Chapter 5 :
Compound Data Types

Aviral Janveja

1 Introduction

A compound data type is one that is made up of multiple values or data types. The main compound data types in Python are **Tuples**, **Lists**, **Sets** and **Dictionaries**. These compound data types allows us 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. **Ordered** meaning that the elements have a defined order and can be accessed via indexing. **Immutable** meaning that once a tuple is created, its elements cannot be modified. Tuples are defined using parentheses **( )** and can contain elements of any data type, including other tuples. For example :

```
# Defining a tuple
tuple1 = (10, 20, 30)
tuple2 = (10, 30, 20)

print(tuple1 == tuple2)  # Output : False
```

In the above example, we see what is meant by ordered. Despite having the same elements, the tuples are different because the order of elements is different.

Tuples are commonly used for storing collections of related data that should not be modified, such as location coordinates or database records. Here is a simple example that demonstrates some basic operations and functions on Tuples like length, indexing, slicing and iterating : 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. 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, 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 Tuple 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 a 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 be used 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 : a 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, the list is concatenated into a string with an 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 **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]
```

The above methods provide different options for sorting lists in Python as per the requirement, whether to modify the original list or return a new sorted list.

3.6 Reversing

You can reverse a list by using the **reverse( )** function or through **slicing**. 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 without modifying the original :

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

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.

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 list size. The following program : mutability_iteration_faulty shows how mutating a list that is being iterated upon could lead to undesirable 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.

4 Sets

A Set is an **unordered** and **unindexed** collection of **unique** elements, meaning duplicates are not allowed. They are mutable, meaning you can modify them after creation. Sets are quite useful when you need to work with unique elements or perform set operations like union, intersection and difference efficiently. Sets are created using curly braces **{ }**. for Example :

```
my_set = {1, 2, 3, 4, 5}
```

Note that although the set is mutable, the set elements are not. That means only immutable data structures are allowed as elements of a set. So while a Tuple could be a set member, a List cannot be a set member. Further, you cannot reference elements in a set via indexing, however you can loop through the set elements :

```
set1 = {"apple", "banana", "cherry"}

for x in thisset:
    print(x)
```

The following program showcases some of the common operations that are possible on sets : set_operations.py

5 Dictionary

Dictionaries are used to store data in **key-value** pairs. A dictionary is a collection of data which is **ordered** as of Python version 3.7, **mutable** and **does not allow** two values with the same key. Dictionaries are written with curly brackets **{ }**, and can be iterated, modified, and accessed efficiently.:

```
# Empty dictionary
empty_dict = {}

# Dictionary with initial values
my_dict = {"apple" : 5, "banana" : 3, "orange" : 7}
```

Unlike sequences like lists or tuples, dictionaries are indexed by keys rather than numeric indices. Keys in a dictionary must be unique, and attempting to add a duplicate key will overwrite the existing value. The following program showcases some of the common operations that are possible on dictionaries : dictionary.py

5.1 Advantages

- Dictionaries provide **fast lookup**, making them suitable for situations where quick access to values based on keys is required.
- While lists and tuples are indexed by integers, dictionaries allow for **flexible** indexing using keys of **immutable** types, such as strings, integers, tuples.
- Dictionaries are **mutable**, allowing for efficient modification of key-value pairs, which can be useful in scenarios where data needs to be updated frequently.
- Dictionaries can be used to **efficiently** aggregate data of different types, making them suitable for representing complex data structures like JSON objects.

5.2 Fast Fibonacci

Remember the Fibonacci program from chapter 4. We remarked, that the recursive approach used there was elegant, but not really efficient.

The program had exponential run-time as each function call to n, lead to two more recursive function calls to $n-1$ and $n-2$. As an experiment, try running that program to print the first 100 Fibonacci numbers. We observe that the program slows down as we reach around $n = 30$ and almost completely freezes around $n = 35$.

Now, to make that code more efficient, we can make use of **Memoization**, using a dictionary. Which simply means, using a dictionary to store the values of previously computed function calls, so that they need not be repeated unnecessarily.

The following program computes the Fibonacci numbers in a much more efficient manner. Try the above experiment of printing the first 100 Fibonacci numbers with this program and observe the difference : fast_fibonacci.py