CHAPTER 2 DATA STRUCTURES

BSD2333 DATA WRANGLING

DR. MOHD KHAIRUL BAZLI MOHD AZIZ

2.3 DATA STRUCTURES AND SEQUENCES

LISTS

Lists are fundamental Python data structures that have continuous memory locations and can host different d ata t ypes (such a s s trings, n umbers, fl oats, an d do ubles) an d ca n be ac cessed by the index.

We will start with a list and list comprehension. A list comprehension is a syntactic sugar (or shorthand) for a for loop, which iterates over a list. We will generate a list of numbers, and then examine which ones among them are even. We will sort, reverse, and check for duplicates. We will also see the different w ays w e c an a ccess t he l ist e lements, i terating o ver t hem a nd c hecking the membership of an element.

The following is an example of a simple list:

```
[ ]: list_example = [51, 27, 34, 46, 90, 45, -19]
```

The following is also an example of a list:

```
[ ]: list_example2 = [15, "Yellow car", True, 9.456, [12, "Hello"]]
```

As you can see, a list can contain any number of the allowed data types, such as int, float, string, and boolean, and a list can also be a mix of different data types (including nested lists).

List Functions

In this section, we will discuss a few basic functions for handling lists. You can access list elements using the following code:

```
[ ]: list_example = [51, 27, 34, 46, 90, 45, -19]
     list_example[0]
```

To find out the length of a list, we simply use the len f unction. The len function in Python returns the length of the specified list:

```
[ ]: len(list_example)
```

We can append new elements in the list. append is a built-in method in Python for the list data type:

```
[ ]: list_example.append(11)
     list_example
```

Exercise 2.01 : Accessing the List Members

1. Open a new Jupyter Notebook and define a list called ssn. Read from the ssn.csv file using the read_csv command and print the list elements:

```
[ ]: import pandas as pd
     ssn = list(pd.read_csv("ssn.csv"))
     print(ssn)
```

2. Access the first element of ssn using its forward index:

```
[ ]: ssn[0]
```

3. Access the fourth element of ssn using its forward index:

```
[ ]: ssn[3]
```

4. Access the last element of ssn using the len function:

```
[ ]: ssn[len(ssn) - 1]
```

5. Access the last element of ssn using its backward index:

```
[ ]: ssn[-1]
```

6. Access the first three elements of ssn using forward indices:

```
[ ]: ssn[1:3]
```

7. Access the last two elements of ssn by slicing:

```
[ ]: ssn[-2:]
```

8. Access the first two elements using backward indices:

```
[ ]: ssn[:-2]
```

When we leave one side of the colon (:) blank, we are basically telling Python either to go until the end or start from the beginning of the list. It will automatically apply the rule of list slices that we just learned.

9. Reverse the elements in the list:

```
[ ]: ssn[-1::-1]
```

In this exercise, we learned how to access the list members with forward and backward indices. We'll create a list in the next exercise.

Exercise 2.02: Generating and Iterating through a List

In this exercise, we are going to examine various ways of generating a list and a nested list using the same file containing the list of social security numbers (ssn. csv) that we used in the previous exercise.

We are going to use the append method to add new elements to the list and a while loop to iterate through the list. To do so, let's go through the following steps:

1. Open a new Jupyter Notebook and import the necessary Python libraries. Read from the ssn.csv file:

[ ]:
```
import pandas as pd
ssn = list(pd.read_csv("ssn.csv"))
```

2. Create a list using the append method. The append method from the Python library will allow you to add items to the list:

[ ]:
```
ssn_2 = []
for x in ssn:
    ssn_2.append(x)
ssn_2
```

3. Generate a list using the following command:

[ ]:
```
ssn_3 = ["soc: " + x for x in ssn_2]
ssn_3
```

This is list comprehension, which is a very powerful tool that we need to master. The power of list comprehension comes from the fact that we can use conditionals such as for..in inside the comprehension itself. This will be discussed in detail in Chapter 2, Advanced Operations on Built-in Data Structures.

4. Use a while loop to iterate over the list:

[ ]:
```
i = 0
while i < len(ssn_3):
    print(ssn_3[i])
    i += 1
```

5. Search all the social security numbers with the number 5 in them:

[ ]:
```
numbers = [x for x in ssn_3 if "9" in x]
numbers
```

Let's explore a few more list operations. We are going to use the + operator to add the contents of two lists and use the extend keyword to replace the contents of the existing list with another list.

6. Generate a list by adding the two lists. Here, we will just use the + operator:

[ ]:
```
ssn_4 = ["102-90-0314" , "247-17-2338" , "318-22-2760"]
ssn_5 = ssn + ssn_4
ssn_5
```

7. Extend a string using the extend keyword:

```
[ ]: ssn_2.extend(ssn_4)
     ssn_2
```

8. Now, let's loop over the first list and create a nested list inside that loop that goes over the second list:

```
[ ]: for x in ssn_2:
         for y in ssn_5:
             print(str(x) + ' , ' + str(y))
```

In this exercise, we used the built-in methods of Python to manipulate lists. In the next exercise, we'll check whether the elements or members in a dataset are present as per our expectations.

Exercise 2.03: Iterating over a List and Checking Membership

This exercise will demonstrate how we can iterate over a list and verify that the values are as expected. This is a manual test that can often be done while dealing with a reasonably sized dataset for business case scenarios. Let's go through the following steps to check the membership of values and whether they exist in the .csv file:

1. Import the necessary Python libraries and read from the car_models.csv file:

```
[ ]: import pandas as pd
     car_models = list(pd.read_csv("car_models.csv"))
     car_models
```

2. Iterate over a list:

```
[ ]: list_1 = [x for x in car_models]
     for i in range(0, len(list_1)):
         print(list_1[i])
```

However, this is not very Pythonic. Being Pythonic means to follow and conform to a set of best practices and conventions that have been created over the years by thousands of capable developers. In this case, this means we could use the in keyword in the for..in conditional because Python does not have index initialization, bounds checking, or index incrementing, unlike traditional languages. Python uses syntactic sugar to make iterating through lists easy and readable. In other languages, you might have to create a variable (index initialization) as you loop over the list check that variable (bounds checking) since it will be incremented in the loop (index incrementing).

3. Write the following code to see the Pythonic way of iterating over a list:

```
[ ]: for i in list_1:
         print(i)
```

Notice that in the second method, we do not need a counter anymore to access the list index; instead, Python's in operator gives us the element at the ith position directly.

4. Check whether the strings D150 and Mustang are in the list using the in operator:

```
[ ]: "D150" in list_1
```

```
[ ]: "Mustang" in list_1
```

In this exercise, we've seen how to iterate over a list and verified the membership of each element. This is an important skill. Often, when working with large applications, manually checking a list could be useful. If at any time you are unsure of a list, you can easily verify what values are present. Now, we will see how we can perform a sort operation on a list.

Exercise 2.04: Sorting a List

In this exercise, we will sort a list of numbers, first by using the sort method and then by using the reverse method. To do so, let's go through the following steps:

1. Open a new Jupyter Notebook and import the necessary Python libraries:

```
[ ]: import pandas as pd
     ssn = list(pd.read_csv("ssn.csv"))
```

2. Use the sort method with reverse=True:

```
[ ]: list_1 = [*range(0, 101, 1)]
     list_1.sort(reverse=True)
     list_1
```

3. Use the reverse method directly to achieve this result:

```
[ ]: list_1.reverse()
     list_1
```

The difference between the sort method and the reverse method is that we can use sort with customized sorting, whereas we can only use reverse to reverse a list. Also, both methods work in-place, so be aware of this while using them. Now, let's create a list with random numbers. Random numbers can be very useful in a variety of situations and preprocessing data is a common process in machine learning.

Exercise 2.05: Generating a Random List

In this exercise, we will be generating a list with random numbers using the random library in Python and performing mathematical operations on them. To do so, let's go through the following steps:

1. Import the random library:

```
[ ]: import random
```

2. Use the randint method to generate some random integers and add them to a list:

```
[ ]: list_1 = [random.randint(0, 30) for x in range (0, 100)]
```

3. Let's print the list. Note that there will be duplicate values in list_1:

```
[ ]: list_1
```

4. Let's find the square of each element:

```
[ ]: list_2 = [x**2 for x in list_1]
     list_2
```

5. Now let's find the log of the 1 elements of list_2:

```
[ ]: import math
     list_2 = [math.log(x+1,10) for x in list_2]
     list_2
```

In this exercise, we worked on random variables, lists comprehension, and preprocessing data. Let's put what we have learned so far together and go through an activity to practice how to handle lists.

SETS

A set, mathematically speaking, is just a collection of well-defined distinct objects. Python gives us a straightforward way to deal with them using its set data type.

Introduction to Sets

With the last list that we generated in the previous section; we are going to revisit the problem of getting rid of duplicates from it. We can achieve that with the following line of code:

```
[ ]: list_12 = list(set(list_1))
```

If we print this, we will see that it only contains unique numbers. We used the set data type to turn the first list into a set, thus getting rid of all duplicate elements, and then used the list function to turn it into a list from a set once more:

```
[ ]: list_12
```

Union and Intersection of Sets

In mathematical terms, a list of unique objects is a set. There are many ways of combining sets in the same mathematical term. One such way is the use of a union.

This simply means taking everything from both sets but only taking the common elements once. We can implement this concept by using the following code:

```
[ ]: set1 = {"Apple", "Orange", "Banana","Guava"}
     set2 = {"Pear", "Peach", "Mango", "Banana", "Guava"}
     set3 = {"Durian","Kiwi", "Grape","Avocado","Banana", "Guava"}
```

To find the union of the two sets, the following code should be used:

```
[ ]: set1 | set2 | set3
```

Notice that the common element, Banana, appears only once in the resulting set. The common elements of two sets can be identified by obtaining the intersection of the two sets, as follows:

We get the intersection of two sets in Python as follows:

[ ]:
```
set1 & set2 & set3
```

In this section, we went through sets and how we can do basic set functionality. Sets are used throughout database programming and design, and they are very useful for data wrangling.

0.6 Creating Null Sets

In mathematical terms, a set that has nothing inside it is called a null set or an empty set.

You can create a null set by creating a set containing no elements. You can do this by using the following code:

[ ]:
```
null_set_1 = set({})
null_set_1
```

However, to create a dictionary with null values, use the following command:

[ ]:
```
null_set_2 = {}
null_set_2
```

We are going to learn about this in detail in the next section.

DICTIONARY

A dictionary is like a list, which means it is a collection of several elements. However, with the dictionary, it is a collection of key-value pairs, where the key can be anything that can fit into memory. Generally, we use numbers or strings as keys.

To create a dictionary, use the following code:

[ ]:
```
dict_1 = {"key1": "value1", "key2": "value2"}
dict_1
```

This is also a valid dictionary:

[ ]:
```
dict_2 = {"key1": 1, "key2": ["list_element1", 34], \
"key3": "value3","key4": {"subkey1": "v1"}, \
"key5": 4.5}
dict_2
```

The keys must be unique in a dictionary.

Exercise 2.06: Accessing and Setting Values in a Dictionary

In this exercise, we are going to access the elements and set values in a dictionary. When working with dictionaries, it's important to be able to iterate through each key-value pair, which will allow you to process the data as needed. To do so, let's go through the following steps:

1. To access a value in the dictionary, you must provide the key. Keep in mind there is no given order for any pair in the dictionary:

```
[ ]: stocks = \
{"Solar Capital Ltd.":"$920.44M", \
"Zoe's Kitchen, Inc.":"$262.32M",\
"Toyota Motor Corp Ltd Ord":"$156.02B",\
"Nuveen Virginia Quality Municipal Income Fund":"$238.33M",\
"Kinross Gold Corporation":"$5.1B",\
"Vulcan Materials Company":"$17.1B",\
 "Hi-Crush Partners LP":"$955.69M",\
"Lennox International, Inc.":"$8.05B",\
"WMIH Corp.":"$247.66M",\
"Comerica Incorporated":"n/a"}
```

2. Print a particular element from the stocks list:

```
[ ]: stocks["WMIH Corp."]
```

3. Set a value using the same method we use to access a value:

```
[ ]: stocks["WMIH Corp."] = "$300M"
```

4. Define a blank dictionary and then use the key notation to assign values to it:

```
[ ]: dict_3 = {} # Not a null set. It is a dict
dict_3["key1"] = "Value1"
dict_3
```

As we can see, the manipulation techniques of a dictionary are pretty simple. Now, just like a list, iterating through a dictionary is very important in order to process the data.

Exercise 2.07: Iterating over a Dictionary

In this exercise, we are going to iterate over a dictionary and print the values and keys. To do so, let's go through the following steps:

1. Open a new Jupyter Notebook and define a dictionary with the key provided along with it. Keep in mind there is no given order for any pair in the dictionary:

```
[ ]: stocks = \
{"Solar Capital Ltd.":"$920.44M",\
"Zoe's Kitchen, Inc.":"$262.32M",\
"Toyota Motor Corp Ltd Ord":"$156.02B",\
"Nuveen Virginia Quality Municipal Income Fund":"$238.33M",\
"Kinross Gold Corporation":"$5.1B",\
"Vulcan Materials Company":"$17.1B",\
"Hi-Crush Partners LP":"$955.69M",\
"Lennox International, Inc.":"$8.05B",\
"WMIH Corp.":"$247.66M",\
"Comerica Incorporated":"n/a"}
```

2. Remove the $ character from the stocks dictionary:

[ ]:
```
for key,val in stocks.items():
    stocks[key] = val.replace('$', '')
stocks
```

3. Iterate over the stocks dictionary again and split the value into a list with price (val) and multiplier (mult) as separate elements where a single value is assigned to each key:

[ ]:
```
for key,val in stocks.items():
    mult = val[-1]
    stocks[key] = [val[:-1],mult]
stocks
```

Notice the difference between how we did the iteration on the list and how we are doing it h ere. A dictionary always contains a key-value pair, and we always need to access the value of any element in a dictionary with its key. In a dictionary, all the keys are unique.

In the next exercise, we will revisit the problem that we encountered with the list earlier in this chapter to create a list with unique values. We will look at another workaround to fix this problem.

Exercise 2.08: Revisiting the Unique Valued List Problem

In this exercise, we will use the unique nature of a dictionary, and we will drop the duplicate values from a list. First, we will create a random list with duplicate values. Then, we'll use the fromkeys and keys methods of a dictionary to create a unique valued list. To do so, let's go through the following steps:

1. First, generate a random list with duplicate values:

[ ]:
```
import random
list_1 = [random.randint(0, 30) for x in range (0, 100)]
```

2. Create a unique valued list from list_1:

[ ]:
```
list(dict.fromkeys(list_1).keys())
```

Here, we have used two useful methods of the dict data type in Python, fromkeys and keys. fromkeys is a built-in function in which a new dictionary is created from the given sequence of elements with values given by the user, while the keys method gives us the keys of a dictionary.

0.7.4 Exercise 2.09: Deleting a Value from Dictionary

In this exercise, we are going to delete a value from dict using the del method. Perform the following steps:

1. Create list_1 with five elements:

[ ]:
```
dict_1 = {"key1": 1, "key2": ["list_element1", 34], \
"key3": "value3","key4": {"subkey1": "v1"}, \
"key5": 4.5}
dict_1
```

2. We will use the del function and specify the element we want to delete:

[ ]:
```
del dict_1["key2"]
dict_1
```

3. Let's delete key3 and key4:

[ ]:
```
del dict_1["key3"]
del dict_1["key4"]
```

4. Now, let's print the dictionary to see its content:

[ ]:
```
dict_1
```

In this exercise, we learned how to delete elements from a dictionary. This is a very useful functionality of dictionaries, and you will find that it's used heavily when writing Python applications.

In our final exercise on dict, we will go over a less commonly used list comprehension called dictionary comprehension. We will also examine two other ways to create a dict, which can be very useful for processing dictionaries in one line. There could be cases where this could be used as a range of key-value pairs of name and age or credit card number and credit card owner. A dictionary comprehension works exactly the same way as list comprehension, but we need to specify both the key and the value.

Exercise 2.10: Dictionary Comprehension

In this exercise, we will generate a dictionary using the following steps:

1. Generate a dict that has 0 to 9 as the keys and the square of the key as the values:

[ ]:
```
list_1 = [x for x in range(0, 10)]
dict_1 = {x : x**50 for x in list_1}
dict_1
```

Can you generate a dict using dict comprehension without using a list? Let's try this now.

2. Generate a dictionary using the dict function:

[ ]:
```
dict_2 = dict([('Tom', 100), ('Dick', 200), ('Harry', 300)])
dict_2
```

3. You can also a dictionary using the dict function, as follows:

[ ]:
```
dict_3 = dict(Tom=100, Dick=200, Harry=300)
dict_3
```

Dictionaries are very flexible and can be used for a variety of tasks. The compact nature of comprehension makes them very popular. The strange-looking pair of values that just looked at ('Harry', 300) is called a tuple. This is another important fundamental data type in Python. We will learn about tuples in the next section.

TUPLES

A tuple is another data type in Python. Tuples in Python are similar to lists, with one key difference. A tuple is a variant of a Python list that is immutable. Immutable basically means you can't modify it by adding or removing from the list. It is sequential in nature and similar to lists.

A tuple consists of values separated by commas, as follows:

```
[ ]: tuple_1 = 24, 42, 2.3456, "Hello"
     tuple_1
```

Notice that, unlike lists, we did not open and close square brackets here.

When referring to a tuple, the length of the tuple is called its cardinality. This comes from database and set theory and is a common way to reference its length.

Creating a Tuple with Different Cardinalities

This is how we create an empty tuple:

```
[ ]: tuple_1 = ()
```

This is how we create a tuple with only one value:

```
[ ]: tuple_1 = "Hello",
```

Notice the trailing comma here.

We can nest tuples, similar to lists and dicts, as follows:

```
[ ]: tuple_1 = "hello", "there"
     tuple_12 = tuple_1, 45, "Sam"
```

One special thing about tuples is the fact that they are an immutable data type. So, once they're created, we cannot change their values. We can just access them, as follows:

```
[ ]: tuple_1 = "Hello", "World!"
     tuple_1[1] = "Universe!"
```

The last line of the preceding code will result in a TypeError as a tuple does not allow modification. This makes the use case for tuples a bit different than lists, although they look and behave very similarly in a few ways.

We can access the elements of a tuple in the same manner we can for lists:

```
[ ]: tuple_1 = ("good", "morning!" , "how", "are", "you?")
     tuple_1[0]
```

Let's access another element:

```
[ ]: tuple_1[3]
```

Unpacking a Tuple

The expression "unpacking a tuple" simply means getting the values contained in the tuple in different variables:

```
[ ]: tuple_1 = "Hello", "World"
     hello, world = tuple_1
     print(hello)
     print(world)
```

Of course, as soon as we do that, we can modify the values contained in those variables.

Exercise 2.11: Handling Tuples

In this exercise, we will walk through the basic functionalities of tuples. Let's go through the steps one by one:

1. Create a tuple to demonstrate how tuples are immutable. Unpack it to read all the elements, as follows:

```
[ ]: tupleE = "1", "3", "5"
     tupleE
```

2. Try to override a variable from the tupleE tuple:

```
[ ]: tupleE[1] = "5"
```

This step will result in TypeError as the tuple does not allow modification.

3. Try to assign a series to the tupleE tuple:

```
[ ]: 1, 3, 5 = tupleE
```

This step will also result in a SyntaxError, stating that it can't assign to the literal:

4. Print variables at 0th and 1st positions:

```
[ ]: print(tupleE[0])
     print(tupleE[1])
```

We have seen two different types of data so far. One is represented by numbers, while the other is represented by textual data. Now it's time to look into textual data in a bit more detail.

STRINGS

Strings in Python are similar to strings in any other programming language.

This is a string:

```
[ ]: string1 = 'Hello World!'
```

A string can also be declared in this manner:

```
[ ]: string2 = "Hello World 2!"
```

You can use single quotes and double quotes to define a string.

The start and end of a string is defined as:

str[inclusive start position: exclusive end position].

Strings in Python behave similar to lists, apart from one big caveat. Strings are immutable, whereas lists are mutable data structures.

Exercise 2.12: Accessing Strings

In this exercise, we are going perform mathematical operations to access strings. Let's go through the following steps:

1. Create a string called str_1:

```
[ ]: str_1 = "Hello World!"
     str_1
```

You can access the elements of the string by specifying the location of the element, like we did for lists.

2. Access the first member of the string:

```
[ ]: str_1[0]
```

3. Access the fifth member of the string:

```
[ ]: str_1[4]
```

4. Access the last member of the string:

```
[ ]: str_1[len(str_1) - 1]
```

5. Access the last member of the string, in a different way this time:

```
[ ]: str_1[-1]
```

Each of the preceding operations will give you the character at the specific index. The method for accessing the elements of a string is like accessing a list. Let's do a couple of more exercises to manipulate strings.

Exercise 2.13: String Slices

This exercise will demonstrate how we can slice strings the same way as we did with lists. Although strings are not lists, the functionality will work in the same way.

Let's go through the following steps:

1. Create a string, str_1:

```
[ ]: str_1 = "Hello World! I am learning data wrangling"
     str_1
```

2. Specify the slicing values and slice the string:

[ ]:
```
str_1[2:10]
```

3. Slice a string by skipping a slice value:

[ ]:
```
str_1[-31:]
```

4. Use negative numbers to slice the string:

[ ]:
```
str_1[-10:-5]
```

As we can see, it is quite simple to manipulate strings with basic operations.

String Functions

To find out the length of a string, we simply use the len function:

[ ]:
```
str_1 = "Hello World! I am learning data wrangling"
len(str_1)
```

The length of the string is 41. To convert a string's case, we can use the lower and upper methods:

[ ]:
```
str_1 = "A COMPLETE UPPER CASE STRING"
str_1.lower()
```

To change the case of the string, use the following code:

[ ]:
```
str_1.upper()
```

To search for a string within a string, we can use the find method:

[ ]:
```
str_1 = "A complicated string looks like this"
str_1.find("complicated")
str_1.find("hello")
```

The output is -1. Can you figure out whether the find method is case-sensitive or not? Also, what do you think the find method returns when it actually finds the string?

To replace one string with another, we have the replace method. Since we know that a string is an immutable data structure, replace actually returns a new string instead of replacing and returning the actual one:

[ ]:
```
str_1 = "A complicated string looks like this"
str_1.replace("complicated", "simple")
```

Strings have two useful methods: split and join. Here are their definitions:

str.split(separator)

The seperator argument is a delimiter that you define:

string.join(seperator)

Exercise 2.14: Splitting and Joining a String

This exercise will demonstrate how to perform split and join operations on a string. These two string methods need separate approaches as they allow you to convert a string into a list and vice versa. Let's go through the following steps to do so:

1. Create a string and convert it into a list using the split method:

```
[ ]: str_1 = "Name, Age, Sex, Address"
     list_1 = str_1.split(",")
     list_1
```

2. Combine this list into another string using the join method:

```
[ ]: s = " | "
     s.join(list_1)
```

CHAPTER 2 DATA STRUCTURES

2.4 FUNCTIONS

Introduction

We were introduced to the basic concepts of different fundamental data structures in the previous chapter. We learned about lists, sets, dictionaries, tuples, and strings. However, what we have covered so far were only basic operations on those data structures. They have much more to offer once you learn how to utilize them effectively. In this chapter, we will venture further into the land of data structures. We will learn about advanced operations and manipulations and use fundamental data structures to represent more complex and higher-level data structures; this is often handy while wrangling data in real life. These higher-level topics will include stacks, queues, interiors, and file operations.

In this chapter, we will also learn how to open a file using built-in Python methods and about the many different file operations, such as reading and writing data, and safely closing files once we are done. We will also take a look at some of the problems to avoid while dealing with files.

Iterator

Iterators in Python are very useful when dealing with data as they allow you to parse the data one unit at a time. Iterators are stateful, which means it will be helpful to keep track of the previous state. An iterator is an object that implements the next method—meaning an iterator can iterate over collections such as lists, tuples, dictionaries, and more. Practically, this means that each time we call the method, it gives us the next element from the collection; if there is no further element in the list, then it raises a StopIteration exception.

Let's learn about the various functions we can use with itertools. As you execute each line of the code after the import statement, you will be able to see details about what that particular function does and how to use it:

```
[ ]: from itertools import (permutations, combinations, \
     dropwhile, repeat, zip_longest)

     permutations?
     combinations?
     dropwhile?
     repeat?
     zip_longest?
```

Exercise 2.15: Introducing to the Iterator

In this exercise, we're going to generate a long list containing numbers. We will first c heck the memory occupied by the generated list. We will then check how we can use the iterator module to reduce memory utilization, and finally, w e w ill u se t his i terator t o l oop o ver t he l ist. To d o this, let's go through the following steps:

1. Open a new Jupyter Notebook and generate a list that will contain 10000000 ones. Then, store this list in a variable called big_list_of_numbers:

```
[ ]: big_list_of_numbers = [1 for x in range (0, 10000000)]
     big_list_of_numbers
```

2. Check the size of this variable:

```
[ ]: from sys import getsizeof
     getsizeof(big_list_of_numbers)
```

The value shown is 81528048 (in bytes). This is a huge chunk of memory occupied by the list. And the big_list_of_numbers variable is only available once the list comprehension is over. It can also overflow the available system memory if you try too big a number.

3. Let's use the repeat() method from itertools to get the same number but with less memory:

```
[ ]: from itertools import repeat
     small_list_of_numbers = repeat(1, times=10000000)
     getsizeof(small_list_of_numbers)
```

The last line shows that our list small_list_of_numbers is only 48 bytes in size. Also, it is a lazy method, a technique used in functional programming that will delay the execution of a method or a function by a few seconds. In this case, Python will not generate all the elements initially. It will, instead, generate them one by one when asked, thus saving us time. In fact, if you omit the times keyword argument in the repeat() method in the preceding code, then you can practically generate an infinite number of ones.

4. Loop over the newly generated iterator:

```
[ ]: for i, x in enumerate(small_list_of_numbers):
         print(x)
         if i > 10:
             break
```

We use the enumerate function so that we get the loop counter, along with the values. This will help us break the loop once we reach a certain number (10, for example.

In this exercise, we first l earned h ow t o u se t he i terator f unction t o r educe m emory u sage. Then, we used an iterator to loop over a list. Now, we'll see how to create stacks.

Stacks

A stack is a very useful data structure. If you know a bit about CPU internals and how a program gets executed, then you will know that a stack is present in many such cases. It is simply a list with one restriction, Last In First Out (LIFO, meaning an element that comes in last goes out first w hen a value is r ead from a stack.

We will implement a stack using a Python list. Python lists have a method called pop, which does the exact same pop operation that you can see in the preceding illustration. Basically, the pop function will take an element off t he s tack, u sing t he L ast i n F irst O ut (LIFO r ules. We w ill use that to implement a stack in the following exercise.

Let's look at an example and try to understand the working of push() and pop() function:

Exercise 2.16: Stack Using List

```
[ ]: # Python code to demonstrate Implementing
     # stack using list
     stack = ["Goku", "Bezita", "Gohan"]
     # Let's push some other names into our list
     stack.append("Trunks")
     stack.append("Goten")
     print(stack)

     # Removes the last item
     print(stack.pop())

     print(stack)

     # Removes the last item
     print(stack.pop())

     print(stack)
```

Exercise 2.17: Implementing a Stack in Python

In this exercise, we'll implement a stack in Python. We will first c reate a n e mpty s tack a nd add new elements to it using the append method. Next, we'll take out elements from the stack using the pop method. Let's go through the following steps:

1. Import the necessary Python library and define a n e mpty stack:

```
[ ]: import pandas as pd
     stack = []
```

2. Use the append method to add multiple elements to the stack. Thanks to the append method, the element will always be appended at the end of the list:

```
[ ]: stack.append('my_test@test.edu')
     stack.append('rahul.subhramanian@test.edu')
     stack.append('sania.test@test.edu')
     stack.append('alec_baldwin@test.edu')
     stack.append('albert90@test.edu')
     stack.append('stewartj@test.edu')
```

```
stack
```

3. Let's read a value from our stack using the pop method. This method reads the current last index of the list and returns it to us. It also deletes the index once the read is done:

[ ]:
```
tos = stack.pop()
tos
```

As you can see, the last value of the stack has been retrieved. Now, if we add another value to the stack, the new value will be appended at the end of the stack.

4. Append Hello@test.com to the stack:

[ ]:
```
stack.append("Hello@test.com")
stack
```

[ ]:
```
tos = stack.pop()
tos
```

From the exercise, we can see that the basic stack operations, append and pop, are pretty easy to perform.

Let's visualize a problem where you are scraping a web page and you want to follow each URL present there (backlinks. Let's split the solution to this problem into three parts. In the first part, we would append all the URLs scraped off t he page into t he s tack. I n t he s econd p art, we would pop each element in the stack, and then lastly, we would examine every URL, repeating the same process for each page. We will examine a part of this task in the next exercise.

Exercise 2.18: Implementing a Stack Using User-De ined Methods

In this exercise, we will continue the topic of stacks from the last exercise. This time, we will implement the append and pop functions by creating user-defined m ethods. We w ill i mplement a stack, and this time with a business use case example (taking Wikipedia as a source. The aim of this exercise is twofold. In the first f ew s teps, w e w ill e xtract a nd a ppend t he U RLs scraped off a web page i n a s tack, w hich a lso i nvolves t he s tring m ethods d iscussed i n t he l ast c hapter. In the next few steps, we will use the stack_pop function to iterate over the stack and print them. This exercise will show us a subtle feature of Python and how it handles passing list variables to functions. Let's go through the following steps:

1. First, define two functions: stack_push and stack_pop. We renamed them so that we do not have a namespace conflict. Also, create a stack called url_ stack for later use:

[ ]:
```
def stack_push(s, value):
    return s + [value]
def stack_pop(s):
    tos = s[-1]
    del s[-1]
    return tos
url_stack = []
url_stack
```

The first function takes the already existing stack and adds the value at the end of it.

Now, we are going to have a string with a few URLs in it.

2. Analyze the string so that we push the URLs in the stack one by one as we encounter them, and then use a for loop to pop them one by one. Let's take the first line from the Wikipedia article (https://en.wikipedia.org/wiki/Data_mining) about data science:

[ ]:
```
wikipedia_datascience = """Data science is an interdisciplinary
field that uses scientific methods, processes, algorithms and systems
to extract knowledge [https://en.wikipedia.org/wiki/Knowledge] and
insights from data [https://en.wikipedia.org/wiki/Data] in various
forms, both structured and unstructured,similar to data mining
[https://en.wikipedia.org/wiki/Data_mining]"""
```

For the sake of the simplicity of this exercise, we have kept the links in square brackets beside the target words.

3. Find the length of the string:

[ ]:
```
len(wikipedia_datascience)
```

4. Convert this string into a list by using the split method from the string, and then calculate its length:

[ ]:
```
wd_list = wikipedia_datascience.split()
wd_list
```

5. Check the length of the list:

[ ]:
```
len(wd_list)
```

6. Use a for loop to go over each word and check whether it is a URL. To do that, we will use the startswith method from the string, and if it is a URL, then we push it into the stack:

[ ]:
```
for word in wd_list:
    if word.startswith("[https://"):
        url_stack = stack_push(url_stack, word[1:-1])
        print(word[1:-1])
```

Notice the use of string slicing to remove the surrounding double quotes "[" "]".

7. Print the value in url_stack:

[ ]:
```
print(url_stack)
```

8. Iterate over the list and print the URLs one by one by using the stack_ popz function:

[ ]:
```
for i in range(0, len(url_stack)):
    print(stack_pop(url_stack))
```

9. Print it again to make sure that the stack is empty after the final for loop:

[ ]:
```
print(url_stack)
```

In this exercise, we have noticed a strange phenomenon in the stack_pop method. We passed the list variable there, and we used the del operator inside the function in step 1, but it changed the original variable by deleting the last index each time we called the function. If you use languages like C, C++, and Java, then this is a completely unexpected behavior as, in those languages, this can only happen if we pass the variable by reference, and it can lead to subtle bugs in Python code. So, be careful when using the user-defined methods.

Lambda Expressions

In general, it is not a good idea to change a variable's value inside a function. Any variable that is passed to the function should be considered and treated as immutable. This is close to the principles of functional programming. However, in that case, we could use unnamed functions that are neither immutable nor mutable and are typically not stored in a variable. Such an expression or function, called a lambda expression in Python, is a way to construct one-line, nameless functions that are, by convention, side-effect-free and are loosely considered as implementing functional programming.

Let's look at the following exercise to understand how we use a lambda expression.

Exercise 2.19: Implementing a Lambda Expression

In this exercise, we will use a lambda expression to prove the famous trigonometric identity:

Let's go through the following steps to do this: 1. Import the math package:

[ ]:
```
import math
```

2. Define two functions, my_sine and my_cosine, using the def keyword. The reason we are declaring these functions is the original sin and cos functions from the math package take radians as input, but we are more familiar with degrees. So, we will use a lambda expression to define a wrapper function for sine and cosine, then use it. This lambda function will automatically convert our degree input to radians and then apply sin or cos on it and return the value:

[ ]:
```
def my_sine():
    return lambda x: math.sin(math.radians(x))
def my_cosine():
    return lambda x: math.cos(math.radians(x))
```

3. Define sine and cosine for our purpose:

[ ]:
```
sine = my_sine()
cosine = my_cosine()
math.pow(sine(30), 2) + math.pow(cosine(30), 2)
```

Notice that we have assigned the return value from both my_sine and my_cosine to two variables, and then used them directly as the functions. It is a much cleaner approach than using them explicitly. Notice that we did not explicitly write a return statement inside the lambda function; it is assumed.

Now, in the next section, we will be using lambda functions, also known as anonymous functions, which come from lambda calculus. Lambda functions are useful for creating temporary functions that are not named. The lambda expression will take an input and then return the first character of that input.

Exercise 2.20: Lambda Expression for Sorting

In this exercise, we will be exploring the sort function to take advantage of the lambda function. What makes this exercise useful is that you will be learning how to create any unique algorithm that could be used for sorting a dataset. The syntax for a lambda function is as follows:

lambda x :

A lambda expression can take one or more inputs. A lambda expression can also be used to reverse sort by using the parameter of reverse as True. We'll use the reverse functionality as well in this exercise. Let's go through the following steps:

1. Let's store the list of tuples we want to sort in a variable called capitals:

```
[ ]: capitals = [("USA", "Washington"), ("India", "Delhi"), ("France",
     "Paris"), ("UK", "London")]
```

2. Print the output of this list:

```
[ ]: capitals
```

3. Sort this list by the name of the capitals of each country, using a simple lambda expression. The following code uses a lambda function as the sort function. It will sort based on the first element in each tuple:

```
[ ]: capitals.sort(key=lambda item: item[0])
     capitals
```

As we can see, lambda expressions are powerful if we master them and use them in our data wrangling jobs. They are also side-effect-free—meaning that they do not change the values of the variables that are passed to them in place.

We will now move on to the next section, where we will discuss membership checking for each element. Membership checking is commonly used terminology in qualitative research and describes the process of checking that the data present in a dataset is accurate.

Exercise 2.21: Multi-Element Membership Checking

In this exercise, we will create a list of words using for loop to validate that all the elements in the first list are present in the second list. Let's see how:

1. Create a list_of_words list with words scraped from a text corpus:

```
[ ]: list_of_words = ["Hello", "there.", "How", "are", "you", "doing?"]
     list_of_words
```

2. Define a check_for list, which will contain two similar elements of list_of_words:

[ ]:
```
check_for = ["How", "are"]
check_for
```

There is an elaborate solution, which involves a for loop and a few if/else conditions (and you should try to write it), but there is also an elegant Pythonic solution to this problem, which takes one line and uses the all function. The all function returns True if all elements of the iterable are True.

3. Use the in keyword to check membership of the elements in the check_for list in list_of_words:

[ ]:
```
all('b' in list_of_words for b in check_for)
```

It is indeed elegant and simple to reason about, and this neat trick is very important while dealing with lists. Basically, what we are doing is looping over the first list with the comprehension and then looping over the second list using the for loop. What makes this elegant is how compactly we can represent this complex process. Caution should be taken when using very complex list comprehension—the more complex you make it, the harder it is to read.

Let's look at the next data structure: a queue.

Queue

Apart from stacks, another high-level data structure type that we are interested in is queues. A queue is like a stack, which means that you continue adding elements one by one. With a queue, the reading of elements obeys the First in First Out (FIFO strategy.

We will accomplish this first using list methods and will show you that, for this purpose, they are inefficient. Then, we will learn about the dequeue data structure from the collections module of Python. A queue is a very important data structure. We can think of a scenario on a producer-consumer system design. When doing data wrangling, you will often come across a problem where you must process very big files. One of the ways to deal with this problem is to split the chunk the contents of the file into smaller parts and then push them into a queue while creating small, dedicated worker processes, to read off the queue and process one small chunk at a time. This is a very powerful design, and you can even use it efficiently to design huge multi-node data wrangling pipelines.

Below is list implementation of queue. We use pop(0) to remove the first item from a list.

Exercise 2.22: Queue Using List

[ ]:
```
# Python code to demonstrate Implementing
# Queue using list
queue = ["Goku", "Bezita", "Gohan"]
queue.append("Trunks")
queue.append("Goten")
print(queue)

# Removes the first item
print(queue.pop(0))
print(queue)

# Removes the first item
print(queue.pop(0))

print(queue)
```

Exercise 2.23: Implementing a Queue in Python

In this exercise, we'll implement a queue in Python. We'll use the append function to add elements to the queue and use the pop function to take elements out of the queue. We'll also use the deque data structure and compare it with the queue in order to understand the wall time required to complete the execution of an operation. To do so, perform the following steps:

1. Create a Python queue with the plain list methods. To record the time the append operation in the queue data structure takes, we use the %%time command:

```
[ ]: %%time
     queue = []
     for i in range(0, 100000):
         queue.append(i)
     print("Queue created")
     queue
```

2. If we were to use the pop function to empty the queue and check the items in it:

```
[ ]: for i in range(0, 100000):
         queue.pop(0)
     print("Queue emptied")
```

However, this time, we'll use the %%time magic command while executing the preceding code to see that it takes a while to finish:

```
[ ]: %%time
     for i in range(0, 100000):
         queue.pop(0)
     print("Queue emptied")
     queue
```

In a modern MacBook, with a quad-core processor and 8 GB of RAM, it took around 1.20 seconds to finish. With Windows 10, it took around 2.24 seconds to finish. It takes this amount of time because of the pop(0) operation, which means every time we pop a value from the left of the list (the current 0 index), Python has to rearrange all the other elements of the list by shifting them one space left. Indeed, it is not a very optimized implementation.

3. Implement the same queue using the deque data structure from Python's collections package and perform the append and pop functions on this data structure:

[ ]:
```
%%time
from collections import deque
queue2 = deque()
for i in range(0, 100000):
    queue2.append(i)
print("Queue created")
for i in range(0, 100000):
    queue2.popleft()
print("Queue emptied")
```

With the specialized and optimized queue implementation from Python's standard library, the time that this should take for both the operations is only approximately 27.9 milliseconds. This is a huge improvement on the previous one.

We will end the discussion on data structures here. What we discussed here is just the tip of the iceberg. Data structures are a fascinating subject. There are many other data structures that we did not touch on and that, when used efficiently, c an o ffer en ormous ad ded va lue. We strongly encourage you to explore data structures more. Try to learn about linked lists, trees, graphs, and all the different variations o f t hem a s m uch a s y ou c an; y ou w ill fi nd th ere ar e ma ny similarities between them and you will benefit greatly from studying t hem. N ot o nly d o t hey o ffer th e joy of learning, but they are also the secret mega-weapons in the arsenal of a data practitioner that you can bring out every time you are challenged with a di icult data wrangling job.

Exercise 2.24 : Using Deque

[ ]:
```
# Python code to demonstrate Implementing
# Stack using deque
from collections import deque
queue = deque(["Luffy", "Zorro", "Sanji", "Nami"])
print(queue)
queue.append("Franky")
print(queue)
queue.append("Usop")
print(queue)
print(queue.pop())
print(queue.pop())
print(queue)
```

Basic File Operations in Python

In the previous topic, we investigated a few advanced data structures and also learned neat and useful functional programming methods to manipulate them without side effects. In this topic, we will learn about a few OS-level functions in Python, such as working with files, but these could also include working with printers, and even the internet. We will concentrate mainly on file-related functions and learn how to open a file, read the data line by line or all at once, and finally, how to cleanly close the file we o pened. T he c losing o peration o f a fi le sh ould be do ne ca utiously, which is ignored most of the time by developers. When handling file o perations, we o ften r un i nto very strange and hard-to-track-down bugs because a process opened a file and did not close it properly.

We will apply a few of the techniques we have learned about to a file that we will read to practice our data wrangling skills further.

Exercise 2.25: File Operations

In this exercise, we will learn about the OS module of Python, and we will also look at two very useful ways to write and read environment variables. The power of writing and reading environment variables is often very important when designing and developing data-wrangling pipelines.

The purpose of the OS module is to give you ways to interact with OS-dependent functionalities. In general, it is pretty low-level and most of the functions from there are not useful on a day-to-day basis; however, some are worth learning. os.environ is the collection Python maintains with all the present environment variables in your OS. It gives you the power to create new ones. The os.getenv function gives you the ability to read an environment variable:

1. Import the os module.

```
[ ]: import os
```

2. Set a few environment variables:

```
[ ]: os.environ['MY_KEY'] = "MY_VAL"
     os.getenv('MY_KEY')
```

3. Print the environment variable when it is not set:

```
[ ]: print(os.getenv('MY_KEY_NOT_SET'))
```

4. Print the os environment:

```
[ ]: print(os.environ)
```

After executing the preceding code, you will be able to see that you have successfully printed the value of MY_KEY, and when you tried to print MY_KEY_NOT_SET, it printed None. Therefore, utilizing the OS module, you will be able to set the value of environment variables in your system.

File Handling

In this section, we will learn about how to open a file in Python. We will learn about the different modes that we can use and what they stand for when opening a file. Python has a built-in open function that we will use to open a file. The open function takes a few arguments as input. Among them, the first one, which stands for the name of the file you want to open, is the only one that's mandatory. Everything else has a default value. When you call open, Python uses underlying system-level calls to open a file handler and return it to the caller.

Usually, a file can be opened either for reading or writing. If we open a file in one mode, the other operation is not supported. Whereas reading usually means we start to read from the beginning of an existing file, writing can mean either starting a new file and writing from the beginning or opening an existing file and appending to it.

You can open a file for reading with the command that follows. The path (highlighted would need to be changed based on the location of the file on your system.

[ ]:
```
fd = open("data_temporary_files.txt")
```

We will discuss some more functions in the following section.

This is opened in rt mode (opened for the reading+text mode. You can open the same file in binary mode if you want. To open the file in binary mode, use the rb (read, byte mode:

[ ]:
```
fd = open('AA.txt',"rb")
fd
```

This is how we open a file for writing:

[ ]:
```
fd = open("data_temporary_files.txt ", "w")
fd
```

Exercise 2.26: Opening and Closing a File

In this exercise, we will learn how to close a file after opening it.

We must close a file once we have opened it. A lot of system-level bugs can occur due to a dangling file handler, which means the file is still being modified, even though the application is done using it. Once we close a file, no further operations can be performed on that file using that specific file handler.

1. Open a file in binary mode:

[ ]:
```
fd = open("AA.txt", "rb")
```

2. Close a file using close(:

[ ]:
```
fd.close()
```

Python also gives us a closed flag with the file handler. If we print it before closing, then we will see False, whereas if we print it after closing, then we will see True. If our logic checks whether a file is properly closed or not, then this is the flag we want to use.

The 'With' Statement

In this section, we will learn about the with statement in Python and how we can effectively use it in the context of opening and closing files.

The with command is a compound statement in Python, like if and for, designed to combine multiple lines. Like any compound statement, with also affects the execution of the code enclosed by it. In the case of with, it is used to wrap a block of code in the scope of what we call a Context Manager in Python. A context manager is a convenient way to work with resources and will help avoid forgetting to close the resource. A detailed discussion of context managers is out of the scope of this exercise and this topic in general, but it is sufficient to say that if a context manager is implemented inside the open call for opening a file in Python, it is guaranteed that a close call will automatically be made if we wrap it inside a with statement.

Opening a File Using the with Statement

Open a file using the with statement:

```
[ ]: with open("AA.txt") as fd:
         print(fd.closed)
     print(fd.closed)
```

If we execute the preceding code, we will see that the first print will end up printing False, whereas the second one will print True. This means that as soon as the control goes out of the with block, the file descriptor is automatically closed.

Exercise 2.27: Reading a File Line by Line

In this exercise, we'll read a file line by line. Let's go through the following steps to do so: 1 . Open a file and then read the file line by line and print it as we read it:

```
[ ]: with open("Alice`s Adventures in Wonderland, "\
     "by Lewis Carroll", encoding="utf8") as fd:
         for line in fd:
             print(line)
```

Looking at the preceding code, we can see why it is important. With this short snippet of code, you can even open and read files that are many gigabytes in size, line by line, and without flooding or overrunning the system memory. There is another explicit method in the file descriptor object, called readline, which reads one line at a time from a file.

2. Duplicate the same for loop, just after the first one:

```
[ ]: with open("Alice`s Adventures in Wonderland, "\
     "by Lewis Carroll", encoding="utf8") as fd:
         for line in fd:
             print(line)
         print("Ended first loop")
     for line in fd:
         print(line)
```

Exercise 2.28: Writing to a File

In this exercise, we'll look into file operations by showing you how to read from a dictionary and write to a file. We will write a few lines to a file and read the file:

Let's go through the following steps: 1. Use the write function from the file descriptor object:

```
[ ]: data_dict = {"India": "Delhi", "France": "Paris",\
     "UK": "London", "USA": "Washington"}
     with open("data_temporary_files.txt", "w") as fd:
         for country, capital in data_dict.items():
             fd.write("The capital of {} is {}\n"\
                   .format(country, capital))
```

2. Read the file using the following command:

```
[ ]: with open("data_temporary_files.txt", "r") as fd:
         for line in fd:
             print(line)
```

3. Use the print function to write to a file using the following command:

```
[ ]: data_dict_2 = {"China": "Beijing", "Japan": "Tokyo"}
     with open("data_temporary_files.txt", "a") as fd:
         for country, capital in data_dict_2.items():
             print("The capital of {} is {}"\
                 .format(country, capital), file=fd)
```

4. Read the file using the following command:

```
[ ]: with open("data_temporary_files.txt", "r") as fd:
         for line in fd:
             print(line)
```