2 Data structures

In this chapter we'll learn about the python data structures that are often used or appear while analyzing data.

2.1 Tuple

Tuple is a sequence of python objects, with two key characteristics: (1) the number of objects are fixed, and (2) the objects are immutable, i.e., they cannot be changed.

Tuple can be defined as a sequence of python objects separated by commas, and enclosed in rounded brackets (). For example, below is a tuple containing three integers.

```
tuple_example = (2,7,4)
```

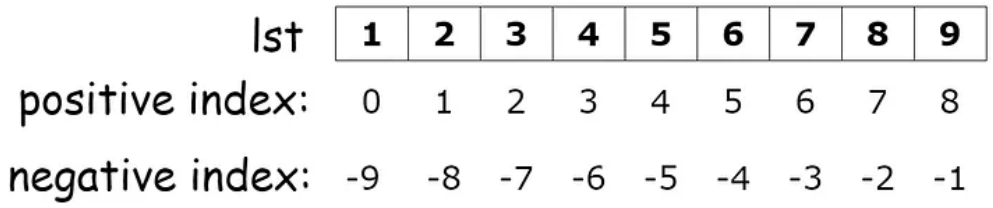
We can check the data type of a python object using the in-built python function *type()*. Let us check the data type of the object *tuple_example*.

```
type(tuple_example)
```

```
tuple
```

2.1.1 Tuple Indexing

Tuple is ordered, meaning you can access specific elements in a list using their index. Indexing in lists includes both positive indexing (starting from 0 for the first element) and negative indexing (starting from -1 for the last element).



Elements of a tuple can be extracted using their index within square brackets. For example the second element of the tuple *tuple_example* can be extracted as follows:

```
tuple_example[1]
```

```
7
```

```
tuple_example[-1]
```

```
4
```

Note that an element of a tuple cannot be modified. For example, consider the following attempt in changing the second element of the tuple *tuple_example*.

```
tuple_example[1] = 8
```

```
---------------------------------------------------------------------
TypeError                                 Traceback (most recent
<ipython-input-6-6ceb38adde52> in <module>
----> 1 tuple_example[1] = 8

TypeError: 'tuple' object does not support item assignment
```

The above code results in an error as tuple elements cannot be modified.

2.1.2 Concatenating tuples

Tuples can be concatenated using the + operator to produce a longer tuple:

```
(2,7,4) + ("another", "tuple") + ("mixed","datatypes",5)
```

```
(2, 7, 4, 'another', 'tuple', 'mixed', 'datatypes', 5)
```

Multiplying a tuple by an integer results in repetition of the tuple:

```
(2,7,"hi") * 3
```

```
(2, 7, 'hi', 2, 7, 'hi', 2, 7, 'hi')
```

2.1.3 Unpacking tuples

If tuples are assigned to an expression containing multiple variables, the tuple will be unpacked and each variable will be assigned a value as per the order in which it appears. See the example below.

```
x,y,z  = (4.5, "this is a string", (("Nested tuple",5)))
```

```
x
```

```
4.5
```

```
y
```

```
'this is a string'
```

```
z
```

```
('Nested tuple', 5)
```

If we are interested in retrieving only some values of the tuple, the expression *_ can be used to discard the other values. Let's say we are interested in retrieving only the first and the last two values of the tuple:

```
x,*_,y,z  = (4.5, "this is a string", (("Nested tuple",5)),"99",
```

```
x
```

```
4.5
```

```
y
```

```
'99'
```

```
z
```

```
99
```

2.1.4 Tuple methods

A couple of useful tuple methods are `count`, which counts the occurrences of an element in the tuple and `index`, which returns the position of the first occurrence of an element in the tuple:

```
tuple_example = (2,5,64,7,2,2)
```

```
tuple_example.count(2)
```

```
3
```

```
tuple_example.index(2)
```

```
0
```

Now that we have an idea about tuple, let us try to think where it can be used.

For which of the following purposes will it be appropriate to use a tuple: (a) storing mean co-ordinates of a city, (b) storing the maximum temperature and humidity of a city, everyday?

both (a) and (b)	(a)
(b)	None

2.2 List

List is a sequence of python objects, with two key characeterisics that differentiates it from tuple: (1) the number of objects are variable, i.e., objects can be added or removed from a list, and (2) the objects are mutable, i.e., they can be changed.

List can be defined as a sequence of python objects separated by commas, and enclosed in square brackets []. For example, below is a list consisting of three integers.

```
list_example = [2,7,4]
```

List indexing works the same way as tuple indexing.

2.2.1 Slicing a list

List slicing is a technique in Python that allows you to extract a portion of a list by specifying a range of indices. It creates a new list containing the elements from the original list within that specified range. List slicing uses the colon : operator to indicate the start, stop, and step values for the slice. The general syntax is: `new_list = original_list[start:stop:step]`

Here's what each part of the slice means: * **start**: The index at which the slice begins (inclusive). If omitted, it starts from the beginning (index 0). * **stop**: The index at which the slice ends (exclusive). If omitted, it goes until the end of the list. * **step**: The interval between elements in the slice. If omitted, it defaults to 1.

```
list_example6 = [4,7,3,5,7,1,5,87,5]
```

Let us extract a slice containing all the elements from the the 3rd position to the 7th position.

```
list_example6[2:7]
```

```
[3, 5, 7, 1, 5]
```

Note that while the element at the `start` index is included, the element with the `stop` index is excluded in the above slice.

If either the `start` or `stop` index is not mentioned, the slicing will be done from the beginning or until the end of the list, respectively.

```
list_example6[:7]
```

```
[4, 7, 3, 5, 7, 1, 5]
```

```
list_example6[2:]
```

```
[3, 5, 7, 1, 5, 87, 5]
```

To slice the list relative to the end, we can use negative indices:

```
list_example6[-4:]
```

```
[1, 5, 87, 5]
```

```
list_example6[-4:-2:]
```

```
[1, 5]
```

An extra colon (':') can be used to slice every nth element of a list.

```
#Selecting every 3rd element of a list
list_example6[::3]
```

```
[4, 5, 5]
```

```
#Selecting every 3rd element of a list from the end
list_example6[::-3]
```

```
[5, 1, 3]
```

```
#Selecting every element of a list from the end or reversing a
list_example6[::-1]
```

```
[5, 87, 5, 1, 7, 5, 3, 7, 4]
```

2.2.2 Adding and removing elements in a list

We can add elements at the end of the list using the *append* method. For example, we append the string 'red' to the list *list_example* below.

```
list_example.append('red')
```

```
list_example
```

```
[2, 7, 4, 'red']
```

Note that the objects of a list or a tuple can be of different datatypes.

An element can be added at a specific location of the list using the *insert* method. For example, if we wish to insert the number 2.32 as the second element of the list *list_example*, we can do it as follows:

```
list_example.insert(1,2.32)
```

```
list_example
```

```
[2, 2.32, 7, 4, 'red']
```

For removing an element from the list, the *pop* and *remove* methods may be used. The *pop* method removes an element at a particular index, while the *remove* method removes the element's first occurence in the list by its value. See the examples below.

Let us say, we need to remove the third element of the list.

```
list_example.pop(2)
```

```
7
```

```
list_example
```

```
[2, 2.32, 4, 'red']
```

Let us say, we need to remove the element 'red'.

```
list_example.remove('red')
```

```
list_example
```

```
[2, 2.32, 4]
```

```
#If there are multiple occurrences of an element in the list, tl
list_example2 = [2,3,2,4,4]
list_example2.remove(2)
list_example2
```

```
[3, 2, 4, 4]
```

For removing multiple elements in a list, either `pop` or `remove` can be used in a `for` loop, or a `for` loop can be used with a condition. See the examples below.

Let's say we need to remove intergers less than 100 from the following list.

```
list_example3 = list(range(95,106))
list_example3
```

```
[95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105]
```

```
#Method 1: For loop with remove, iterating over the elements of
#but updating a copy of the original list
list_example3_filtered = list(list_example3) #
for element in list_example3:
    if element<100:
        list_example3_filtered.remove(element)
print(list_example3_filtered)
```

```
[100, 101, 102, 103, 104, 105]
```

Q1: What's the need to define a new variable `list_example3_filtered` in the above code?

A1: Replace list_example3_filtered with list_example3 and identify the issue. After an element is removed from the list, all the elements that come afterward have their index/position reduced by one. After the elment `95` is removed, `96` is at index `0`, but the for loop will now look at the element at index `1`, which is now `97`. So, iterating over the same list that is being updated in the loop will keep 96 and 98. Using a new list gets rid of the issue by keeping the original list unchanged, so the for-loop iterates over all elements of the original list.

Another method could have been to interate over a copy of the original list and update the original list as shown below.

```
#Method 2: For loop with remove, iterating over the elements of
#but updating the original list
for element in list_example3[:]: #Slicing a list creates a new
    if element<100:
        list_example3.remove(element)
print(list_example3)
```

```
[100, 101, 102, 103, 104, 105]
```

Below is another method that uses a shorthand notation - list comprehension (explained in the next section).

```
#Method 3: For loop with condition in list comprehension
list_example3 = list(range(95,106))
[element for element in list_example3 if element>=100]
```

```
[100, 101, 102, 103, 104, 105]
```

2.2.3 List comprehensions

List comprehensions provide a concise and readable way to create new lists by applying an expression to each item in an iterable (e.g., a list, tuple, or range) and optionally filtering the items based on a condition. They are a powerful and efficient way to generate lists without the need for explicit loops. The basic syntax of a list comprehension is as follows:

```
new_list = [expression for item in iterable if condition]
```

- **expression**: This is the expression that is applied to each item in the iterable. It defines what will be included in the new list.
- **item**: This is a variable that represents each element in the iterable as the comprehension iterates through it.
- **iterable**: This is the source from which the elements are taken. It can be any iterable, such as a list, tuple, range, or other iterable objects.
- **condition (optional)**: This is an optional filter that can be applied to control which items from the iterable are included in the new list. If omitted, all items from the iterable are included.

Example: Create a list that has squares of natural numbers from 5 to 15.

```
sqrt_natural_no_5_15 = [(x**2) for x in range(5,16)]
print(sqrt_natural_no_5_15)
```

```
[25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225]
```

Example: Create a list of tuples, where each tuple consists of a natural number and its square, for natural numbers ranging from 5 to 15.

```
sqrt_natural_no_5_15 = [(x,x**2) for x in range(5,16)]
print(sqrt_natural_no_5_15)
```

```
[(5, 25), (6, 36), (7, 49), (8, 64), (9, 81), (10, 100), (11,
121), (12, 144), (13, 169), (14, 196), (15, 225)]
```

Example: Creating a list of words that start with the letter ‘a’ in a given list of words.

```
words = ['apple', 'banana', 'avocado', 'grape', 'apricot']
a_words = [word for word in words if word.startswith('a')]
```

```
print(a_words)
```

```
['apple', 'avocado', 'apricot']
```

Example: Create a list of even numbers from 1 to 20.

```
even_numbers = [x for x in range(1, 21) if x % 2 == 0]
print(even_numbers)
```

```
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

List comprehensions are not only concise but also considered more Pythonic and often more efficient than using explicit loops for simple operations. They can make your code cleaner and easier to read, especially for operations that transform or filter data in a list.

2.2.4 Practice exercise 1

Below is a list consisting of responses to the question: "At what age do you think you will marry?" from students of the STAT303-1 Fall 2022 class.

```
exp_marriage_age=['24','30','28','29','30','27','26','28','30+'
```

Use list comprehension to:

2.2.4.1

Remove the elements that are not integers - such as *'probably never', '30+'*, etc. What is the length of the new list?

Hint: The built-in python function of the `str` class - isdigit() may be useful to check if the string contains only digits.

Solution:

```
exp_marriage_age_num = [x for x in exp_marriage_age if x.isdigi
print("Length of the new list = ",len(exp_marriage_age_num))
```

```
Length of the new list =  181
```

2.2.4.2

Cap the values greater than 80 to 80, in the clean list obtained in (1). What is the mean age when people expect to marry in the new list?

```
exp_marriage_age_capped = [min(int(x),80) for x in exp_marriage_
print("Mean age when people expect to marry = ", sum(exp_marriag
```

```
Mean age when people expect to marry =  28.955801104972377
```

2.2.4.3

Determine the percentage of people who expect to marry at an age of 30 or more.

```
print("Percentage of people who expect to marry at an age of 30
```

```
Percentage of people who expect to marry at an age of 30 or
more = 37.01657458563536 %
```

2.2.4.4

Redo Q2.2.4.2 using the if-else statement within list comprehension.

2.2.5 Practice exercise 2

Below is a list consisting of responses to the question: “What do you expect your starting salary to be after graduation, to the nearest thousand dollars? (ex: 47000)” from students of the STAT303-1 Fall 2023. class.

```
expected_salary = ['90000', '110000', '100000', '90k', '80000',
```

Clean `expected_salary` using list comprehensions only, and find the mean expected salary.

2.2.6 Concatenating lists

As in tuples, lists can be concatenated using the `+` operator:

```
import time as tm
```

```
list_example4 = [5,'hi',4]
list_example4 = list_example4 + [None,'7',9]
list_example4
```

```
[5, 'hi', 4, None, '7', 9]
```

For adding elements to a list, the `extend` method is preferred over the `+`

operator. This is because the `+` operator creates a new list, while the `extend` method adds elements to an existing list. Thus, the `extend` operator is more memory efficient.

```
list_example4 = [5,'hi',4]
list_example4.extend([None, '7', 9])
list_example4
```

```
[5, 'hi', 4, None, '7', 9]
```

2.2.7 Sorting a list

A list can be sorted using the `sort` method:

```
list_example5 = [6,78,9]
list_example5.sort(reverse=True) #the reverse argument is used
list_example5
```

```
[78, 9, 6]
```

2.2.8 Practice exercise 3

Start with the list [8,9,10]. Do the following:

2.2.8.1

Set the second entry (index 1) to 17

```
L = [8,9,10]
L[1]=17
```

2.2.8.2

Add 4, 5, and 6 to the end of the list

```
L = L+[4,5,6]
```

2.2.8.3

Remove the first entry from the list

```
L.pop(0)
```

8

2.2.8.4

Sort the list

```
L.sort()
```

2.2.8.5

Double the list (concatenate the list to itself)

```
L=L+L
```

2.2.8.6

Insert 25 at index 3

The final list should equal [4,5,6,25,10,17,4,5,6,10,17]

```
L.insert(3,25)
L
```

```
[4, 5, 6, 25, 10, 17, 4, 5, 6, 10, 17]
```

Now that we have an idea about lists, let us try to think where it can be used.

For which of the following purposes will it be appropriate to use a list: (a) storing the names of all countries, (b) storing the population of all countries each year?

(a)	both (a) and (b)
None	(b)

2.2.9 Other list operations

You can test whether a list contains a value using the `in` operator.

```
list_example6
```

```
[4, 7, 3, 5, 7, 1, 5, 87, 5]
```

```
6 in list_example6
```

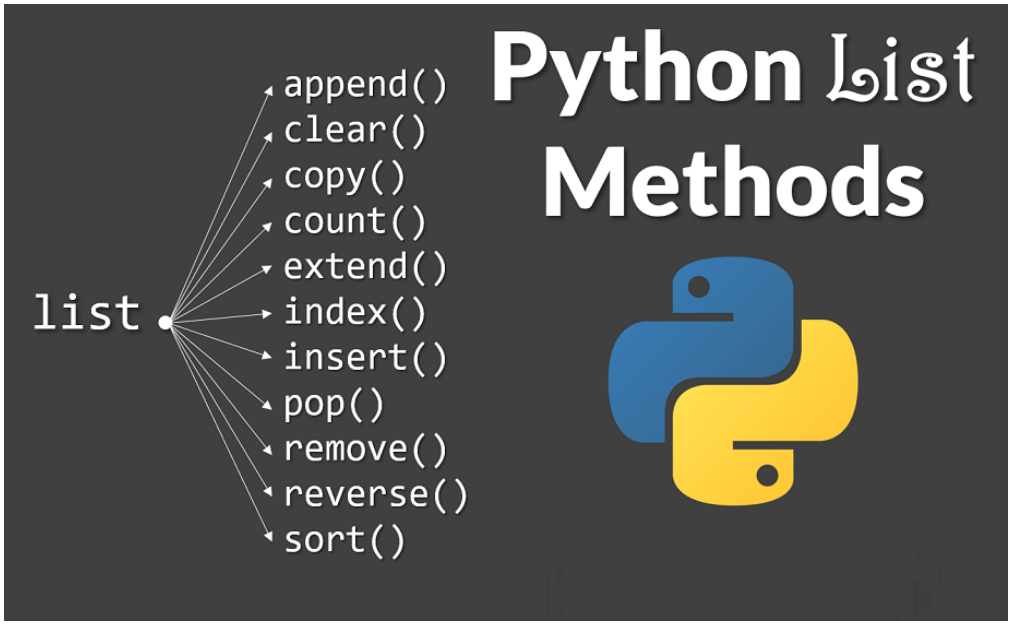
```
False
```

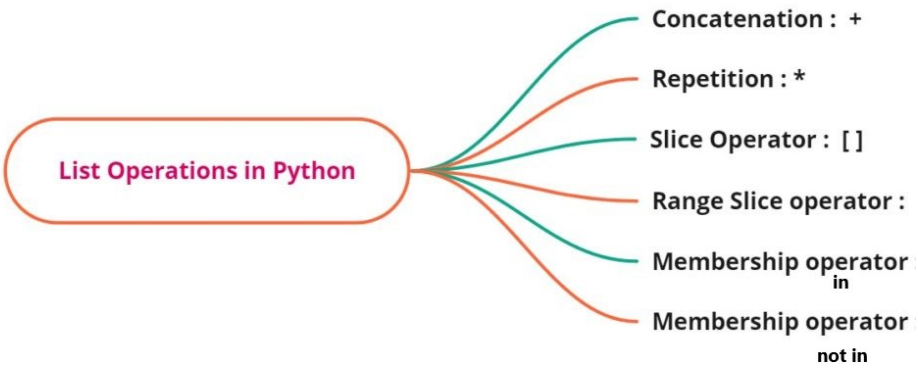
```
7 in list_example6
```

```
True
```

2.2.10 Lists: methods

Just like strings, there are several in-built methods to manipulate a list. However, unlike strings, most list methods modify the original list rather than returning a new one. Here are some common list operations:





List operations in Python (itvoyagers.in)

miro

2.2.11 Lists vs tuples

Now that we have learned about lists and tuples, let us compare them.

Q2: A list seems to be much more flexible than tuple, and can replace a tuple almost everywhere. Then why use tuple at all?

A2: The additional flexibility of a list comes at the cost of efficiency. Some of the advantages of a tuple over a list are as follows:

1. Since a list can be extended, space is over-allocated when creating a list. A tuple takes less storage space as compared to a list of the same length.
2. Tuples are not copied. If a tuple is assigned to another tuple, both tuples point to the same memory location. However, if a list is assigned to another list, a new list is created consuming the same memory space as the original list.
3. Tuples refer to their element directly, while in a list, there is an extra layer of pointers that refers to their elements. Thus it is faster to retrieve elements from a tuple.

The examples below illustrate the above advantages of a tuple.

```
#Example showing tuples take less storage space than lists for
tuple_ex = (1, 2, 'Obama')
list_ex = [1, 2, 'Obama']
print("Space taken by tuple =",tuple_ex.__sizeof__()," bytes")
print("Space taken by list =",list_ex.__sizeof__()," bytes")
```

```
Space taken by tuple = 48  bytes
Space taken by list = 64  bytes
```

```
#Examples showing that a tuples are not copied, while lists can
tuple_copy = tuple(tuple_ex)
print("Is tuple_copy same as tuple_ex?", tuple_ex is tuple_copy
list_copy = list(list_ex)
print("Is list_copy same as list_ex?",list_ex is list_copy)
```

```
Is tuple_copy same as tuple_ex? True
Is list_copy same as list_ex? False
```

```
#Examples showing tuples takes lesser time to retrieve elements
import time as tm
tt = tm.time()
list_ex = list(range(1000000)) #List containinig whole numbers
a=(list_ex[::-2])
print("Time take to retrieve every 2nd element from a list = ",

tt = tm.time()
tuple_ex = tuple(range(1000000)) #tuple containinig whole numbe
a=(tuple_ex[::-2])
print("Time take to retrieve every 2nd element from a tuple = "
```

```
Time take to retrieve every 2nd element from a list =
0.03579902648925781
Time take to retrieve every 2nd element from a tuple =
0.02684164047241211
```

2.3 Dictionary

Unlike lists and tuples, a dictionary is an unordered collection of items. Each item stored in a dictionary has a key and value. You can use a key to retrieve the corresponding value from the dictionary. Dictionaries have the type `dict`.

Dictionaries are often used to store many pieces of information e.g. details about a person, in a single variable. Dictionaries are created by enclosing key-value pairs within braces or curly brackets `{` and `}`, colons to separate keys and values, and commas to separate elements of a dictionary.

The dictionary keys and values are python objects. While values can be any

python object, keys need to be immutable python objects, like strings, integers, tuples, etc. Thus, a list can be a value, but not a key, as elements of list can be changed.

```
dict_example = {'USA':'Joe Biden', 'India':'Narendra Modi', 'Ch
```

Elements of a dictionary can be retrieved by using the corresponding key.

```
dict_example['India']
```

```
'Narendra Modi'
```

2.3.1 Viewing keys and values

```
dict_example.keys()
```

```
dict_keys(['USA', 'India', 'China'])
```

```
dict_example.values()
```

```
dict_values(['Joe Biden', 'Narendra Modi', 'Xi Jinping'])
```

```
dict_example.items()
```

```
dict_items([('USA', 'Joe Biden'), ('India', 'Narendra Modi'),
('China', 'Xi Jinping')])
```

The results of `keys`, `values`, and `items` look like lists. However, they don't support the indexing operator `[]` for retrieving elements.

```
dict_example.items()[1]
```

```
---------------------------------------------------------------
TypeError                                 Traceback (most recent
Cell In[65], line 1
----> 1 dict_example.items()[1]

TypeError: 'dict_items' object is not subscriptable
```

2.3.2 Adding and removing elements in a dictionary

New elements can be added to a dictionary by defining a key in square brackets and assiging it to a value:

```
dict_example['Japan'] = 'Fumio Kishida'
dict_example['Countries'] = 4
dict_example
```

```
{'USA': 'Joe Biden',
 'India': 'Narendra Modi',
 'China': 'Xi Jinping',
 'Japan': 'Fumio Kishida',
 'Countries': 4}
```

Elements can be removed from the dictionary using the `del` method or the `pop` method:

```
#Removing the element having key as 'Countries'
del dict_example['Countries']
```

```
dict_example
```

```
{'USA': 'Joe Biden',
 'India': 'Narendra Modi',
 'China': 'Xi Jinping',
 'Japan': 'Fumio Kishida'}
```

```
#Removing the element having key as 'USA'
dict_example.pop('USA')
```

```
'Joe Biden'
```

```
dict_example
```

```
{'India': 'Narendra Modi', 'China': 'Xi Jinping', 'Japan':
'Fumio Kishida'}
```

New elements can be added, and values of exisiting keys can be changed using the `update` method:

```
dict_example = {'USA':'Joe Biden', 'India':'Narendra Modi', 'Ch
```

```
dict_example
```

```
{'USA': 'Joe Biden',
 'India': 'Narendra Modi',
 'China': 'Xi Jinping',
 'Countries': 3}
```

```
dict_example.update({'Countries':4, 'Japan':'Fumio Kishida'})
```

```
dict_example
```

```
{'USA': 'Joe Biden',
 'India': 'Narendra Modi',
 'China': 'Xi Jinping',
 'Countries': 4,
 'Japan': 'Fumio Kishida'}
```

2.3.3 Iterating over elements of a dictionary

The items() attribute of a dictionary can be used to iterate over elements of a dictionary.

```
for key,value in dict_example.items():
    print("The Head of State of",key,"is",value)
```

```
The Head of State of USA is Joe Biden
The Head of State of India is Narendra Modi
The Head of State of China is Xi Jinping
The Head of State of Countries is 4
The Head of State of Japan is Fumio Kishida
```

2.3.4 Practice exercise 4

The GDP per capita of USA for most years from 1960 to 2021 is given by the dictionary `D` given in the code cell below.

Find:

1. The GDP per capita in 2015
2. The GDP per capita of 2014 is missing. Update the dictionary to include the GDP per capita of 2014 as the average of the GDP per capita of 2013 and 2015.
3. Impute the GDP per capita of other missing years in the same manner as in (2), i.e., as the average GDP per capita of the previous year and the next year.

Note that the GDP per capita is not missing for any two consecutive years.

4. Print the years and the imputed GDP per capita for the years having a missing value of GDP per capita in (3).

```
D = {'1960':3007,'1961':3067,'1962':3244,'1963':3375,'1964':357
```

Solution:

```
print("GDP per capita in 2015 =", D['2015'])
D['2014'] = (D['2013']+D['2015'])/2
for i in range(1960,2021):
    if str(i) not in D.keys():
        D[str(i)] = (D[str(i-1)]+D[str(i+1)])/2
        print("Imputed GDP per capita for the year",i,"is $",D[
```

```
GDP per capita in 2015 = 56763
Imputed GDP per capita for the year 1969 is $ 4965.0
Imputed GDP per capita for the year 1977 is $ 9578.5
Imputed GDP per capita for the year 1999 is $ 34592.0
```

2.4 Functions

If an algorithm or block of code is being used several times in a code, then it can be separately defined as a function. This makes the code more organized and readable. For example, let us define a function that prints prime numbers between `a` and `b`, and returns the number of prime numbers found.

```
#Function definition
def prime_numbers (a,b=100):
    num_prime_nos = 0

    #Iterating over all numbers between a and b
    for i in range(a,b):
        num_divisors=0

        #Checking if the ith number has any factors
        for j in range(2, i):
            if i%j == 0:
                num_divisors=1;break;

        #If there are no factors, then printing and counting th
        if num_divisors==0:
            print(i)
```

```
            num_prime_nos = num_prime_nos+1

    #Return count of the number of prime numbers
    return num_prime_nos
```

In the above function, the keyword `def` is used to define the function, `prime_numbers` is the name of the function, `a` and `b` are the arguments that the function uses to compute the output.

Let us use the defined function to print and count the prime numbers between 40 and 60.

```
#Printing prime numbers between 40 and 60
num_prime_nos_found = prime_numbers(40,60)
```

```
41
43
47
53
59
```

```
num_prime_nos_found
```

```
5
```

If the user calls the function without specifying the value of the argument `b`, then it will take the default value of `100`, as mentioned in the function definition. However, for the argument `a`, the user will need to specify a value, as there is no value defined as a default value in the function definition.

2.4.1 Global and local variables with respect to a function

A variable defined within a function is local to that function, while a variable defined outside the function is global to that function. In case a variable with the same name is defined both outside and inside a function, it will refer to its global value outside the function and local value within the function.

The example below shows a variable with the name `var` referring to its local value when called within the function, and global value when called outside the function.

```
var = 5
def sample_function(var):
```

```
    print("Local value of 'var' within 'sample_function()'= ",va

sample_function(4)
print("Global value of 'var' outside 'sample_function()' = ",va
```

```
Local value of 'var' within 'sample_function()'=  4
Global value of 'var' outside 'sample_function()' =  5
```

2.4.2 Practice exercise 5

The object `deck` defined below corresponds to a deck of cards. Estimate the probablity that a five card hand will be a flush, as follows:

1. Write a function that accepts a hand of 5 cards as argument, and returns whether the hand is a flush or not.
2. Randomly pull a hand of 5 cards from the deck. Call the function developed in (1) to determine if the hand is a flush.
3. Repeat (2) 10,000 times.
4. Estimate the probability of the hand being a flush from the results of the 10,000 simulations.

You may use the function shuffle() from the `random` library to shuffle the deck everytime before pulling a hand of 5 cards.

```
deck = [{'value':i, 'suit':c}
for c in ['spades', 'clubs', 'hearts', 'diamonds']
for i in range(2,15)]
```

Solution:

```
import random as rm

#Function to check if a 5-card hand is a flush
def chck_flush(hands):

    #Assuming that the hand is a flush, before checking the car
    yes_flush =1

    #Storing the suit of the first card in 'first_suit'
    first_suit = hands[0]['suit']

    #Iterating over the remaining 4 cards of the hand
    for j in range(1,len(hands)):

        #If the suit of any of the cards does not match the sui
```

```
        if first_suit!=hands[j]['suit']:
            yes_flush = 0;

            #As soon as a card with a different suit is found,
            break;
    return yes_flush

flush=0
for i in range(10000):

    #Shuffling the deck
    rm.shuffle(deck)

    #Picking out the first 5 cards of the deck as a hand and ch
    #If the hand is a flush it is counted
    flush=flush+chck_flush(deck[0:5])

print("Probability of obtaining a flush=", 100*(flush/10000),"%
```

```
Probability of obtaining a flush= 0.18 %
```

2.5 Practice exercise 6

The code cell below defines an object having the nutrition information of drinks in starbucks. Assume that the manner in which the information is structured is consistent throughout the object.

```
starbucks_drinks_nutrition={'Cool Lime Starbucks Refreshers™ Be
```

Use the object above to answer the following questions:

2.5.1

What is the datatype of the object?

```
print("Datatype=",type(starbucks_drinks_nutrition))
```

```
Datatype= <class 'dict'>
```

2.5.1.1

If the object in (1) is a dictonary, what is the datatype of the values of the dictionary?

```
print("Datatype=",type(starbucks_drinks_nutrition[list(starbucks
```

```
Datatype= <class 'list'>
```

2.5.1.2

If the object in (1) is a dictonary, what is the datatype of the elements within the values of the dictionary?

```
print("Datatype=",type(starbucks_drinks_nutrition[list(starbucks
```

```
Datatype= <class 'dict'>
```

2.5.1.3

How many calories are there in `Iced Coffee`?

```
print("Calories = ",starbucks_drinks_nutrition['Iced Coffee'][0
```

```
Calories =  5
```

2.5.1.4

Which drink(s) have the highest amount of protein in them, and what is that protein amount?

```
#Defining an empty dictionary that will be used to store the pro
protein={}

for key,value in starbucks_drinks_nutrition.items():
    for nutrition in value:
        if nutrition['Nutrition_type']=='Protein':
            protein[key]=(nutrition['value'])

#Using dictionary comprehension to find the key-value pair havi
{key:value for key, value in protein.items() if value == max(pro
```

```
{'Starbucks® Doubleshot Protein Dark Chocolate': 20,
 'Starbucks® Doubleshot Protein Vanilla': 20,
 'Chocolate Smoothie': 20}
```

2.5.1.5

Which drink(s) have a fat content of more than 10g, and what is their fat content?

```
#Defining an empty dictionary that will be used to store the fa
fat={}
for key,value in starbucks_drinks_nutrition.items():
    for nutrition in value:
        if nutrition['Nutrition_type']=='Fat':
            fat[key]=(nutrition['value'])

#Using dictionary comprehension to find the key-value pair havi
{key:value for key, value in fat.items() if value>=10}
```

```
{'Starbucks® Signature Hot Chocolate': 26.0, 'White Chocolate
Mocha': 11.0}
```

2.5.1.6

Answer Q2.5.1.5 using only dictionary comprehension.