

# Lecture 4

Dictionaries, Named Tuples, Nested Data Structs

# Dictionaries

Dictionaries are like sets where the items in it are unordered and unique because of this you can't index into it and you can't iterate through it like an array.

This is my absolute favorite built in data structure and from what I'm aware of it's unique to python. **Note:** in other languages you can build a hash map and this is similar.

```
1 #empty dictionary
2 myDict = {}
3 print(myDict)
4
5 #Dictionary with objects
6 myDict = {"key1" : "a", "key2" : "b", "key3" : "c"}
7 print(myDict)
```

```
{ }
{'key1': 'a', 'key2': 'b', 'key3': 'c'}
```

# How do you use a dictionary?

To use a dictionary we have to understand it's two terms.

## Key

- Used to access the dictionary's contents
- Keys are **UNIQUE and IMMUTABLE**

## Value

- The objects stored with every key
- Accessed by `dictionary[key] = value`
- Values are **MUTABLE and** don't have to be **UNIQUE**

# Dictionary Example

To add to a dictionary you give the dictionary a key and assign it to a value

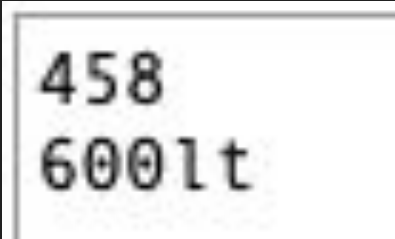
```
1 myDict = {}  
2  
3 #Adds the key "Ferrari" with a value of "458" to the dictionary  
4 myDict["Ferrari"] = "458"  
5 #Adds the key "Mclaren" with a value of "600lt" to the dictionary  
6 myDict["Mclaren"] = "600lt"  
7  
8 print(myDict)
```

```
{'Ferrari': '458', 'Mclaren': '600lt'}
```

# Dictionary Example

You can get the value associated with its key by specifying a dictionary and giving it a key

```
1 myDict = {"Ferrari" : "458", "McLaren" : "600lt"}  
2  
3 print(myDict["Ferrari"])  
4 print(myDict["McLaren"])  
5
```



458  
600lt

# Key Error

You can't access a dictionary with a key that does not exist. If you do you will get a key error.

```
1 myDict = {"Ferrari" : "458", "McLaren" : "600lt"}  
2  
3 print(myDict["Aston Martin"])
```

KeyError: 'Aston Martin'

# Common Dictionary Methods

`.keys()`

- Returns a list of the dictionary's keys

`.values()`

- Returns a list of the dictionary's values

`.clear()`

- Removes all objects in the dictionary

`.copy()`

- Makes a copy of the dictionary

`.items()`

- Returns a list of tuples of every key, value pair

`.pop()`

- Removes an object given a key
- I use `del` more

# Common Methods

## dict.keys()

```
1 myDict = {"Ferrari" : "458", "McLaren" : "600lt"}
2
3 print(myDict.keys())
```

```
dict_keys(['Ferrari', 'McLaren'])
```

Note: it looks like this when you print it cause .keys() and .values() is similar to the range() function which is what we call a generator function but that is out of the range of this course. **Just know you can treat it like a regular list when iterating.**

## dict.values()

```
1 myDict = {"Ferrari" : "458", "McLaren" : "600lt"}
2
3 print(myDict.values())
```

```
dict_values(['458', '600lt'])
```

## Example:

```
1 myDict = {"Ferrari" : "458", "McLaren" : "600lt"}
2
3 print(myDict.keys())
4 for i in myDict.keys():
5     print(i)
```

```
dict_keys(['Ferrari', 'McLaren'])
Ferrari
McLaren
```



# Common Methods

dict.items()

```
1 myDict = {"Ferrari" : "458", "McLaren" : "600lt"}
2
3 print(myDict.items())
4 for i in myDict.items():
5     print(i)
```

```
dict_items([('Ferrari', '458'), ('McLaren', '600lt')])
('Ferrari', '458')
('McLaren', '600lt')
```

These are the main 3 methods that you will be using most of the time the other methods such as copy and clear should be self explanatory.

Notice that .items() is also a generator function just like keys() and values()

# Named Tuples

Just like lists, tuples, dictionaries, etc. this is also a type of data structure but it is not natively built into the language and will need to be imported.

Note: This is taught in 31 but it is never used again and quite frankly it isn't very useful but it is something that you should know just incase it comes up on the ICS 31 exam

# How do we use Named Tuples

We will go more in depth about what external modules and how to use them more later in the course but for now to use them this is what you do:

```
from collections import namedtuple
```

We then assign it to a variable with a “blueprint” of what we want it to store

```
3 Cars = namedtuple("Cars", "brand model year")
```

We can now use this blueprint to create more objects that

```
5 german = Cars(brand = "BMW", model = "M3", year = 2008)
6 japanese = Cars(brand = "Honda", model = "s2000", year = 2004)
7
```

# What is named tuple's type?

The type of objects that you create using a named tuple template is the name of the template itself from the first parameter.

```
1 from collections import namedtuple
2
3 Cars = namedtuple("Cars", "brand model year")
4
5 german = Cars(brand = "BMW", model = "M3", year = 2008)
6 print(type(german))
```

```
<class '__main__.Cars'>
```

# Accessing values in Named Tuples

To access objects that we store in a named tuple we would call on the named tuple and state the object that we want to retrieve

```
1 from collections import namedtuple
2
3 Cars = namedtuple("Cars", "brand model year")
4
5 german = Cars(brand = "BMW", model = "M3", year = 2008)
6 japanese = Cars(brand = "Honda", model = "s2000", year = 2004)
7
8 print(german.brand)
9 print(german.model)
10 print(german.year)
```

BMW
M3
2008

# Named Tuple Methods

To change a value stored in a named tuple we need to use the `._replace()` method

We have to reassign the variable `german` since named tuples are immutable.

This is also why we can't do `german.brand = "Audi"`

```
1 from collections import namedtuple
2
3 Cars = namedtuple("Cars", "brand model year")
4
5 german = Cars(brand = "BMW", model = "M3", year = 2008)
6
7 print(german.brand)
8 print(german.model)
9 german = german._replace(brand = "Audi", model = "r8")
10 print(german.brand)
11 print(german.model)
```

BMW  
M3  
Audi  
r8

```
1 from collections import namedtuple
2
3 Cars = namedtuple("Cars", "brand model year")
4
5 german = Cars(brand = "BMW", model = "M3", year = 2008)
6
7 #Doesnt work
8 print(german.brand)
9 german.brand = "Audi"
10 print(german.brand)
```

AttributeError: can't set attribute

**Single-value data structures (all immutable):** `int` `float` `bool` (We often treat **string** information—textual data—as a single value, though it's technically multiple-value as shown below.)

Type	Constant values	Selected operators	Selected functions	Uses
<b>int</b>	0    -17    238653489	+ - * / % // ** == != > < >= <=	<code>abs()</code> <code>max()</code> <code>min()</code>	Counting things; indexing strings or lists
<b>float</b>	3.0   -2.00053   3.5e19	+ - * / % // ** == != > < >= <=	<code>abs()</code> <code>max()</code> <code>min()</code>	Calculating real-world quantities
<b>bool</b>	True    False	<code>and</code> <code>or</code> <code>not</code> <code>==</code> <code>!=</code>		Results of comparisons, yes-or-no questions

**Multiple-value data structures:** `string` `list` `namedtuple` `file` `dict` `set` `tuple`

Type	Mutable (can change in place)?	Sequence (are items in order)?	Constant values	Selected operators and functions	Selected methods	Main characteristics	Uses
<b>string</b>	No	Yes	<code>' '</code> <code>'Four score and 7 ...'</code> <code>"We the people ..."</code> <code>''' Multi-line string can span many lines '''</code>	<code>in</code> <code>+</code> <code>*</code> <code>[]</code> <code>[:]</code> <code>len()</code> <code>==</code> <code>!=</code> <code>&gt;</code> <code>&lt;</code> <code>&gt;=</code> <code>&lt;=</code> <code>int()</code> <code>float()</code>	<code>S.find()</code> <code>S.split()</code> <code>S.strip()</code> <code>S.format()</code> <code>S.translate()</code>	Indexable sequence of characters, can take "slices" (substrings)	Text, names, words, "ASCII graphics" like *****
<b>list</b> (also called array)	Yes	Yes	<code>[ ]</code> <code>[1, 2, 3]</code> <code>['Hello!']</code>	<code>in</code> <code>+</code> <code>*</code> <code>[]</code> <code>[:]</code> <code>len()</code> <code>min()</code> <code>max()</code> <code>==</code> <code>!=</code> <code>&gt;</code> <code>&lt;</code> <code>&gt;=</code> <code>&lt;=</code>	<code>L.count()</code> <code>L.sort()</code> <code>L.append()</code> <code>L.extend()</code>	Indexable sequence, can take "slices"	General collection of same-type objects
<b>namedtuple</b> (also called structure or record)	No	Yes	<code>Restaurant('Taillevent', 'French', '333-4444', 'Gateau Marjolaine', 33.00)</code>	<code>.</code> [to specify field/attribute] <code>print(R.name, 'has', R.dish, 'for \$', R.price)</code>	<code>R = R._replace(name='Woma')</code>	Access to components by attribute/field name	from collections <code>import namedtuple</code> Grouping components of an object (e.g., student name, ID, major, GPA)
<b>file</b>	No	Yes		<code>inf = open('blah.txt', 'r')</code> <code>outf = open('ans.txt', 'w')</code>	<code>nextline = inf.readline()</code> <code>linelist = inf.readlines()</code> <code>outf.writelines()</code> <code>outf.close()</code>	Files give you <b>persistence</b> : Their contents stay around after the program stops running.	Saving data from one run to the next; preparing input to another program; packaging data to send over the net
<b>dict</b> (also map, table, dictionary)	Yes	No	<code>{ }</code> <code>{ 'Joe':25, 'Jan': 47, 'Jill':44, 'Jim': 45 }</code>	<code>in</code> <code>[ ]</code> <code>len()</code> <code>==</code> <code>!=</code> <code>&gt;</code> <code>&lt;</code> <code>&gt;=</code> <code>&lt;=</code>	<code>d.items()</code> <code>#view</code> <code>d.keys()</code> <code>#view</code> <code>d.values()</code> <code>#view</code>	No order; look up value by key	Programmer-defined indexes, fast lookup
<b>set</b>	Yes	No	<code>set()</code> <code>{ 'a', 'b', 'c' }</code>	<code>in</code> <code>len()</code> <code>&gt;</code> <code>&lt;</code> <code>&gt;=</code> <code>&lt;=</code>	<code>S.add()</code> <code>S.remove()</code>	No order, no duplicates	Eliminate duplicates
<b>tuple</b>	No	Yes	<code>( )</code> <code>('Joe', 'Smith', 19, 'Informatics')</code> <code>('Donald',) # One item</code>	<code>in</code> <code>[ ]</code> <code>len()</code> <code>==</code> <code>!=</code> <code>&gt;</code> <code>&lt;</code> <code>&gt;=</code> <code>&lt;=</code>	<code>T.count()</code>	Grouped data, access by position	Short multi-part data packages where field names not needed

# What exactly is an object

This can be quite a difficult explanation with only the knowledge that we have covered but for what we need to know for this next part you can think of objects as a collection of data.

So far everything that we've covered, lists, dictionaries, sets, integers, booleans, strings, including named tuples, etc. are all objects and we can work on these objects using built in functions and methods.

Remember that our data structures are nothing more than objects that hold other objects so what's stopping them from holding each other? And with this we will start going into multi-dimensional arrays.

We are not going to get super deep into this cause it can get complicated and its out of the skill level for this course so this is simply an introduction.



# Nesting Data Structures: 2D arrays

The most standard nested data structure that you will find is a list of lists. A singular list is a one dimensional array but when you make a list of lists you create a matrix or a two dimensional array.

```
1 cars = ["GTC4 Lusso", "812 SF", "488 GTB", "Portofino"],  
2       ["Huracan", "Aventador", "Urus"],  
3       ["570s", "600lt", "GT", "720s"]  
4
```

Note: you can do this in one line but when formatting nested arrays you should do something similar to this example

# How do we access values in nested arrays?

```
1 cars = ["GTC4 Lusso", "812 SF", "488 GTB", "Portofino"],
2       ["Huracan", "Aventador", "Urus"],
3       ["570s", "600lt", "GT", "720s"]
4
```

Accessing nested arrays is the same as accessing one dimensional arrays but require more indexes depending on how deep you want to go

Example: can you tell me what these will print?

```
1 cars = ["GTC4 Lusso", "812 SF", "488 GTB", "Portofino"],
2       ["Huracan", "Aventador", "Urus"],
3       ["570s", "600lt", "GT", "720s"]
4
5 print(cars) #This gives you a list of lists
6 print(cars[0]) #This gives you a list of strings
7 print(cars[0][0]) #This gives you a string
```

# Nested Arrays

```
1 cars = ["GTC4 Lusso", "812 SF", "488 GTB", "Portofino"],
2       ["Huracan", "Aventador", "Urus"],
3       ["570s", "600lt", "GT", "720s"]]
4
5 print(cars) #This gives you a list of lists
6 print(cars[0]) #This gives you a list of strings
7 print(cars[0][0]) #This gives you a string
8
```

```
[['GTC4 Lusso', '812 SF', '488 GTB', 'Portofino'], ['Huracan', 'Aventador', 'Urus'], ['570s', '600lt', 'GT', '720s']]
['GTC4 Lusso', '812 SF', '488 GTB', 'Portofino']
GTC4 Lusso
```

# How do we iterate through nested arrays

We recently learned that we could use loops to iterate through arrays so to iterate through a nested array we would use a nested loop

```
1 cars = ["GTC4 Lusso", "812 SF", "488 GTB", "Portofino"],  
2       ["Huracan", "Aventador", "Urus"],  
3       ["570s", "600lt", "GT", "720s"]  
4  
5 for brand in cars:  
6     print(brand)  
7     for model in brand:  
8         print(model)
```

```
['GTC4 Lusso', '812 SF', '488 GTB', 'Portofino']  
GTC4 Lusso  
812 SF  
488 GTB  
Portofino  
['Huracan', 'Aventador', 'Urus']  
Huracan  
Aventador  
Urus  
['570s', '600lt', 'GT', '720s']  
570s  
600lt  
GT  
720s
```

# List of Named Tuples

This one should be pretty self explanatory and the elements are accessed the same way

```
11 Cars = namedtuple("Cars", "brand model year")
12
13 german = Cars(brand = "BMW", model = "M3", year = 2008)
14 japanese = Cars(brand = "Honda", model = "s2000", year = 2004)
15
16 myList = [german, japanese]
17 print(myList[0]) #This gets you a named tuple at index 0
18 print(myList[0].model) #This gets you the model of index 0
```

```
Cars(brand='BMW', model='M3', year=2008)
M3
```

# Iteration can also be done in a similar way

```
Cars = namedtuple("Cars", "brand model year")  
  
german = Cars(brand = "BMW", model = "M3", year = 2008)  
japanese = Cars(brand = "Honda", model = "s2000", year = 2004)  
  
myList = [german, japanese]  
  
for car in myList:  
    print(car.model)
```

```
M3  
s2000
```

Note: I am not using a nested loop because this the named tuple object is not an array.

I am going through my list of named tuples which are countries and printing every the group's model

The same can be done with the other variables stored in the named tuple

# Named tuples of Lists

```
Country = namedtuple("Country", "brands models")  
  
germany = Country(brands = ["BMW", "VW", "Mercedes"], models = ["M3", "Golf R", "SLR"])  
  
print(germany.brands)
```

```
['BMW', 'VW', 'Mercedes']
```

Take note of how I am accessing the named tuples. For example germany itself is a named tuple which is not an array but by accessing one of it's values the accessed variable IS an array

# Iterating through arrays stored in data structures

Say I wanted to iterate through the list of songs that is stored in the named tuple seen in last slide. Remember that Named tuple is can not be iterated through but that does not mean it can't store iterable objects within it.

```
8 Country = namedtuple("Country", "brands models")
9
10 germany = Country(brands = ["BMW", "VW", "Mercedes"], models = ["M3", "Golf R", "SLR"])
11
12 for brand in germany.brands:
13     print(brand)
```

```
BMW
VW
Mercedes
```



# Nested Arrays Cont.

The three that we just went over are the ones stated in the ICS documentation but this can be extended with all data structures to infinite dimensions. The possibilities are limitless.

For example in ICS 33 you will have a dictionary of dictionaries of lists of dictionaries.

# Dictionary of lists

These are some of my most used nested data structures so I just wanted to go over it cause you might like them as well. (Note: I tend to use nested dictionaries a lot but it can get very messy if you aren't use to it so we're gonna start here)

```
1 cars = {"BMW" : ["M2", "M3", "M4", "M5"],  
2         "VW" : ["Golf R", "GTI"],  
3         "Mercedes" : ["SLR", "SLS", "GTS"]}  
4
```

These are really nice for storing a lot of data and being able to keep them more organized

# Accessing Values

```
1 cars = {"BMW" : ["M2", "M3", "M4", "M5"],  
2         "VW" : ["Golf R", "GTI"],  
3         "Mercedes" : ["SLR", "SLS", "GTS"]}  
4  
5 print(cars["BMW"])  
6 print(cars["BMW"][0])
```

```
['M2', 'M3', 'M4', 'M5']  
M2
```

# Iterating

```
1 cars = {"BMW" : ["M2", "M3", "M4", "M5"],  
2         "VW" : ["Golf R", "GTI"],  
3         "Mercedes" : ["SLR", "SLS", "GTS"]}  
4  
5 for brand in cars.keys():  
6     for model in cars[brand]:  
7         print(model)
```

```
M2  
M3  
M4  
M5  
Golf R  
GTI  
SLR  
SLS  
GTS
```

First I am going through all the keys which are groups.

Second I am going through each member as cars[brand] returns a list of models from every brand

You can do it this way too but I like the first way better

```
5 for brand in cars.values():  
6     for model in brand:  
7         print(model)
```

# Nested Data Structures Drawing Example / Visualization

# What are functions?

Functions are objects that we create that will help us reduce the redundancy of our code.

```
1 def myFunction(parameter):  
2     pass
```