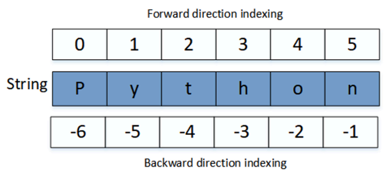
Python Ordered Data Structures

Introduction

Welcome to today's lecture on ordered data structures in Python (Lists and Tuples). Lists and tuples are fundamental data structures used for storing collections of items. In this lesson, we'll explore the capabilities and limitations of ordered data structures, understand the concepts of mutability and immutability, and delve into the usage of various built-in methods.

Table of Contents

Ordered data structures, also called *sequences*, are ordered grouping of elements, each with their own index, which is a number that marks their place in the sequence. Just like you learned with Strings (which are Python sequences), this indexing starts at 0.



Lists

Lists are containers that hold objects in a given order. Lists are *mutable*, meaning they allow you to add, remove, or overwrite individual elements inside them. In Python, lists are wrapped in square brackets `[]`. Unlike some languages, Python lists can contain mixed data types.

Creating Lists

To initialize or create an empty list, Python has a built-in method `list()`, or you can use empty brackets.

```
In [ ]: my_list = list()
        my_list
```

```
Out[ ]: []
```

- Note: "list" is a Python built-in method name. Like reserved words, they have a special use case right out of the box and are usually presented with a special text formatting in an IDE. Unlike reserved words, they **CAN BE OVERWRITTEN**. If you were to name a new list with just the name "list", you will overwrite the built-in method.

  ```
  list = [1,2,3,4,5]
  ```

  ```
  my_new_list = list()  # Will result in an error.
  ```
  If you find that you have written over a built-in Python function, you will need to restart the Python kernel.

```
In [ ]: my_other_list = []
        my_other_list
```

```
Out[ ]: []
```

You can also create a list with data, or elements, already inside. You just need to wrap the items with the brackets and separate each item with a comma.

```
In [ ]: num_list = [0,1,2,3,4,5,6,7,8,9]
        num_list
```

```
Out[ ]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
In [ ]: str_list = ['I', 'love', 'Python']
        str_list
```

```
Out[ ]: ['I', 'love', 'Python']
```

Lists can contain other collections, such as lists. Lists stored inside of other lists are considered *nested lists*.

```
In [ ]: list_with_lists = ['a', 'b', 'c', ['d', 'e', 'f']]
        list_with_lists
```

```
Out[ ]: ['a', 'b', 'c', ['d', 'e', 'f']]
```

List Operations and Methods

There are a number of methods (i.e. functions) that can be applied to lists.

Let's initialize a list of our "bucket list" travel locations and explore the various list operations and methods available to us.

```
In [ ]: travel_bucket_list = ['New Zealand', "Antarctica", "Italy", "Chile", "Fiji", "Ireland", "Morocco"]
```

Like accessing individual elements in a string, you can add the square brackets after the list name to access individual elements in a list. Unlike strings, however, you can also use an assignment operator (`=`) to overwrite the element at that location with a new value.

Syntax	Description
`list_variable[n]`	Accesses the item at index `[n]`
`list_variable[n] = x`	Sets the item at index `[n]` to `x`

Related to grabbing a single index, you can also slice python lists like youdid with strings.

Syntax	Description
`list_variable[start:stop]`	Accesses the sub-list from `start` index up to but not including `stop`

```
In [ ]: travel_bucket_list[4] # grabbing a single index
```

```
Out[ ]: 'Fiji'
```

```
In [ ]: travel_bucket_list[2:4] # slicing a string
```

```
Out[ ]: ['Italy', 'Chile']
```

```
In [ ]: travel_bucket_list[4] = "Texas"
        travel_bucket_list
```

```
Out[ ]: ['New Zealand', 'Antarctica', 'Italy', 'Chile', 'Texas', 'Ireland', 'Morocco']
```

List Operations

List operations include checking the length of a list, checking membership (results in a boolean), or using loops to iterate and perform operations on each element.

```
In [ ]: len(travel_bucket_list)
```

```
Out[ ]: 7
```

```
In [ ]: print("Fiji" in travel_bucket_list)
        print("Texas" in travel_bucket_list)
```

```
False
True
```

List Methods:

Method	Description
`list.append(x)`	Add an item to the end of the list.
`list.extend(iterable)`	Extend the list by appending elements from the iterable.
`list.insert(i, x)`	Insert an item at a given position.
`list.clear()`	Remove all items from the list.
`list.index(x)`	Return the index of the first item with a specific value.
`list.count(x)`	Return the number of times a specific value appears in the list.
`list.sort()`	Sort the items of the list in place.
`list.reverse()`	Reverse the elements of the list in place.
`list.copy()`	Return a shallow copy of the list.

```
In [ ]: travel_bucket_list.append("Germany")
        travel_bucket_list
```

```
Out[ ]: ['New Zealand',
         'Antarctica',
         'Italy',
         'Chile',
         'Texas',
         'Ireland',
         'Morocco',
         'Germany']
```

```
In [ ]: travel_bucket_list.extend(["Argentina", "Thailand", "Australia"])
        travel_bucket_list
```

```
Out[ ]: ['New Zealand',
         'Antarctica',
         'Italy',
         'Chile',
         'Texas',
         'Ireland',
         'Morocco',
         'Germany',
         'Argentina',
         'Thailand',
         'Australia']
```

```
In [ ]: travel_bucket_list.insert(0, "Canada")
        travel_bucket_list
```

```
Out[ ]: ['Canada',
         'New Zealand',
         'Antarctica',
         'Italy',
         'Chile',
         'Texas',
         'Ireland',
         'Morocco',
         'Germany',
         'Argentina',
         'Thailand',
         'Australia']
```

While the `.append()` method and `.extend()` method are very similar, they are also very different. Look at the examples below and note how the two work differently.

In [ ]:
```
travel_bucket_list.append(["Peru", "Dominican Republic"])
travel_bucket_list
```

Out[ ]:
```
['Canada',
 'New Zealand',
 'Antarctica',
 'Italy',
 'Chile',
 'Texas',
 'Ireland',
 'Morocco',
 'Germany',
 'Argentina',
 'Thailand',
 'Australia',
 ['Peru', 'Dominican Republic']]
```

In [ ]:
```
travel_bucket_list.extend("Japan")
travel_bucket_list
```

Out[ ]:
```
['Canada',
 'New Zealand',
 'Antarctica',
 'Italy',
 'Chile',
 'Texas',
 'Ireland',
 'Morocco',
 'Germany',
 'Argentina',
 'Thailand',
 'Australia',
 ['Peru', 'Dominican Republic'],
 'J',
 'a',
 'p',
 'a',
 'n']
```

As we see here, `.append()` will always add the argument to the list as a single item, even if your argument is a list of things. On the other hand, `.extend()` will always treat the argument as a sequence of things and try to add each individual item to the list.

The `.remove()` method searches a list for a value specified and removes the first instance of that value from the collection. The `.pop()` method also removes an item from the list. But instead of specifying a value, we have to give it an index location. Furthermore, `.pop()` removes the item but also *returns* it, meaning that it is made available to be, for instance, stored in a variable.

Method	Description
`list.remove(x)`	Removes the first appearance of `x` in the list
`list.pop(i)`	Removes the item at index `i` in the list and returns that item

Let's clean up the mistake we made above when using the `.extend()` method with a single item ('Japan'). We'll have to remove each letter one by one.

In [ ]:
```
travel_bucket_list.remove('J')
travel_bucket_list.remove('a') # Removes only the first entry so there will be another one
travel_bucket_list.remove('p')
travel_bucket_list.remove('n')
```

In [ ]:
```
travel_bucket_list
```

Out[ ]:
```
['Canada',
 'New Zealand',
 'Antarctica',
 'Italy',
 'Chile',
 'Texas',
 'Ireland',
 'Morocco',
 'Germany',
 'Argentina',
 'Thailand',
 'Australia',
 ['Peru', 'Dominican Republic'],
 'a']
```

In [ ]:
```
travel_bucket_list.remove('a')
travel_bucket_list
```

Out[ ]:
```
['Canada',
 'New Zealand',
 'Antarctica',
 'Italy',
 'Chile',
 'Texas',
 'Ireland',
 'Morocco',
 'Germany',
 'Argentina',
 'Thailand',
 'Australia',
 ['Peru', 'Dominican Republic']]
```

In [ ]:
```
travel_bucket_list.pop(-1) # removes and returns the last entry from list
```

Out[ ]:
```
['Peru', 'Dominican Republic']
```

In [ ]:
```
travel_bucket_list
```

```
Out[ ]: ['Canada',
         'New Zealand',
         'Antarctica',
         'Italy',
         'Chile',
         'Texas',
         'Ireland',
         'Morocco',
         'Germany',
         'Argentina',
         'Thailand',
         'Australia']
```

The `.index()` method searches a list for a given value, and returns the index position of the first instance of the value. If the value is not in the list, this method throws an error. The `.count()` method counts the number of times a value appears in a list.

Syntax	Description
`list_variable.index(x)`	Returns the index where `x` first appears in the list; Throws an error if `x` is not contained in the list
`list_variable.count(x)`	Counts the number of times `x` appears in the list

```
In [ ]: travel_bucket_list.index("Texas")
```

```
Out[ ]: 5
```

```
In [ ]: travel_bucket_list.count("Texas")
```

```
Out[ ]: 1
```

The `sort()` method sorts a list. It sorts in an ascending order by default. You can pass in the `reverse=True` flag to sort in descending order.

- Note: This method mutates the original list, so if the original order is important, please do not use this method.

The `sorted()` method returns a copy of the sorted list that you can assign to a new variable but leaves the original list intact.

```
In [ ]: travel_bucket_list.sort(reverse=True)
        travel_bucket_list
```

```
Out[ ]: ['Thailand',
         'Texas',
         'New Zealand',
         'Morocco',
         'Italy',
         'Ireland',
         'Germany',
         'Chile',
         'Canada',
         'Australia',
         'Argentina',
         'Antarctica']
```

```
In [ ]: other_sorted_travel_bucket_list = sorted(travel_bucket_list)
        print(travel_bucket_list)
        print(other_sorted_travel_bucket_list)
```

```
['Thailand', 'Texas', 'New Zealand', 'Morocco', 'Italy', 'Ireland', 'Germany', 'Chile', 'Canada', 'Australia', 'Argentina', 'Antarctica']
['Antarctica', 'Argentina', 'Australia', 'Canada', 'Chile', 'Germany', 'Ireland', 'Italy', 'Morocco', 'New Zealand', 'Texas', 'Thailand']
```

Exercise:

Create your own list of places you want to travel. Use list methods to add them to the `travel_bucket_list`

```
In [ ]: ## Your code goes here
```

Tuples

Creating Tuples

Tuples are ordered collections, similar to lists, but are *immutable*. They are created using parentheses instead of square brackets. Like all immutable objects, you cannot add, subtract, or edit items in place. To change data in a tuple, you must overwrite the entire structure. This attribute makes tuples great for data that stays static, such as days of the week or coordinate pairs.

Tuple Operations

Since tuples are ordered collections like lists, some operations will work with tuples as they will with lists(iteration, checking length, checking membership).

```
In [ ]: coordinates = (3, 4)
        length = len(coordinates)  # Result: 2
        print(length)
        is_five_present = 5 in coordinates  # Result: False
        print(is_five_present)
```

```
2
False
```

Here are some examples of methods that work with tuples.

```
In [ ]: my_tuple = (1,4,3,8,45,234,5,45,346,678,23,4,2,99,100)

        print(min(my_tuple))
        print(max(my_tuple))
        print(sum(my_tuple))
```

```
1
678
1597
```

Other methods that we find with lists that change the order of elements, or alter the elements in any way, will not work with tuples. the list method `sort()`, while useful for lists, will not work with tuples because they re-order the elements. In the example below, you can see that the `sort()` method is not highlighting as you would see with a list.

```
In [ ]: coordinates.sort()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[29], line 1
----> 1 coordinates.sort()

AttributeError: 'tuple' object has no attribute 'sort'
```

Conclusion

In this lecture, you've explored the capabilities and limitations of Python lists and tuples. Lists are mutable and offer extensive methods for manipulation, making them suitable for dynamic collections. Tuples, on the other hand, are immutable and provide security for data that should not change. Understanding when and how to use each structure is crucial in Python programming.

Remember to practice and apply these concepts in your Python projects to become proficient in working with lists and tuples.

Assignments

- Doubles
- Traveler
- Museum