Lecture 0 - Part 2: Data structures

1 Data Structures

- Real-world problems often require complex **data structures** to solve it.
- In programming, a data structure is any specific object used for collecting and organizing data.
- The four main types of data structures in Python are **lists, tuples, sets,** and **dictionaries**.
- Lists, tuples, sets, and dictionaries are all **iterable objects**, meaning they can be looped over and their values accessed one at a time, permitting it to be iterated over in a **for-loop**.

2 Mutable vs Immutable

- In Python, objects can be either mutable or immutable.
- Immutable objects are objects whose values cannot be changed after they are created. Examples of immutable objects include integers, floating-point numbers, tuples, and strings.
- Mutable objects are objects whose values can be changed after they are created. Examples of mutable objects include lists, sets, and dictionaries.
- When a mutable object is modified, it remains the same object in memory, but its value is changed but when an immutable object is "modified," a new object is created with the new value, and the original object remains unchanged.
- Immutable objects are generally considered to be more "safe" because they cannot be accidentally changed, while mutable objects can lead to unexpected behavior. However, mutable objects are more flexible and can be more efficient in certain situations because they can be modified in place rather than requiring the creation of a new object.

[ ]:
```
import ctypes
#Lists are iterable objects
my_list=["Ali","Python",15.5,2];
for item in my_list:
    print(item)
#Lists are mutable
print("The id of my_list is ",id(my_list))
my_list[0]="Omar"
print(my_list)
print("The id of my_list is ",id(my_list))
#integers are immutable
x=5
print("The id of x is ",id(x))
```

```
y=x
print("The id of y is ",id(y))
x=x+1
print("The id of x is ",id(x))
print(ctypes.cast(id(y), ctypes.py_object).value)
```

2.1 1. Lists

2.1.1 What are lists?

- Lists are a type of ordered collection in Python that allow you to store a collection of values and you can access them by their index position in the list.
- Lists are mutable, meaning you can add, remove, or modify elements after the list has been created.
- Lists are dynamically sized, which means they can grow or shrink as needed to accommodate new elements or remove existing ones.
- Lists can contain elements of different types, including numbers, strings, and even other lists.
- Lists are iterable objects, meaning you can loop over each element in the list using a for loop or other iteration methods.

2.1.2 Create a list in Python:

- To create a list in Python, we use square brackets and separate the elements with commas. For example: `my_list = [1, 2, 3, 'hello', True]`.

2.1.3 Indexing and slicing:

- Lists are indexed starting from 0, so the first element of a list is at index 0, the second at index 1, and so on.
- To access a specific element in a list, we can use its index with square brackets. For example: `my_list[0]` would return the first element of the list, which is 1.
- We can also use slicing to extract a portion of the list. Slicing is done with the colon `:` symbol. For example: `my_list[1:3]` would return a new list containing the second and third elements of `my_list`.

2.1.4 Add/Change List Elements:

- We can add an element to a list using the `append()` method and `extend()` to add a range of items. For example: `my_list.append(4)` would add the number 4 to the end of the list.
- We can also change the value of an existing element in a list by assigning a new value to its index. For example: `my_list[3] = 'world'` would change the fourth element of the list to 'world'.
- We can concatenate two lists using the operator `+` and duplicate (repeat) a list using the operator `*`.

- We can insert an item at a desirable location using the function `insert(<position>,<item>)`.

2.1.5 Delete List Elements:

- We can delete elements from a list using the del keyword followed by the index of the element we want to delete. For example: `del my_list[2]` would delete the third element from the list.
- We can also use the `remove()` method to remove the first occurrence of a specific element. For example: `my_list.remove('hello')` would remove the first occurrence of the string 'hello' from the list.
- There are two other methods, `pop([<index>])` and `clear()` to remove an element of given index (by default the last one) and all elements respectively.

2.1.6 Python List Methods:

Python provides many `built-in` methods that can be used with lists. Some examples include: - `index(item)`: gives the index of first occurrence of `item`. - `copy()`: returns a copy of the list. - `count()`: returns the number of times a specific element appears in the list - `sort()`: sorts the list in ascending order - `reverse()`: reverses the order of the elements in the list.

[ ]:
```python
# Create a list of integers
print("# Create a list of integers")
my_list = [1, 2, 3, 4, 5]
print(my_list)
# Indexing and Slicing
print("# Indexing and Slicing")
print(my_list[0])    # Output: 1
print(my_list[1:3])  # Output: [2, 3]
print(my_list[-1])   # Output: 5

# Add/Change List Elements
print("# Add/Change List Elements")
my_list[2] = 6       # Change the 3rd element to 6
my_list.append(7)    # Add a new element 7 at the end of the list
my_list.insert(0, 0)  # Add a new element 0 at the beginning of the list
print(my_list)

# Delete List Elements
print("# Delete List Elements")
del my_list[1]       # Delete the 2nd element
my_list.remove(4)    # Remove the first occurrence of 4 from the list
my_list.pop()        # Remove the last element from the list
print(my_list)
# Python List Methods
print("# Python List Methods")
print("the first occurence of the item 5 have the index:",my_list.index(5))
```

```
my_list_2=my_list.copy()
print("my_list_2=",my_list_2)
my_list.sort()      # Sort the list in ascending order
print(my_list)
my_list.reverse()   # Reverse the order of the list
print(my_list)
print(len(my_list)) # Output: 4, because we have removed two elements from the␣
 ↪original list
my_list.clear()
type(my_list)
```

2.2 2. Tuples

2.2.1 Create a Tuple in Python:

A tuple is an **ordered** and **immutable** collection of elements in Python. Tuples are defined using parentheses **()** and separating the elements by **commas**.

Example:

```
# Creating a tuple
my_tuple = (1, 2, 3, "four", 5.0)
print(my_tuple)  # Output: (1, 2, 3, 'four', 5.0)
```

2.2.2 Access Tuple Elements:

You can access the elements of a tuple using their index. Indexing starts at 0 for the first element and ends at n-1 for the nth element.

Example:

```
# Accessing elements of a tuple
my_tuple = (1, 2, 3, "four", 5.0)
print(my_tuple[0])  # Output: 1
print(my_tuple[3])  # Output: 'four'
```

2.2.3 Modifying a Tuple:

As tuples are immutable, you cannot modify its elements. However, you can create a new tuple by concatenating two or more tuples.

Example:

```
# Modifying a tuple
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)
new_tuple = tuple1 + tuple2
print(new_tuple)  # Output: (1, 2, 3, 4, 5, 6)
```

2.2.4 Delete Tuple Elements:

As tuples are immutable, you cannot delete a single element. However, you can delete the entire tuple.

Example:

```
# Deleting a tuple
my_tuple = (1, 2, 3, "four", 5.0)
del my_tuple
```

2.2.5 Tuple methods and operations:

Tuples have various `built-in` methods and operations that can be performed on them. Some of them are:

- `count()`: returns the number of times a specified element occurs in the tuple.
- `index()`: returns the index of the first occurrence of a specified element in the tuple.
- `len()`: returns the number of elements in the tuple.

Example:

```
# Tuple methods and operations
my_tuple = (1, 2, 2, 3, 4, 4, 4, "four")
print(my_tuple.count(4))  # Output: 3
print(my_tuple.index(2))  # Output: 1
print(len(my_tuple))  # Output: 8
```

2.2.6 Tuples vs Lists

When it comes to comparing lists and tuples in Python, there are some differences and advantages to consider:

- **Mutability**: Lists are mutable, meaning you can add, remove, or modify elements in place, while tuples are immutable, meaning once they are created, you cannot change them.
- **Performance**: Tuples are generally faster than lists because they are immutable and can be optimized by the interpreter. Lists, on the other hand, require more memory allocation and deallocation operations.
- **Use cases**: Lists are best suited for scenarios where you need to modify the contents of the data structure frequently, while tuples are more suitable for situations where you want to ensure data integrity and prevent accidental modification.
- **Comparison**: Lists and tuples can be compared using the == operator, which checks if both objects have the same elements in the same order. However, if you want to compare the identity of the objects, you can use the `is` operator.
- **Advantages**: Tuples have some advantages over lists, such as being **hashable** (a hash is integer identificator of that object which never changes during its lifetime) and therefore suitable for use as keys in dictionaries, and being more memory-efficient for storing small, fixed-size collections of data.

```
[ ]: # Create a list and a tuple with the same elements
my_list = [4, 2, 3]
my_tuple = (1, 2, 3)

# Modify the first element of the list
my_list[0] = 1
print(my_list)
# Attempt to modify the first element of the tuple (will raise an␣
 ↪error)my_tuple[0] = 4
try:
    my_tuple[0] = 4
except TypeError as e:
    print(e)

# Compare the list and tuple
print(my_list)
print(my_tuple)
print(my_list == list(my_tuple))  # True
print(my_list is list(my_tuple))  # False
```

2.3 3. Sets

2.3.1 Definition of Python Sets:

- A set in Python is an unordered collection of unique and immutable elements. It is defined by enclosing a comma-separated list of values within curly braces `{}` or by using the built-in `set()` function.
- Sets in python imitate the mathematical sets notion.
- Sets can be used to effectively prevent duplicate values. Example:

```
# Defining a set
fruits = {'apple', 'banana', 'orange'}
print(fruits) # Output: {'orange', 'apple', 'banana'}
# Using set() function
numbers = set([1, 2, 3, 4])
print(numbers) # Output: {1, 2, 3, 4}
```

2.3.2 Creating Python Sets:

As mentioned before, we can create a set by enclosing a comma-separated list of values within curly braces `{}` or using the `set()` function. However, if we want to create an empty set, we cannot use the curly braces as it will create an empty dictionary. Instead, we need to use the `set()` function. Example:

```
# Creating an empty set
empty_set = set()
print(empty_set) # Output: set()

# Creating a non-empty set
```

```
fruits = {'apple', 'banana', 'orange'}
print(fruits) # Output: {'orange', 'apple', 'banana'}
```

2.3.3 Modifying a set in Python:

- Since sets are mutable in Python, we can add or remove elements from it. We can add an element to a set using the `add()` method or add multiple elements using the `update()` method.
- However, since they are unordered, indexing has no meaning.
- We cannot access or change an element of a set using indexing or slicing.

Example:

```
# Adding an element to a set
fruits = {'apple', 'banana', 'orange'}
fruits.add('grapes')
print(fruits) # Output: {'orange', 'apple', 'grapes', 'banana'}

# Adding multiple elements to a set
fruits.update(['pineapple', 'watermelon'])
print(fruits) # Output: {'orange', 'apple', 'pineapple', 'grapes', 'banana', 'watermelon'}

# Removing an element from a set
fruits.remove('banana')
print(fruits) # Output: {'orange', 'apple', 'pineapple', 'grapes', 'watermelon'}

# Removing an element using discard()
fruits.discard('banana')
print(fruits) # Output: {'orange', 'apple', 'pineapple', 'grapes', 'watermelon'}
```

2.3.4 Removing elements from a set:

Similarly, we can remove an element from a set using the `remove()` or `discard()` method. However, if we try to remove an element that is not present in the set using the `remove()` method, it will raise a `KeyError`. To avoid this error, we can use the `discard()` method instead. Another method to remove elements from a set is the `pop()` method. This method removes and returns an arbitrary element from the set.

Example:

```
# Using remove() method
fruits = {'apple', 'banana', 'orange'}
fruits.remove('banana')
print(fruits) # Output: {'orange', 'apple'}

# Using discard() method
fruits.discard('banana') # No error is raised
print(fruits) # Output: {'orange', 'apple'}

# Using pop() method
```

```
fruits.pop()
print(fruits) # Output: {'apple'}
```

2.3.5 Python Set Methods and Operations:

Python sets come with a variety of built-in methods and operations that can be used to perform various set operations. Some of these methods and operations include `union()`, `intersection()`, `difference()`, `symmetric_difference()`, `issubset()`, `issuperset()`, `copy()`, `clear()` and more. - **Union**: returns a new set containing all the unique elements from both sets.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
union_set = set1.union(set2)
print(union_set) # output: {1, 2, 3, 4, 5}
```

- **Intersection**: returns a new set containing only the common elements from both sets.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
intersection_set = set1.intersection(set2)
print(intersection_set) # output: {3}
```

- **Difference**: returns a new set containing the elements from the first set that are not in the second set.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
difference_set = set1.difference(set2)
print(difference_set) # output: {1, 2}
```

- **Symmetric Difference**: returns a new set containing the elements that are in either of the sets, but not in both.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
symmetric_difference_set = set1.symmetric_difference(set2)
print(symmetric_difference_set) # output: {1, 2, 4, 5}
```

- **Subset**: checks if one set is a subset of another set.

```
set1 = {1, 2, 3, 4}
set2 = {1, 2, 3}
is_subset = set2.issubset(set1)
print(is_subset) # output: True
```

- **Superset**: checks if one set is a superset of another set.

```
set1 = {1, 2, 3, 4}
set2 = {1, 2, 3}
is_superset = set1.issuperset(set2)
print(is_superset) # output: True
```

- **Copy**: creates a copy of the set.

```
set1 = {1, 2, 3}
set2 = set1.copy()
print(set2) # output: {1, 2, 3}
```

- **Clear**: to remove all elements of a set.

```
set1 = {1, 2, 3}
set2 = set1.clear()
print(set2) # output: None
```

2.4 4. Dictionnaries

2.4.1 What's a Dictionary?

A dictionary is a built-in data type in Python that allows you to store data in `key:value` pairs. In other words, it is a collection of elements that are stored as a `key:value` pair, where each key maps to a corresponding value.

Example:

```
# creating a dictionary
my_dict = {"name": "John", "age": 25, "gender": "Male"}

# accessing values using keys
print(my_dict["name"])  # output: John

# modifying value of a key
my_dict["age"] = 30

# adding a new key-value pair
my_dict["city"] = "New York"

# removing a key-value pair
del my_dict["gender"]
```

2.4.2 How to Create a Dictionary?

A dictionary is created using curly braces `{}` and the key-value pairs are separated by a colon (`:`). Each key-value pair is separated by a comma (`,`).

Example:

```
# creating an empty dictionary
empty_dict = {}

# creating a dictionary with some key-value pairs
my_dict = {"name": "John", "age": 25, "gender": "Male"}

# creating a dictionary using the dict() constructor
another_dict = dict(name="Jane", age=30, city="San Francisco")
```

2.4.3 Types of Values and Keys

In a dictionary, keys must be unique and immutable (cannot be changed). Values can be of any data type and can be changed. There are several built-in data types that can be used as keys, such as integers, strings, and tuples.

Example:

```
# dictionary with integer keys
my_dict = {1: "apple", 2: "banana", 3: "cherry"}

# dictionary with string keys
my_dict = {"name": "John", "age": 25}

# dictionary with tuple keys
my_dict = {("name", 1): "John", ("age", 1): 25}
```

2.4.4 Accessing Elements from Dictionary

You can access the values of a dictionary by specifying its corresponding key in square brackets. If the key is not present in the dictionary, it will raise a KeyError.

Example:

```
# accessing a value using a key
my_dict = {"name": "John", "age": 25}
print(my_dict["name"])  # output: John

# using get() method to access a value
print(my_dict.get("age"))  # output: 25

# handling KeyError
print(my_dict.get("city"))  # output: None
print(my_dict["city"])  # raises KeyError
```

2.4.5 Changing and Adding Dictionary Elements

You can change the value of an existing key or add a new key-value pair to a dictionary using the assignment operator (=). We can also access a using the method `get(<key>)`.

Example:

```
# changing value of a key
my_dict = {"name": "John", "age": 25}
my_dict["age"] = 30

# adding a new key-value pair
my_dict["city"] = "New York"
```

2.4.6 Removing Elements from Dictionary

You can remove a key-value pair from a dictionary using the `del` statement or the `pop()` method. The `pop()` method also returns the value of the deleted key. We can also use the methods `popitem()` to remove and return an arbitrary (key, value) item pair from the dictionary, and `clear()` to remove all the items at once.

Example:

```
# removing a key-value pair using del statement
my_dict = {"name": "John", "age": 25, "city": "New York"}
del my_dict["city"]

# removing a key-value pair using pop() method
my_dict = {"name": "John", "age": 25, "city": "New York"}
age = my_dict.pop("age")
my_dict.popitem()
```