# Introduction to Programming W5

Lists and dictionaries I.

2022/05/06

## 1 Lists

Remember from W3: In Python, a **list** is a container, a data structure that can group different data types together. Lists can be created using square brackets, and their elements can be accessed by their index, where indexing starts at 0. Their length is defined by the number of elements the list contain. Try the following code, read the instructional comments and observe the output:

```
our_list = ['Pythagoras', 427, 'Plato', ['apple', 'a plum', 3], True, 'Aristotle']
print('Our list:', our_list)
print('The first element in the list container is', our_list[0])
print('Type of the second element is', type(our_list[1])) #427 is an integer
print('Length of our list is', len(our_list)) #how many elements does the list have?
#the following will correspond to the number of characters in the string:
print("Length of 'Pythagoras' is", len(our_list[0]))

print('List inside our list:', our_list[3]) #lists can contain another list!
print(type(our_list[3]))
print('Length of the list inside our list is', len(our_list[3]))
print('The first element in the list inside our list is', our_list[3][0])
print('The second element in the list inside our list is', our_list[3][1])
print(type(our_list[3][0])) #the type of 'apple'
```

Note that lists can be nested inside lists, and their elements can be accessed by their index, using the same square bracket notation (no additional space is required). For example, `our_list[3][0]` corresponds to the first element of the fourth element in `our_list` (the string `'apple'`). Remember that you can group *different* data types in the list (e.g., Boolean, string, list, float, etc.). Try the following for loop to check the type of the list elements in iteration:

```
for element in our_list:
    print(type(element))
```

As you remember, indexing starts from 0, but one can reduce the index the start accessing elements from the end of the list:

```
print('The last element in our list is', our_list[-1])
print('The third to last element in our list is', our_list[-3])
```

Using *slicing*, one can access more than one element from a list by using a colon (`:`) between the indices. Note that the lower index is inclusive and the upper index is exclusive (i.e., the starting index element will be included, but the end index element will not be included). Try the following code and observe the output:

```
print('The first two elements from our_list are', our_list[0:2])
print('With notation index 3 to index 5:', our_list[3:5])
our_new_list = our_list[1:4]
print(our_new_list)
#the following will return only one element 'Plato', because the upper index is exclusive!
print(our_new_list[1:2])
```

Note that slicing a list will result in a list (notice the square brackets in the output)! Optionally, the lower and upper indices can be omitted if one wishes to slice from the beginning or up to the end of the list:

```
print(our_new_list[:2]) #we start at the beginning (index 0) up to index 2 (exclusive)
#next, we start at index 1 (inclusive) up to the end of the list (would be exclusive index 3)
print(our_new_list[1:])
```

In programming languages, an object is called *mutable* if you can explicitly change it after it is created. If the object cannot be changed once it is created, it is *immutable*. Whether an object is mutable or immutable depends on its data type. Numbers (integers, floating point numbers), Booleans, and strings are immutable, but lists are mutable and can be modified after their creation using indexing and/or slicing. Consider the following example:

```
my_list2 = [1, True, 'three', 4.5]
print(my_list2)
my_list2[2] = 'four'
print(my_list2)
my_list2[1] = '4'
print(my_list2)
```

Above, `my_list2` changed because we replaced the third element (index 2) 'three' with another string 'four'. Then we changed the second element (index 1) that was the Boolean `True` to integer 4. Note that the string and the Boolean are *not modified*, the list `my_list2` is modified. If we try modifying an immutable object (such as a string), we will encounter an error:

```
#trying to "replace" the first character (index 0) from the third element (index 2) of my_list2:
my_list2[2][0] = 'a' #this will not work, and you will encounter a TypeError
```

Operators `in` and `not in` can be used to check the membership.

```
another_list = ['B', 'This is a sentence.', '459', ['B', 'C', 127.45]]
for element in another_list: #iterating through all the elements in another_list
    if 'This' in element: #conditional: if the string "This" is in the current element
        print(element) #only the second element will have 'This' inside

for element in another_list:
    if 'B' in element:
        print(element)
```

In the second for loop, the first element is 'B', so naturally, it has 'B'. The third element (a list) also has 'B' inside (the first, index 0 element in that list). In the next example, `not in` is used to check if the string `a` is in an element or not. (The function `enumerate` is used to create an iterator that has both the indices and values of a list (W4 material).)

```
for index, element in enumerate(another_list):
    if 'a' not in element:
        #string 'a' is in the second element (index 1), so it won't be printed
        print('Index:', index, 'Element:', element)
```

## 2 Dictionaries

A **dictionary** is another useful, and commonly used data type in Python. Data is stored in a dictionary using *key* and *value* pairs. Dictionaries can be created using curly brackets, and the mapping between keys and values are denoted using colons: `key:value`. It is very important that unlike the values, the keys need to be unique, and of an immutable data type (i.e., you cannot have multiples of the same key in a dictionary, and keys needs to be unchangeable). If by mistake, the same key is used more than once in a dictionary, the value corresponding to it will be the latest defined in the dictionary. Keys and values do not need to be the same data type, and both keys and values can be different data types. Try the following code:

```
#here, keys are strings and values are integers
scientist_dictionary = {'Da Vinci':1452, 'Galilei':1564, 'Newton':1642}
print(len(scientist_dictionary)) #the dictionary has 3 elements
random_dictionary = {23:'abcd', 14.678:34, 'abc':[1,2,3]} #mix of different data types
```

Values of a dictionary can be accessed by their corresponding keys, using a square bracket notation:

```
print(scientist_dictionary['Galilei']) #will print the corresponding value
```

Like lists, dictionaries are mutable, so their elements can be changed, and new elements can be added. Try the following code without deleting the assignment of `scientist_dictionary`.

```
print(scientist_dictionary)
#the following will change the value corresponding to the key 'Da Vinci'
scientist_dictionary['Da Vinci'] = 1519
print(scientist_dictionary)
scientist_dictionary['Darwin'] = 1809 #adding a new key:value pair to the dictionary!
print(scientist_dictionary)
```

One can check the membership of keys using the `in` and `not in` operators in a similar way to lists:

```
#the following will print the Boolean True, since the key Newton is in the dictionary
print("Is the key 'Newton' in our dictionary?", 'Newton' in scientist_dictionary)
```

Both lists and dictionaries are *ordered* in Python 3.6<, so the order of elements are reserved. A for loop can be used to access the keys of the dictionary in the order they were initially defined:

```
#this will print the keys of the dictionary
for name in scientist_dictionary:
    print(name)
```

This is equivalent to using the `.keys()` method on the dictionary:

```
#this will print the keys of the dictionary
for name in scientist_dictionary.keys():
    print(name)
```

The values for iteration can be accessed by using the `.values()` method on the dictionary:

```
#this will print the values of the dictionary
for date in scientist_dictionary.values():
    print(date)
```

Using the `.items()` method, keys and values of the dictionary are returned at the same time. Note that two loop variables are needed (here, `name` corresponds to the keys, and `date` corresponds to the values).

```
#this will print the keys and values of the dictionary
for name, date in scientist_dictionary.items():
    print('The name of the scientist:', name, ', Corresponding date:', date)
```

Dictionaries can be especially useful for creating *compound data structures* to organize data. Consider the following example:

```
gpu_dict = {
"nvidia":{
"GPU model": "3090 Ti",
"MSRP": False,
"launch year": 2022,
"manufacturers": ['Gigabyte', 'MSI', 'Zotac']},
"amd":{
"GPU model": "RX 6800 XT",
"MSRP": False,
"launch year": 2020,
"manufacturers": ['MSI', 'Asrock']}
}
```

Lists and dictionaries do not need to be defined in one line, and to make large lists or dictionaries easily readable, one can insert new lines when it is appropriate (do not forget the comma between elements). Above, the dictionary `gpu_dict` has two items. One can confirm this by checking the length of the dictionary with `print(len(gpu_dict))` or by printing out the keys in a for loop. The keys of these two items are "nvidia" and "amd", and the corresponding values for these keys are dictionaries themselves. These dictionaries behave the same way, e.g., values can be accessed with square brackets using the keys, etc. Try the following code:

```
print("Printing the value for the key 'nvidia'") #will be a dictionary
print(gpu_dict['nvidia'])
print("Printing the value for the key 'amd'") #will be a dictionary
print(gpu_dict['amd'])
print("Manufacturers for Nvidia:") #will be a list
print(gpu_dict['nvidia']['manufacturers'])
```

When printing the manufacturers for Nvidia, first, the key 'nvidia' is used to access the dictionary that is its corresponding value. Then, the key 'manufacturers' is used to access its corresponding value which is a list. Try to add one more item to the dictionary, where the value itself is a dictionary! (The content, naming of keys and values are irrelevant, just add one more item.)

# 3 Homework

1. Practice operators, iteration, conditionals using lists and dictionaries!
2. Practice slicing lists and accessing its elements by indexing!
3. Practice creating dictionaries involving different data types!
4. Practice modifying elements of lists and dictionaries!
5. Practice creating compound data structures!