**Instructions**: You must read the material and create an outline of the topics in your OWN words.  Do not copy the text from the tutorials into your notes. Make sure your outline contains notes for each subsection of the reading assignment. Thoroughly cover each topic to show you have a firm understanding of the programming concept or construct.

| **Ques** | **NOTES:** |
| --- | --- |
| List  Python – Access List Items  Python - Loop Lists  Python Program to Slice Lists  Looping  Python  while Loop | Sample list:  mylist = [“apple”, “banana”, “cherry”]  Lists are used to store multiple items in a single variable like the one above.  In addition to the list (data type), Python has three other containers used to store the collection of data such as Tuple, Set, and Dictionary, all with different qualities and usage.  Lists are created using square brackets:  thislist = ["apple", "banana", "cherry"] print(thislist)  **List Items**  They are ordered, changeable, and allow duplicate values.  They are also indexed, with the first starting from index [0], and the second item with index [1] etc.  **Ordered**  This means that the items in a list have a defined order, and that order will not change.  When adding to a list, the new items will be placed at the end of the list.  Note: There are some list methods that will change the order, but in general, the order of the items will not change.  **Changeable**  A list is changeable, meaning that we can change, add, and remove items from a list after it has been created.  **Allow Duplicates**  The advantage of being indexed allows the list to have duplicated values with different indexes.  thislist = ["apple", "banana", "cherry", "apple", "cherry"] print(thislist)  **List Length**  To get the length of a list we use the len() function:  thislist = ["apple", "banana", "cherry"] print(len(thislist))  List Items – Data Types  List items can be of any type:  String, int, and Boolean data types:  list1 = ["apple", "banana", "cherry"] list2 = [1, 5, 7, 9, 3] list3 = [True, False, False]  A list can contain different data types:  list1 = ["abc", 34, True, 40, "male"]  type()  Lists are defined as objects with the data type ‘list’:  <class ‘list’>  **To get the data type:**  mylist = ["apple", "banana", "cherry"] print(type(mylist))  The list() Constructor  It is also possible to use the list() constructor when creating a new list.  thislist = list(("apple", "banana", "cherry")) # note the double round-brackets print(thislist)  **Python Collection (Arrays)**  There are four collection data types in the Python programming language:   * **List** is a collection which is ordered and changeable. Allows duplicate members. * [**Tuple**](https://www.w3schools.com/python/python_tuples.asp) is a collection which is ordered and unchangeable. Allows duplicate members. * [**Set**](https://www.w3schools.com/python/python_sets.asp) is a collection which is unordered, unchangeable\*, and unindexed. No duplicate members. * [**Dictionary**](https://www.w3schools.com/python/python_dictionaries.asp) is a collection which is ordered\*\* and changeable. No duplicate members.   **Access Items**  Items in a list are indexed and you can access them by referring to the index number:  thislist = ["apple", "banana", "cherry"] print(thislist[1])  The above will return the second item from the list as it has an index of [1].  **Negative Indexing**  Here we start from the end of the list to access items  “-1” refers to the last item, “-2” refers to the second last item etc.  Example:  thislist = ["apple", "banana", "cherry"] print(thislist[-1])  The above returns “cherry” as this is the last item.  **Range of Indexes**  Here we use a range of indexes to specify where to start and where to end the range. When specifying a range, the return type value will be a new list with the specified items.  thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"] print(thislist[2:5])  This returns the third, fourth, and fifth items.  **Note:** The search will start at index 2 (included) and end at index 5 (not included).  By leaving out the start value, the range will start at the first item:  thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"] print(thislist[:4])  And by leaving out the end value, the range will go on to the end of the list:  thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"] print(thislist[2:])  Basically, using range of indexes has the following format:  List(start:stop:step)  **Range of Negative Indexes**  You specify negative indexes if you want to start the search from the end of the list:  thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"] print(thislist[-4:-1])  **Check if Item Exists**  To check if an item exists in a list we use the “in” keyword:  thislist = ["apple", "banana", "cherry"] if "apple" in thislist:   print("Yes, 'apple' is in the fruits list")  **Loop Through a List**  You can loop over a list of items using a “for” loop:  thislist = ["apple", "banana", "cherry"] for x in thislist:   print(x)  **Loop Through the Index Numbers**  Looping can also be done by referring to the index number of the list. We use the “range()” and “len()” functions to create a suitable iterable.  thislist = ["apple", "banana", "cherry"] for i in range(len(thislist)):   print(thislist[i])  The iterable created above is [0, 1, 2]  **Using a While Loop**  We can loop through a list of items by using a while loop.  Use the “len()” function to determine the length of the list, and starting at 0, we loop through the list by referring to their indexes, but we should remember to increase the index by 1 after each iteration.  thislist = ["apple", "banana", "cherry"] i = 0 while i < len(thislist):   print(thislist[i])   i = i + 1  **Looping Using List Comprehension**  List Comprehensions are the shortest syntax used for looping over a list and a shorthand for the “for” loop:  thislist = ["apple", "banana", "cherry"] [print(x) for x in thislist]  The format for list slicing is [start:stop:step]   * **start** is the index of the list where slicing starts. * **stop** is the index of the list where slicing ends. * **step** allows you to select **nth** item within the range **start** to **stop**.   List slicing works like the Python slice() function:  Slice:  This function returns a slice object that is used to slice any sequence (string, tuple, list, range, or bytes).  Example  text = 'Python Programing'  # get slice object to slice Python  sliced\_text = slice(6)  print(text[sliced\_text])  # Output: Python  Slice() Syntax  Slice(start, stop, step)  List Contd.  Get all the items  my\_list = [1, 2, 3, 4, 5]  print(my\_list[:])  Output:  [1, 2, 3, 4, 5]  my\_list = [1, 2, 3, 4, 5]  print(my\_list[2:])  Output:  [3, 4, 5]  Get all the Items Before a Specific Position  my\_list = [1, 2, 3, 4, 5]  print(my\_list[:2])  [1, 2]  Get all the Items from One Position to Another Position  my\_list = [1, 2, 3, 4, 5]  print(my\_list[2:4])  [3, 4]  Get the Items at Specified Intervals  my\_list = [1, 2, 3, 4, 5]  print(my\_list[::2])  [1, 3, 5]  You can use negative indexing to start from the last item:  my\_list = [1, 2, 3, 4, 5]  print(my\_list[::-2])  [5, 3, 1]  If you want the items from one position to another, you can mention them from “start” to “stop.”  my\_list = [1, 2, 3, 4, 5]  print(my\_list[1:4:2])  [2, 4]  Computers are designed to perform repetitive tasks and can perform millions of calculations per second.  **Two Types**  In computer programming, there are two types of loops: indefinite loops, which repeat till a condition is met, and definite loops, which repeat a specified number of times.  **Indefinite Loops**  These are called “while loops” in programming. The first line here is a conditional statement that evaluates to True or False (known as a Boolean expression). If the expression evaluates to True, the program execution enters the body of the loop (called the block), to execute any other program statements. Blocks begin with an indentation, and code within the block maintains that indentation. The block ends when the indentation ends.  Example code:  x = 0  while (x<5):  print(“Python rocks!”)  x += 1  One should be very careful when creating while loops as new programmer unintentionally creates indefinite while loops in error.  **Application of Indefinite Loops: Continuous Play**  These types of loops are useful in situations where we want the user to continue the process like say in a game. In Python an indefinite loop is implemented using a while loop. Here we use a “condition” to determine if the body of the while loop is executed or skipped.  Let’s use an example to illustrate this:  play = True # set Boolean flag to True  while(play):  print(“Playing the game…”)  answer = input(“would you like to play again (y/n)? “)  if(answer.upper() == “N”):  play = False  print(“Thank you for playing!!”)  The flag here is the variable play initially set to True, allowing the program to enter the loop. We now ask the user if he/she wishes to continue, and depending on the response, we set the value of play to False or leave it as it was. If the value remains True, the game continues, or it ends.  In the code, we make use of the upper() function to translate user input to upper case so we can easily compare it, and again, the expected inputs are “N” or “n”. Any other response won’t be considered as valid.  **Definite Loop**  Before working with this type of loop, we need to look at a classification term called “iterable.” Iterables are countable data types, and in Python, we have five such types.  Strings:  Since we can count the number of letters therein, it is called an iterable.  List:  These contain elements of different types, start with an index of 0, and can be counted.  Tuple:  This is a special type of list that, though similar to a list, cannot be changed (immutable) once created.  Dictionary:  This collection of items has a key:value pair structure, which is countable.  Range:  This type of object generates a list, and the syntax is range(start, end, increment). For example, range(2,100,2) will produce a list from 2 to 98, jumping by twos [2,4,6 ...94, 96, 98]. *But notice the ending value is never reached!* That is an important characteristic of ranges.  Syntax for creating a for loop:  myList = [2, 4, 6, 8, 10]  for i in myList:  print(“Hello, World!”)  From the above example, Line 2 starts off with the required keyword **'for,'** then names an arbitrary index variable (*here called****i***), which keeps track of the current iterable element pointed to in the loop's progression. An **iterable** (here *myList*) is used to dictate the duration of the loop: if it has four elements, the loop will repeat four times. Finally, the **colon** introduces the **loop body** (block), which describes the code executed during each trip through the loop.  **The Usefulness of Range**  Though all iterables are functionally the same, some behave quite differently from others. The range iterable creates a list based on the arguments supplied to the range function.  Although the range function was described as having three parameters, two of those have defaults (the default start is 0, and the default increment is 1). So the simple statement range(10) will produce a list of ten elements, [0-9].  This statement, range(len(list)), will generate a list of numbers exactly corresponding to the indexes of each of the elements in my list, no matter its size.  This characteristic of lists makes it possible to process parallel lists: multiple lists whose corresponding elements are related.  For example the names list ["John," "Paul," "George," "Ringo"] corresponds with the year born list [1940,1942,1943,1940]. If the lists are parallel, that means that John (names[0]) was born in 1940 (yearborn[0]), and Paul (names[1]) was born in 1942 (yearborn[1]), etc.  This is used in Python to repeat a block of code until a certain condition is met.  number = 1  while number <= 3:  print(number)  number = number + 1  **Output**  1  2  3  While Loop Syntax  While condition:  # body of while loop  1. The while loop evaluates the condition.  2. If the condition is true, the body of the while loop is executed. The condition is evaluated again.  3. This process continues until the condition is False.  4. Once the condition evaluates to False, the loop terminates. |
|  | **Flowchart of Python while Loop** |
| Python For Loop | Example: Python while Loop  # Calculate the sum of numbers until user enters 0  number = int(input('Enter a number: '))  total = 0  # iterate until the user enters 0  while number != 0:  total += number  number = int(input('Enter a number: '))  print('The sum is', total)  **Output**  Enter a number: 3  Enter a number: 2  Enter a number: 1  Enter a number: -4  Enter a number: 0  The sum is 2  Here is how the above program works:   1. It asks the user to enter a number. 2. If the user enters a number other than **0**, it adds the number to the total and asks the user to enter a number again. 3. If the user enters **0**, the loop terminates and the program displays the total.   **Infinite while Loop**  Here, if the while loop condition is always True, the loop runs for infinite times, forming an infinite while loop. For example  age = 32  # the test condition is always True  while age > 18:  print('You can vote')  **Output**  You can vote  You can vote  You can vote  .  .  .  The above example is equivalent to:  age = 32    # the test condition is always True  while True:  print('You can vote')  We use a “for” loop to iterate over sequences such as lists, strings, tuples, etc.  languages = ['Swift', 'Python', 'Go']  # access elements of the list one by one  for i in languages:  print(i)  **Output**  Swift  Python  Go  In the above example, we created a list of languages. As the list has 3 elements, the loop iterates **3** times.  The value of i is   * Swift in the first iteration. * Python in the second iteration. * Go in the third iteration.   For loop Syntax  for val in sequence:  # statement(s)  **Flowchart of Python for Loop** |
| The Basic of Python For Loops: A Tutorial | **Example: Loop Through a String**  language = 'Python'  # iterate over each character in language  for x in language:  print(x)  **Output**  P  y  t  h  o  n  **For Loop with Python range()**  Range function returns a sequence of numbers.  Values = range(4)  This will return 0, 1, 2, 3  Iterating over this range we have this:  # iterate from i = 0 to i = 3  for i in range(4):  print(i)  **Output**  0  1  2  3  This is how the above code works:   |  |  |  |  | | --- | --- | --- | --- | | Iteration | Value of i | print(i) | Last item in sequence? | | 1st | 0 | Prints 0 | No | | 2nd | 1 | Prints 1 | No | | 3rd | 2 | Prints 2 | No | | 4th | 3 | Prints 3 | Yes The loop terminates. |   **What Are Foor Loops?**  These Python code blocks are used to perform various tasks on each item in an iterable such as a string, list, tuple, etc. It uses a predefined number of steps to carry out these tasks and checks to ensure no step is skipped and no extra step is added.  An example:  our\_list = ['Lily', 'Brad', 'Fatima', 'Zining']  for name in our\_list:  print(name)  Output:  Lily  Brad  Fatima  Zining  Note that we used the variable called name in the code even though we hadn’t pre-defined it. So Python will interpret any variable we use in that spot as a reference to each list item in any sequence we are working on.  So we could have also used “x” like so:  for x in our\_list:  print(x)  The same can be done with a string:  for letter in 'Lily':  print(letter)  L  i  l  y  **Using For Loops with Lists of Lists**  List of lists are used to represent tabular data in Python and the for loop is used to work in these data:   |  |  |  | | --- | --- | --- | | **vehicle** | **range** | **price** | | Tesla Model 3 LR | 310 | 49900 | | Hyundai Ioniq EV | 124 | 30315 | | Chevy Bolt | 238 | 36620 |   The above table can be represented like so:  ev\_data = [['vehicle', 'range', 'price'],  ['Tesla Model 3 LR', '310', '49900'],  ['Hyundai Ioniq EV', '124', '30315'],  ['Chevy Bolt', '238', '36620']]  First thing we do here is to convert the ranges into int and reassign them to their former slots  for row in ev\_data[1:]: # loop through each row in ev\_data starting with row 2 (index 1)  ev\_range = row[1] # each car's range is found in column 2 (index 1)  ev\_range = int(ev\_range) # convert each range number from a string to an integer  row[1] = ev\_range # assign range, which is now an integer, back to index 1 in each row  print(ev\_data)  We start processing from the position [1:], as the [0] is the table header information row. We then picked the second item (range) from each subsequent list item, converted it to int, and reassigned it to the original position.  Output:  [['vehicle', 'range', 'price'], ['Tesla Model 3 LR', 310, '49900'], ['Hyundai Ioniq EV', 124, '30315'], ['Chevy Bolt', 238, '36620']]  We now move on to process the data for total range and total number of cars with the below code:  total\_range = 0 # create a variable to store the total range number  for row in ev\_data[1:]: # loop through each row in ev\_data starting with row 2 (index 1)  ev\_range = row[1] # each car's range is found in column 2 (index 1)  total\_range += ev\_range # add this number to the number stored in total\_range  number\_of\_cars = len(ev\_data[1:]) # calculate the length of our list, minus the header row  print(total\_range / number\_of\_cars) # print the average range  We can also use nested If Else statements and even other loops, within our for loop:  long\_range\_car\_list = [] # creating a new list to store our long range car data  for row in ev\_data[1:]: # iterate through ev\_data, skipping the header row  ev\_range = row[1] # assign the range number, which is at index 1 in the row, to the range variable  if ev\_range > 200: # append the whole row to long-range list if range is higher than 200  long\_range\_car\_list.append(row)  print(long\_range\_car\_list)  We use a nested if statement to find and append to an empty list any cars with a range greater than 200 miles.  Output:  [['Tesla Model 3 LR', 310, 49900], ['Chevy Bolt', 238, 36620]]  **Other Useful Techniques: Range, Break, and Continue**  Range:  For loops can be used together with Python’s range() function to iterate through each number in a specified range.  For x in range(5, 9):  Print(x)  Output:  5  6  7  8  Python doesn’t include the maximum value of a range in the range count, which is why the number 9 doesn’t appear above. To count up to 9 (inclusive), we’d need to change range(5,9) to range(5, 10)  If we specified a single number in the range() function then Python take that as the maximum value, and assign a default minimum value of zero:  for x in range(3):  print(x)  Output  0  1  2  A third argument can be added to range() function to specify that you’d like to use increments when counting:  for x in range(0,9,3):  print(x)  0  3  9  **Break**  When we need to stop the loop if a certain condition is met, we use the “break” statement; this helps us to break out of the loop before its conclusion.  for name in our\_list:  break  print(name)  We never get to print any name as the break statement prevents this from happening. To print the name or any name we can adjust the code as follows:  for name in out\_list:  if name == ‘Zinning’:  break  print(name)  Lily  Brad  Fatima   1. Python checks to see if the first name is 'Zining'. It *isn't*, so it continues executing the code below our if statement, and prints the first name. 2. Python checks to see if the second name is 'Zining'. It *isn't*, so it continues executing the code below our if statement, and prints the second name. 3. Python checks to see if the third name is 'Zining'. It *isn't*, so it continues executing the code below our if statement, and prints the third name. 4. Python checks to see if the fourth name is 'Zining'. It *is*, so break is executed and the for loop ends.   Adding a “break” statement to the code for car data previous we get:  long\_range\_car\_list = [] # creating our empty long-range car list again  for row in ev\_data[1:]: # iterate through ev\_data as before looking for cars with a range > 200  ev\_range = row[1]  if ev\_range > 200:  long\_range\_car\_list.append(row)  if 'Tesla' in row[0]: # but if 'Tesla' appears in the vehicle column, end the loop  break  print(long\_range\_car\_list)  Output:  [['Tesla Model 3 LR', 310, 49900]]  **Continue**  When Python encounters a “continue” statement, it skips an iteration of the loop and moves on to the next item.  for name in our\_list:  if name == “Brad”:  continue  print(name)  Lily  Fatima  Zining  Brad was skipped from the above code as demanded by the presence of the continue statement.  In summary:   * break ends the loop *entirely*. When Python executes break, the for loop is over. * continue ends a *specific iteration* of the loop and moves to the next item in the list. When Python executes continue it moves immediately to the next loop iteration, but it does not end the loop entirely.   Adding the “continue” statement to our car code:  short\_range\_car\_list = [] # creating our empty short-range car list  for row in ev\_data[1:]: # iterate through ev\_data as before  ev\_range = row[1]  if ev\_range > 200: # if the car has a range of > 200  continue # end the loop here; do not execute the code below, continue to the next row  short\_range\_car\_list.append(row) # append the row to our short-range car list  print(short\_range\_car\_list) |

I learned about Python repetition and how data structures like Lists, Tuples, Dictionaries, Strings, and Ranges are used to repeat tasks in Python. I also looked at how to combine these data structures in processing data and the advantages one has over the other. I also looked at “continue” and “break” statements, what they are used for, and how they are used. Looping techniques using “for” and “while” loops were treated, and the differences between them were considered using examples. Finally, I walked through a simple tutorial that showed me how to use these Python objects.