Chapter 3: Data structures in Python

After reading this chapter, readers will be able to:

```
1. create and manipulate lists, dictionaries, tuples and sets
2. Understand differences between different types of data struc
tures
```

3.1 Type of data structures

Data structures are a way of storing and organizing data so that they can be accessed and worked with in an efficient manner. In this tutorial, you'll learn about the various Python data structures and see how they are implemented.

Data structures into Python can be broadly categorized into two broad categories:

```
1. Primary structures
2. Non-primitive structures
```

Let us discuss these two in details with help of examples.

3.2 Primary data structures

These are the most primitive or the basic data structures. They are the building blocks for data manipulation and contain pure, simple values of a data. Python has four primitive variable types:

```
1.Integers: Numeric data; whole numbers from -infinity to +infi
nity.
2.Float:Numbers with decimal numbers such as 3.14, 6.53
3.Strings: Collection of alphabets,characters, or words such as
 "Toyota", "Hyundai", "a", "yo".
4.Boolean: True or False
```

3.2.1 Integers

Operations that we can do on integers include various mathematical operations such add, substract, multiply, divide, calculate remainder,etc. Let us create a variable ***x*** and assign an integer value of 4 to it. Let us create another variable **y** and assign a value of 7 to it. We will perform addition, substraction, multiplication, and remainder calculations using these two variables. Note that in python you can add comments to make code readable by adding # at beginning of the line.

In [1]:

```
# Comment: Declare variables  x and y and assign them values of 3 and 7
x = 3
y = 7
```

In [2]:

```
# Add x and y. as name suggests, print command prints the output.
print(x+y)
```

```
10
```

In [3]:

```
# substract y from x.
print(x-y)
```

```
-4
```

In [4]:

```
# print the remainder of x/y
print(x%y)
```

```
3
```

3.2.2 Floats

Floats can be used for decimal numbers. Float stands for floating point number. We will assign values of 3.3 and 7.5 to ***x*** and ***y*** respectively. Note that previous values get overwritten when we assign new values to the same variable.

In [5]:

```
# Comment: Declare variables  x and y and assign them values of 3.3 and
x = 3.3
y = 7.5
```

In [6]:

```
# Add x and y
print(x+y)
```

```
10.8
```

In [7]:

```
# substract y from x.
print(x-y)
```

```
-4.2
```

In [8]:

```
# print the remainder of x/y
print(x%y)
```

```
3.3
```

3.2.3 Strings

Strings are collections of alphabets, words or other characters. We will assign values of car make and car model to ***x*** and ***y*** respectively. Note that string is enclosed within "" or ''

In [9]:

```
x = " Toyota"
y = "camry"
```

In [10]:

```
#Adding two strings
print(x+y)
```

```
 Toyotacamry
```

In [11]:

```
#Capitalize the string. Use command str.capitalize()
print(str.capitalize(y))
```

```
Camry
```

In [12]:

```
#Compute length of string. Use command len() to compute length
print(len(x+y))
```

```
12
```

3.2.4 Boolean

Boleans take the values of True and False. Let us reassign the value of 3 and 4 to x and y respectively and see if they are equal. Can you guess the result: has to be false!!

In [13]:

```
x = 3
y = 4
print(x==y)
print(x < 4)
```

```
False
True
```

Note: In most programming languages, **=** assigns the value whereas == compares the two values.

3.3 Non-primitive Data Structures

Non-primitive data types are not defined by the programming language, but are instead created by the programmer. Examples of non-primitive data structures include:

```
1. List
2. Dictionaries
3. Tuples
```

3.3.1 List

Lists in Python are used to store collection of heterogeneous items. You can change the elements of list making them mutable. You can recognize lists by their square brackets [and]. For example, x = [] creates an empty list.

In [14]:

```
x = []
print(type(x)) # Prints what is the type of x
```

```
<class 'list'>
```

Let us create a list named **car** in which we save car make, car model, and year of manufacturing. Note that car make and car models are strings whereas year of manufacturing is integer. Lists are quite powerful in saving such heterogenous data types.

In [15]:

```
car = ['Toyota', 'Camry', 2018]
print(car)
```

```
['Toyota', 'Camry', 2018]
```

We can also add variables to the list. Let us create three separate variables, namely, car make, car model, and year of manufacturing and add these to list.

In [16]:

```
make = "Toyota"
model = "Camry"
year = 2018

car_2 = [make, model, year]
print(car_2)
```

```
['Toyota', 'Camry', 2018]
```

Indexing lists

Lists are ordered sets of objects. This means that you can access elements of list by their index. Python starts counting from zero. Hence, if you have to access first element of list ***car_2***, we have to type ***car_2[0]***

In [17]:

```
print(car_2[0])
```

```
Toyota
```

Similarly, if you have to access second element of list ***car_2***, you can type ***car_2[1]***

In [18]:

```
print(car_2[1])
```

```
Camry
```

If you have to access last element of list ***car_2***, you can type ***car_2[-1]***. [-1] refers to the last element in list.

In [19]:

```
print(car_2[-1])
```

```
2018
```

Adding elements to the list: append function

Oo! We forgot to add the price to our car_2 list. Without recreating the list, we have to add an element to the list. To do this, we use ***append*** command.

In [20]:

```
car_2.append("25000")
print(car_2)
```

```
['Toyota', 'Camry', 2018, '25000']
```

Removing elements from list: pop, del and remove function.

There are three ways to remove elements from the list.

1. pop: **list_name.pop(index)** removes the element and displays what is being removed.
2. del: **del list_name[index]** removes the element without displaying what is being removed.

3. remove: **list_name.remove(element_name)**: removes the first item from thr list whose value is

In [21]:

```
# Prints intact list
print(car_2)

#removes element at index 1
car_2.pop(1)
print(car_2)

#same as pop. removes item at last position. (index = -1)
del car_2[-1]
print(car_2)


#removes first occurence of word "Toyota"
car_2.remove("Toyota")
print(car_2)
```

```
['Toyota', 'Camry', 2018, '25000']
['Toyota', 2018, '25000']
['Toyota', 2018]
[2018]
```

3.3.2 Dictionaries

The list car_2 = ['Toyota', 'Camry', 2018, '25000'] is a nice way to store heterogenous data but not necessarily the best way. For example, it is impossible for users of the list to understand what each element refers to. Either you need to add lot of comments to the code or assign name to each elements. This is where Python ***dictionaries*** come to our help. Elements in the dictionary have following characteristis:

```
1. Every entry has a name (or key) and a value.
2. Ordering does not matter. Hence, indexing will not work
3. Elements are accessed using key values
```

Let us create a dictionary containing brand name, model, year and price of our car. Recall that we used square brackets [] to declare the list. However, to create dictionaries, we use curly bracket {}. Let us create an empty dictionary and name it car_dict

In [22]:

```
car_dict = {}
```

Now, we will add the required keys and respective values to our dictionary.

In [23]:

```
car_dict['brand'] = 'Toyota'
car_dict['model'] = 'Camry'
car_dict['year'] = 2018
car_dict['price in $'] = 28000

print(car_dict)
```

```
{'brand': 'Toyota', 'model': 'Camry', 'year': 2018, 'price
in $': 28000}
```

Modifying values in dictionary

To modify values in dictionary, simply reassign the value to key. Let us say, dealership reduced the price of the car by $2000. To modify the price of our car, we ccan simply write:

In [24]:

```
car_dict['price in $'] = 28000-2000
print(car_dict)
```

```
{'brand': 'Toyota', 'model': 'Camry', 'year': 2018, 'price
in $': 26000}
```

Deleting keys and corresponding values

We will use the pop function again to pop out the keys. For example, dealership is no longer interested in keeping year of car model in their record. So, they want to remove ***year*** key from the dictionary.

In [25]:

```
car_dict.pop('year')
print(car_dict)
```

```
{'brand': 'Toyota', 'model': 'Camry', 'price in $': 26000}
```

Using keys and lists simultaneously

Now, our anonymous car dealer got so excited with lists and dictionaries that he wish to use these two to keep stock of their inventory. He is not sure how to add information about to different models into same variable. Specifically, he wants to add make, model, price, year of manufacturing, cars left in inventory for two cars: Toyota Corolla and Toyota Camry.

He has created two dictionaries, named corolla and camry separately:

In [26]:

```
corolla ={}
corolla['make'] = 'Toyota'
corolla['model'] = 'Corolla'
corolla['year'] = 2018
corolla['price'] = 19000
corolla['cars_left'] = 43

camry ={}
camry['make'] = 'Toyota'
camry['model'] = 'Camry'
camry['year'] = 2018
camry['price'] = 26000
camry['cars_left'] = 19
```

Now he wants to add these two dictionaries to the same list so that he can glance at all the information all at once. We will use append function that we learnt with the list. Let us create an empty list named ***dealership*** and add information about ***corolla*** and ***camry*** to the list:

In [27]:

```
dealership = []

dealership.append(corolla)
dealership.append(camry)

print(dealership)
```

```
[{'make': 'Toyota', 'model': 'Corolla', 'year': 2018, 'pri
ce': 19000, 'cars_left': 43}, {'make': 'Toyota', 'model':
'Camry', 'year': 2018, 'price': 26000, 'cars_left': 19}]
```

Recall that we can access elements of lists by their indices. Here, corolla is stored at index 0 and camry is stored at index 1. Hence, we can access the information about ***Corolla*** using:

In [28]:

```
print(dealership[0])
```

```
{'make': 'Toyota', 'model': 'Corolla', 'year': 2018, 'pric
e': 19000, 'cars_left': 43}
```

In [29]:

```
print(dealership[0]['model'])
```

```
Corolla
```

3.3.3 Tuples

Tuples are very similar to lists with one difference. Unlike lists, you cannot change the values of tuples once these are defined (i.e., tuples are immutable). For example, you want to create a tuple with whole numbers from 0 to 5. Notice that tuples are identified by first parenthesis.

In [30]:

```
tuple1 = (0,1,2,3,4,5)
```

Tuples are also ordered object like list. Hence, you can access elements of tuple by their index.

In [31]:

```
print(tuple1[1])
```

```
1
```

You can also print the range of elements. That is if we are interested to print elements 2 through end from the tuple that we just created. We can use tuple_name[starting_point:] to accomplish this operation

In [32]:

```
print(tuple1[1:])
```

```
(1, 2, 3, 4, 5)
```

Now, let us try to change the first value of tuple to 10. You should end up getting an error because tuples are immutable and that's what differentiates these from lists.

In [34]:

```
print(tuple1[0]) = 10
```

```
  File "<ipython-input-34-7dde4f3c5e1c>", line 1
    print(tuple1[0]) = 10
                        ^
SyntaxError: can't assign to function call
```

You can always go back and forth between tuples and lists and convert one format to another. We are so adamant to change first element to 10 that we will first change tuple to list, update the value, and reconvert list to tuple:

In [35]:

```
#Convert tuple to list
list_1 = list(tuple1)
print(list_1)

#Update first element to list
list_1[0] = 10
print(list_1)


#Reconvert list back to tuple
tuple_2 = tuple(list_1)
print(tuple_2)
```

```
[0, 1, 2, 3, 4, 5]
[10, 1, 2, 3, 4, 5]
(10, 1, 2, 3, 4, 5)
```

3.4: Conclusions and next steps

Data structures are lifelines of any programming language. Hence, it is incredibly important to make ourselves comfortable with fundamentals of various datatypes. As we move forward, you will see these data structures in action with real-world examples. In next chapter, we will get ourselves familiar with control structures (such as ***for*** loop, conditional statements, while loop, etc.)