Introduction to Programming and Problem Solving

Week 9: List

Lecturer: Lemi Agrey Oliver

Department of Information Technology

Kumi University

Content

- Introduction list
- Creating Lists
- Accessing the Elements of a List
- Negative List Indices
- List Slicing [Start: end]
- List Slicing with Step Size,
- Python Inbuilt Functions for Lists, The List Operator,
- List Comprehensions, List Methods, List and Strings, splitting a String in List, Passing List to a Function, Returning List from a Function

Introduction to List

- Just like a string, a list is also a series of values in Python.[1]
- List in python is a versatile and commonly used data structure that allows you to store and manipulate collections of items.
- Lists are ordered and mutable, which means you can change their contents by adding, removing, or modifying elements.
- Each element in a list is identified by its position or index, starting from 0 for the first element.

Characteristics of a list

- Mutable: Lists are mutable, meaning you can change their contents by adding, removing, or modifying elements.
- Heterogeneous: Lists can contain elements of different data types, including numbers, strings, other lists, and more.
- Ordered: Lists are ordered collections, which means that elements are stored in a specific sequence, and you can access elements by their index.
- Access: You can access elements in a list using square brackets and the index of the element. Indexing starts from 0 for the first element.

Characteristics of a list+

- Common Operations: Lists support various operations, such as append(), extend(), insert(), remove(), and more, for modifying and manipulating their contents.
- List Comprehensions: You can use list comprehensions to create new lists based on existing lists or iterables.

Creating a list

- Creating a list in Python is straightforward. You can create a list by enclosing a comma-separated sequence of elements within square brackets. Here's how you can create a list:
- my_list = [1, 2, 3, 4, 5]
- In this example, my_list is a list that contains five integer elements. You can create a list with elements of any data type, and the elements can also be of mixed types. For instance:
- mixed_list = [1, "apple", 3.14, True, [7, 8, 9]]

Creating a list+

- In this case, mixed_list contains a variety of elements, including integers, strings, floats, Boolean values, and even another list as one of its elements.
- Using the list() Constructor:
- Examples

```
1. lst=list([2, 52, 56, 89, 78])#integer literals
2. print(lst)
3. Output: [2, 52, 56, 89, 78]
4. lst1=list(range(30, 60, 4))#Range function
5. print(lst1)
6. Output: [30, 34, 38, 42, 46, 50, 54, 58]
7. lst2=list(["John", "Peter", "James", "Loboa", "Kwaje"])
8. print(lst2)
9. Output: ['John', 'Peter', 'James', 'Loboa', 'Kwaje']
```

Creating a list++

- We can also create an empty list using the list constructor
- Example: lst=list()
- You can create a list by passing an iterable (e.g., a tuple, string, or another list) to the list() constructor.

```
1. my_tuple = (1, 2, 3, 4, 5)
2. my_list = list(my_tuple),
3. print(my_list)
4. ([1, 2, 3, 4, 5],)
5. num_lst=list("ALEMIN")
6. print(num_lst)
7. Output: ['A', 'L', 'E', 'M', 'I', 'N']
```

Creating a list++++

- You can also create an empty list and then add elements to it later:
- empty_list = []
- empty_list.append(10)
- empty_list.append("banana")
- This creates an empty list empty_list and then adds an integer and a string to it using the append() method.
- Lists are flexible and versatile data structures that allow you to store and manipulate collections of items in Python.

Accessing Elements

- Accessing elements in a Python list is done by using the index of the element within square brackets. Lists are zero-indexed, meaning the index for the first element is 0, the second element is at index 1, and so on. Here's how you can access elements in a list:
- **Accessing a Single Element:**
- To access a single element at a specific index, use square brackets with the index value:
- my_list = [10, 20, 30, 40, 50]
- element = my_list[2] # Access the element at index 2 (which is 30)

Accessing Multiple Elements (Slicing)

- You can access a range of elements in a list using slicing. Slicing is done by specifying a start and end index separated by a colon. It returns a new list containing the elements from the start index up to, but not including, the end index.
- my_list = [10, 20, 30, 40, 50]
- The syntax for accessing the elements of a list is the same as the syntax for accessing the characters of a string[2] example my_list[2] will be 30
- sub_list = my_list[1:4] # Creates a new list [20, 30, 40]
- You can also omit the start or end index to slice from the beginning or up to the end of the list:
- first_three = my_list[:3] # Creates a new list [10, 20, 30]
- last_two = my_list[3:]   # Creates a new list [40, 50]

More on List Slicing

- The start index is inclusive, and the end index is exclusive. The slicing operation includes all elements from start up to, but not including, end.
- If you omit start, it defaults to 0, and if you omit end, it defaults to the length of the list, effectively selecting the entire list.
- You can use negative indices for start and end, which count from the end of the list.
- You can add a third argument, step, to slice with a specific step size. This allows you to skip elements in the original list.

More on List Slicing

```
1. my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9]
2. # Get a sublist from index 2 to index 5 (inclusive at the start, exclusive at
   the end)
3. sublist1 = my_list[2:6]
4. # Result: [3, 4, 5, 6]
5. # Get the last three elements (from index -3 to the end)
6. sublist3 = my_list[-3:]
7. # Result: [7, 8, 9]
8. # Get every second element starting from index 1 (step by 2)
9. sublist4 = my_list[1::2]
10.# Result: [2, 4, 6, 8]
11.# Reverse the list using slicing
12.reversed_list = my_list[::-1]
13.# Result: [9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Negative Indexing

- Python supports negative indexing, where -1 refers to the last element, -2 refers to the second-to-last element, and so on. It's useful for accessing elements from the end of the list:
- my_list = [10, 20, 30, 40, 50]
- last_element = my_list[-1] # Access the last element (which is 50)
- Keep in mind that if you attempt to access an index that is out of the list's range, it will result in an "IndexError." So make sure to check the length of the list before accessing elements to avoid errors.

List Operations

- Python lists support various operations for adding, removing, and modifying elements, as well as other common list-related tasks. Here are some of the fundamental list operations:

- **Appending Elements:**

- You can add an element to the end of a list using the append() method.

- my_list = [1, 2, 3]

- my_list.append(4)

- # Result: [1, 2, 3, 4]

Extending Lists

- You can append multiple elements from another iterable (e.g., another list) to the end of a list using the extend() method.
- my_list = [1, 2, 3]
- my_list.extend([4, 5, 6])
- # Result: [1, 2, 3, 4, 5, 6]
- **Inserting Elements:**
- You can insert an element at a specific index using the insert() method.
- my_list = [1, 2, 3]
- my_list.insert(1, 4) # Insert 4 at index 1
- # Result: [1, 4, 2, 3]

Removing elements

- You can remove elements from a list in various ways:
- Using remove() to remove the first occurrence of a specific value.
- Using pop() to remove and return an element at a specific index.
- Using del to remove an element at a specific index.
- my_list = [1, 2, 3, 4, 5]
- my_list.remove(3) # Removes the first occurrence of 3
- my_list.pop(2) # Removes and returns the element at index 2 (which is 4)
- del my_list[0] # Removes the element at index 0 (which is 1)

Sorting and Reversing

- You can sort a list in ascending or descending order using sort() and reverse() methods.
- my_list = [3, 1, 4, 2, 5]
- my_list.sort() # Sort in ascending order
- my_list.reverse() # Reverse the list

Concatenating Lists:

- You can combine two or more lists into a new list using the + operator.

Concatenating Lists

- list1 = [1, 2, 3]
- list2 = [4, 5, 6]
- combined_list = list1 + list2
- # Result: [1, 2, 3, 4, 5, 6]

List Functions

- Python provides several built-in functions that are commonly used with lists to perform various operations and retrieve information about the list. Here are some of the most commonly used list functions:
- len(list):
- Returns the number of elements in the list.
- my_list = [1, 2, 3, 4, 5]
- length = len(my_list) # Result: 5
- max(list) and min(list):

List Functions+

- Return the maximum and minimum values in the list, respectively. Note that the list should contain comparable elements.
- my_list = [1, 2, 3, 4, 5]
- max_value = max(my_list) # Result: 5
- min_value = min(my_list) # Result: 1
- sum(list):
- Returns the sum of all elements in the list.
- my_list = [1, 2, 3, 4, 5]
- total = sum(my_list) # Result: 15

List Functions++

- sorted(list):
- Returns a new sorted list without modifying the original list. By default, it sorts in ascending order.
- my_list = [3, 1, 4, 2, 5]
- sorted_list = sorted(my_list) # Result: [1, 2, 3, 4, 5]
- sorted(list, reverse=True):

List Functions++

- You can sort the list in descending order by specifying the reverse=True argument.
- my_list = [3, 1, 4, 2, 5]
- sorted_list_desc = sorted(my_list, reverse=True) # Result: [5, 4, 3, 2, 1]
- any(iterable) and all(iterable):
- These functions are not exclusive to lists but can be used with any iterable. any(iterable) returns True if at least one element in the iterable is True. all(iterable) returns True if all elements in the iterable are True.
- my_list = [True, False, True]
- any_true = any(my_list) # Result: True
- all_true = all(my_list) # Result: False

enumerate(list)

- Returns an iterator that provides both the index and the value of each element in the list. This can be useful for looping through a list with access to the index.
- my_list = ['apple', 'banana', 'cherry']
- for index, value in enumerate(my_list):
- print(f"Index {index}: {value}")
- These functions are handy when working with lists and can simplify various common operations on lists.

List Comprehensions

- List comprehensions are concise and powerful way to create lists in Python.
- They allow you to generate new lists by applying an expression to each element of an existing iterable (such as a list) and optionally applying conditions to filter elements.
- List comprehensions have a compact syntax and are often used to make code more readable and efficient.

The basic structure of a list comprehension is as follows:

- new_list = [expression for element in iterable if condition]
- expression is the operation or transformation to apply to each element.

List Comprehensions+

- element is a variable that represents each element in the iterable.
- iterable is the original iterable (e.g., a list) you are iterating over.
- condition (optional) is a filtering condition to include only elements that meet certain criteria.
- Here are some examples of list comprehensions:
- Creating a list of squares:
- numbers = [1, 2, 3, 4, 5]

List Comprehensions++

- squares = [x**2 for x in numbers]
- # Result: [1, 4, 9, 16, 25]
- Filtering elements with a condition:
- numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]
- even_numbers = [x for x in numbers if x % 2 == 0]
- # Result: [2, 4, 6, 8]

A Program to convert temperatures from degree Celsius to Frahrenheit

- Consider a list with five different Celsius values. Convert all the Celsius values into Fahrenheit[3]

```
1. print("Celsius= ",end="")
2. Celsius=[10,20,31.3,40,39.2] #List with Celsius Value
3. print(Celsius)
4. print(" Celsius to Fahrenheit Conversion ")
5. print("Fahrenheit = ",end="")
6. Fahrenheit=[ ((float(9)/5)*x + 32) for x in Celsius]
7. print(Fahrenheit)
```

Output

- Celsius=[10, 20, 31.3, 40, 39.2]
- Celsius to Fahrenheit Conversion
- Fahrenheit = [50.0, 68.0, 88.34, 104.0, 102.56]

Creating a list of tuples:

- names = ["Alice", "Bob", "Charlie"]
- name_lengths = [(name, len(name)) for name in names]
- # Result: [("Alice", 5), ("Bob", 3), ("Charlie", 7)]
- **Nested list comprehensions:**
- You can also create lists of lists using nested list comprehensions:
- matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
- flattened = [x for row in matrix for x in row]
- # Result: [1, 2, 3, 4, 5, 6, 7, 8, 9]

List Methods

- Python lists come with a variety of built-in methods that allow you to perform common operations and manipulations on lists. Here are some of the most commonly used list methods:

- append(x):

- Adds the element x to the end of the list.

```
1. my_list = [1, 2, 3]
2. my_list.append(4)
3. # Result: [1, 2, 3, 4]
```

- extend(iterable):

List Methods+

- Appends elements from an iterable (e.g., another list) to the end of the list.

```
1.  list1 = [1, 2, 3]
2.  list2 = [4, 5, 6]
3.  list1.extend(list2)
```

- # Result: [1, 2, 3, 4, 5, 6]

- insert(i, x):

- Inserts element x at the specified index i.

List Methods+

- `my_list = [1, 2, 3]`
- `my_list.insert(1, 4)  # Insert 4 at index 1`
- `# Result: [1, 4, 2, 3]`
- remove(x):
- Removes the first occurrence of element x from the list.

```
1. my_list = [1, 2, 3, 4, 3]
2. my_list.remove(3)
3. # Result: [1, 2, 4, 3]
4. pop(i):
```

List Methods++

- Removes and returns the element at index i. If no index is provided, it removes and returns the last element.
- my_list = [1, 2, 3, 4, 5]
- popped_element = my_list.pop(2) # Remove and return the element at index 2 (which is 3)
- # Result: [1, 2, 4, 5], popped_element = 3
- index(x):
- Returns the index of the first occurrence of element x.
- my_list = [10, 20, 30, 40, 30]

List Methods+++

- index = my_list.index(30)
- # Result: 2 (index of the first occurrence of 30)
- count(x):
- Returns the number of times element x appears in the list.
- my_list = [1, 2, 3, 2, 4, 2]
- count = my_list.count(2)
- # Result: 3 (2 appears 3 times)

List Methods+++

- sort():
- Sorts the list in ascending order. It modifies the original list in place.
- my_list = [3, 1, 4, 2, 5]
- my_list.sort()
- # Result: [1, 2, 3, 4, 5]

List Methods++++

- reverse():Reverses the order of elements in the list in place.
- my_list = [1, 2, 3, 4, 5]
- my_list.reverse()# Result: [5, 4, 3, 2, 1]
- clear():
- Removes all elements from the list, leaving it empty.
- my_list = [1, 2, 3, 4]
- my_list.clear()
- # Result: []

String

- Immutable: Strings are immutable, which means you cannot change the characters within a string. Any operation that appears to modify a string actually creates a new string.
- Homogeneous: Strings contain characters and are homogeneous in that they consist of a single data type.
- Ordered: Strings are ordered sequences of characters, allowing you to access individual characters by their index.

String+

- Access: You can access characters in a string using square brackets and the index of the character. Like lists, indexing starts from 0 for the first character.
- Common Operations: Strings have a variety of methods for common string operations, such as upper(), lower(), replace(), and more. These methods return new strings since strings are immutable.
- String Formatting: Strings support various formatting techniques, including f-strings, %-formatting, and the str.format() method.

Difference between a String and a list

- my_list = [1, 2, 3]
- my_string = "Hello, World!"
- # Modifying the list
- my_list.append(4)
- my_list[1] = 5
- # Attempting to modify the string (which is not possible)
- # This creates a new string with the desired change

Difference between a String and a list+

- print(my_list)   # Result: [1, 5, 3, 4]
- print(my_string) # Result: "Hello, Python!“
- my_string = my_string.replace("World", "Python")

Splitting a String into List

- You can split a string into a list of substrings using the split() method in Python. This is a common operation when you have a string containing words or values separated by a delimiter, such as a space or a comma. The split() method splits the string based on the specified delimiter and returns a list of the resulting substrings.
- Here's how you can use the split() method to split a string into a list:
- # Example 1: Splitting a string using a space as the delimiter
- my_string = "This is a sample string"
- word_list = my_string.split() # By default, it splits on spaces
- print(word_list)# Result: ['This', 'is', 'a', 'sample', 'string']

Splitting a String into List+

- # Example 2: Splitting a string using a comma as the delimiter
- csv_data = "apple,banana,orange,grape"
- fruit_list = csv_data.split(',')
- print(fruit_list)
- # Result: ['apple', 'banana', 'orange', 'grape']

Splitting a String into List++

- In the first example, we split the string my_string using spaces as the delimiter. Since we didn't specify a delimiter in the split() method, it used spaces by default to split the string into words.
- In the second example, we split the string csv_data using a comma , as the delimiter. This is often used for parsing CSV (Comma-Separated Values) data into a list of values.
- You can also specify an optional maxsplit parameter to limit the number of splits. For example:
- my_string = "apple banana cherry date"
- word_list = my_string.split(' ', 2) # Split into a maximum of 3 substrings
- print(word_list)
- # Result: ['apple', 'banana', 'cherry date']

Passing List to a Function

- You can pass a list as an argument to a Python function, allowing you to perform operations on that list within the function. When you pass a list to a function, you can modify the list within the function, and these modifications will affect the original list since lists are mutable. Here's how to pass a list to a function:

```
1. def process_list(my_list):
2.     # Perform operations on the list
3.     my_list.append(4)  # This modifies the original list
4. my_list = [1, 2, 3]
5. process_list(my_list)
6. print(my_list)
7. # Result: [1, 2, 3, 4]
```

Passing List to a Function+

- In this example, we define a function process_list that takes a single parameter my_list, which is a list. Inside the function, we append the value 4 to the list. When we call process_list(my_list), the original list my_list is passed to the function, and any changes made to it within the function are reflected in the original list outside the function.
- Here are a few key points to keep in mind when passing lists to functions:
- Lists are mutable, so any modifications made to the list within the function are applied to the original list unless you explicitly create a new list within the function.

Passing List to a Function++

- If you want to avoid modifying the original list, you can create a copy of the list using slicing or the copy module before passing it to the function.

```
import copy
def process_list(my_list):
    new_list = copy.deepcopy(my_list)  # Create a copy of the list
    new_list.append(4)
    return new_list
my_list = [1, 2, 3]
result_list = process_list(my_list)
print(my_list)        # Original list remains unchanged: [1, 2, 3]
```

Passing List to a Function+++

- print(result_list) # Result of the function: [1, 2, 3, 4]
- Functions can also return lists as their results, allowing you to create and return new lists based on the input list.
- Passing lists to functions is a powerful way to encapsulate and modularize your code when working with collections of data.

Returning List from a Function

➢You can return a list from a Python function by using the return statement. This allows you to create and return a new list or modify an existing list within the function and then use the returned list outside the function. Here's how to return a list from a function:

```
1. def create_list():
2.     new_list = [1, 2, 3, 4, 5]
3.     return new_list
4. result_list = create_list()
5. print(result_list)
6. # Result: [1, 2, 3, 4, 5]
```

Returning List from a Function +

- In this example, the create_list function creates a new list and then returns it using the return statement. When you call the function and assign its result to the result_list variable, you get the newly created list.
- You can also pass arguments to the function and use those arguments to generate the list to be returned:

```
def generate_sequence(n):
    sequence = [i for i in range(1, n + 1)]
    return sequence
result_sequence = generate_sequence(10)
```

Returning List from a Function ++

- print(result_sequence)
- # Result: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
- In this example, the generate_sequence function takes an argument n and creates a list of numbers from 1 to n, which is then returned.
- You can also modify an existing list within the function and return the modified list:

Returning List from a Function +++

```
1.  def modify_list(my_list):
2.      my_list.append(6)
3.      my_list.extend([7, 8])
4.      return my_list
5.  original_list = [1, 2, 3, 4, 5]
6.  result_list = modify_list(original_list)
7.  print(result_list)
8.  # Result: [1, 2, 3, 4, 5, 6, 7, 8]
```

Returning List from a Function ++++

- In this case, the modify_list function takes an existing list my_list, appends elements to it, and then returns the modified list.
- Returning lists from functions allows you to encapsulate and reuse code, making your code more modular and readable. You can use the returned lists for further processing or to update the original data.

Program to check whether a list is a palindrome

```
1. def is_Palindrome(ls_num):
2.     # Create a reversed version of the list
3.     reversed_Lst = ls_num[::-1]
4.     # Compare the original list with the reversed list
5.     return ls_num == reversed_Lst
6. # Example usage:
7. my_list = [1, 2, 3, 2, 1]
8. result = is_Palindrome(my_list)
9. print(result)  # Result: True
10.another_list = [1, 2, 3, 4, 5]
11.result = is_Palindrome(another_list)
12.print(result)  # Result: False
```

Explanation

- In this function, ls_num is the list you want to check for palindrome status.
- We create a reversed version of the list using slicing (ls_num[::-1]) and then compare it to the original list using ==.
- If they are the same, the function returns True, indicating that the list is a palindrome; otherwise, it returns False.

Summary

- In this week's lecture, we learned about list which we said is a mutable collection and we also discussed some of its characteristics such as heterogeneous, ordered, etc. we further delved into the different operations that can be performed in a list.
- We moved onto learning about list comprehension which is said to be a concise and powerful way to create lists in Python,
- Throughout our discussion, we used a number of examples to simplify the absorption of the key concepts.
- Understanding list is very fundamental in the field of Programming and problem solving
- In our next lecture we shall delved into searching and sorting which is yet an other fundament concept in programming and problem solving

Reference

[1]Balagurusamy E. (2016) *Introduction to computing and problem-solving using python*,(p.127) McGraw Hill Education (India) Private Limited

[2] Wentworth P., Elkner J., Downey A.B, MeyersHow C. (2020) *Think Like a Computer Scientist: Learning with Python 3 Documentation Release* 3rd Edition (p.111)

[3] Kamthane, A. N., & Kamthane , A. A. (2018). *PROGRAMMING AND PROBLEM SOLVING WITH PYTHON* (p. 203). McGraw Hill Education (India) Private Limited.