Hyperiondev

TASK

# Data Structures - Lists and Dictionaries

Visit our website

# Introduction

## WELCOME TO THE DATA STRUCTURES - LISTS AND DICTIONARIES TASK!

This task aims to ensure that you have a concrete understanding of strings and list manipulation and also to give you a little introduction to dictionaries. In the **example files**, you will see examples that deal with operations that can be applied to elements in lists as well as dictionaries. This task also touches on functions and how they can be used to compute certain values on list elements as well as dictionaries (otherwise known as hash maps).

## LOOKING AT LISTS

Lists have indexes that you can use to call, change, add, or delete elements. Have a look at the examples below:

What will each example print? Jot your answers down on a piece of paper first, and then run the code samples to check your understanding.

Creating a list:

```
string_list = ["John", "Mary", "Harry"]
```

Indexing a list:

```
pet_list = ["cat", "dog", "hamster", "goldfish", "parrot"]
print (pet_list[0])
```

Slicing a list:

```
num_list = [1, 4, 2, 7, 5, 9]
print (num_list[1:2])
```

Changing an element in a list:

```
name_list = ["James", "Molly", "Chris", "Peter", "Kim"]
name_list[2] = "Tom"
```

Adding an element to a list:

```
new_list = [34, 35, 75, "Coffee", 98.8]
new_list.append("Tea")
```

Deleting an element in a list:

```
char_list = ['P', 'y', 't', 'h', 'o', 'n']
del char_list[3]
```

## PYTHON LIST METHODS

There are many useful built-in list methods available for you to use. We have already looked at the `append()` method.

Some other list methods can be found below, with more information easily located **online**:

- `extend()` - Adds all elements of a list to another list
- `insert()` - Inserts an item at the defined index
- `remove()` - Removes an item from the list
- `pop()` - Removes and returns an element at the given index
- `index()` - Returns the index of the first matched item
- `count()` - Returns the count of the items passed as an argument
- `sort()` - Sorts items in a list in ascending order
- `reverse()` - Reverses the order of items in the list

## NESTED LISTS

Lists can include other lists as elements. These inner lists are called nested lists. Look at the following example:

```
a = [1,2,3]
b = [4,9,8]
c = [a,b, 'tea', 16]
print(c)          # prints [[1, 2, 3],[4,9,8], tea, 16]
c.remove(b)
print(c)          # prints [[1, 2, 3], tea, 16]
```

## COPYING LISTS

There are several ways to make a copy of a list. For example, you could use the *slice operator*. The slice operator always creates a new list by making a copy of a portion

of another list. Slice a whole list to make a copy of that list. See below for an example of this:

```
a = [1,2,3]
b = a[:]
b[1] = 10
print(a)          # prints [1, 2, 3]
print(b)          # prints [1, 10, 3]
```

Taking the slice **[:]** creates a new copy of the list. However, it only copies the outer list. Any sublist inside is still a reference to the sublist in the original list. This is called a *shallow copy*. For example,

```
a = [4, 5, 6]
b = a
a[0] = 10
print(b) # prints [10, 5, 6]
```

Alternatively, you could use the `copy()` method of the copy module. Using the `copy()` method ensures that if you modify the copied list (list B), the original list (list A) remains the same. However, if list A contains other lists as items, those inner lists can still be modified if the corresponding inner lists in list B are modified. The `copy.copy()` method makes a shallow copy in the same way that slicing a list does. However, the copy module also contains a function called `deepcopy()`. This makes a copy of the list and any lists contained in it.

To use the `deepcopy()` and `copy()` methods you must import the *copy* module. You use the `deepcopy()` function of the copy module, as shown below:

```
import copy
a = [[1, 2, 3], [4, 5, 6]]
b = a[:]
c = copy.deepcopy(a)
b[0][1] = 10             # changes position [0][1] in both b and a
c[1][1] = 12
print(a)                 # prints [[1, 10, 3], [4, 5, 6]]
print(b)                 # prints [[1, 10, 3], [4, 5, 6]]
print(c)                 # prints [[1, 2, 3], [4, 12, 6]]
```

This is all quite complex stuff! If you're feeling in any way confused about `copy()` and `deepcopy()`, copy and paste the code sample above into your editor and run it. Look at the results, and then try changing aspects of the code and running it again to see how the results change. You'll quickly start to get a feel for what is happening!

Explore the **copy module documentation** for more information.

In summary, the two main methods for copying a list are using the *slice* operator or using the *copy* module. Using the *copy* module allows one to make use of the `deepcopy()` method, which is the best method to use if a list contains other lists.

## LIST COMPREHENSION

List comprehension can be used to construct lists elegantly and concisely. It is a powerful tool that will apply some operation to every element in a list and then put the element into a new list. List comprehension consists of an expression followed by a `for` statement inside square brackets.

For example:

```
num_list = ['1', '5', '8', '14', '25', '31']
new_num_list_ints = [int(element) for element in num_list]
```

For each element in `num_list`, we are casting it to an integer and putting it into a new list called `new_num_list_ints`.

## DICTIONARIES

Dictionaries are used to store data and are very similar to lists. However, lists are *ordered* sets of elements, whereas dictionaries are *unordered* sets. Also, elements in dictionaries are accessed via *keys* and not via their index positions the way lists are. When the key is known, you can use it to retrieve the value associated with it.

## CREATING A DICTIONARY

To create a dictionary, place the items inside curly braces (`{ }`) and separate them with commas (`,`). An item has a *key* and a *value*, which is expressed as what is called a key-value pair `(key: value)`. Items in a dictionary can have a value of any data type. However, the *key* must be immutable (basically anything but a list or dictionary) and unique.

For example:

```
int_key_dict = {1: 'apple',
                2: 'banana',
                3: 'orange'
                }
```

Dictionaries can also be created from a list with the `dict()` function. For example:

```
int_key_list = [(1, 'apple'), (2, 'banana'), (3, 'orange')]
int_key_dict = dict(int_key_list)
```

You'll notice some strange things here: what does `(1, 'apple')` mean? This is a data type called a tuple. A **tuple** is similar to a list, but with some important properties:

- It is immutable (or unchangeable)
- It is ordered. This means that the order in which elements appear is important in some way. In the example above, it is important that the key appears before the value in the tuple.

When reading tuples, it is often useful to use something called *pattern matching*. This is where you assign certain values to certain variables, as long as the tuple matches a certain pattern. For example,

```
my_tuple = (1, 'apple')
key, value = my_tuple
print(key) # prints 1
print(value) # prints apple
```

In this example, the pattern that needs to be matched is that the tuple must contain two values. The first value in the tuple is assigned to `key`, and the second value is assigned to `value`.

## ACCESSING ELEMENTS FROM A DICTIONARY

While we use indexing to access elements in a list, dictionaries use keys. Keys can be used to access values either by placing them inside square brackets (**[]**), such as with indices in lists, or with the `get()` method. However, if you use the `get()` method, it will return 'None' instead of 'KeyError', if the key is not found.

For example:

```
profile_dict = {'name': 'Chris',
                'surname': 'Smith',
                'age': 28,
                'cell': '083 233 3242'
               }

print (profile_dict['surname'])    # prints out 'Smith'
print (profile_dict.get('cell'))   # prints out '083 233 3242'
```

You can also iterate over all the keys and all the values with the **`.keys()`** and **`.values()`** methods respectively. For example, following on from above:

```
keys = profile_dict.keys()
values = profile_dict.values()

print(keys)
print(values)
```

**Output:**

```
dict_keys(['name', 'surname', 'age', 'cell'])
dict_values(['Chris', 'Smith', 28, '083 233 3242'])
```

## CHANGING ELEMENTS IN A DICTIONARY

We can add new items or change items using the assignment operator (=). If there is already a key present, the value gets updated. Otherwise, if there is no key, a new key: value pair is added.

## DICTIONARY MEMBERSHIP TEST

You can test if a key is in a dictionary by using the keyword **`in`**. Enter the key you want to test for membership, followed by the **`in`** keyword and, lastly, the name of the dictionary. This will return either **`True`** or **`False`**, depending on whether the dictionary contains the key or not. The membership test is for keys only, not for values.

```
doubles = { 1: 2,
            2: 4,
            3: 6,
            4: 8,
            5: 10
          }

print(1 in doubles) # prints out True
```

## Instructions

First, read and run the **example files** using VS Code or the IDE/editor of your choice. Feel free to write and run your own example code before doing the compulsory task to become more comfortable with the concepts covered in this task.

## Compulsory Task 1

Follow these steps:

- Imagine you are running a cafe. Create a new Python file in your folder called **cafe.py**.
- Create a list called `menu`, which should contain at least 4 items in the cafe.
- Next, create a dictionary called `stock`, which should contain the stock value for each item on your menu.
- Create another dictionary called `price`, which should contain the prices for each item on your menu.
- Next, calculate the `total_stock` worth in the cafe. You will need to remember to loop through the appropriate dictionaries and lists to do this.
  **Tip:** when you loop through the menu list, the 'items' can be set as keys to access the corresponding 'stock' and 'price' values. Each 'item_value' is calculated by multiplying the stock value by the price value. For example:
  `item_value = (stock[items] * price[items])`
- Finally, print out the result of your calculation.

Rate us

# Share your thoughts

Hyperion strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved, or think we've done a good job?

**Click here** to share your thoughts anonymously.