HyperionDev

Data Structures – The List

Task

Visit our website

Introduction

In this lesson, you will learn about data structures in programming. A data structure is a specialised format for organising and storing data so that it may be used efficiently. General data structure types include arrays, lists, files, tables, and trees. Different data structures are suited to different kinds of applications, and some are highly specialised for specific tasks. The most common data structure in Python is a list, and this is what we will be focusing on in this task.

Lists

The list is a data type, just like strings, integers, and floats. Lists are known as sequence types because they behave like an ordered collection of items. Sequence types are different from numeric types, such as integers and floats, because they are compound data types, which are data types that are used to store collections of other data types. Lists can store a collection of any data type, with the list of values (items) being separated by commas between square brackets (`[]`). The values or items in a list can be strings, integers, floats, or even other lists or a mix of different types.

Creating a list

To create a list, give it an appropriate name and provide it with values. Enter the name of the list, followed by the assignment operator, and then the list of comma-separated values, placed in between square brackets.

For example:

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

A list can have any number of items and they may even be of different types. It can even have another list as an item. Lists can also start off empty. For example:

```
string_list = []
```

From there, you can add items to your list, as you will learn later in this lesson.

Indexing lists

We can access all elements in a list using the index operator (`[]`). Indices must be integers, with the index starting from `0` for the leftmost item. Therefore, working from left to right, a list containing `10` elements will have indices from `0` to `9`. Alternatively, you can use negative indices to read the items in a list backwards. The index `-1` refers to the rightmost item in a list. Therefore, working from right to left, a list having 10 elements will have indices from `-10` to `-1`. The item at index `0` will be the same as that at index `-10`. This is very similar to how you would access individual characters in a string.

For example:

```
pet_list = ["cat", "dog", "hamster", "goldfish", "parrot"]
print(pet_list[0])  # prints the value of the list at position 0, cat

# What types are we working with? Let's see:
print(type(pet_list[0]))  # prints <class 'str'>
print(type(pet_list))  # prints <class 'list'>
```

This will print out the string `cat`, the type `<class 'str'>` (i.e., a string), and the type `<class 'list'>` (i.e., a list).

Slicing lists

We've already discussed string slicing, and you may have reviewed **this resource** that also deals with list/array slicing. Slicing a list enables you to extract multiple items from that list. We can access a range of items in a list by using the slicing operator (`:`). To slice a list, you need to first indicate a **start** and **end** position for the items you would like to access. Then, place these positions between the index operator (`[]`) and separate them with the colon. The item in the start position is included in the sliced list, while the item in the end position is **not** included.

For example:

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

The above code prints everything from the 1st to the 2nd element of the list, NOT including the 2nd element, so `[4]` will be printed.

What would be printed from the list above using the print statement, `print(num_list[2:4])`? If you're not sure, copy and paste the code and run it to find out.

Changing elements in a list

You can use the assignment operator (=) to change single or multiple elements in a list.

For example, to replace “Chris” with “Tom”, do the following:

```
name_list = ["James", "Molly", "Chris", "Peter", "Kim"]
name_list[2] = "Tom"
print(name_list)  # Prints ["James", "Molly", "Tom", "Peter", "Kim"]
```

Remember that you can also use negative indexing, working from the right-hand side of the list. Thus the code below is equivalent to the code above:

```
name_list = ["James", "Molly", "Chris", "Peter", "Kim"]
name_list[-3] = "Tom"
print (name_list) # This will print ["James", "Molly", "Tom", "Peter", "Kim"]
```

Adding elements to a list

You can add an element to the end of a list using the `append()` method. For example:

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

Deleting elements from a list

You can use the `.pop()` function to remove and return the last element in a list, or an element at a specified index. You could also use the `.remove()` function to remove the first occurrence of a specific element in a list. For example:

```
char_list = ['P', 'y', 't', 'h', 'o', 'n']
print(char_list)  # ['P', 'y', 't', 'h', 'o', 'n']

char_list.pop()
print(char_list)  # ['P', 'y', 't', 'h', 'o']

char_list.pop(0)
print(char_list)  # ['y', 't', 'h', 'o']

char_list.remove('t')
print(char_list)  # [y', 'h', 'o']
```

Checking if something is in a list

You can simply use an `if` statement to check if a certain item is in a list. This uses the `in` operator, which also has a converse, `not in`. These operators (`in` and `not in`) are what we call **membership operators**; they return Booleans (`True` or `False`) and we can use them to test whether or not something is in a list.

Operator	What is returned
`in`	True if value is found in the sequence.
`not in`	True if value is not found in the sequence.

For example:

```
grocery_list = ["Bread", "Milk", "Butter", "Cheese", "Cereal"]

if "Apples" in grocery_list:
    print("The item Apples was found in the list grocery_list")
else:
    print("The item Apples was not found in the list grocery_list")
```

We could shorten and simplify the above example to work without the `if` statement, like this:

```
grocery_list = ["Bread", "Milk", "Butter", "Cheese", "Cereal"]
print("Are apples included?", "Apples" in grocery_list)
# Output: Are apples included? False
```

Quickly populating lists

We can do some pretty nifty things to rapidly instantiate lists, as follows:

```
test_list1 = [7]*5  # Instantiate a list of five 7s
print(test_list1)

test_list2 = ["Bob"]*4  # Instantiate a list of four of the same string
print(test_list2)

test_var = 6
test_list3 = [test_var]*2  # Instantiate a list of two of the same values from a
variable
print(test_list3)
```

Output:

```
[7, 7, 7, 7, 7]
['Bob', 'Bob', 'Bob', 'Bob']
[6, 6]
```

Casting to a list

It is possible to cast almost anything to a list using the `list()` function. For example:

```
hello_string = "Hello world"
hello_list = list(hello_string)

print(hello_list)
```

Output:

```
['H', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
```

The `range()` function is a special Python function that will automatically generate a list of integers within a specified range. The `range()` function needs two integer values, a start number and a stop number. For the function `range(start index: end index)`, the start index is included, and the end index is not included. The `range()` then needs to be cast as a list.

An example of how the `range()` function is used to add values to a list is shown below:

```
num_til_10 = list(range(0, 11))
print(num_til_10)
```

Output:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Instructions

Read and run the accompanying **example file** provided to become more comfortable with the concepts covered in this lesson before doing the practical tasks.

Take note

The task below is **auto-graded**. An auto-graded task still counts towards your progression and graduation. Give it your best attempt and submit it when you are ready.

When you select “Request Review”, the task is automatically complete, and you do not need to wait for it to be reviewed by a mentor.

You will then receive an email with a link to a model answer, as well as an overview of the approach taken to reach this answer.

Take some time to review and compare your work against the model answer. This exercise will help solidify your understanding and provide an opportunity for reflection on how to apply these concepts in future projects.

In the same email, you will also receive a link to a survey, which you can use to self-assess your submission.

Once you’ve done that, feel free to progress to the next task.

Auto-graded task

Follow these steps:

- Create a new Python file in this folder called **list_types.py**.
- Imagine you want to store the names of three of your friends in a list of strings. Create a list variable called `friends_names`, and write the syntax to store the full names of three of your friends.
- Now, write statements to print out the name of the **first friend**, then the name of the **last friend**, and finally the length of the list.

- Now, define a list called `friends_ages` that stores the age of each of your friends in a corresponding manner, i.e., the first entry of this list is the age of the friend named first in the other list.
- Print each friend's name and age in a sentence similar to this:

  ```
  Sophia is 25 years old
  ```

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

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.

Reference list

University of Manchester. (1948). *Williams–Kilburn tube*. Computer History Museum. **https://www.computerhistory.org/revolution/memory-storage/8/308/948**