PER SCHOLAS

Python: Lists

Learning Objectives

This lesson provides learners with information about what a List is and what it does in Python. This presentation is designed for beginners. By the end of this lesson, learners will be able to:

- Identify the similarities between Python lists and strings.
- Differentiate between the **`append()`** and **`extend()`** methods in Python.
- Concatenate two or more lists in Python using the **`+`** operator.
- Assign a new value to a specific index in a list to modify an element in Python.
- Use the **`insert()`** method to add elements to a list at a specific index in Python.
- Remove elements from a list using the **`pop()`** and **`remove()`** methods in Python.
- Sort a list in Python using the built-in **`sort()`** method.

PER SCHOLAS

Table of Contents

Section 1
Lists and List Methods

Lists and Similarities to Strings

Lists are a data type in Python that are used to store a collection of data within a single variable using brackets `[]`, with commas separating values.

- Lists are **mutable** objects, which means that they can be changed in-place after creation.
- Lists have many similarities to strings, which are effectively collections of characters. Lists and strings are similar in:
  - Indexing, Slicing, Concatenation, Repetition, and Iteration

To access a specific piece of data within a list, use the notation `list[index]`, where `index` is the position of the data in the list. **The first element of a list is at index 0**.

```
# Create a list of strings.
string_list = [“Hello”, “Python”, “World”]

# Create a list of numbers.
number_list = [3, 4, 5, 6, 8, 10]

# Create a list of boolean values.
boolean_list = [True, False, False, True]

# Create mixed list.
mixed_list = [3, 4, “Python”, True]
print(mixed_list[2])        # output: Python
```

PER SCHOLAS

Converting Strings to Lists

Since strings are already effectively lists of characters, Python includes built-in functions for converting between these two similar data types.

- Using the `list()` built-in function, we can easily change a string into a list of characters. It is important to note that white spaces are treated as characters, including leading or trailing white spaces.
- We can also use the `split()` built-in function to turn a string into a list of substrings based on a delimiter passed into the `split()` function.

```
# Create a string.
my_string = "Hello World"

# Create a list of characters from my_string.
character_list = list(my_string)

# Create a list of substrings from my_string.
substring_list = my_string.split()

# Print the results.
print(my_string)        # output: "Hello World"
print(character_list)   # output: ['H', 'e', 'l',
'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd']
print(substring_list)   # output: ['Hello', 'World']
```

Modifying a List Element

To modify an element in a list, you can access it by its index and assign a new value.

The syntax for modifying an element is `list[index] = new_value`, where `index` is the index of the element to be modified and `new_value` is the new value you want to assign to that element.

The new value can be of any valid data type, including other lists or objects.

```
# Create a list.
my_list = [1, 2, 3, 4, 5]

# Modify an element in the list.
# Note that the first element in
# a list is at index 0.
my_list[2] = 7

# Print the result.
print(my_list)  # output: [1, 2, 7, 4, 5]
```

List Methods: append() and extend()

The `append()` method is used to add an element to the end of a list. The argument passed to `append()` can be any valid Python object.

The `extend()` method is used to add all the elements of an iterable (e.g., a list) to the end of a list. The argument passed to `extend()` must be an iterable (e.g., a list, tuple, set, string, etc.).

The return value of both methods is `None`, which means that they modify the list in place and do not create a new list.

Example: append() and extend()

The `append()` method is used to add an element to the end of a list.

The `extend()` method is used to add all the elements of an iterable to the end of a list.

Using append()

```
# Create an empty list.
my_list = []

# Add elements to the list using append().
my_list.append(1)
my_list.append(2)
my_list.append(3)

# Print the new list.
# Note that my_list *is* modified.
print(my_list)        # output: [1, 2, 3]
```

Using extend()

```
# Create an empty list.
my_list = []

# Add elements to the list using extend().
my_list.extend([1, 2, 3])

# Print the new list.
# Note that my_list *is* modified.
print(my_list)        # output: [1, 2, 3]
```

List Methods: insert()

The `insert()` method is used to insert an item at a specific index in a list.

The syntax for using `insert()` is `list.insert(index, item)`, where `index` is the index where the item should be inserted and `item` is the item to be inserted.

All items in the list after the specified index are shifted to the right to make space for the new item.

```
# Create a list.
my_list = [1, 2, 4, 5]

# Insert elements using insert().
my_list.insert(2, 3)

# Print the new list.
# Note that my_list *is* modified.
print(my_list)   # output: [1, 2, 3, 4, 5]
```

List Methods: pop()

The `pop()` method is used to remove and return an item from a specific index in a list.

The syntax for using `pop()` is `list.pop(index)`, where `index` is the index of the item to be removed. If no index is specified, the last item in the list is removed.

The `pop()` method can also be used to store the removed item in a variable for later use, such as `item = list.pop(index)`.

```
# Create a list.
my_list = [1, 2, 3, 4, 5]

# Remove an element by index via pop().
item = my_list.pop(2)

# Print the new list and the returned element.
# Note that my_list *is* modified.
print(my_list)   # output: [1, 2, 4, 5]
print(item)   # output: 3
```

List Methods: remove()

The `remove()` method is used to remove the first occurrence of a specified value from a list.

The syntax for using `remove()` is `list.remove(value)`, where `value` is the item to be removed.

If the specified value is not found in the list, a `ValueError` is raised.

```
# Create a list.
my_list = [1, 2, 3, 4, 5]

# Remove a value via remove().
my_list.remove(3)

# Print the new list.
# Note that my_list *is* modified.
print(my_list)   # output: [1, 2, 4, 5]
```

List Methods: sort()

The `sort()` method is used to sort a list in ascending order by default.

The syntax for using `sort()` is `list.sort()`.

The `sort()` method can also accept a `reverse=True` parameter to sort the list in reverse order. Additionally, it can accept a `key` parameter to specify a custom sorting criterion.

```
# Create a list.
my_list = [5, 2, 6, 4, 1, 1, 3]

# Sort the list via sort().
my_list.sort()

# Print the new list.
# Note that my_list *is* modified.
print(my_list)   # output: [1, 1, 2, 3, 4, 5, 6]
```

Concatenating Lists

Concatenating a list is the process of combining two or more lists into a single list.

The `+` operator is used to concatenate two lists in Python.

When concatenating lists, **a new list object is created** and the elements from the original lists are copied into the new list in the order in which they appear.

```
# Create two lists.
list1 = [1, 2, 3]
list2 = [4, 5, 6]

# Add elements to the list using append().
new_list = list1 + list2

# Print the new list.
print(new_list)    # output: [1, 2, 3, 4, 5, 6]
```

Traversing Lists

Traversing a list means iterating through all of its elements one by one.

The most common way to traverse a list is to use a `for` loop with the `in` keyword.

- For example, `for element in my_list`.
- Inside the loop, you can access and work with each element individually, like printing it or modifying it.

```
# Create a list.
my_list = [1, 2, 3, 4, 5]

# Traverse the list with a for loop.
for element in my_list:
    print(element)

# Traverse the list by accessing the
# indexes with the range() and len() functions.
for i in range(len(my_list)):
    print(f"Index {i} contains: {my_list[i]}")
```

PER SCHOLAS

Nested Lists

Nested lists are lists that contain other lists as elements. These can be used to represent complex data structures, such as matrices, tables, or trees.

To access elements of a nested list, you can use multiple indexing operations. For example, if you have a nested list called `my_list` and you want to access the element in the second row and third column, you can use `my_list[1][2]`.

Two-Dimensional List Example and Visualization Table

```
# Create a two-dimensional list with three sublists.
# Each sublist contains three elements.
my_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

# Print the middle element of the 2D list.
# Remember that list indexes start at 0.
print(my_list[1][1])        # output: 5
```

my_list	[y][0]	[y][1]	[y][2]
[0][x]	1	2	3
[1][x]	4	5	6
[2][x]	7	8	9

Nested Lists: Iteration

You can also use nested `for` loops to iterate through the elements of a nested list.

A two-dimensional list requires two loops, and each additional nested list adds a layer of complexity in the form of an additional nested loop.

```
# Create a two-dimensional list with three sublists.
# Each sublist contains three elements.
my_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

# Iterate through each sublist.
for sublist in my_list:
    # Iterate through each sublist element.
    for element in sublist:
        print(element)

# output (each on a new line): 1 2 3 4 5 6 7 8 9
```

my_list	[y][0]	[y][1]	[y][2]
[0][x]	1	2	3
[1][x]	4	5	6
[2][x]	7	8	9

PER SCHOLAS

GLAB: Lists

Please follow the link below to the guided lab for Strings.

- GLAB - 340.5.1 - Lists
- You can also find this lab on Canvas under the Assignments section.
- If you have questions while performing the lab activity, ask your instructors for assistance.

Hands-On Activities

Please follow the link below for Python List practice. Open following links on **Google colaboratory.**

ACT 340.5.1 - List Exercises - Basic.ipynb

ACT 340.5.2 - List Exercises - Advanced.ipynb

Knowledge Check

- What are some similarities between Python strings and lists?
- How can you convert a string into a list in Python, and what built-in function is used to do this?
- What is the difference between the `append()` and `extend()` methods for adding elements to a list in Python?
- What is the purpose of the `insert()` method in Python, and how is it used?
- What is the purpose of the `pop()` method in Python, and how is it used?
- What is the purpose of the `remove()` method in Python, and how is it used?
- What is the purpose of the `sort()` method in Python, and how is it used?
- How can you concatenate two lists in Python, and what is the operator used to do this?
- What is a nested list in Python, and what is an example of a use case for a nested list?

Summary

In this lesson, we explored Lists. Lists are a commonly used data structure in programming that are similar to strings, in that they both store sequences of elements:

- Nested lists are lists that contain other lists as elements, and can be used to represent multidimensional data structures.
- The `append()` method is used to add a single element to the end of a list, while the `extend()` method is used to add multiple elements to the end of a list.
- To concatenate two lists in Python, you can use the `+` operator, which creates a new list that contains all the elements from both lists.
- Traversing a list involves iterating over each element of a list to access, process, or manipulate them individually, and it is a fundamental technique in list manipulation.
- You can modify an element in a list by assigning a new value to a specific index in the list.
- Python provides several built-in methods for modifying lists, such as `insert()`, `pop()`, `remove()`, and `sort()`.

PER SCHOLAS

Questions