Introducing Lists

University of Wyoming COSC 1010

Adapted from: *Python Crash Course 3rd Ed* By Eric Matthes

Introducing Lists

Introuduction

- We will start learning about lists, and for to start working with them
- Lists allow you to store sets of information in one place
- Lists can range from a few items to millions
- One of the most powerful programming features
- Tie together many important concepts

What is a List?

- A collection of items ina particular order
- The list can include
  - The digits 0 to 9
  - The names of all the people in your family
  - Really anything you want
- The items in a list don't have to be related

What is a List?

- Lists usually contain more than one element so their names are the plural form of the word
  - letters
  - digits
  - names
- In Python **square brackets** `[]` indicate a list
- Individual items in a list are separated by commas

In [ ]:
```
# create our list of bikes
bikes = ["trek","cannondale","specialized"]
# and print the list
print(bikes)
```

```
['trek', 'cannondale', 'specialized']
```

Accessing Elements in a List

- Lists are ordered collections
- You can access any element in a list by telling Python the position or *index*
- To access an element you write the name of the list followed by the index of the item enclosed in square brackets

In [ ]:
```
bikes = ["trek","cannondale","specialized"]
print(bikes[0])
```

```
trek
```

Accessing Elements in a List

- When you ask for a single item python will return just that
- Notice it didn't include the brackets as the previous code did
- It just returned the string element, so you can use the string methods we talked about earlier on it!

In [ ]:
```
bikes = ["trek","cannondale","specialized"]
print(bikes[0].title())
```

```
Trek
```

Index Positions Start at 0 **Not** 1

- You may have noticed in the code slides, for our list `["trek","cannondale","specialized"]` we used `bikes[0]` to get `trek`
- This is because python considers the first item in a list to be at position `0`
- This applies to most programming languages
- If you have strange behavior in your code, check to see if you're making a simple but common *off-by one* error

Index Positions Start at 0 **Not** 1

- The second item in a list has an index of 1

- Using this counting method you can get any element you want from a list by subtracting one from its position in the list
- To get the 4th item you use index 3, etc.
- Python has a special syntax for when you wish to access the last element in an array `-1`

In [ ]:
```
bikes = ["trek","cannondale","specialized"]
print(bikes[-1].title())
```

```
Specialized
```

Index Positions Start at 0 **Not** 1

- The previous code returned `specialized`
- The syntax can be quite useful
- This enables you to access the last element without needing to know (or check) how long the array is
- This convention extends to other negative index values
- `-2` will return the second to last element
- `-3` returns the third to last
- And onwards

Using Individual Values from a List

- You can use an individual value from a list as you would a variable
- You can even assign it to a variable, but you don't need to. It's already stored!
- f-strings can be used to create a message based on a value from the list

In [ ]:
```
bikes = ["trek","cannondale","specialized"]
message = f"My first bike was a {bikes[0].title()}"
print(message)
```

```
My first bike was a Trek
```

Modifying, Adding, and Removing Elements

- Most lists created will be dynamic
- This means the list will be built then elements can be added or removed as the program runs
- For example, you could create a game with aliens for a player to shoot
  - As a player shoots aliens they will be removed
  - As more aliens spawn they can be added to the list
- The length of the array will expand and contract as the game runs

Modifying Elements in a List

- the syntax for modifying an element is similar to the syntax for access ing an element in a list
- To change an element use the name of the list followed by the index of the element
- Then provide a new value to be stored in tht index position
- This is done in the same way you assign a variable, just with the value being stored in a list

In [ ]:
```
cities = ["Laramie","Casper","Jackson Hole","Cheyenne"]
print(cities)

cities[3] = "Sheridan"
print(cities)
```

```
['Laramie', 'Casper', 'Jackson Hole', 'Cheyenne']
['Laramie', 'Casper', 'Jackson Hole', 'Sheridan']
```

Modifying Elements in a List

- In the previous cell we defines the list `cities`
- Then we overwrite `Cheyenne` with the value `Sheridan`
- Printing the list at each stage shows the output
- You can change the value of any item, based on it's index

Adding an Element to a List

- Recall lists are dynamic
- Meaning things can be added to lists, not just at their declaration
- Perhaps you are storing cities in which people live, and want to add more as more

people say where they're from
- Python provides several ways to add new data to existing lists

Appending items to the End of a List

- The easiest way to add an element is to *append* it to the list
- When an item is appended it is added to the end of the list
- Using the previous cities we can show how to add items using the `.append()` function

In [ ]:
```
cities = ["Laramie","Casper","Jackson Hole","Cheyenne"]
print(cities)

cities.append("Sheridan")
print(cities)
```

```
['Laramie', 'Casper', 'Jackson Hole', 'Cheyenne']
['Laramie', 'Casper', 'Jackson Hole', 'Cheyenne', 'Sheridan']
```

Appending items to the End of a List

- Here we are able to append `Sheridan` without also have to remove `Cheyenne`
- The append method makes it easy to build lists dynamically
- You could even start with an empty list and use append to build it out as needed!

In [ ]:
```
#declare the empty list
animals = []
#now add in different animals
animals.append("dog")
animals.append("cat")
animals.append("bird")

print(animals)
```

```
['dog', 'cat', 'bird']
```

Appending items to the End of a List

- Append can be used to build lists out nicely
- Items are indexed in order of when they are appended
- So if starting with an empty list the first element appended will be the first element in the list
- Building lists this way is common, as you won't always know what data a user will want to store in a program until after the program is running
- To put users in control, start by defining an empty list that will hold user's values

- Append each new value to the empty list

Inserting Elements into a List

- New elements can be added at any position in a list by using `.insert()`
- You can do this by specifying the index of the new element and the value of the new item

In [ ]:
```
cities = ["Laramie","Casper","Jackson Hole","Cheyenne"]
print(cities)

cities.insert(1,"Sheridan")
print(cities)
```

```
['Laramie', 'Casper', 'Jackson Hole', 'Cheyenne']
['Laramie', 'Sheridan', 'Casper', 'Jackson Hole', 'Cheyenne']
```

Inserting Elements into a List

- We inserted Sheridan at the second position (index value 1)
- The `.insert()` function opens a space at the position specified and stores the value in that position
- This shifts every other value after the index listed one position to the right

Removing Elements from a List

- The other side of the dynamic coin means that elements ca also be removed from lists
- Going back to the alien example, if a player shoots down one
- Or if you're voting on cities and the one with the least amount of votes gets removed
- Or perhaps you are writing code for an application and users are able to delete their accounts
- Items can be removed from a list according to its position in the list ot its value

Removing an Item Using the del Statement

- If you know the position of the item you want to remove from a list you can use the `del` statement
- this statement can be used to delete any position, not just the first
- Once the `del` statement is used the value can no longer be accessed

In [ ]:
```
cities = ["Laramie","Casper","Jackson Hole","Cheyenne"]
print(cities)

del cities[1]
print(cities)
```

```
['Laramie', 'Casper', 'Jackson Hole', 'Cheyenne']
['Laramie', 'Jackson Hole', 'Cheyenne']
```

Removing an Item Using the `.pop()` method

- Sometimes you want to get the value of an item before it is removed
- Suppose you want to get the x and y position of an alien that was removed from the screen
- Or if you want to remove a user from a list of active members and mve them to a list of inactive
- The `.pop()` method removes the last item in a list
  - It lets you work with that item after removing it
- The term *pop* comes from thinking of a list as a stack of item off the top of the stack
- In this analogy the top of a stack corresponds to the end of a list

In [ ]:
```
cities = ["Laramie","Casper","Jackson Hole","Cheyenne"]
print(cities)

print(f"We removed {cities.pop()}")

print(cities)
```

```
['Laramie', 'Casper', 'Jackson Hole', 'Cheyenne']
We removed Cheyenne
['Laramie', 'Casper', 'Jackson Hole']
```

Removing an Item Using the `.pop()` method

- In the previous cell we
  - Defined and printed the `cities` list
  - Popped a value from the list
  - While using that value ina print statement
  - Printed the list to show the city had been removed
- Alternatively you could assign the popped city to a variable
- `city = cities.pop()`

Removing an Item Using the `.pop()` method

- How might `.pop()` be useful?
- Imagine we had a list of motorcycles that store motorcycles in chronological order
  - That is to say the order in which they were owned
- Pop can then be used to print the last bike owned

Popping Items from Any Position in a List

- In addition to removing items from the end of the list `.pop()` can remove elements from any index
- This is done by including the index position within the parenthesis

In [ ]:
```
cities = ["Laramie","Casper","Jackson Hole","Cheyenne"]
print(cities)

print(f"We removed {cities.pop(1)}")

print(cities)
```

```
['Laramie', 'Casper', 'Jackson Hole', 'Cheyenne']
We removed Casper
['Laramie', 'Jackson Hole', 'Cheyenne']
```

Popping Items from Any Position in a List

- Remember each time `.pop()` is used the item will no longer be stored within the list
- If you are unsure if `del` or `.pop()` is better just think about if you want to use the removed item or not
- If you want to use the item as it is removed use `.pop()`
- If you don't need to use the value then `del`

Removing an Item by Value

- Sometimes you won't know the index position of an item
- But you will know the Value!
- If you know the value then the `.remove()` method can be used

In [ ]:
```
cities = ["Laramie","Casper","Jackson Hole","Cheyenne"]
print(cities)

cities.remove('Casper')

print(cities)
```

```
['Laramie', 'Casper', 'Jackson Hole', 'Cheyenne']
['Laramie', 'Jackson Hole', 'Cheyenne']
```

Removing an Item by Value

- The `.remove()` method tells Python to figure out where `Casper` appears and remove it
- Remove can also be used to work with a value being removed
  - By assigning the value to a variable
  - Then using that variable in remove
- Remove only removes the first occurrence of the specified value
- If there is a possibility that the value occurs more than one time a loop is needed
  - We'll talk about loops during a later lecture

Organizing a List

- Often lists will be created in an unpredictable order
- AS programmers you can't always control what order users put data in
- Although this is unavoidable, it isn't final
- You may want your data to be ina certain order
  - Sometimes you will want to preserve the original order
  - Other times you'll want to change the original order
- Python provides a number of different ways to organize lists

Sorting a List Permanently with the `.sort()` Method

- Python's `.sort()` method makes it relatively easy to sort a list
- Imagine you want to sort the list of Wyoming cities from earlier alphabetically
- And that you want to change the order of the list to store them as such
- To keep the task simple everything should be of the same casing

In [ ]:
```
cities = ["Laramie","Casper","Jackson Hole","Cheyenne"]
print(cities)

cities.sort()
print(cities)
```

```
['Laramie', 'Casper', 'Jackson Hole', 'Cheyenne']
['Casper', 'Cheyenne', 'Jackson Hole', 'Laramie']
```

Sorting a List Permanently with the `.sort()` Method

- The `.sort()` method changes the order of the list permanently
- The cities are now in alphabetical order, and cannot be reverted back to the original order
- You can also sort the list in reverse-alphabetical order
- This is done by passing the argument `reverse=True` to `.sort()`
  - `cities.sort(reverse=True)`
  - This sorting is also permanent

In [ ]:
```
cities = ["Laramie","Casper","Jackson Hole","Cheyenne"]
print(cities)

cities.sort(reverse=True)
print(cities)
```

```
['Laramie', 'Casper', 'Jackson Hole', 'Cheyenne']
['Laramie', 'Jackson Hole', 'Cheyenne', 'Casper']
```

Sorting a List Temporarily with the `.sorted()` Function

- Sometimes you will want to maintain the original order of a list
- this can be done with the `sorted()` function
- The `sorted()` function lets you display the list in a particular order
- It doesn't change the original order of the list

In [ ]:
```
cities = ["Laramie","Casper","Jackson Hole","Cheyenne"]
print(cities)

print(sorted(cities))

print(cities)
```

```
['Laramie', 'Casper', 'Jackson Hole', 'Cheyenne']
['Casper', 'Cheyenne', 'Jackson Hole', 'Laramie']
['Laramie', 'Casper', 'Jackson Hole', 'Cheyenne']
```

Sorting a List Temporarily with the `.sorted()` Function

- Note the list still exists in its original order after `sorted()` being used
- The `sorted()` function can also accept a `reverse=True`
- Sorting alphabetically is harder when casing isn't consistent

Printing a List in reverse Order

- To reverse the original order of a list the `reverse()` method can be used
- The `.reverse()` method changes the order permanently
- You can revert back by simply calling `.reverse()` again!

In [ ]:
```
cities = ["Laramie","Casper","Jackson Hole","Cheyenne"]
print(cities)
cities.reverse()
print(cities)
cities.reverse()
print(cities)
```

```
['Laramie', 'Casper', 'Jackson Hole', 'Cheyenne']
['Cheyenne', 'Jackson Hole', 'Casper', 'Laramie']
['Laramie', 'Casper', 'Jackson Hole', 'Cheyenne']
```

Finding the Length of a List

- You can quickly find the length of a list by using `len()`
- It will give you the number of items in a list
- Python counts the items in a list starting at **one**

In [ ]:
```
cities = ["Laramie","Casper","Jackson Hole","Cheyenne"]
print(len(cities))
```

```
4
```

Avoiding Index Errors When Working With Lists

- There is one type of error when working with lists
- Let's assume you have a list with four cities and try to access the city at index position 4

In [ ]:
```
cities = ["Laramie","Casper","Jackson Hole","Cheyenne"]
print(cities[4])
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
/tmp/ipykernel_1339/3269608785.py in <module>
      1 cities = ["Laramie","Casper","Jackson Hole","Cheyenne"]
----> 2 print(cities[4])

IndexError: list index out of range
```

Avoiding Index Errors When Working With Lists

- Python attempts to give you the item at index four
- But, when it searches the list there are no items with that index
- Because of the *off-by-one* nature of indexing in lists, this error is typical
- People think the third item is number 3, as they start counting at 1
- But, the third item is really 2

Avoiding Index Errors When Working With Lists

- An index error means Python can't find an item at the index requested
- If you encounter an index error, try adjusting your count by `-1`
- Run the program again to ensure if you are correct
- Remember, when you want to access the last item in a list you can use `cities[-1]`