Hyperiondev

TASK

Handling Strings, Lists, and Dictionaries

Visit our website

Introduction

WELCOME TO THE HANDLING STRINGS, LISTS, AND DICTIONARIES TASK!

One of the most crucial concepts to grasp when it comes to programming is string handling. You will need to be very comfortable with string handling, so it is important to refresh and consolidate your knowledge. In this task, you will learn to create more advanced programs with strings, which use more functions and programming techniques.

We will also aim to ensure that you have a concrete understanding of dictionaries (otherwise known as hash maps).

INDEXING STRINGS

You can think of the string 'Hello world!' as a list and each character in the string as an item with a corresponding index.

```
'  H  e  l  l  o     w  o  r  l  d  !  '
   0  1  2  3  4  5  6  7  8  9  10 11
```

The space and the exclamation point are included in the character count, so 'Hello world!' is 12 characters long, from 'H' at index 0 to '!' at index 11.

```
String = "Hello"
print(String[0]) # H
print(String[1]) # e
print(String[2]) # l
print(String[3]) # l
print(String[4]) # o
```

Remember that if you specify an index, you'll get the character at that position in the string. You can also slice strings by specifying a range from one index to another; remember that the character at the starting index is included and the character at the ending index is not.

Note that slicing a string does not modify the original string. You can capture a slice from one variable in a separate variable. Try typing the following into the interactive shell:

```
original_string = "Hello world!"
new_string = original_string[0:5]
print(new_string)
```

What will be printed by the code above? By slicing and storing the resulting substring in another variable, you can have both the whole string and the substring handy for quick, easy access.

STRING METHODS

Once you understand strings and their indexing, the next step is to master using some of the common string methods. These are built-in modules of code that perform certain operations on strings. These methods are useful as they save time since there is no need to write the code over and over again to perform certain operations. The most common string methods are (where `s` is the variable that contains the string we are working with):

- `s.lower()` and `s.upper()` – convert a string to either uppercase or lowercase.
- `s.strip()` – removes any whitespace from the beginning or end of a string.
- `s.strip(',')` – optional input to `s.strip()`. The string provided as input will be removed from the target string. In this case, the "`,`" character gets removed from the `s` string.
- `s.find('text')` – searches for a specific text and returns its position in the string you are searching. If the string isn't found, `-1` is returned.
- `s.replace('oldText', 'newText')` – replaces any occurrence of '`oldText`' with '`newText`'.
- `s.split('word')` – breaks down a string into a list of smaller pieces. The string is separated based on what is called a *delimiter*. This is a string or *char* value that is passed to the method. If no value is given it will automatically split the string using whitespace as the delimiter and create a list of the characters.
- `s.append(item)` – to add an item to the end of a list, you use the `append()` method. For example, `list.append(item)` adds the single item within the brackets to the end of a list.

- `“”.join(string_list)` – takes a list of strings or characters and joins them to create one string. You can specify the character, if any, you wish to use to join the list elements. For example, `“@”.join([“apples”, “bananas”, “carrots”])` would output “`apples@bananas@carrots`”.

Examine **example.py** to see how each of these methods can be used.

ESCAPE CHARACTER

Python uses the backslash (\) as an escape character. The backslash (\) is used as a marker character to tell the compiler/interpreter that the next character has some special meaning. The backslash, together with certain other characters, is known as an escape sequence.

Some useful escape sequences are listed below:

- `\n` – Newline
- `\t` – Tab

The escape character can also be used if you need to include quotation marks within a string. You can put a backslash (\) in front of a quotation mark so that it doesn't terminate the string. What would the code below print out? Try it and see!

```
print("Hello \n\"bob\"")
```

You can also put a backslash in front of another backslash to include a backslash in a string.

```
print("The escape sequence \\n creates a new line in a print statement")
# Output: The escape sequence \n creates a new line in a print statement
```

STRING BUILDING

The best practice is to create strings through `.format()` and, by extension, f-strings. For example:

```
name = "Peter"
print("Hello, {}!".format(name))
```

```
name = "Peter"
```

```
print(f"Hello, {name}!")
```

However, sometimes a form of concatenation is needed in order to *build* a string. One of the main reasons for writing a program is to manipulate information. We turn meaningless data (e.g. 24) into useful information (e.g. Tom is 24 years old). String building allows us to put data in a format that turns data into information. This is important for working with text files, databases, and when you send data from the back end to the front end. Below is an example of string building using a *for loop*:

```
number_builder = ""
i = 0

while i <= 50:
    if i % 2 == 0:
        number_builder += str(i) + " "
    i += 1
print(number_builder)
```

Here, every time **`i`** is even, it gets cast as a string and added to the **`number_builder`** string (which starts off empty – (**“”**)) until **`i`** is greater than 50.

Another way to do this is with the **`.join()`** method you've just learned about. As mentioned, this function takes a list and joins the elements together to make a string. We can rewrite the above example as below to incorporate **`.join()`**:

```
number_builder = [] #note the variable has to be a list rather than a string
i = 0

while i <= 50:
    if i % 2 == 0:
        number_builder.append(str(i))
    i += 1
print(" ".join(number_builder))
```

Here, we have made **`number_builder`** a list, and for each iteration of the loop, an even number gets appended to the list. Finally, in the print statement, the elements are all joined together with a space in between. You may have noticed that **`i`** is cast to a string before being appended. This is because you cannot make an integer act like a string without casting it – only strings and characters can be joined together.

For both examples, the output would be:

```
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50
```

Now, let's transition into the realm of using lists in Python, a versatile and fundamental data structure for organising and manipulating collections of data.

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] 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()**. 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 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.

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 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 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** (unchangeable – basically anything but a list or dictionary) and unique.

Let's have a look at an 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 is 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 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 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.

```
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 to become more comfortable with the concepts covered in this task.

Practical Task 1

Follow these steps:

- Create a file called **alternative.py**.
- Write a program that reads in a string and makes each alternate **character** into an upper case character and each other alternate character a lower case character.

  e.g. The string "`Hello World`" would become "`HeLlO WoRlD`"

- Now, try starting with the same string but making each alternative **word** lower and upper case.

  e.g. The string: "`I am learning to code`" would become "`i AM learning TO code`".

  Tip: Using the `split()` and `join()` functions will help.

Practical Task 2

Follow these steps:

- Imagine you are running a café. Create a new Python file in your folder called **cafe.py**.
- Create a list called `menu`, which should contain at least four items sold in the café.
- 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 café. 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.