

Lecture 3

Introduction to Arrays

Solution to Assignment 3

```
1 brand = input("What is your favorite car brand? ")
2 car = input("What is your favorite car from that brand? ")
3 print(car + " is my favorite car from " + brand)
4 cont = input("Do you have any other favorite car? ")
5
6 while cont != "No":
7     if cont == "Yes"
8         brand = input("What is your favorite car brand? ")
9         car = input("What is your favorite car from that brand? ")
10        print(car + " is my favorite car from " + brand)
11    else:
12        print("Please enter either Yes or No")
13
14    cont = input("Do you have any other favorite car? ")
```

Introduction to Basic Data Structures

If you are coming from a language like C++ you would know that you have to dynamically allocate memory to create data structures.

Since python is dynamically typed these data structures are built in and do not require memory management!

Built in Data Structures:

- Lists
- Tuples
- Sets
- Dictionaries

Note: frozensets and frozendictionaries do exist as well but we will not be covering them as they aren't really used.

Introduction to Lists

Note: These are also referred to as arrays in other languages

To initialize a list the format would be:

`object = [contents]`

`object = []` for an empty list

Example:

```
1 "A list with three items"
2 carBrands = ["Honda", "Ford", "Hyundai"]
3
4 "An empty list"
5 emptyList = []
6
```

Small thing to note about Lists in Python

- Unlike other languages, the objects stored in a list can be different, they don't all have to be the same type

```
1 myList = ["S2000", 239, "Mustang", 129.39023, True]
2 print(myList)
```

```
['S2000', 239, 'Mustang', 129.39023, True]
```

List's Methods/Functions

Basic functions

- `len()` - Returns the length of the list
- `sorted()` - Returns the sorted version of the list
- `print()` - prints the list and its contents

Special Case Functions

- `max()` - returns the highest value in the list
- `min()` - returns the lowest value in the list
- `sum()` - returns the sum of the list

Basic Methods (Member Functions)

- `.append()` - adds to the end of the list
- `.remove()` - removes item from the list
- `.sort()` - sorts the original list
- `.count()` - counts how many times an item appears in the list
- `.reverse()` - reverses the order of the list
- `.copy()` - returns a copy of the list
 - `List[:]` does the same thing
- `.index()` - returns the index of the item's first appearance in the list
- `.clear()` - removes all items in the list
- `.pop()` - removes an item at an index and returns the item
- `.extend()` - iterates through an object and appends each item.

Common Methods

len()

```
1 list_cars = ["BMW", "Hyundai", "Ford"]  
2 print(len(list_cars))
```

3

.append()

```
1 list_cars = ["BMW", "Hyundai", "Ford"]  
2 list_cars.append("Honda")  
→ 3 print(list_cars)
```

```
['BMW', 'Hyundai', 'Ford', 'Honda']
```

Common Methods

.sort()

```
1 list_cars = ["BMW","Hyundai","Ford"]
2 list_cars.sort()
3 print(list_cars)
```

```
['BMW', 'Ford', 'Hyundai']
```

sorted()

```
1 list_cars = ["BMW","Hyundai","Ford"]
2 sorted_list_cars = sorted(list_cars)
3 print(sorted_list_cars)
```

```
['BMW', 'Ford', 'Hyundai']
```


Common Methods

.remove()

```
1 list_cars = ["BMW", "Hyundai", "Ford"]
2 list_cars.remove("Ford")
3 print(list_cars)
```

```
['BMW', 'Hyundai']
```

.pop()

```
1 list_cars = ["BMW", "Hyundai", "Ford"]
2 popped_car = list_cars.pop(0)
3 print(popped_car)
4 print(list_cars)
```

```
BMW
['Hyundai', 'Ford']
```

Special Case Functions

max()

```
list_nums = [1,2,3]
```

min()

```
print("Max =",max(list_nums))
```

```
print("Min =",min(list_nums))
```

sum()

```
print("Sum =",sum(list_nums))
```

Max = 3

Min = 1

Sum = 6

Indexing

Indexing is how we get the items stored in a list

As stated before when programming 0 is **the first** index, not 1 and the last index in a range is **non-inclusive**

The last index can be gotten with the index -1

Going between 0 and length-1 will get you the items in between

For example:

```
1 carBrands = ["Honda", "Ford", "Hyundai"]
2
3 #The first index
4 print(carBrands[0])
5
6 #The second index
7 print(carBrands[1])
8
9 #The last index
10 print(carBrands[2])
11
```

```
Honda
Ford
Hyundai
```


Negative Indexing

This isn't used very often if ever cause most of the time there are better ways of getting around having to do this but it can come in handy at times.

Seen in the last slide the index -1 is the last index

We can continue this, -2, -3.. Or until the start of the list

```
1 carBrands = ["Honda", "Ford", "Hyundai"]
2
3 #The last index
4 print(carBrands[-1])
5
6 #The rest
7 print(carBrands[-2])
8 print(carBrands[-3])
9
```



Hyundai
Ford
Honda

IndexError

Python won't let you index an object out of bounds and will raise an `IndexError` if you do

Example

```
1 carBrands = ["Honda", "Ford", "Hyundai"]  
2 print(carBrands[5])
```

```
IndexError: list index out of range
```

Splicing

Splicing is how you get a specific range of a list

Note: The start is always inclusive and the end is always non inclusive

What do you think these will print?

```
1 carBrands = ["Honda", "Ford", "Hyundai", "BMW"]
2
3 #From index to end
4 print(carBrands[2:])
5
6 #Between 2 indexes
7 print(carBrands[2:4])
8
9 #From start to index
10 print(carBrands[:3])
```

Splicing Solution

```
1 carBrands = ["Honda", "Ford", "Hyundai", "BMW"]
2
3 #From index to end
4 print(carBrands[2:])
5
6 #Between 2 indexies
7 print(carBrands[2:4])
8
9 #From start to index
10 print(carBrands[:3])
```

```
['Hyundai', 'BMW']
['Hyundai', 'BMW']
['Honda', 'Ford', 'Hyundai']
```

Skip Indexing

I honestly don't find this very useful but just note that it is something that can be done.

Remember from last lecture we learned that we can splice an array using : and specifying a start and an end but we can also use the form [start:end:step interval]

```
1 carBrands = ["Honda", "Ford", "Hyundai", "BMW", "Toyota"]  
2 print(carBrands[0:5:2])  
3 print(carBrands[1:5:2])
```

```
['Honda', 'Hyundai', 'Toyota']  
['Ford', 'BMW']
```


Mutating lists by index/splicing

Just like variables you can reassign them to different values.

Using Indexes:

```
1 myList = [1,2,3,4,5]
2 myList[3] = "Ferrari"
3 print(myList)
```

```
[1, 2, 3, 'Ferrari', 5]
```

Using Splicing:

```
1 myList = [1,2,3,4,5]
2 myList[1:3] = ["Ferrari"]
3 print(myList)
```

```
[1, 'Ferrari', 4, 5]
```

Arithmetic Operations on Lists

This isn't used very often if ever cause most of the time there are better ways of getting around having to do this but it can come in handy at times.

You can add two lists using +

You can also repeat the contents of the list with *

Addition example:

```
1 a = [1,2]
2 b = [2]
3
4 print(a + b)
```

```
[1, 2, 2]
```

Multiplication example:

```
1 a = [1,2]
2
3 print(a * 3)
```

```
[1, 2, 1, 2, 1, 2]
```

Iteration

Since lists are a type of data structure we need a way to iterate or navigate through it.

This is where loops come in handy and specifically for loops when it comes to basic lists.

Remember that the general format of a for loop is:

for object in container:

Do task

In the last lecture we used a function called `range()`. Although `range` is technically a special kind of function that won't be covered until ICS 33 you can simply think of it for now as a type of container

Iteration cont.

Note: When working with lists you will most likely be working with for loops. Although while loops will come into play in certain scenarios.

Example of iteration through a list:

Output:

```
1 car = ["Ferrari", "McLaren", "Aston Martin"]
2 for i in car:
3     print(i)
```

```
Ferrari
McLaren
Aston Martin
```

We will be expanding on the application of lists and their functions and methods with loops in a coding demonstration

What does immutable mean?

Being immutable simply means the object can not change and in python all the basic data structures have an immutable form.

- Lists -> Tuple
- Sets -> Frozenset
- Dictionary -> Frozendictionary

Why would we want to make something immutable?

For the level of this course as well as almost 90% of the ICS 30 series this isn't really useful for the topics that the classes cover.

If you are coming from a different language you might be familiar with “private” and “public.”

In python nothing is private therefore everything is public and one of the ways to get around this by using immutable data structures as well as specific naming conventions that you will learn later on.

What are tuples?

As stated before, tuples are the immutable version of a list which means they can not be modified. Otherwise you can do the same things to them as you would a list.

You can distinguish a tuple from a list by its use of `()` instead of `[]` or by using the `type()` function

```
1 car = ()
2 print(car)
3
4 car = ("Ferrari", "McLaren" , "Aston Martin")
5 print(car)
6
```

```
()
('Ferrari', 'McLaren', 'Aston Martin')
```

What can we do with tuples

Anything that you can do with a list, you can do with a tuple.
Iteration, indexing, common methods, etc **AS LONG AS YOU
ARE NOT MODIFYING THE TUPLE ITSELF!**

Strings Review

Strings are a lot like tuples since they are an array of immutable singular characters and therefore behave very similarly but with a few special methods.

```
1 myString = "This is a string"  
2 myString = 'This is also a string'
```

You can also use `str()` to convert anything into a string

Strings are special

If you are coming from a C variant you probably know that strings are not a built in data type but it is in python.

Strings are special because although they are a basic built in data type they have special properties similar to arrays/lists

Essentially strings are nothing more than **an array of singular character values** called “chars” (not found in python but are in other languages)

As a result a lot of the things that you can do with lists you can do the same with strings

Strings are special

Strings are immutable so technically they are more like tuples in Python terms.

So all the things that you can do with tuples you can do with strings plus strings also have their own special functions that we will be covering in the next lecture.

Something that I noticed: When looking over some of the code for assignment 2 that people have sent me I noticed that some of you did “string” * int.

This is possible as you can do the same thing with lists/tuples.

String Similarities

Indexing:

```
1 myString = "Ferrari"  
2 print(myString[0])  
3 print(myString[5])
```

```
F  
r
```

Just like lists and tuples you can index them and loop through them the same way. Like tuples you can not mutate the string since they are immutable. Ex: `myString[0] = "a"` won't work

Arithmetic:

```
1 myString = "Ferrari"  
2 myString2 = " is cool"  
3 print(myString + myString2)
```

```
Ferrari is cool
```

```
1 myString = "Ferrari"  
2 print(myString * 3)  
3
```

```
FerrariFerrariFerrari
```

String Iteration

Just to show how strings are like arrays when you iterate through them you get each and every character just like if you were to index them.

```
1 myString = "Ferrari"  
2 for i in myString:  
3     print(i)  
4
```

F
e
r
r
a
r
i

Common Methods and Functions on Strings

Basic Functions

- `print()`
- `len()`
- `str()`

Common Methods

- `.split()`
- `.strip()`
- `.upper()`
- `.lower()`
- `.format()`
 - Don't use this use f string instead. Works the same way but isn't as clean
- `.replace()`

Common Methods

string.split("")

```
1 myString = "Ferrari.Mclaren.AstonMartin"
2 print(myString.split("."))
3
```

```
['Ferrari', 'Mclaren', 'AstonMartin']
```

string.strip()

```
1 myString = "    Ferrari    "
2 print(myString.strip())
3
```

```
Ferrari
```

Common Methods

string.upper()

```
1 myString = "Ferrari"  
2 print(myString.upper())
```

FERRARI

string.lower()

```
1 myString = "Ferrari"  
2 print(myString.lower())  
3
```

ferrari

.upper() and .lower()

```
1 while cout.upper() != "NO":|
```

Although I didn't ask for it I noticed that some of you did do it for assignment 3.

Based on the solutions for assignment 3 you might of thought what if they enter NO or no, those will count as invalid inputs even though the user meant "No"

This could be fixed using .upper() or .lower() so we can modify all possible inputs to be uniform.

Common Methods

string.format()

```
1 myString = "{} is my favorite car from {}"  
2 print(myString.format("S2000", "Honda"))
```

```
S2000 is my favorite car from Honda
```

string.replace()

```
1 myString = "hello.my.name.is.Trung"  
2 print(myString.replace(".", " "))
```

```
hello my name is Trung
```

F strings

This is the best way to format a string and should be the only way to do it. F string stands for “formatted” string. It works the same way as `.format` but is more intuitive since you can put the variables directly into the string itself.

```
1 car = "s2000"
2 brand = "Honda"
3 print(f"{car} is my favorite car from {brand}")
```

```
s2000 is my favorite car from Honda
```

Strings Type Conversion with lists and tuples

Since strings are a lot like lists/tuples they can be implicitly type converted to lists or tuples but not the other way around

Example: Strings -> list/tuple

```
1 myString = "BMW"
2 print(list(myString))
3 print(tuple(myString))
```

```
['B', 'M', 'W']
('B', 'M', 'W')
```

Example: list/tuple -> string

(does not work)

```
1 myList = ['a', 'b', 'c']
2 print(str(myList))
```

```
['a', 'b', 'c']
```

Introduction to Sets

Sets are a little weird....

Just like lists and tuples they store objects but they are unordered and every item in a set is unique.

I honestly don't find them very useful but it's something that you have to know

Note: sets are defined by {} but it is the only data structure where you can not use its container format to initialize an empty version of the structure

```
1 #Empty set
2 mySet = set()
3 print(mySet)
4
5 #Set with objects
6 mySet = {1,2,3,4}
7 print(mySet)
```

```
set()
{1, 2, 3, 4}
```

Sets Type Conversion

Again this is a little bit weird

You can implicitly type convert sets to strings, lists and tuples as well as the other way around.

Since sets are unordered and only contain unique items you can get weird results when converting them.

```
1 myList = [1,2,3,3,4]
2 print(set(myList))
3
4 myTuple = (1, 2, 3, 3, 4)
5 print(set(myTuple))
6
7 myString = "Ferrari"
8 print(set(myString))
```

```
{1, 2, 3, 4}
{1, 2, 3, 4}
{'e', 'r', 'a', 'F', 'i'}
```

As you can see all items that weren't unique got removed and the ordering of the object can be messed up.

Common Set Methods

- `.add()`
 - Adds an item to a set
- `.remove()`
 - Removes an item from a set
- `.issubset()`
 - Returns a boolean depending on if the set is a subset or not
- `.copy()`
 - Returns a copy
- `.clear()`
 - Clears the set
- `.intersection()`
 - Returns the intersection

Common Methods

.add()

```
1 set_cars = {"BMW", "Hyundai", "Ford"}
2 set_cars.add("Honda")
3 print(set_cars)
```

```
{'Ford', 'Hyundai', 'BMW', 'Honda'}
```

.remove()

```
1 set_cars = {"BMW", "Hyundai", "Ford"}
2 set_cars.remove("BMW")
→ 3 print(set_cars)
```

```
{'Hyundai', 'Ford'}
```


Common Methods

.issubset()

```
1 set_cars = {"BMW", "Hyundai", "Ford", "Honda"}
2
3 set_cars_sub = {"BMW", "Hyundai", "Ford"}
4
5 print(set_cars_sub.issubset(set_cars))
```

True

.copy()

```
1 set_cars = {"BMW", "Hyundai", "Ford", "Honda"}
2
3 set_cars_copy = set_cars.copy()
4
5 print(set_cars_copy)
```

{'BMW', 'Honda', 'Hyundai', 'Ford'}

Common Methods

.clear()

```
1 set_cars = {"BMW", "Hyundai", "Ford", "Honda"}
2
3 set_cars.clear()
4
5 print(set_cars)
```

set()

.intersection()

```
1 set_cars = {"BMW", "Hyundai", "Ford", "Honda"}
2 set_cars_1 = {"BMW", "Hyundai", "Ford", "Kia"}
3 print(set_cars.intersection(set_cars_1))
```

```
{'Hyundai', 'BMW', 'Ford'}
```

Basic List/Tuple Algorithms

These are the some of the basic algorithms regarding lists and tuples that you will need to know and understand. This is simply a list for you to reference but they are all demonstrated in the coding demo for this lecture.

- Traversing
- Mapping
- Filtering
- Counting
- Calculation