HyperionDev

# Data Structures – Lists and Dictionaries

## Task

Visit our website

# Introduction

This lesson aims to ensure that you have a concrete understanding of list manipulation and also to provide an introduction to dictionaries (otherwise known as hash maps). We also touch on built-in functions and how they can be used to access and manipulate certain values in list elements as well as dictionaries.

Some of you may have learnt about lists before. If so, the first part of this task will be revision for you. Nonetheless, we encourage you to work through it so as to have the concepts fresh in your mind before starting the section on dictionaries.

# Lists

Lists have indexes that you can use to 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. Some 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, and these inner lists are called nested lists. Look at the following for an 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, which always creates a new list by making a copy of a portion of another list. You can also slice a whole list to make a copy of that list. See below for an example:

```
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] showing that b reflects the current state of a
```

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()`, which makes a copy of the list and any lists contained in it. To use the `deepcopy()` and `copy()` methods, you must first import the `copy` module.

You use the `deepcopy()` function of the `copy` module in the following way:

```
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 for a beginner! 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.

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.

### Extra resource

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

## Looping over lists

What if you have a list of items and you want to do something to each item? It doesn't matter if the list is comprised of 100 items or 3 items, the logic is the same and can be done as shown in the following examples.

We could use a `while` loop to iterate over every item in the list, like this:

```
food_list = ["Pizza", "Burger", "Fries", "Pasta", "Salad"]
i=0

# Run from index 0 to index 1 less than the list length
while i < len(food_list):
    # Print the element at that position
    print(food_list[i])
    # Increment i
    i+=1
```

This loop prints out every item in the list.

We could also use a `for` loop to iterate over every item in the list, as in the following example:

```
food_list = ["Pizza", "Burger", "Fries", "Pasta", "Salad"]
# Loop through each item in the food list
for food in food_list:
        print(food)
```

This loop also prints out every item in the list.

## 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 resulting 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`.

Could you use this approach to multiply every element of `new_num_list_ints` by 2 and put the results into a new list called `by_two_num_list`? How would you do it? Try running the code sample above, and then building on it.

# 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 dictionaries

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 a key–value pair `(key: value)`. Items in a dictionary can have a value of any data type, however, the key must be immutable (unchangeable; 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 the first element of the list at `int_key_list[0]`, containing `(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.
- It is ordered. This means that the order in which elements appear is important in some way. In the example above, it's important that the key appears before the value in the tuple.

The code sample above creates a list of three tuples and then creates a dictionary with three entries, where each tuple becomes an entry representing a key–value pair.

When reading tuples, it's 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.

```
profile_dict["age"] = 29  # Changing value for “age”
print(profile_dict)

profile_dict["gender"] = "male"  # Adding new key-value pair
print(profile_dict)
```

## Dictionary membership test

You can test whether 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.

See below for an example:

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

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

# Instructions

Read and run the accompanying **example files** provided before doing the tasks to become more comfortable with the concepts covered in these tasks.

## Practical task 1

1. Write a Python program called **John.py** that takes in a user's input as a string.
2. While the string is not "John", add every string entered to a list until "John" is entered. This program basically stores all **incorrectly entered strings** in a list where "John" is the only correct string.
3. Print out the list of incorrect names.
4. Example program run (what should show up in the Python console when you run it)

```
Enter your name: <user enters Tim>
Enter your name: <user enters Mark>
Enter your name: <user enters John>
Incorrect names: ['Tim', 'Mark']
```

**HINT:** When testing your `while` loop, you can make use of `.upper()` or `.lower()` to eliminate case sensitivity.

## Practical task 2

Imagine you are running a café. You would like to know the total worth of the stock in your café.

1. Create a new Python file in your folder called **cafe.py**.
2. Create a list called `menu` that should contain at least four items sold in the café.
3. Next, create a dictionary called `stock` that should contain the stock value for each item on your menu.
4. Create another dictionary called `price` that should contain the prices for each item on your menu.
5. Next, calculate the total worth of the stock in the café and then store the results inside a variable called `total_stock`. 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[item] * price[item])`
6. Finally, print out the result of your calculation.

**Important:** Be sure to upload all files required for the task submission inside your task folder and then click "Request review" on your dashboard.

---

## Challenge

Use this opportunity to extend yourself by completing an optional challenge activity.

1. Create a new file in this folder called **loop_lists.py**.
2. Inside it, define a list of strings of your five favourite movies.
3. Now, loop over the list. For each item in the list, print out “Movie: ” plus the movie's name.
4. Can you figure out how to print out “Movie 1: <Movie 1's name>, Movie 2: ...”, etc.?

**Hint:** You will need to look up the `enumerate` command in Python using a Google search. This command allows you to loop over a list retaining both the item at every position and its index (i.e., position in the list).

## Share your thoughts

Please take some time to complete this short feedback **form** to help us ensure we provide you with the best possible learning experience.