Chapter 4 : Compound Data Types

Aviral Janveja

1 Introduction

In Python, a compound data type is a type of data that is made up of multiple values or data types. The main compound data types in Python are **Tuples**, **Lists**, **Dictionaries** and **Sets**. These compound data types allow you to store and manipulate collections of data in various ways, providing flexibility and efficiency in handling complex data structures in Python programs. Let us look into them one by one.

2 Tuples

A Tuple is an ordered collection of elements that is immutable, meaning that once a tuple is created, its elements cannot be modified, added, or removed. Tuples are defined using parentheses ( ) and can contain elements of any data type, including other tuples. This is how you define a Tuple :

```
# Creating a tuple
my_tuple = (1, 2, 3, 'a', 'b', 'c')
```

Tuples are commonly used for storing collections of related data that should not be modified, such as coordinates, database records, or configurations. They are also useful for returning multiple values from a function in a single return call. Here's a simple example that demonstrates some basic operations and functions on Tuples : tuples.py

3 Lists

A List is an ordered collection of elements that is mutable, meaning it can be modified after creation. Lists are defined using square brackets [ ] and can contain elements of any data type, including other Lists. This is how you define a List :

```
# Define a list
my_list = [1, 2, 3, 4, 5]
```

Lists are versatile data structures in Python and are commonly used for storing collections of related data that may need to be modified. They provide flexibility and efficiency in handling complex data structures in programs. Here's a simple example that demonstrates some basic operations and functions on Lists like indexing, slicing, length and iteration : lists.py. There are a lot more operations that we can do on Lists because of their mutability aspect than we can do on Tuples or Strings for example. Some of these operations are shown in the following sections.

3.1 Add

In Python, you can add new elements to a list using several functions. Here are a few common ways to do it:

1. **append( )** : We can add elements to the end of list using this function.

   ```
   my_list = [1, 2, 3]
   my_list.append(4)
   print(my_list)  # Output: [1, 2, 3, 4]
   ```

   The **dot notation** used with append above, is used for calling functions related to a certain data type. For example, the append function above is related to the list data type and you cannot use it with a string for example.

2. **insert( )** : You can use this function to insert an element at a specific index in the list. For example :

   ```
   my_list = [1, 2, 3]
   my_list.insert(1, 5)  # Inserting 5 at index 1
   print(my_list)  # Output: [1, 5, 2, 3]
   ```

3. **extend( )** : This function is used to add elements of another list to the end of the list. For example :

   ```
   my_list = [1, 2, 3]
   my_list.extend([4, 5, 6])
   print(my_list)  # Output: [1, 2, 3, 4, 5, 6]
   ```

3.2 Delete

We can delete items from a list using various functions. Here are some common ways to delete items from a list :

1. **del( )** : The del statement can be used to remove an item or a slice of items from a list by specifying the index or slice to be removed.

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

   del(my_list[2])
   # Removes the item at index 2 (value: 3)

   print(my_list)  # Output: [1, 2, 4, 5]
   ```

```
# You can also delete a slice of items
del(my_list[1:3])
# Removes items at indices 1 and 2

print(my_list)  # Output: [1, 5]
```

2. **remove( )** : The remove function can be used to remove the first occurrence of a specific element from the list.

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

my_list.remove(3)
# Removes the value 3 from the list

print(my_list)  # Output: [1, 2, 4, 5]
```

3. **pop( )** : The pop function can be used to remove and return an item from a specific index in the list. If no index is provided, it removes and returns the last item in the list.

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

my_list.pop(2)
# Removes and returns the
item at index 2 (value: 3)

print(my_list)  # Output: [1, 2, 4, 5]

# If no index is provided,
it removes and returns the last item
last_item = my_list.pop()

print(last_item)  # Output: 5
print(my_list)  # Output: [1, 2, 4]
```

These are some of the common ways to add and delete items from a list in Python. The appropriate method to use depends on the specific requirements of your program.

3.3 String to List

You can use the **split( )** function to split a string into a list of sub-strings based on a specified parameter. For example :

```
my_string = "Hello World !"
my_list = my_string.split()
print(my_list)  # Output: ['Hello', 'World', '!']
```

In the above example, the string is split into sub-strings based on the default parameter : white-space, resulting in a list of sub-strings. You can also split a string based on a custom parameter :

```
my_string = "apple,banana,orange"
my_list = my_string.split(',')
print(my_list)  # Output: ['apple', 'banana', 'orange']
```

In the above example, the string is split into sub-strings based on the comma parameter, resulting in a list of sub-strings.

3.4 List to String

You can use the **join( )** function to concatenate elements of a list into a single string with a specified parameter in between :

```
my_list = ['a', 'b', 'c']
my_string = ' '.join(my_list)
print(my_string)  # Output: a b c
```

In the above example, elements of the list are concatenated into a single string with a single space between them.

```
my_list = ['a', 'b', 'c']
my_string = '_'.join(my_list)
print(my_string)  # Output: a_b_c
```

Whereas, In the above example, list is concatenated into a single string with a underscore between each element. These functions provide convenient ways to convert between lists and strings in Python, allowing you to manipulate and work with data in various formats.

3.5 Sorting Lists

You can sort lists using either **sorted( )** or the **sort( )** function directly on the list object.

1. **sorted( )** : This function returns a new sorted list without modifying the original list.

    ```
    my_list = [3, 1, 4, 1, 5, 9, 2, 6, 5]
    sorted_list = sorted(my_list)
    print(sorted_list)
    # Output: [1, 1, 2, 3, 4, 5, 5, 6, 9]
    ```

2. **sort( )** : This function sorts the list in-place, meaning it modifies the original list and does not return a new list.

```
my_list = [3, 1, 4, 1, 5, 9, 2, 6, 5]
my_list.sort()
print(my_list)
# Output: [1, 1, 2, 3, 4, 5, 5, 6, 9]
```

These methods provide flexibility in sorting lists in Python based on various requirements and whether to modify the original list or return a new sorted list.

3.6 Reversing

You can reverse a list using either the **reverse( )** function or by using **slicing**. Both functions achieve the same result. The reverse function modifies the original list in place :

```
my_list = [1, 2, 3, 4, 5]
my_list.reverse()
print(my_list)  # Output: [5, 4, 3, 2, 1]
```

Whereas, slicing creates a new list with the elements in reverse order without modifying the original list :

```
my_list = [1, 2, 3, 4, 5]
reversed_list = my_list[::-1]
print(reversed_list)  # Output: [5, 4, 3, 2, 1]
```

Choose the method that best fits your requirements.

3.7 Aliasing

Aliasing refers to the situation where two or more variable names refer to the same data object in memory. When dealing with mutable objects like lists, aliasing can lead to unexpected behavior.

If modifications are made to the data object through one alias, all other aliases will now refer to the modified data object as well. Checkout the following example that illustrates the same : list_alias.py

Aliasing can sometimes lead to unintended side effects, especially in complex programs where multiple aliases are involved and modifications are made in different parts of the program. To avoid such issues, it's important to be aware of aliasing and carefully manage references to data objects, especially when dealing with mutable objects like lists in Python.

3.8 Cloning

Cloning refers to creating a separate and independent copy of a list, such that modifications to the copy do not affect the original list. Cloning is particularly useful when you need to work with a copy of the list while preserving the original list. Checkout the following example which demonstrates the above concept and two ways you can clone a list : cloning.py

Regardless of the way used, cloning ensures that modifications made to the copy, do not affect the original list, hence providing a safer way of working with lists in Python.

3.9 Mutability and Iteration

When we use a for loop to iterate over a list, Python internally keeps track of the index of the current element being processed. It typically starts iterating from the first element (index 0) and proceeds sequentially to the last element of the list.

When you mutate the list during the iteration process, you are changing the size and content of the list while Python's internal loop counter is still iterating through it. This can lead to unexpected behavior such as skipped elements. This happens because the Python loop counter continues on to the next index without adjusting for the modified size of the list. The following program : mutability_iteration_faulty shows how mutating a list that is being iterated upon could lead to undesired results.

To avoid such issues, it is recommended to avoid mutating a list while iterating over it. If you need to modify the list based on certain conditions, consider creating a separate **copy** of the original list first. This approach ensures that the iteration process remains consistent and predictable. The following program : mutability_iteration_corrected displays how that could be achieved by providing a corrected version of the faulty program shown above.

4 Sets

5 Dictionary

5.1 Song Lyrics Frequenct Example