TASK

Beginner Data Structures — The List

Visit our website

Introduction

WELCOME TO THE BEGINNER DATA STRUCTURES — THE LIST TASK!

In this task, 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 to specific tasks. The most common data structure in Python is a list and this is what we will be focusing on.

Get in touch

Connect for support

Remember that with our courses, you're not alone! You can contact an expert code reviewer to get support on any aspect of your course.

The best way to get help is to login to Discord at **https://discord.com/invite/hyperdev** where our specialist team is ready to support you.

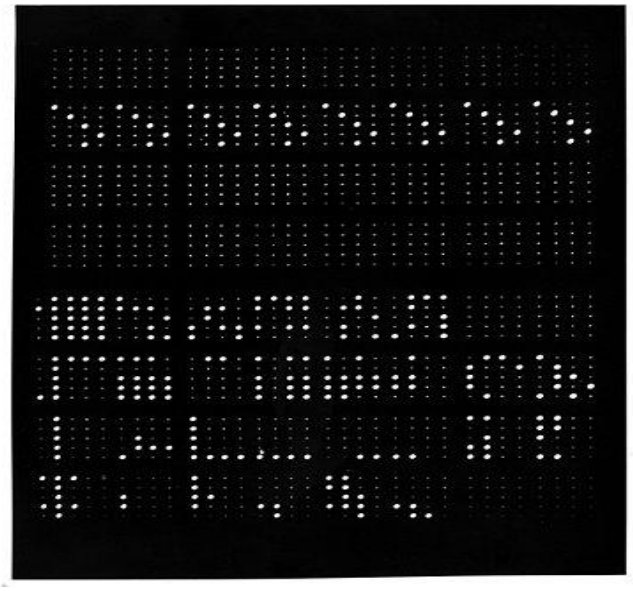
Our team is happy to offer you support that is tailored to your individual career or education needs. Do not hesitate to ask a question or for additional support!

A note from the
Hyperion Team

The Williams-Kilburn Tube

The Williams-Kilburn Tube (or the Williams Tube) was an early computer storage device that used cathode-ray tubes to store bits (0s and 1s). This device was the first electronic memory that stored its bits as dots on a screen. On the face of the tube the dots represent the 1s and the spaces represent the 0s, as seen in the image below. The dots faded after less than a second and needed to be continuously refreshed.

Fred Williams invented the device at Manchester University in 1946. It was developed in collaboration with Tom Kilburn. The device was used in the Manchester Mark I Computer and by IBM, only having 2k bits of memory.



To find out more about how this storage device worked, visit
http://www.radiomuseum.org/forum/williams_kilburn_williams_kilburn_ram.html.

WHAT ARE 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. Compound data types are used to store collections of other data types. Lists can store a collection of any data type. Lists are written as a list of values (items), 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.

NOTE:

If you have previous coding experience, numpy.array([]) and lists are different.

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 further down in *Adding items to a list.*

INDEXING LISTS

We can access all elements in a list using the index operator ([]). Indices must be integers. The index starts 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're already discussed string slicing, and you may have reviewed **this resource** which 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 first to 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':

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

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 comprises 100 items, 3 items, or no 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 #initialise i
while i < len(food_list): #run from index 0 to index one less than the list length
  print (food_list[i]) #print the element at that position
  i+=1 #increment i, the loop counter
```

This loop prints out every item in the list.

We could also use a *for loop* to iterate over every item in the list.

```
food_list = ["Pizza", "Burger", "Fries", "Pasta", "Salad"]
for food in food_list:
        print (food)
```

This loop also prints out every item in the list.

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. This also has a converse, *not in*. *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)
#will print Are apples included? False
```

QUICKLY POPULATING LISTS

We can do some pretty nifty things to rapidly instantiate lists.

```
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 strings
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

First read **example.py**. Open it using VS Code or Anaconda.

- **example.py** should help you understand some simple Python. Every task will have example code to help you get started. Make sure you read all of **example.py** and try your best to understand.

- You may run **example.py** to see the output. Feel free to write and run your own example code before doing the Task to become more comfortable with Python.

Compulsory Task 1

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.

Compulsory Task 2

Follow these steps:

- Create a new Python file in this folder called **loop1000.py**.
- You are asked to print out all the numbers from 1 to 1000. Write 2 lines of code in your file to print out all numbers from 1 to 1000.
- Once you've got this to work, add logic inside your loop to only print out the numbers from 1 to 1000 that are even (i.e. divisible by 2). Remember the modulo command — i.e., %. 10%5 will give you the remainder of 10 divided by 5. 10 divided by 5 equals 2 with a remainder of 0. Hence, this

statement returns 0. Any even number is a number that can be divided by 2 with no remainder.

Compulsory Task 3

Follow these steps:

- Write a Python program called **John.py** that takes in a user's input as a string.
- 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.
- Print out the list of incorrect names.
- 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.

If you are having any difficulties, please feel free to contact our specialist team **on Discord** for support.

Optional Bonus Task

Follow these steps:

- Create a new file in this folder called **loop_lists.py**.
- Inside it, define a list of strings of your 5 favourite movies.
- Now, loop over the list. For each item in the list, print out "Movie: " plus the movie's name.

- 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).

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.