects can be reached in multiple ways.**
  - A list is stored in memory with pathways to stored elements rather than as seen side by side.



  - A list can even have multiple spots referring to the same object



    - [36, “woo”, [2], [2]]
- **Mutating an object affects all variables with paths to that object**
  - One change can affect multiple saved variables.



3.5.1 Mutability, Part I

- Mutability needs to be paid attention to. A = sign simply signifies pointing.
  -
```
2.7 Mutability.py
2.7.1_Mutabilty_Example.pptx
2.8 - Tuples.ipynb
2.9 - Ranges.ipynb
Old version of 2.5 More about Mutability.ipynb
name_game.py
Pauls-MacBook-Pro:week_2 paul$ python3 "2.7 Mutability.py"
first_list now has the value [1000, 2, 3, 4, 5]
```
  -
```
first_list = [1,2,3,4,5]
second_list = first_list
second_list[0] = 1000
print("first_list now has the value",first_list)
```
  - Being mutable means that the base list can be changed by affecting a higher level list.
-
```
a_list = ["tu", "tu"]
b_list = [a_list]
c_list = [a_list]
b_list[0][1] = "ba"
print("c_list now has the value", c_list)
```
  - Start with b_list. Since we are setting b_list = [a_list], then b_list = [[a_list]]
    - If were to append 0 to b_list, then it would be [[a_list], 0]
  - B_list[0] is pointing to the first element in it’s list, [a_list] or [“tu”, “tu”]

    - Therefore, B_list[0][1] is finding the 2nd index from a_list. Replacing with “ba” creating a_list = [“tu”, “ba”]
    - Finally c_list outputs [a_list] which is [“tu”, “ba”] but inside a list, so it is **[[“tu”, “ba”]]**
- If lists were immutable, the pathing would not cause changes.

Will’s Content (3.7.1 - 3.11.1):

3.7.1 Mutability, Part II

- **Mutable Objects** (e.g. some “container objects” like lists, dictionaries, sets, etc.) can be changed in-place (elements can be added, removed, or altered) without creating a new object in memory.
- This submodule focuses specifically on the mutability properties of *lists*: a mutable object that holds references to other objects — the **memory addresses** of the objects it points to – rather than the objects themselves.
- When you modify a list, such as appending an item or changing an element, it affects the memory reference to that list and any other references to the same list. This means that if you have nested lists, even making an ordinary .copy() of the parent list might not be enough to avoid accidentally editing the child list within your original nested list. In the below code, for example:

```
a_list = [1, 2,["Bill","Kay"]]
b_list = a_list.copy()
a_list[2].append("Paul") # makes a_list = [1, 2,["Bill","Kay", “Paul”]]
print("b_list now has the value", b_list)

# b_list is identical to a_list, because their child list reference points to the same memory address
```

- The deepcopy() function copies *all* levels of a nested list, fully separating the two objects in memory

3.8.1 Tuples

- This was pronounced as “Tuh-ples” by the video’s instructor, but some guy wrote on Reddit that “you should say it exactly the way your boss says it”
- Tuples are effectively just **immutable** lists, otherwise they are the same: ordered sequences of objects
- Because of their immutability, you’ll get an error like this if you attempt to change their contents

```
x = (10,11)
x[0] = 100
```

⊗ 0.0s Python

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call l
Cell In[6], line 2
      1 x = (10,11)
----> 2 x[0] = 100

TypeError: 'tuple' object does not support item assignment
```

- Syntax differences:
  - a_list = [1,2,3]
  - a_tuple = (1,2,3)
  - You can easily typecast between the two using the constructors tuple(a_list) / list(a_tuple)

Takeaway: Use mutable objects like lists when you want to dynamically add to, subtract from, or update the object as your application runs; use immutable objects like Tuples when you want stronger data integrity, memory efficiency, and faster application performance. Beware that nesting mutable objects can lead to surprising results

3.9.1 Ranges

- Created by range(*start [inclusive & defaults to zero], stop [exclusive], step [defaults to 1*)
- Produces a **list**
- Only accepts integer inputs:

```
x=range(10,100,0.1)
print (x)

x=range(10,100,2)
print (x)

x=range(10,100,2)
print (x)
```

⊗ 0.0s Python

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[15], line 1
----> 1 x=range(10,100,0.1)
      2 print (x)
      4 x=range(10,100,2)

TypeError: 'float' object cannot be interpreted as an integer
```

3.10.1 Dictionaries

- The main purpose of a dictionary is to map **keys** to **values**. Each key is unique, and it maps to a specific value.
- They are constructed with the syntax dict_name = {“key”: 1, “key_2”: “str”, …} or dict_name = dict(a=1,b=”str”,...). Note that keys don’t have to be strings; they can be any immutable object. Values can be anything.
- Unlike tuples, lists, etc., they are not ordered
- You can access specific dictionary values using dict_name[‘key’]
- You can retrieve all keys, values, or both using the .keys(), .values(), and .items() methods.
- Nested dictionaries have the same issue in which you need to deepcopy the dictionary to keep *all* of the copied version separate in memory.
- If you’ve ever seen a JSON file, these are very similar in structure.
  The differences:
  - JSON is a data transmission format, a dictionary is a data structure
  - JSON keys must be strings and are always enclosed in double quotes, whereas dictionary keys can be any immutable type and strings can be wrapped in single or double quotes
  - JSON supports a smaller set of data types compared to Python dictionaries. **JSON only supports strings, numbers, booleans, null, arrays, and objects**. Python dictionaries can contain any valid Python object as values.
  - Example JSON file:

```
{
  "string": "Hello, world!",
  "number": 42,
  "float": 3.14,
  "boolean": true,
  "null": null,
  "array": [1, 2, 3, "four", false],
  "object": {
    "nested": "value"
  }
}
```

  - Example Python Dictionary:

```
{
    "string": "Hello, world!",
    "number": 42,
    "float": 3.14,
    "boolean": True,
    "none": None,
    "list": [1, 2, 3, "four", False],
    "dict": {
        "nested": "value"
    },
    "tuple": (1, 2, 3),
    "set": {1, 2, 3},
    "complex": 1+2j,
    "bytes": b"binary data",
    "function": lambda x: x*2
}
```

3.11.1 Nested Lists and Dictionaries

- Nested dictionaries are extremely common. In many cases this is because JSONs have a **wrapper** or **envelope** at the top-level that shares metadata about the transmission. For example, the JSON file:

```
{
  "status": "success",
  "timestamp": "2024-09-12T10:00:00Z",
  "data": {
    "name": "John",
    "age": 30,
    "city": "New York"
  }
}
```

After running the method dict_name = json_loads(json_string) — or dict_name = response.json() if you got it from an API — using the **requests** module…

Becomes the dictionary:

```
{
  'status': 'success',
  'timestamp': '2024-09-12T10:00:00Z',
  'data': {
    'name': 'John',
    'age': 30,
    'city': 'New York'
  }
}
```

- You can access these nested value ‘age’ in the above example using the syntax Dict_name[‘data’][‘age’]

- Similarly, with lists, eg:

```
animals = [['Tiger', 3,15.0],
           ['Lion',7,8.91],
            'Camel']
```

  You can access the int 7 with the syntax animals[1][1].

  Beware though that since a string is also a type of sequence,
  If you were to run animals[1][0], the operation would return ‘C’

- In a hybrid list-dictionary scenario, you may need to use something like [1][‘age’]

Generalizable rule:

- 1) Check the type of the item you are accessing
- 2) If it's dict type:
  - Use the .keys() to look up the dictionary keys
  - Check access to one of the keys and start again
- 3) If it's a list type:
  - You can print what is in the list
  - Also check the length of the list
  - Check an item in the list and start again with #1
- Repeat the steps as necessary for each new item until you get to the end!