TechOrigin

Intro to Python

Access this notebook on GITHUB or COLAB

Table of Contents

Click on the links to go directly to specific sections on the notebook.

Python - Let's get you writing some Python Code now!</h1>

Welcome! This notebook will teach you about the tuples, list, sets and dictionaries in the Python Programming Language. By the end of this lab, you'll know their basic operations in Python, including indexing, slicing, sorting, updating, deleting and so on.

Import Dependencies

In [ ]:
```
%%expect_exception TypeError
```

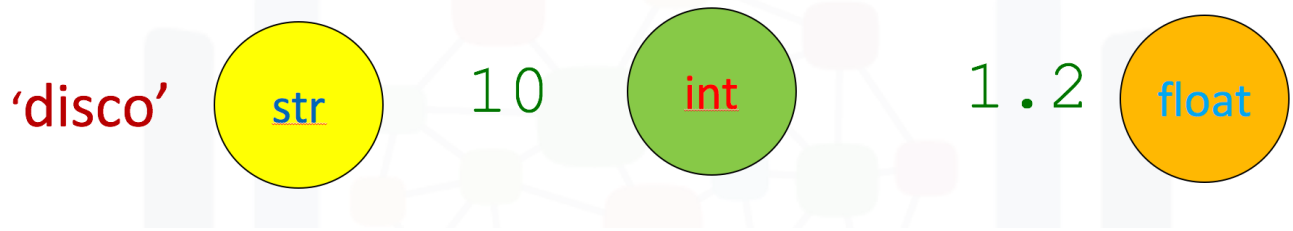
Tuples

Welcome! This notebook will teach you about the tuples in the Python Programming Language. By the end of this lab, you'll know the basics tuple operations in Python, including indexing, slicing and sorting.

About this Dataset

What is in a Tuple

In Python, there are different data types: string, integer and float. These data types can all be contained in a tuple as follows:



Now, let us create your first tuple with string, integer and float.

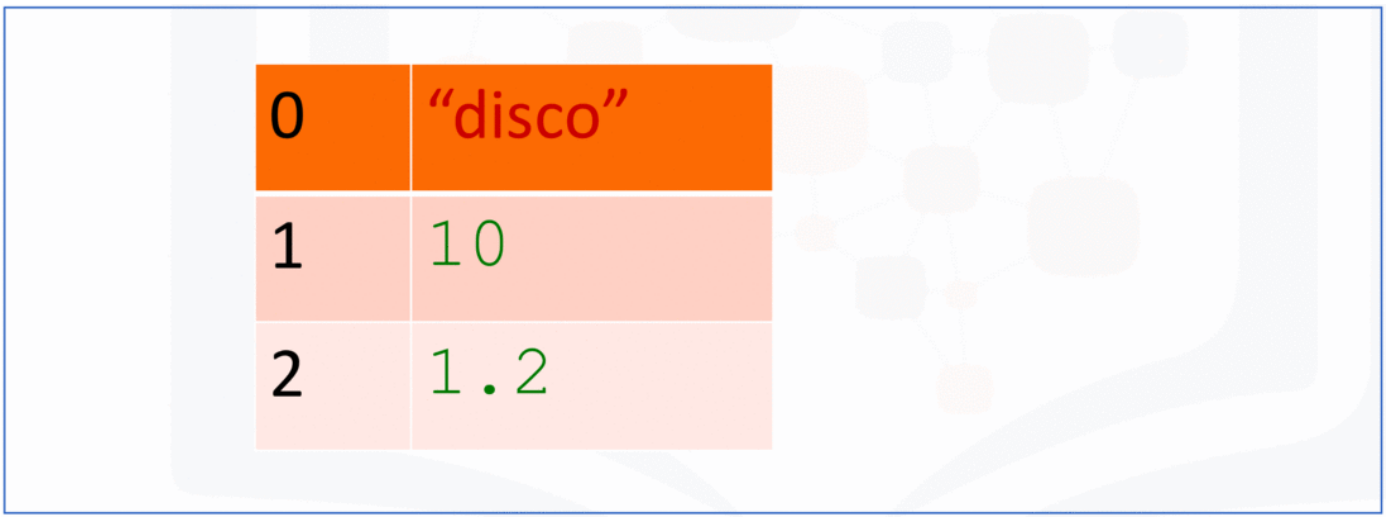
In [ ]:
```
# Create your first tuple

tuple1 = ("disco",10,1.2 )
tuple1
```

The type of variable is a **tuple**.

In [ ]:
```
# Print the type of the tuple you created

type(tuple1)
```

Indexing Tuples

Each element of a tuple can be accessed via an index. The following table represents the relationship between the index and the items in the tuple. Each element can be obtained by the name of the tuple followed by a square bracket with the index number:



We can print out each value in the tuple:

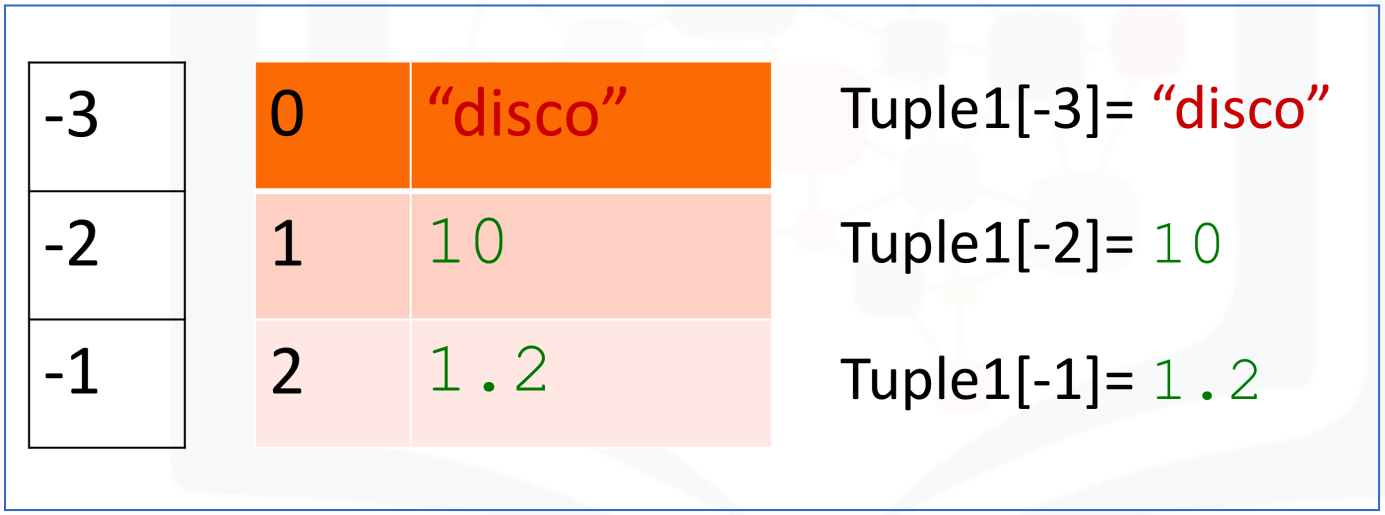
In [ ]:
```
# Print the variable on each index
```

```
print(tuple1[0])
print(tuple1[1])
print(tuple1[2])
```

We can print out the **type** of each value in the tuple:

In [ ]:
```
# Print the type of value on each index

print(type(tuple1[0]))
print(type(tuple1[1]))
print(type(tuple1[2]))
```

We can also use negative indexing. We use the same table above with corresponding negative values:



We can obtain the last element as follows (this time we will not use the print statement to display the values):

In [ ]:
```
# Use negative index to get the value of the last element

tuple1[-1]
```

We can display the next two elements as follows:

In [ ]:
```
# Use negative index to get the value of the second last element

tuple1[-2]
```

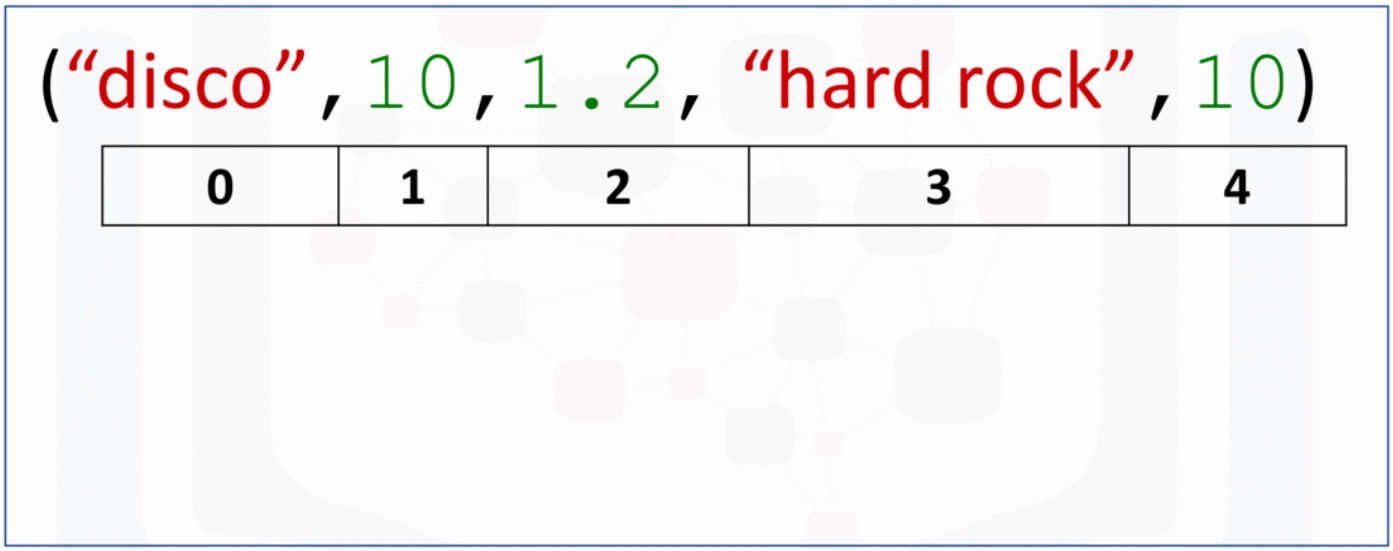
In [ ]:
```
# Use negative index to get the value of the third last element

tuple1[-3]
```

Concatenate Tuples

We can concatenate or combine tuples by using the + sign:

In [ ]:
```
# Concatenate two tuples

tuple2 = tuple1 + ("hard rock", 10)
tuple2
```

We can slice tuples obtaining multiple values as demonstrated by the figure below:



Slicing Tuples

We can slice tuples, obtaining new tuples with the corresponding elements:

In [ ]:
```
# Slice from index 0 to index 2

tuple2[0:3]
```

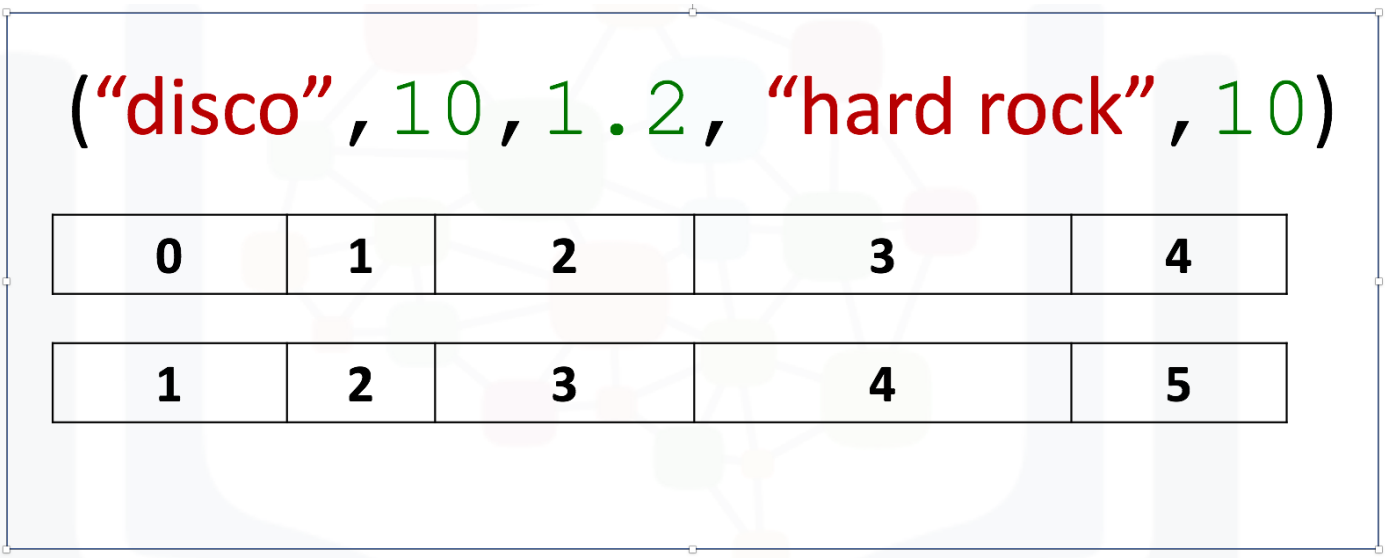
We can obtain the last two elements of the tuple:

In [ ]:
```
# Slice from index 3 to index 4

tuple2[3:5]
```

We can obtain the length of a tuple using the length command:

In [ ]:
```
# Get the length of tuple

len(tuple2)
```

This figure shows the number of elements:



Sorting Tuples

Consider the following tuple:

In [ ]:
```
# A sample tuple

Ratings = (0, 9, 6, 5, 10, 8, 9, 6, 2)
```

We can sort the values in a tuple and save it to a new tuple:

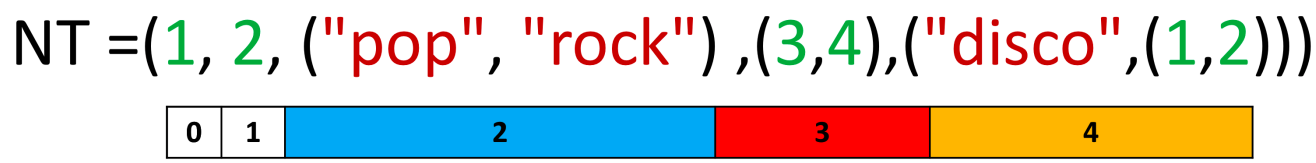
In [ ]:
```
# Sort the tuple

RatingsSorted = sorted(Ratings)
RatingsSorted
```

Nested Tuple

A tuple can contain another tuple as well as other more complex data types. This process is called 'nesting'. Consider the following tuple with several elements:

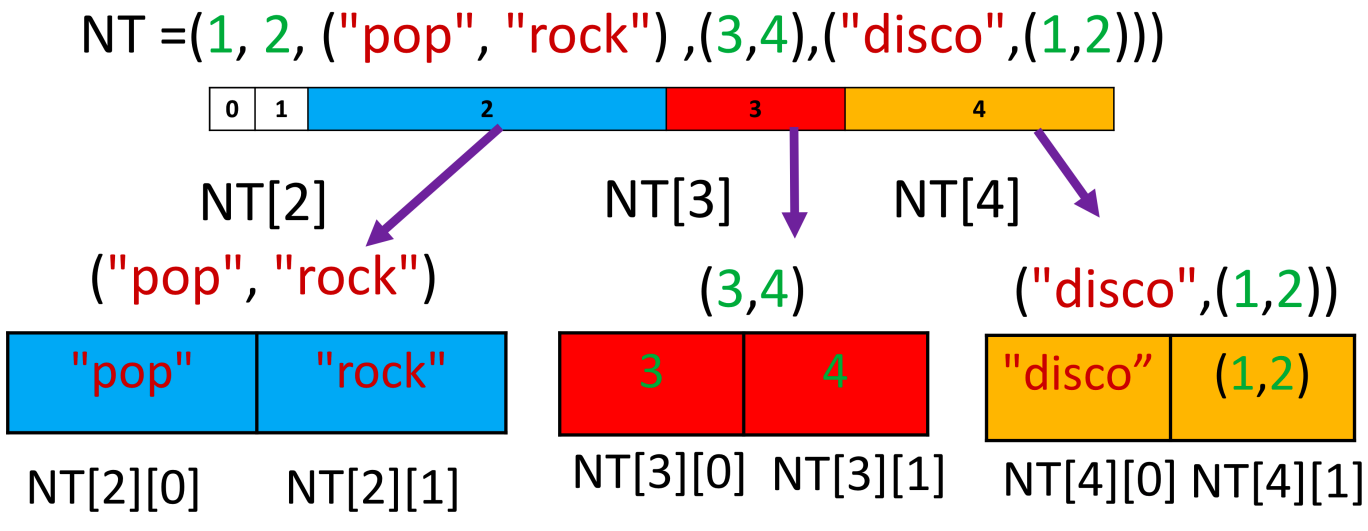
In [ ]:
```
# Create a nest tuple

NestedT =(1, 2, ("pop", "rock") ,(3,4),("disco",(1,2)))
```

Each element in the tuple including other tuples can be obtained via an index as shown in the figure:



In [ ]:
```
# Print element on each index

print("Element 0 of Tuple: ", NestedT[0])
print("Element 1 of Tuple: ", NestedT[1])
print("Element 2 of Tuple: ", NestedT[2])
print("Element 3 of Tuple: ", NestedT[3])
print("Element 4 of Tuple: ", NestedT[4])
```

We can use the second index to access other tuples as demonstrated in the figure:



We can access the nested tuples :

In [ ]:
```
# Print element on each index, including nest indexes

print("Element 2, 0 of Tuple: ",   NestedT[2][0])
print("Element 2, 1 of Tuple: ",   NestedT[2][1])
print("Element 3, 0 of Tuple: ",   NestedT[3][0])
print("Element 3, 1 of Tuple: ",   NestedT[3][1])
print("Element 4, 0 of Tuple: ",   NestedT[4][0])
print("Element 4, 1 of Tuple: ",   NestedT[4][1])
```

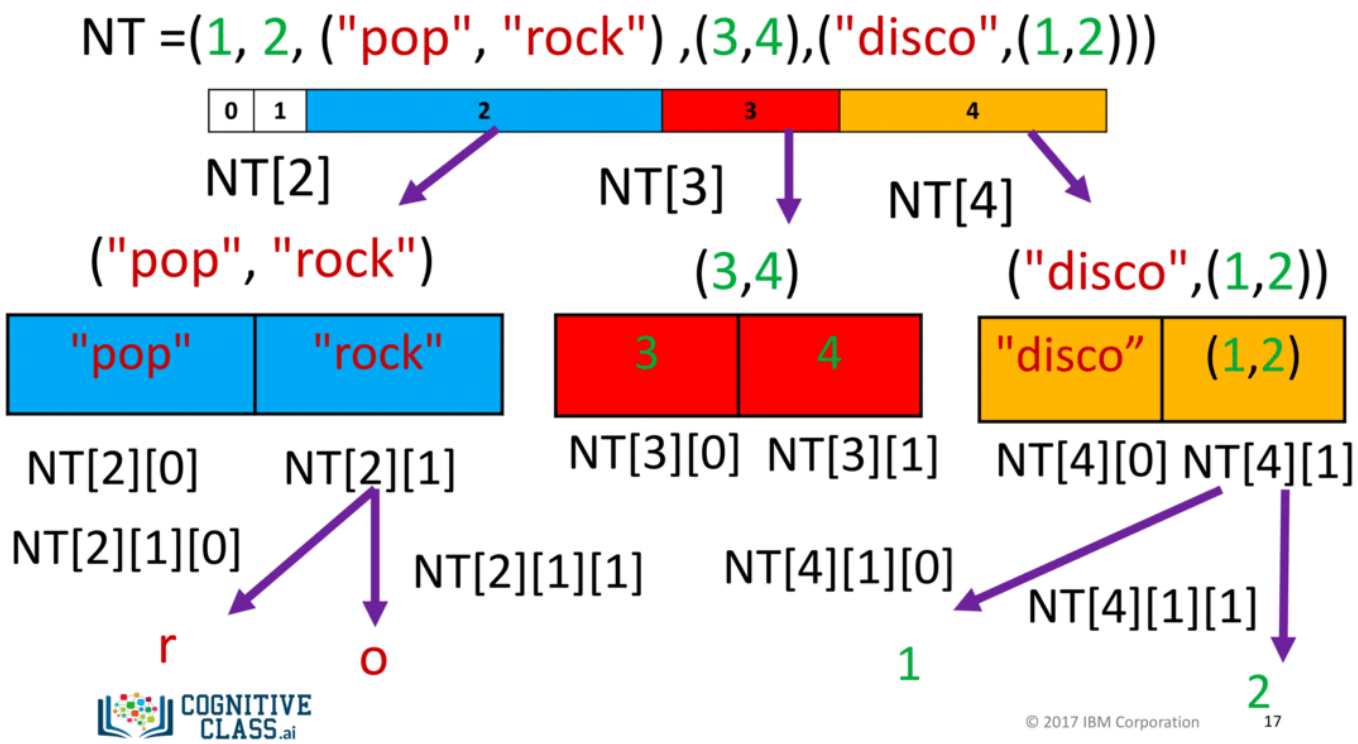
We can access strings in the second nested tuples using a third index:

In [ ]:
```
#print the first element in the first nested tuples

NestedT [2][0][0]
```

In [ ]:
```
# Print the first element in the second nested tuples

NestedT[2][1][0]
```

In [ ]:
```
# Print the second element in the second nested tuples

NestedT[2][1][1]
```

We can use a tree to visualise the process. Each new index corresponds to a deeper level in the tree:



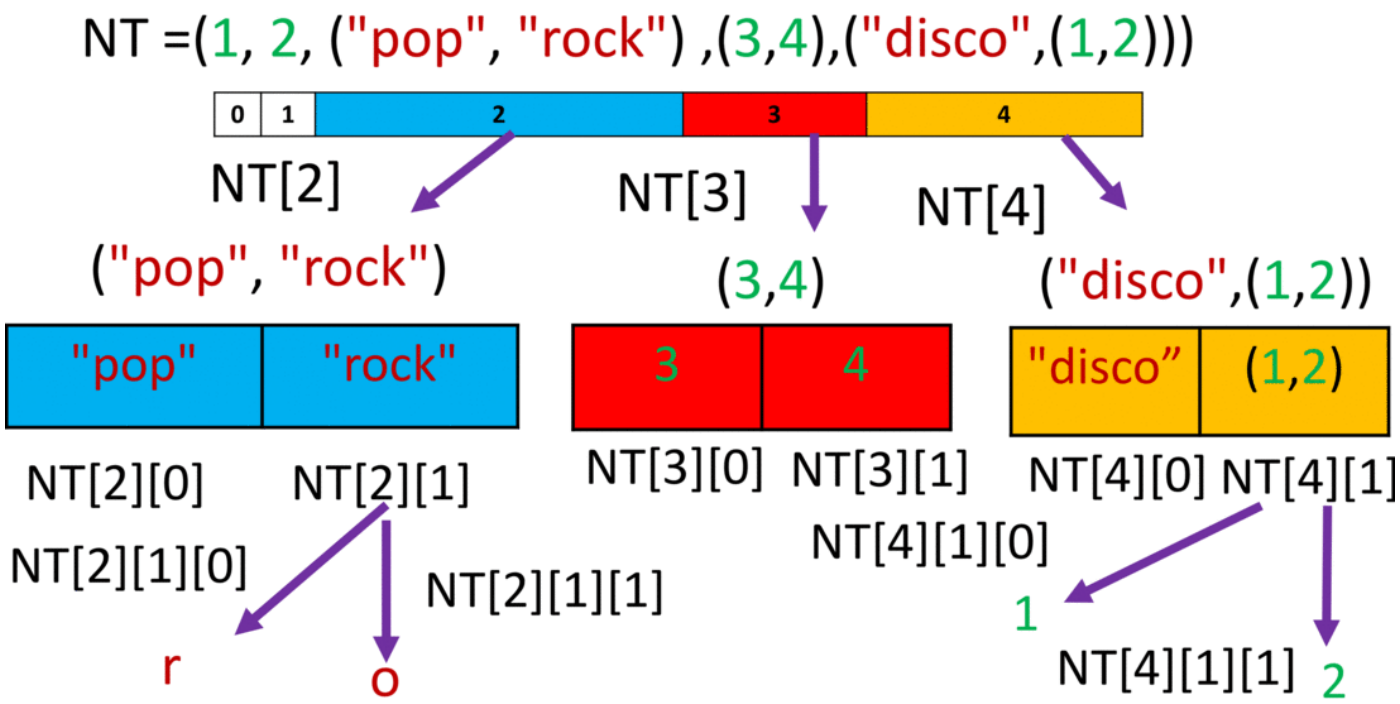
Similarly, we can access elements nested deeper in the tree with a fourth index:

In [ ]:
```
# Print the first element in the second nested tuples

NestedT[4][1][0]
```

In [ ]:
```
# Print the second element in the second nested tuples

NestedT[4][1][1]
```

In [ ]:
```
# Print the second element in the first nested tuples of the fourth index
NestedT [4][0][3]
```

The following figure shows the relationship of the tree and the element `NestedT[4][1][1]`:



While we can retrieve data through indexing (because a `tuple` is ordered), we cannot modify it (because a `tuple` is immutable).

In [ ]:
```
tuple3 = ('Dylan', 26, 167.6, True)
print(example_tuple)
```

While we can retrieve data through indexing (because a `tuple` is ordered), we cannot modify it (because a `tuple` is immutable).

In [ ]:
```
print(tuple3[0])
print(tuple3[1])
print(tuple3[2])
```

In [ ]:
```
%%expect_exception TypeError

tuple3[2] = 169.3
```

Deleting Tuples

In [ ]:
```
%%expect_exception TypeError

# deletion also fails
del example_tuple[-1]
```

For clarity enclose tuples with `()`, Python will assume we want a `tuple` if we don't use any symbols to enclose comma separated values as seen below.

In [ ]:
```
tuple4 = 'Jill', 36, 162.3, True
print(tuple4)
print(type(tuple4))
```

This implicit `tuple` comes up most often when working with functions that return multiple outputs. For example, we might have a function that returns the first and last letter of a string.

In [ ]:
```
def first_last(s):
    return s[0], s[-1]

chars = first_last('hello!')
print(chars)
```

Tuple Unpacking

In such cases, we'll sometimes want to store the multiple outputs in separate variables.

In [ ]:
```
first_char, last_char = first_last('hello!')

print(first_char)
print(last_char)
```

This syntax above is called *unpacking*. We can use it with any `tuple`, whether it was returned by a function or not.

In [ ]:
```
def first_last(s):
    return s[0], s[-1]

first_char, last_char = first_last('computerised!')

print(first_char)
print(last_char)
```

In [ ]:
```
tuple5 = ('Dylan', 26, 167.6, True)
print(tuple5)

name, age, height, has_dog = tuple5

print(name)
print(age)
print(height)
print(has_dog)
```

Both the Python `list` and `tuple` are ordered and heterogeneous. However, unlike the `list`, the `tuple` is immutable, meaning it cannot be modified after it is created. Therefore, a `list` might be better for representing data that is expected to change over the course of a program, like a to-do list. A `tuple` might be better for representing data that is expected to be fixed, like the responses of an individual subject to a survey.

Gotcha

One common mistake people make with immutability and especially with tuples is to assume data structures inside the tuple are immutable because the tuple is immutable. Lets see an example.

Even though the tuple itself is immutable, we cannot change the exact objects which it contains, those objects themselves can be mutated if they are mutable. As with anywhere mutability shows up, this requires the programmer to be careful and not assume data has not been modified in some context.

In [ ]:
```
tuple6 = tuple([[], 'a'])
print(tuple6)
tuple6[0].append(1)
print(tuple6)
```

In [ ]:
```
tuple7 = tuple([[], 'a'])
print(tuple7)
tuple7[0].append(90)
print(tuple7)
```

In [ ]:
```
example_set = {'Dylan', 26, 167.6, True}
print(example_set)
```

Even though we entered the data in one order, the `set` printed out in a different order. Even more significantly, we cannot index or slice a `set`.

In [ ]:
```
%%expect_exception TypeError

print(example_set[0])
```

Lists

In [ ]:
```

```

Lists in Python

Welcome! This notebook will teach you about the lists in the Python Programming Language. By the end of this lab, you'll know the basics list operations in Python, including indexing, other list operations and copy/clone list.

About the Dataset

Imagine you received album recommendations from your friends and compiled all of the recommandations into a table, with specific information about each album.

The table has one row for each movie and several columns:

- **artist** - Name of the artist
- **album** - Name of the album
- **released_year** - Year the album was released
- **length_min_sec** - Length of the album (hours,minutes,seconds)
- **genre** - Genre of the album
- **music_recording_sales_millions** - Music recording sales (millions in USD) on SONG://DATABASE
- **claimed_sales_millions** - Album's claimed sales (millions in USD) on SONG://DATABASE
- **date_released** - Date on which the album was released
- **soundtrack** - Indicates if the album is the movie soundtrack (Y) or (N)
- **rating_of_friends** - Indicates the rating from your friends from 1 to 10

The dataset can be seen below:

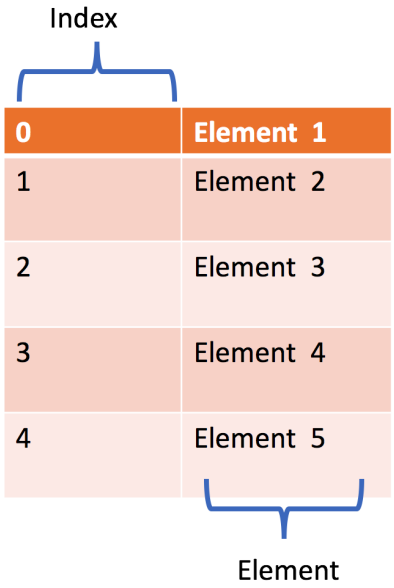
<table font-size:xx-small> Artist Album Released Length Genre Music recording sales (millions) Claimed sales (millions) Released Soundtrack Rating (friends) Michael Jackson Thriller 1982 00:42:19 Pop, rock, R&B 46 65 30-Nov-82 10.0 AC/DC Back in Black 1980 00:42:11 Hard rock 26.1 50 25-Jul-80 8.5 Pink Floyd The Dark Side of the Moon 1973 00:42:49 Progressive rock 24.2 45 01-Mar-73 9.5 Whitney Houston The Bodyguard 1992 00:57:44 Soundtrack/R&B, soul, pop 26.1 50 25-Jul-80 Y 7.0 Meat Loaf Bat Out of Hell 1977 00:46:33 Hard rock, progressive rock 20.6 43 21-Oct-77 7.0 Eagles Their Greatest Hits (1971-1975) 1976 00:43:08 Rock, soft rock, folk rock 32.2 42 17-Feb-76 9.5 Bee Gees Saturday Night Fever 1977 1:15:54 Disco 20.6 40 15-Nov-77 Y 9.0 Fleetwood Mac Rumours 1977 00:40:01 Soft rock 27.9 40 04-Feb-77 9.5 </table>

Below:

Artist	Album	Relased	Length	Genre	Music recording sales (millions)	Claimed sales (millions)	Released	Soundtrack	Rating (friends)
Michael Jackson	Thriller	1982	00:42:19	Pop, rock, R&B	46	65	30-Nov-82	b	10.0
Michael Jackson	Thriller	1982	00:42:19	Pop, rock, R&B	46	65	30-Nov-82	b	9.0
Michael Jackson	Thriller	1982	00:42:19	Pop, rock, R&B	46	65	30-Nov-82	b	10.0
Michael Jackson	Thriller	1982	00:42:19	Pop, rock, R&B	46	65	30-Nov-82	b	10.0

Indexing

We are going to take a look at lists in Python. A list is a sequenced collection of different objects such as integers, strings, and other lists as well. The address of each element within a list is called an **index**. An index is used to access and refer to items within a list.



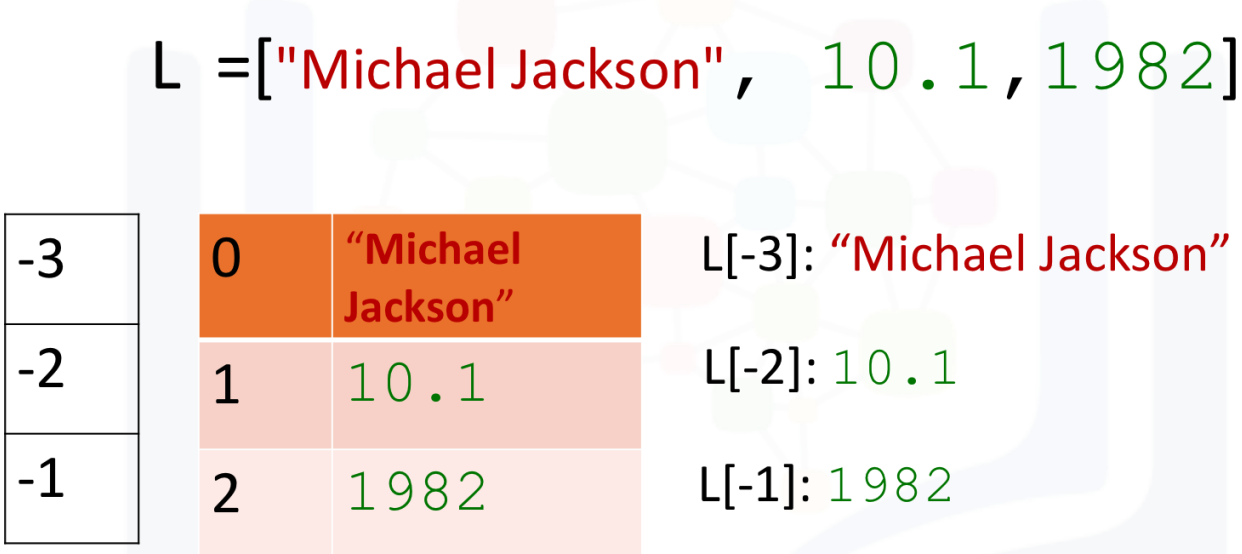
[Element 1 , Element 2 , Element 3 , Element 4, Element 5]

Index 0 1 2 3 4

To create a list, type the list within square brackets **[ ]**, with your content inside the parenthesis and separated by commas. Let's try it!

In [ ]:
```
# Create a list

L = ["Michael Jackson", 10.1, 1982, True]
L
```

We can use negative and regular indexing with a list :



In [ ]:
```
# Print the elements on each index

print('the same element using negative and positive indexing:\n Postive:',L[0],
'\n Negative:' , L[-3] )
print('the same element using negative and positive indexing:\n Postive:',L[1],
'\n Negative:' , L[-2]  )
```

```
print('the same element using negative and positive indexing:\n Postive:',L[2],
'\n Negative:' , L[-1]  )
```

List Content

Lists can contain strings, floats, and integers. We can nest other lists, and we can also nest tuples and other data structures. The same indexing conventions apply for nesting:

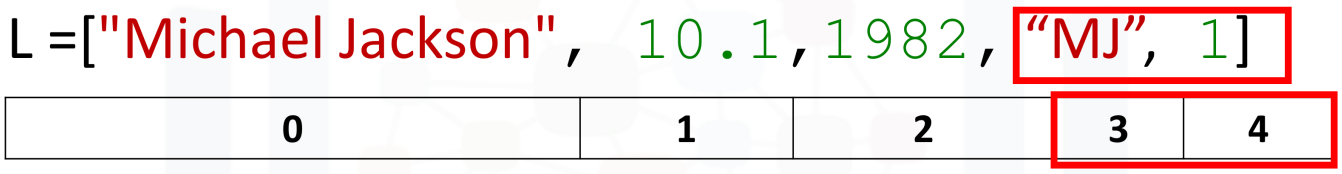
In [ ]:
```
# Sample List
["Michael Jackson", 10.1, 1982, [1, 2], ("A", 1)]
```

Python `list` are: 'ordered', 'heterogeneous', and 'mutable'. Because it is heterogeneous and mutable, the `list` is very flexible. We need to be careful about the changes we make to a `list`, because they can be very unpredictable. We could break our code or lose data!

List Operations: Slicing

We can also perform slicing in lists. For example, if we want the last two elements, we use the following command:

In [ ]:
```
# Sample List
List = ["Michael Jackson", 10.1, 1982, "MJ" ,1]
List
```



In [ ]:
```
# List slicing
List[3:5]
```

Extend Method

We can use the method `extend` to add new elements to the list:

In [ ]:
```
# Use extend to add elements to list
List = [ "Michael Jackson", 10.2]
List.extend(['pop', 10])
List
```

In [ ]:
```
List = ["Michael Jackson", 10.2]
List.extend([90])
List
```

Another similar method is `extendend`. If we apply `append` instead of `extend`, we add one element to the list:

In [ ]:
```
# Use append to add elements to list
List = [ "Michael Jackson", 10.2]
List.append(['pop', 10])
List
```

Each time we apply a method, the list changes. If we apply `extend` we add two new elements to the list. The list `L` is then modified by adding two new elements:

In [ ]:
```
# Use extend to add elements to list
List = [ "Michael Jackson", 10.2]
List.extend(['pop', 10])
List
```

If we append the list `['a','b']` we have one new element consisting of a nested list:

In [ ]:
```
# Use append to add elements to list
List.append(['a','b'])
List
```

As lists are mutable, we can change them. For example, we can change the first element as follows:

In [ ]:
```
# Change the element based on the index
List2 = ["disco", 10, 1.2]
print('Before change:', A)
List2[0] = 'hard rock'
print('After change:', A)
```

Deleting Elements in List

We can also delete an element of a list using the `del` command:

In [ ]:
```
# Delete the element based on the index

print('Before change:', List2)
del(List2[0])
print('After change:', List2)
```

We can convert a string to a list using `split`. For example, the method `split` translates every group of characters separated by a space into an element in a list:

In [ ]:
```
# Split the string, default is by space

'hard rock'.split()
```

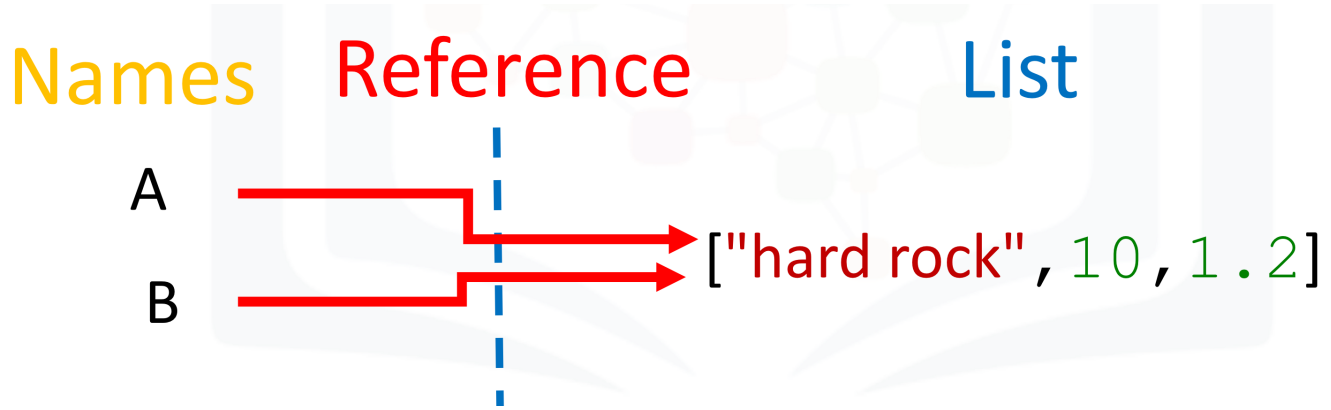
We can use the split function to separate strings on a specific character. We pass the character we would like to split on into the argument, which in this case is a comma. The result is a list, and each element corresponds to a set of characters that have been separated by a comma:

In [ ]:
```
# Split the string by comma

'A,B,C,D'.split(",")
```

Copy and Clone List

When we set one variable **List4** equal to **List3**; both **List3** and **List4** are referencing the same list in memory:

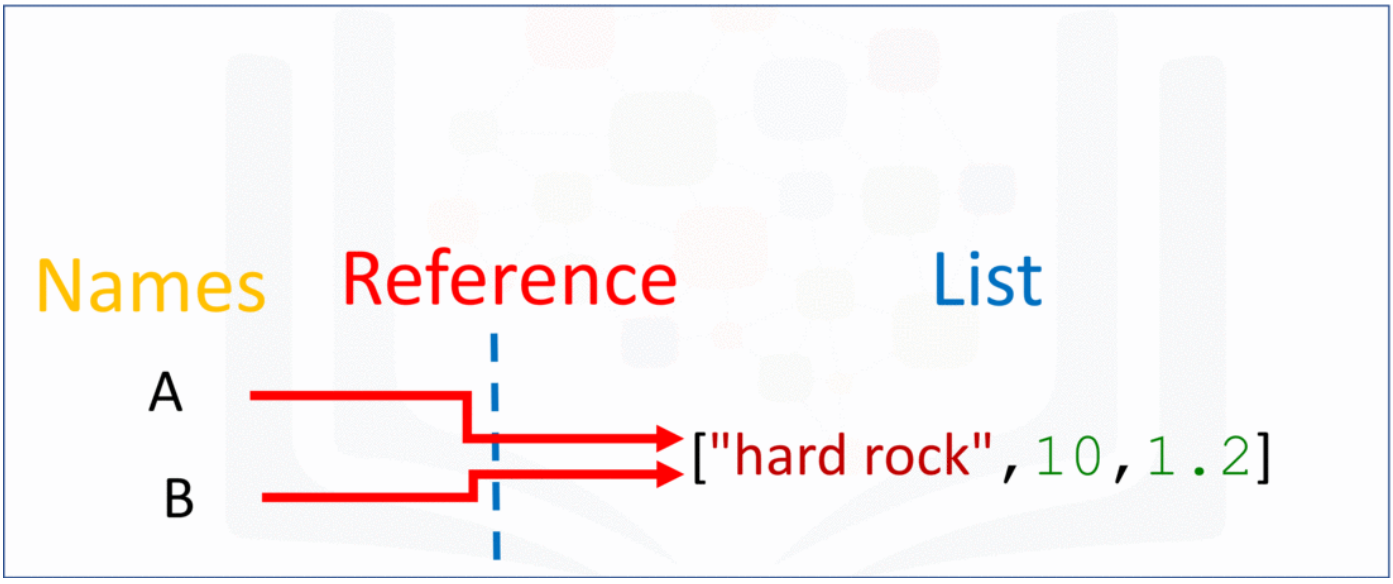
In [ ]:
```
# Copy (copy by reference) the list A
List3= ["hard rock", 10, 1.2]
List4 = List3
print('List3:', List3)
print('List4:', List4)
```



Initially, the value of the first element in **List4** is set as hard rock. If we change the first element in **List3** to **banana**, we get an unexpected side effect. As **List3** and **List4** are referencing the same list, if we change list **List3**, then list **List4** also changes. If we check the first element of **List4** we get banana instead of hard rock:

In [ ]:
```
# Examine the copy by reference
print('List3[0]:', List3[0])
List3[0] = "banana"
print('List4[0]:', List4[0])
```

This is demonstrated in the following figure:



You can clone list **List3** by using the following syntax:

In [ ]:
```
# Clone (clone by value) the list A

List4 = List3[:]
List4
```

In [ ]:
```
List4[0] = "cherry"
List4
```

In [ ]:
```
List3
```

Variable **B** references a new copy or clone of the original list; this is demonstrated in the following figure:

Now if you change **A**, **B** will not change:

In [ ]:
```
print('List4[0]:', List4[0])
List3[0] = "hard rock"
print('List4[0]:', List4[0])
```

In [ ]:
```

```

We see we successfully dealt with both a shorter list so far.

Collections (or **containers** as they're known in Python) can be very useful for tackling complex data problems. Python provides several types of containers, which we will explore. Each one has different properties and structure that make them useful for specific tasks. Later in the course, we'll also introduce powerful, highly-structured containers that Python users have invented and shared with the Python community.

In the context of data science, we'll often call a collection of data that logically belongs together a **data set**, and the type of variable we use to store it in Python a **data structure**. These terms are meant to emphasize the relationships between the individual pieces of information that creates their meaning as a whole.

Complex List

In [ ]:
```
grocery_a = 'chicken'
grocery_b = 'onions'
grocery_c = 'rice'
grocery_d = 'peppers'
grocery_e = 'bananas'

grocery_list = ['chicken', 'onions', 'rice', 'peppers', 'bananas']
print(grocery_list)

grocery_list = [grocery_a, grocery_b, grocery_c, grocery_d, grocery_e]
print(grocery_list)
```

So far we've worked with `list`s of strings (and `range`), but we're not limited to only that data type.

In [ ]:
```
int_list = [2, 6, 3049, 18, 37]
float_list = [3.7, 8.2, 178.245, 63.1]
mixed_list = [26, False, 'some words', 1.264]

print(int_list)
print(float_list)
print(mixed_list)
```

We can store any `type` of data in a `list`. We can even put a `list` inside of a `list`.

In [ ]:
```
list_of_lists = [['a', 'list', 'of', 'words'], [1, 5, 209], [True, True, False]]
print(list_of_lists)
```

There are very few restrictions on how we structure a list or what we put in it. This can lead to a very complicated nested structure.

In [ ]:
```
confusing_list = [[23, 73, 50], 'some words', 12.308, [[False, True], 'more words']]
print(confusing_list)
```

In [ ]:
```
confusing_list[0]
```

In [ ]:
```
confusing_list[0][2]
```

In [ ]:
```
confusing_list[3]
```

Again, the Python `list` is *heterogeneous* because it can hold a collection of mixed objects. This is one of the major defining properties of the Python `list`.

You may have also noticed that when we put data into a `list` in particular order, it stays in that order when we `print` or use the `list` in a `for` loop. Because `list` preserves order, we say it is *ordered*. We can use this property to retrieve particular items from a `list` based on their position (or **index**) in the list.

In [ ]:
```
#again let's use the list
grocery_a = 'chicken'
grocery_b = 'onions'
grocery_c = 'rice'
grocery_d = 'peppers'
grocery_e = 'bananas'

grocery_list = ['chicken', 'onions', 'rice', 'peppers', 'bananas']
print(grocery_list)

grocery_list = [grocery_a, grocery_b, grocery_c, grocery_d, grocery_e]
print(grocery_list)
```

Printing `grocery_list[2]` returned the third item in the list: 'rice'. Why did it return the third item if we asked for the item at index 2? Python `list`s are *zero-indexed*.

In [ ]:
```
print(grocery_list[0])
print(grocery_list[1])
print(grocery_list[2])
```

'again' only the last line will print. We can also retrieve a *slice* of items from a list.

In [ ]:
```
grocery_list[0]
grocery_list[1]
grocery_list[2]
```

In [ ]:
```
# hint of the list is given as ; ['chicken', 'onions', 'rice', 'peppers', 'bananas']
print(grocery_list[1:4]) #prints the 2th to the element before the 5th element
print(grocery_list[3:]) #prints the 4th to the last element
print(grocery_list[:3]) #print from 1st to the element before the 4th element
```

Python also has a negative indexing syntax, allowing us to access the list from the end instead of the beginning. The last element is indexed by -1.

In [ ]:
```
print(grocery_list[-1])
print(grocery_list[-3:])
```

We can also slice a list using a step-size other than 1. For instance, we can slice every other item of the list, or even reverse the list by making negative steps.

In [ ]:
```
print(grocery_list[::2])
print(grocery_list[4:1:-1])
print(grocery_list[4:1:-2])
```

We can of course also retrieve information from a list by using a `for` loop.

In [ ]:
```
for item in grocery_list:
    print('we have ... %s' % item)
```

While we'll usuaally use the syntax `for item in list`, sometimes we will combine a `for` loop with indexing. The `range` function (which we used in the last notebook) is useful for this. For example, we can pick out every other item in the list.

In [ ]:
```
for i in range(0, len(grocery_list), 2):
    print(i, grocery_list[i])
```

The `range` function returns a sequence of integers between the first and second argument, using the third argument as the step size. Notice that the upper bound (i.e. second argument) is not included in the output.

In [ ]:
```
print(range(0, 10, 3)) # starts at 0 and counts by 3 and exclude now
print(range(104, 100, -1)) # starts counting backwward from 104 to 101
print(range(5)) # starts at 0 and counts by 1 by default
```

We can also use indexing/slicing to replace items in the list.

In [ ]:
```
grocery_list = ['chicken', 'onions', 'rice', 'peppers', 'bananas']
print(grocery_list)
grocery_list[-1] = 'oranges' # replace bananas with oranges
```

```
print(grocery_list)
grocery_list[1:3] = ['carrots', 'couscous'] #replace onions and rice with carrots and cou
print(grocery_list)
```

modifying list

Since we can modify lists after they are created, we call them *mutable* (the modifications are called *mutations*). Some Python data types are *immutable*, meaning once they are created they cannot be changed. We'll explore this further as we introduce more data types.

Another way we can mutate a `list` is to `append` new items.

In [ ]:
```
grocery_list = ['chicken', 'onions', 'rice', 'peppers', 'bananas']
print(grocery_list)
grocery_list.append('squash')
print(grocery_list)
grocery_list.append(['bread', 'salt'])
print(grocery_list) # what happened?
```

Since lists can contain lists, we have to be careful about adding multiple items to our list. Instead of `append`, we might want to use `extend`.

In [ ]:
```
grocery_list = ['chicken', 'onions', 'rice', 'peppers', 'bananas', 'squash']
print(grocery_list)
grocery_list.extend(['bread', 'salt'])
print(grocery_list)
```

Sorting List

Another mutation we can make to a list is to sort it.

In [ ]:
```
grocery_list.sort()
print(grocery_list)
```

Sets

A Python `set` is also similar to a `list`, except it is unordered. It can store heterogeneous data and it is mutable, but what does it mean to be unordered? The simplest explanation is simply to look at an example. We can create a set by enclosing our data with curly brackets `{}`.

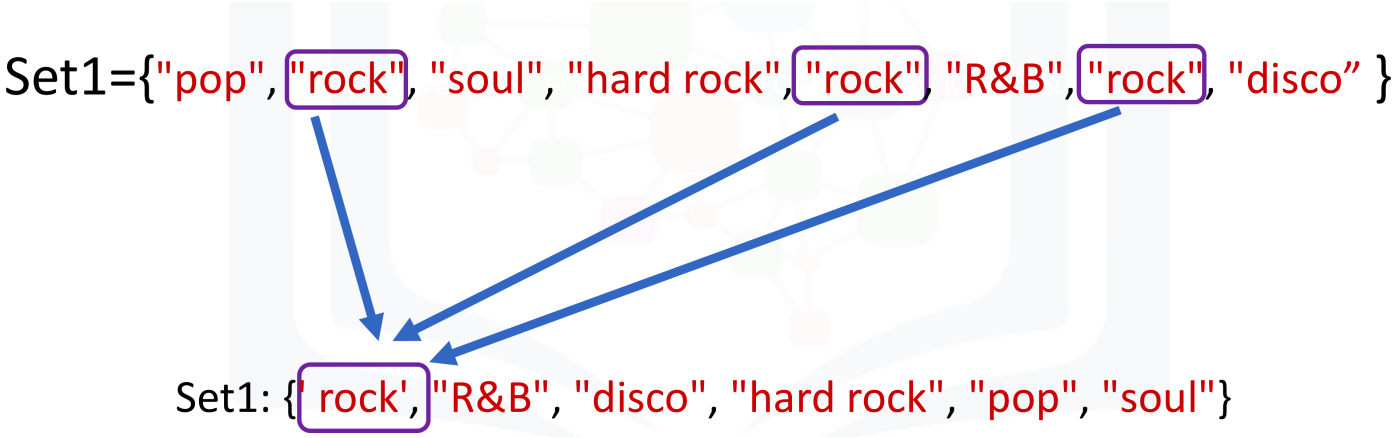
Set Content

A set is a unique collection of objects in Python. You can denote a set with a curly bracket **{}**. Python will automatically remove duplicate items: A Python `set` is also similar to a `list`, except it is unordered. It can store heterogeneous data and it is mutable, but what does it mean to be unordered? The simplest explanation is simply to look at an example. We can create a set by enclosing our data with curly brackets `{}`.

In [ ]:
```
# Create a set
```

```
set1 = {"pop", "rock", "soul", "hard rock", "rock", "R&B", "rock", "disco"}
set1
```

The process of mapping is illustrated in the figure:



You can also create a set from a list as follows:

In [ ]:
```
# Convert list to set

album_list = [ "Michael Jackson", "Thriller", 1982, "00:42:19", \
               "Pop, Rock, R&B", 46.0, 65, "30-Nov-82", None, 10.0]
album_set = set(album_list)
album_set
```

Now let us create a set of genres:

In [ ]:
```
# Convert list to set

music_genres = set(["pop", "pop", "rock", "folk rock", "hard rock", "soul", \
                    "progressive rock", "soft rock", "R&B", "disco"])
music_genres
```

Set Operations

Let us go over set operations, as these can be used to change the set. Consider the set **A**:

In [ ]:
```
# Sample set

A = set(["Thriller", "Back in Black", "AC/DC"])
A
```

We can add an element to a set using the `add()` method:

In [ ]:
```
# Add element to set

A.add("NSYNC")
A
```

If we add the same element twice, nothing will happen as there can be no duplicates in a set:

In [ ]:
```
# Try to add duplicate element to the set

A.add("NSYNC")
A
```

We can remove an item from a set using the `remove` method:

In [ ]:
```
# Remove the element from set

A.remove("NSYNC")
A
```

We can verify if an element is in the set using the `in` command:

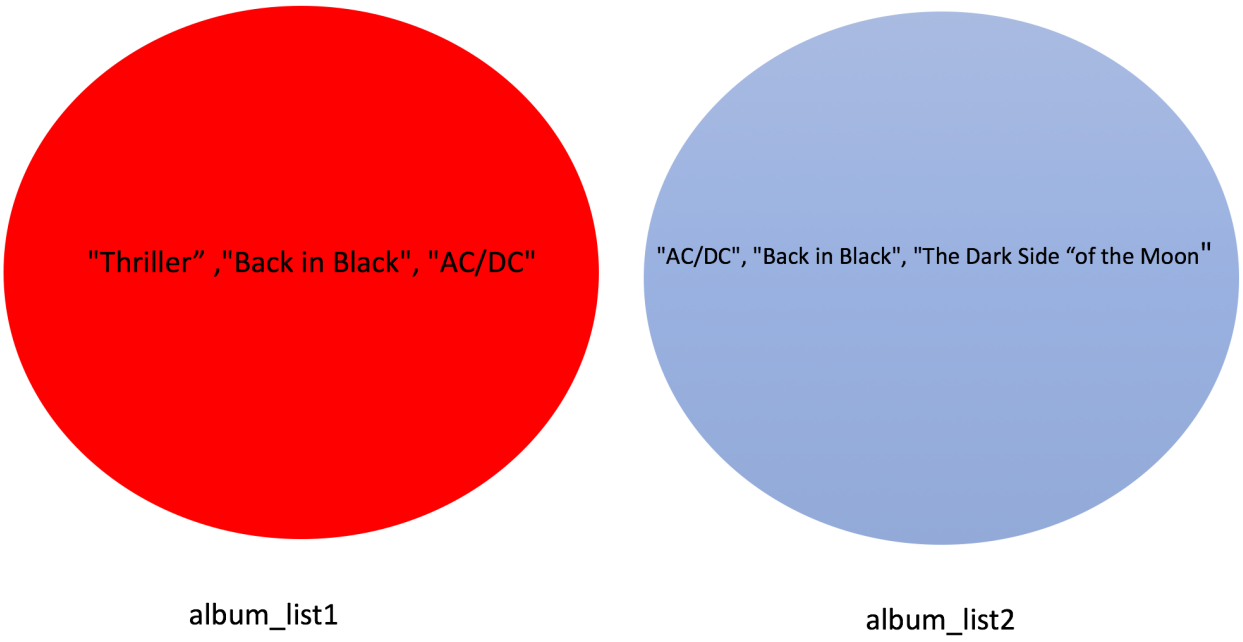
In [ ]:
```
# Verify if the element is in the set

"AC/DC" in A
```

Sets Logic Operations

Remember that with sets you can check the difference between sets, as well as the symmetric difference, intersection, and union:

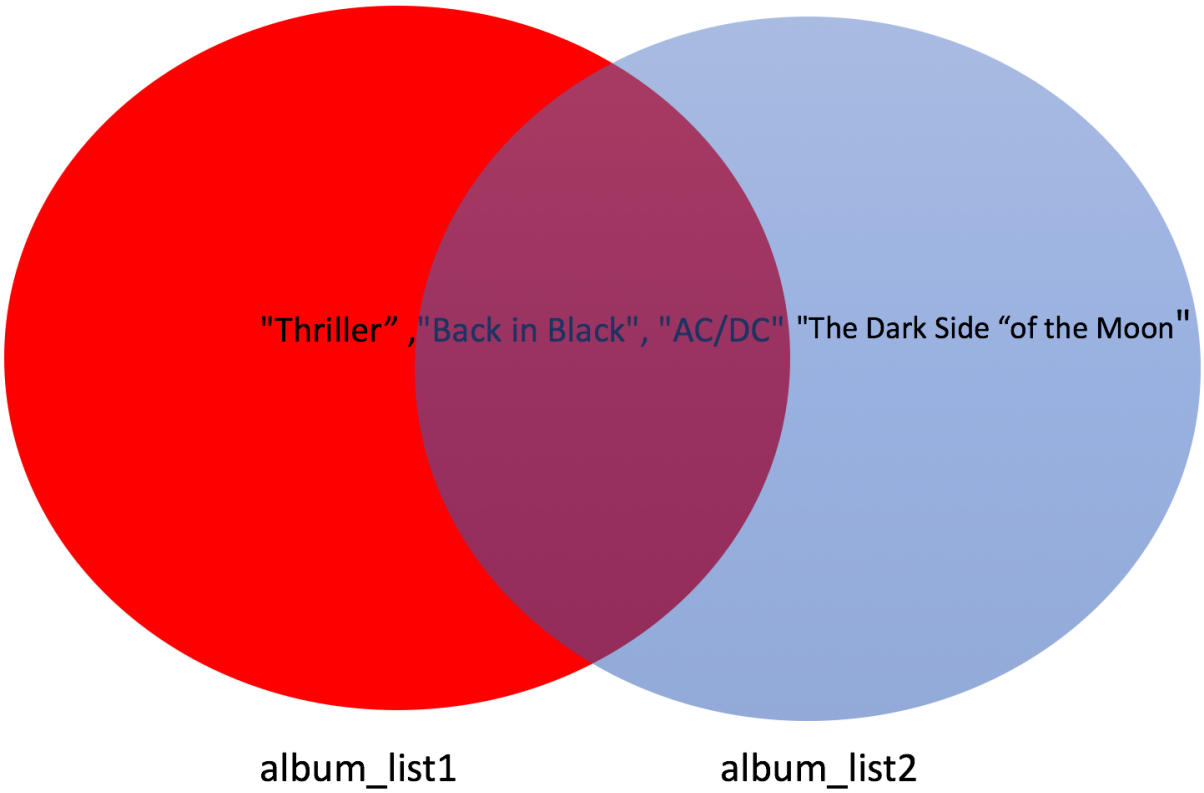
Consider the following two sets:

In [ ]:
```
# Sample Sets

album_set1 = set(["Thriller", 'AC/DC', 'Back in Black'])
album_set2 = set([ "AC/DC", "Back in Black", "The Dark Side of the Moon"])
```



In [ ]:
```
# Print two sets

album_set1, album_set2
```

As both sets contain **AC/DC** and **Back in Black** we represent these common elements with the intersection of two circles.



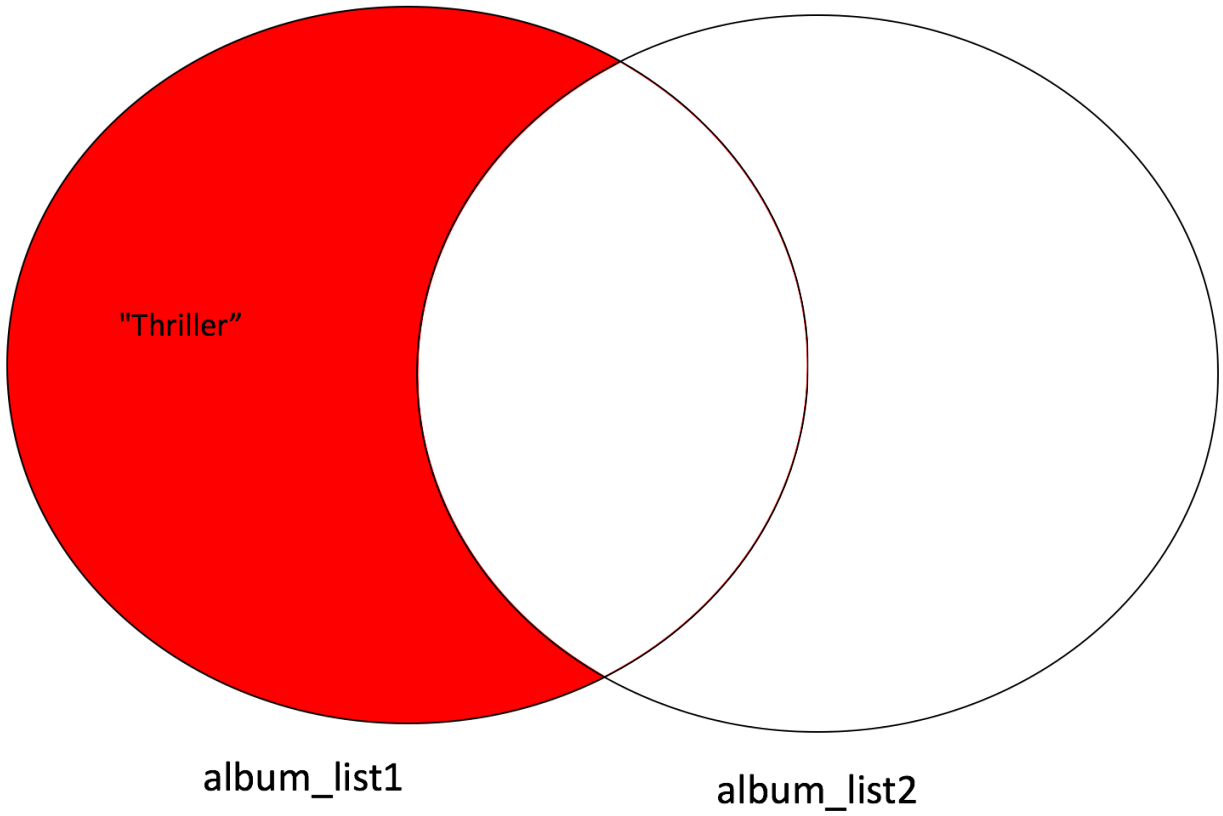
You can find the intersect of two sets as follow using `&`:

In [ ]:
```
# Find the intersections

intersection = album_set1 & album_set2
intersection
```

You can find all the elements that are only contained in `album_set1` using the `difference` method:

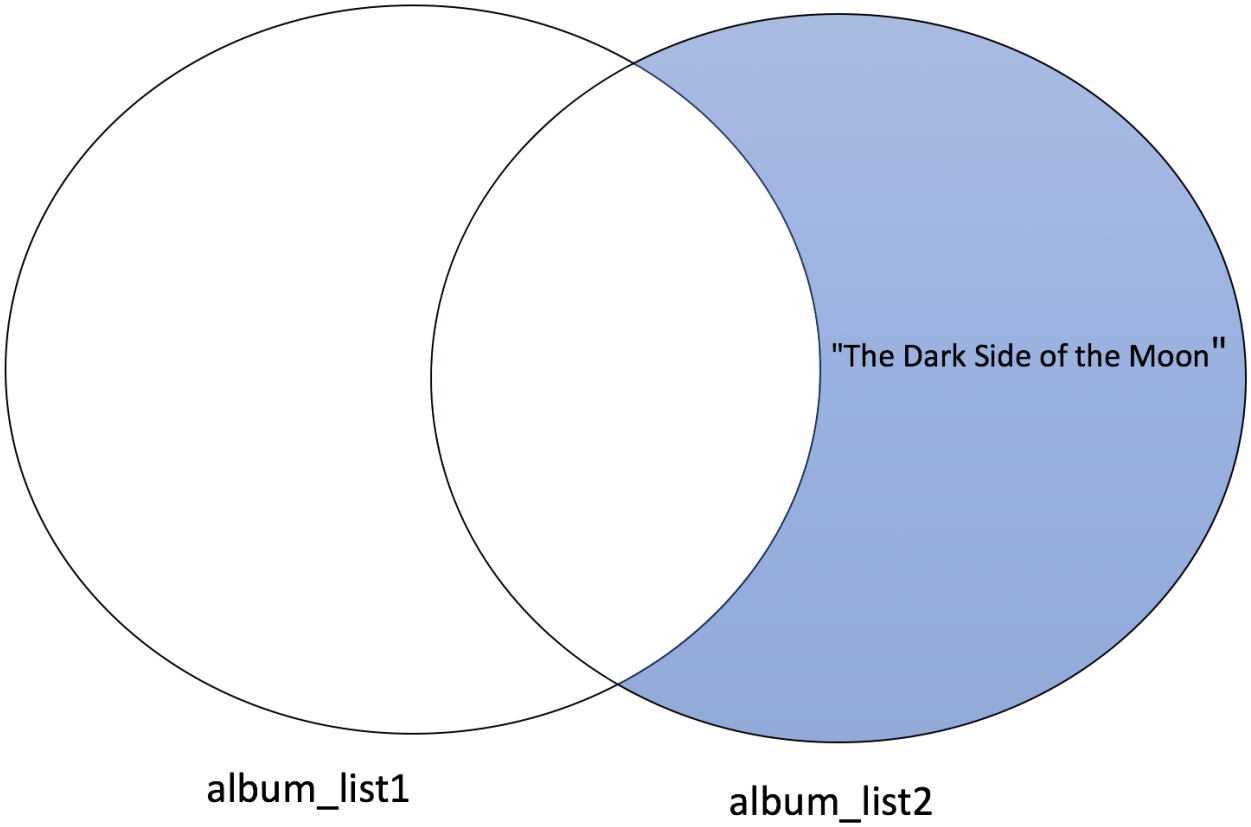
In [ ]:
```
# Find the difference in set1 but not set2

album_set1.difference(album_set2)
```

You only need to consider elements in `album_set1`; all the elements in `album_set2`, including the intersection, are not included.



The elements in `album_set2` but not in `album_set1` is given by:

In [ ]:
```
album_set2.difference(album_set1)
```

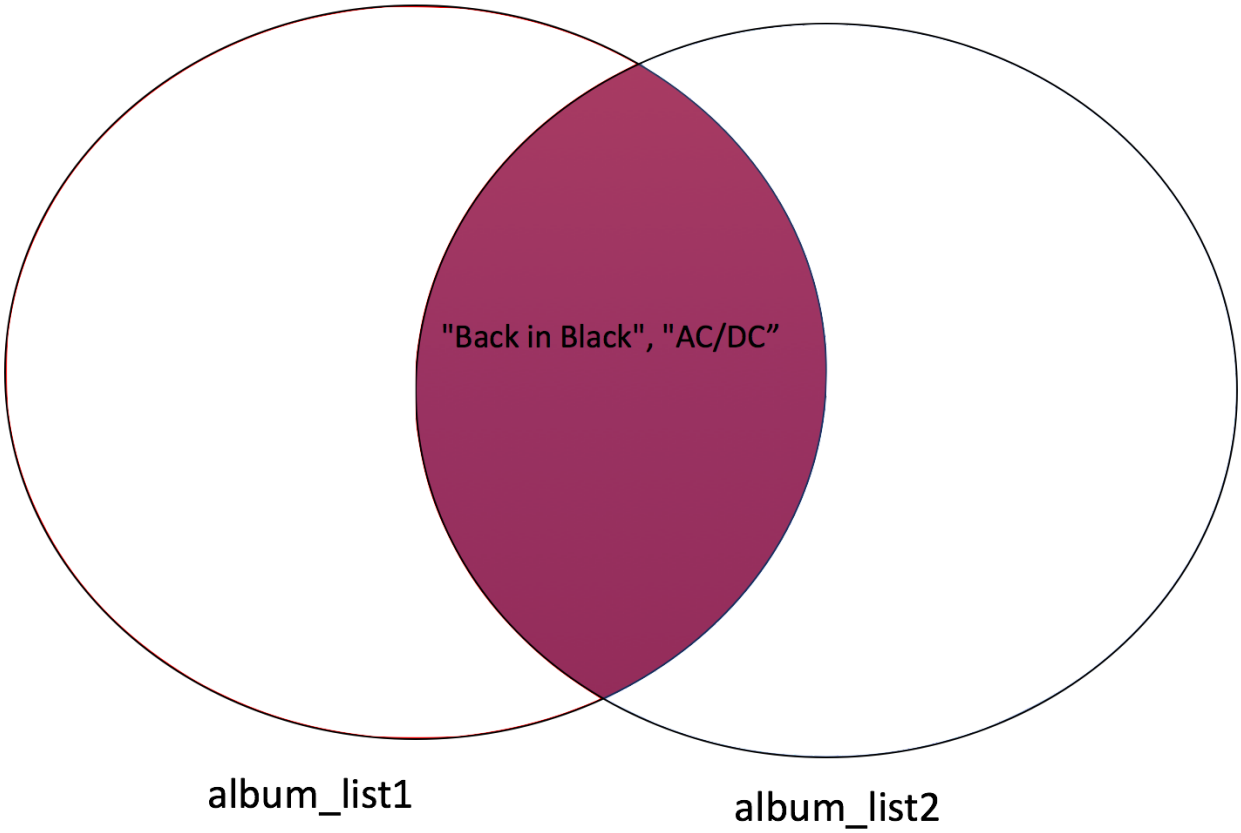


You can also find the intersection of `album_list1` and `album_list2`, using the `intersection` method:

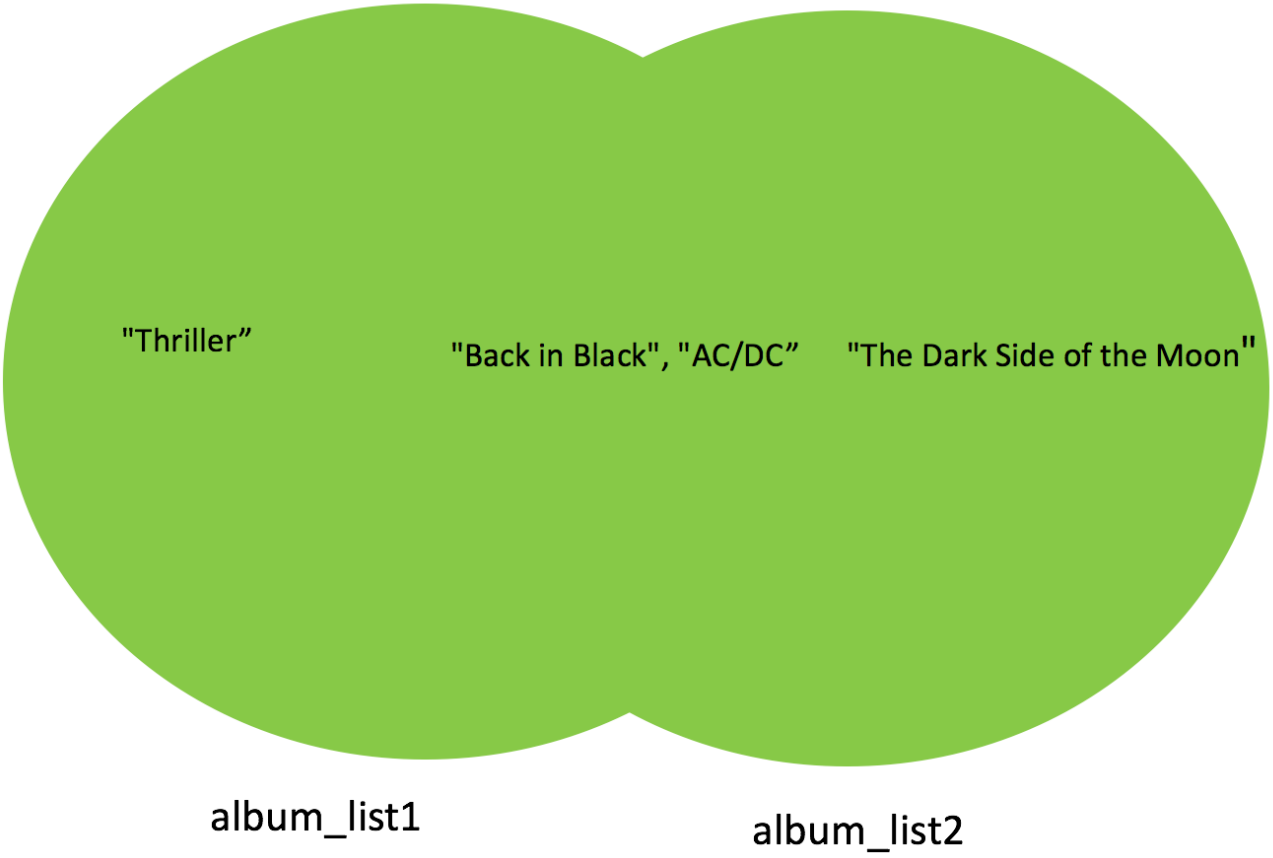
In [ ]:

```
# Use intersection method to find the intersection of album_list1 and album_list2

album_set1.intersection(album_set2)
```

This corresponds to the intersection of the two circles:



The union corresponds to all the elements in both sets, which is represented by coloring both circles:



The union is given by:

In [ ]:
```
# Find the union of two sets

album_set1.union(album_set2)
```

And you can check if a set is a superset or subset of another set, respectively, like this:

In [ ]:
```
# Check if superset

set(album_set1).issuperset(album_set2)
```

In [ ]:
```
# Check if subset

set(album_set2).issubset(album_set1)
```

Here is an example where `issubset()` and `issuperset()` return true:

In [ ]:
```
# Check if subset
set({"Back in Black", "AC/DC"}).issubset(album_set1)
```

In [ ]:
```
# Check if superset
album_set1.issuperset({"Back in Black", "AC/DC"})
```

Consider the list `A = [1, 2, 2, 1]` and set `B = set([1, 2, 2, 1])`, does `sum(A) = sum(B)`

In [ ]:
```
# Write your code below and press Shift+Enter to execute
A = [1, 2, 2, 1]
B = set([1, 2, 2, 1])
```

```
print("the sum of A is:", sum(A))
print("the sum of B is:", sum(B))
```

Create a new set `album_set3` that is the union of `album_set1` and `album_set2`:

In [ ]:
```
# Write your code below and press Shift+Enter to execute

album_set1 = set(["Thriller", 'AC/DC', 'Back in Black'])
album_set2 = set([ "AC/DC", "Back in Black", "The Dark Side of the Moon"])
```

In [ ]:
```
album_set3 = album_set1.union(album_set2)
album_set3
```

Find out if `album_set1` is a subset of `album_set3`:

In [ ]:
```
# Write your code below and press Shift+Enter to execute
album_set1.issubset({"Back in Black", "AC/DC"})
```

More set methods

However, we can still add and delete items from a set.

In [ ]:
```
print(example_set)
print(example_set.pop())
print(example_set)

example_set.add('True')
print(example_set)
example_set.update([58.1, 'brown'])
print(example_set)
```

In [ ]:
```
# Adding an element
example_set.add(10)
example_set
```

In [ ]:
```
example_set.add(10)
example_set
```

The `add` method of a `set` works similarly to the `append` method of a `list`. The `update` method of a `set` works similarly to the `extend` method of a `list`.

Why is `set` useful?

It seems strange that we might want an *unordered* data structure. We can't access or modify the data through indexing. How does giving up order benefit us? The answer is that it gives us flexibility about how the data is stored in memory, and that flexibility can make data retrieval much faster.

Imagine we have ten boxes and ten piles of money. We put the ten piles of money in the ten boxes. Now say we want to find the box that has $5.37 in it. We don't know which box this is, so we start with the first box and check. If it isn't in the first box, we move on to the second box. We keep checking boxes until we find it. This might take awhile.

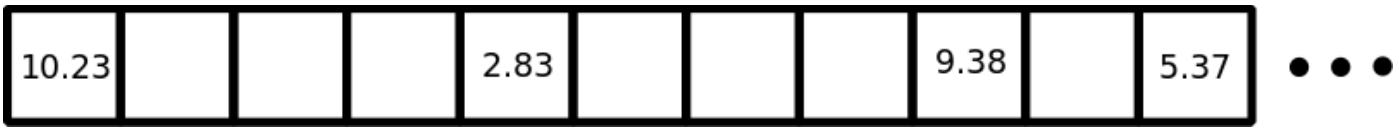
Now imagine we have the same ten piles of money, but we have 31 boxes. Instead of putting each pile of money into the boxes in order, instead put each pile into a box based on the amount of money in the pile. First we multiply the amount of money by 100, and then take modulus division by 31. This gives the number of the box we should put the pile of money in.

In [ ]:
```
piles = [2.83, 8.23, 9.38, 10.23, 25.58, 0.42, 5.37, 28.10, 32.14, 7.31]
```

In [ ]:
```
def hash_function(x):
    return int(x * 100 % 31)

[hash_function(pile) for pile in piles]
```

In [ ]:
```
def hash_function(x):
    return int(x * 100 % 31)

[hash_function(pile) for pile in piles]
```

In [ ]:
```
#Now say we want to find the box with \$5.37 in it. We don't have to search through box a
print(int(5.37 * 100 % 31))
```



Box number 10 contains the $5.37 pile.

This technique of assigning boxes (i.e. memory) based on the object it contains is called **hashing**. It makes searching for data very fast (as we've illustrated), but at the cost of increase memory allocation (we needed more boxes). It also means that we cannot assign an order to the objects as they are stored in memory.

Hashing also puts two major restrictions on the `set`. First of all, objects in a `set` must be immutable. If an object were to change, its position in memory would no longer correspond with its **hash**. Secondly, the objects in a `set` must be unique. Identical objects end up with the same hash. Since we can't store multiple objects in the same chunk of memory, we simply discard any duplicates.

This second restriction means we can use a `set` to easily determine the unique objects in a `list` or `tuple`.

suppose we have 50 boxes with with we want to put in 10 piles of money, we can use the code below

In [ ]:
```
def hash_function(x):
    return int(x * 100 % 50)

[hash_function(pile) for pile in piles]
```

In [ ]:
```
(print (int(7.3 *100 % 50)))
```

Unique function for set is hashing as displayed below:

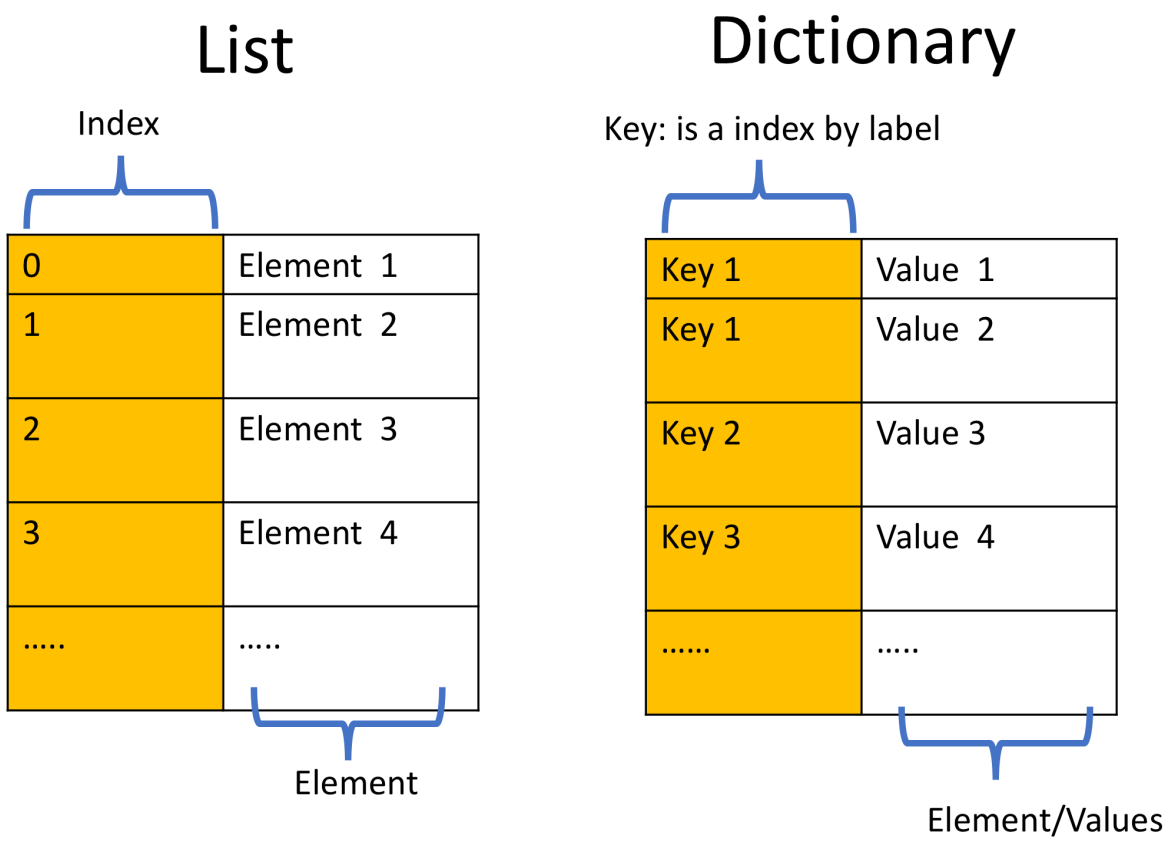
In [ ]:
```
new_list = 'my string'
hash(new_list)
```

In [ ]:
```
print(set([23, 609, 348, 10, 5, 23, 340, 82]))
print(set(('a', 'b', 'q', 'c', 'c', 'd', 'r', 'a')))
```

In [ ]:
```
student_a_courses = {'history', 'english', 'biology', 'theatre'}
student_b_courses = {'biology', 'english', 'mathematics', 'computer science'}

print(student_a_courses.intersection(student_b_courses))
print(student_a_courses.union(student_b_courses))
print(student_a_courses.difference(student_b_courses))
print(student_b_courses.difference(student_a_courses))
print(student_a_courses.symmetric_difference(student_b_courses))
```

Dictionaries

What are Dictionaries?



An example of a Dictionary `Dict`:

In [ ]:
```
# Create the dictionary
```

```
Dict = {"key1": 1, "key2": "2", "key3": [3, 3, 3], "key4": (4, 4, 4), ('key5'): 5, (0, 1)
Dict
```

The keys can be strings:

Keys can also be any immutable object such as a tuple:

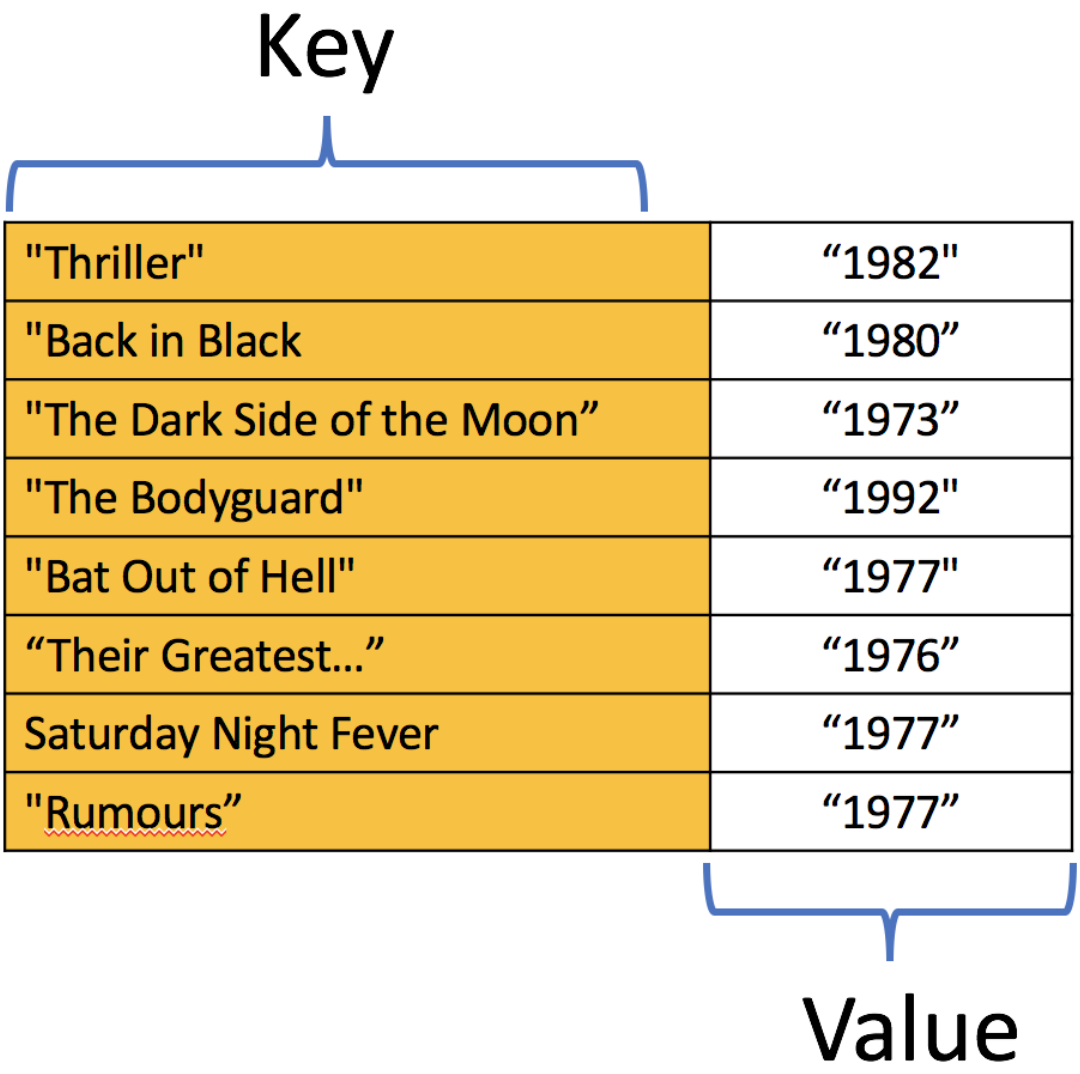
Each key is separated from its value by a colon "`:`". Commas separate the items, and the whole dictionary is enclosed in curly braces. An empty dictionary without any items is written with just two curly braces, like this "`{}`".

In [ ]:

```
# Create a sample dictionary

release_year_dict = {"Thriller": "1982", "Back in Black": "1980", \
                     "The Dark Side of the Moon": "1973", "The Bodyguard": "1992", \
                     "Bat Out of Hell": "1977", "Their Greatest Hits (1971-1975)": "1976",
                     "Saturday Night Fever": "1977", "Rumours": "1977"}
release_year_dict
```

In summary, like a list, a dictionary holds a sequence of elements. Each element is represented by a key and its corresponding value. Dictionaries are created with two curly braces containing keys and values separated by a colon. For every key, there can only be one single value, however, multiple keys can hold the same value. Keys can only be strings, numbers, or tuples, but values can be any data type.

It is helpful to visualize the dictionary as a table, as in the following image. The first column represents the keys, the second column represents the values.



Keys

You can retrieve the values based on the names:

In [ ]:
```
# Get value by keys

release_year_dict['Thriller']
```

This corresponds to:

"Thriller"	“1982"
"Back in Black	“1980”
"The Dark Side of the Moon”	“1973”
"The Bodyguard"	“1992"
"Bat Out of Hell"	“1977"
“Their Greatest…”	“1976”
“Saturday Night Fever”	“1977”
"Rumours”	“1977”

Similarly for **The Bodyguard**

In [ ]:
```
# Get value by key

release_year_dict['The Bodyguard']
```

"Thriller"	“1982"
"Back in Black	“1980”
"The Dark Side of the Moon”	“1973”
"The Bodyguard"	“1992"
"Bat Out of Hell"	“1977"
“Their Greatest…”	“1976”
“Saturday Night Fever”	“1977”
"Rumours”	“1977”

Now let you retrieve the keys of the dictionary using the method `release_year_dict()`:

In [ ]:
```
# Get all the keys in dictionary

release_year_dict.keys()
```

You can retrieve the values using the method `values()`:

In [ ]:
```
# Get all the values in dictionary

release_year_dict.values()
```

We can add an entry:

In [ ]:
```
# Append value with key into dictionary
```

```
release_year_dict['Graduation'] = '2007'
release_year_dict
```

We can delete an entry:

In [ ]:
```
# Delete entries by key

del(release_year_dict['Thriller'])
del(release_year_dict['Graduation'])
release_year_dict
```

We can verify if an element is in the dictionary:

In [ ]:
```
# Verify the key is in the dictionary

'The Bodyguard' in release_year_dict
```

We can o understand the Python `dict`, let's start again with the Python `list`.

In [ ]:
```
me = ['chisom', 25, 175.6, 60.1, 'black', 'grey', True]
```

This `list` describes me: my name, my age, my height (in centimeters), my weight (in kilograms), my hair color, my eye color, and whether or not I have a dog. We know we can access this information individually by index.

In [ ]:
```
print('My name is %s' % me[0])
print('I have %s hair' % me[4])
```

It would be easy to get mixed up about which data is which (for example, which `'brown'` is hair color and which is eye color?), or where I should find it (will age always be at index 1?).

A better solution would be a data structure where we could index using meaningful values. For example instead of using `me[0]` to recover `Dylan`, I could use `me['name']`. Instead of hair color being `me[4]`, it could be `me['hair']`. This feature is the central characteristic of the Python `dict`.

In [ ]:
```
me_dict = {'name': 'Chisom', 'age': 25, 'height': 170.1, 'weight': 60.1,
           'hair': 'black', 'eyes': 'black', 'has dog': True}

print('My name is %s' % me_dict['name'])
print('I have %s hair' % me_dict['hair'])
print('I will be %d years old next month' % me_dict['age'])
```

In [ ]:
```
me_dict.values()
```

In [ ]:
```
me_dict.keys()
```

Instead of calling `'name'` and `'hair'` the index, we call them **keys**. Each key is associated with a **value** in a **key-value pair**. We can see the key-value pairs in the `{}` syntax used for creating a `dict`. Each key-value pair is separated by a comma, and within a pair the key and value are separated by a colon `:`.

Exercises

1. When might a `dict` be more useful than a `list`?
2. Compare the flexibility of a `dict` which contains other `dict` object to that of a multi dimensional array.

Answers

1. A 'dict' is more useful when want to reference each elemebt in a list and hence for easy reference we use a 'dict' rather than a list.

2.

Dictionary	Multi-dimensional array
they are very effective in xyz	they are not effective in xyz

`zip`

The `zip` function can be very handy for creating a `dict`. Let's go back to the `list` we made before that contained all the values describing me. We'll make a second `list` containing all the keys we would want for putting these values in a dictionary

In [ ]:
```
value_list = me
key_list = ['name', 'age', 'height', 'weight', 'hair', 'eyes', 'has dog']

print(value_list)
print(key_list)
```

Currently we have two lists: one of values and one of keys. They have no relationship to each other within Python, but we can see that they belong together logically. How do we combine them in Python? By using the `zip` function.

In [ ]:
```
key_value_pairs = list(zip(key_list, value_list))
print(key_value_pairs)
```

We now have a list of tuples. We interpret the first element of each tuple as a key, and the second element as a value. We can turn this list of tuples directly into a `dict`.

In [ ]:
```
me_dict = dict(key_value_pairs)
print(me_dict)
```

You may have noticed that even though our list of tuples began with `('name', 'Dylan')`, when we printed `me_dict` it started with `'eyes': 'brown'`. If you guessed this means that a `dict` is unordered, you are correct! The keys are hashed to assign key-value pairs to memory. Therefore, keys must be immutable and unique, similar to the elements of a `set`. However, values don't have these restrictions.

Some ways we can't write a 'dict'

In [ ]:
```
%%expect_exception TypeError
```

```
# this doesn't work
invalid_dict = {[1, 5]: 'a', 5: 23}
```

The `dict` is also mutable. We can add new key-value pairs by simple assignment.

In [ ]:
```
print(me_dict)
me_dict['favorite book'] =  'The Little Prince'
print(me_dict)
```

We can also use `update`, similar to the way we used it for a `set`, except now with key-value pairs.

In [ ]:
```
print(me_dict)
me_dict.update({'favorite color': 'orange', 'siblings': 3, 'nieces/nephews': 0})
print(me_dict)
```

We can replace or delete key-value pairs from the `dict`.

In [ ]:
```
del me_dict['favorite book']
print(me_dict)
```

In [ ]:
```
print(me_dict.pop('siblings'))
print(me_dict)
```

Because the `dict` uses hashing, searching it is very fast (as with `set`). Sometimes dictionaries are referred to as **lookup tables** or **hash tables**. It is incredibly useful for referencing data through meaningful keys. While the data in a `dict` is unordered, it remains organized by the keys. We can retrieve a list of the keys and values directly, or as key-value pairs, using the appropriate methods of `dict`.

In [ ]:
```
print(me_dict.keys())
print(me_dict.values())
print(me_dict.items())
```

Switching data structures

Each of the containers we've introduced has different properties and characteristics. Sometimes we will want to change one data structure into another to take advantage of these differences. We've already seen some methods for transforming a `dict` into a `list` of `tuple`s or vice versa. We can easily transform between `list`, `tuple`, and `set`.m

In [ ]:
```
example_list = ['a', 'b', 23, 10, True, 'a', 10]
example_tuple = tuple(example_list)
example_set = set(example_tuple)
example_list = list(example_set)

print(example_tuple)
print(example_set)
print(example_list) # note we lost the duplicates because of set
```

Search

We discussed the idea of searching for data in our data structures when describing what makes `set` (and `dict`) so special. What does search look like in Python? We search for data using the keyword `in`.

In [ ]:
```
print(example_list)
print('a' in example_list)
print('c' in example_list)
```

When dealing with a `dict`, we can search keys, but not values.

In [ ]:
```
print(me_dict)
print('hair' in me_dict)
print('has cat' in me_dict)
print('brown' in me_dict)
```

Searching for keys is important in dictionaries so that we don't accidentally try to retrieve a key-value pair that doesn't exist.

In [ ]:
```
%%expect_exception KeyError

print(me_dict['has cat'])
```

In [ ]:
```
if 'has dog' in me_dict:
    print('Has dog: %s' % me_dict['has dog'])
else:
    print(None)

if 'has cat' in me_dict:
    print('Has cat: %s' % me_dict['has cat'])
else:
    print(None)
```

In [ ]:
```
# can use get method for same results

print('Has dog: %s' % me_dict.get('has dog'))
print('Has cat: %s' % me_dict.get('has cat'))
```

Sorting

Since a `tuple` is immutable, can we sort it? Or is that a mutation? What would it mean to sort a `set` or a `dict`, which has no order?

Out of the data structures we've studied so far, only `list` has a `sort` method. However, Python also has a `sorted` function, which will create a sorted `list` out of other data structures. By default `sorted` applied to a `dict` makes a `list` of sorted keys. We must use the `items` method if we want our output to be key-value pairs.

In [ ]:
```
print(sorted(map(str, example_tuple)))
print(sorted(map(str, example_set)))
print(sorted(me_dict.items()))
print(sorted(me_dict))
```

Iteration

As we've seen in some examples already, it will often be useful to iterate through a data structure, whether to execute some task based on the information contained or to transform or analyze a data set. We will most often use `for` loops to iterate over data structures. With a `list`, `tuple`, or `set` the elements of the container are returned one after another. With a `dict` things are a little more complicated: do we want to iterate over keys, values, or key-value pairs?

In [ ]:
```
# by default we iterate over keys of a dict
for f in me_dict:
    print(f)
```

In [ ]:
```
# to iterate over values...
for y in me_dict.values():
    print(y)
```

In [ ]:
```
# or to iterate over key-value pairs...
for k, v in me_dict.items():
    print('%s:%s' % (k, v))
```

Notice we used `tuple` unpacking in the `for` loop in the last example!

After Thoughts ??

What do you make of this rather simplistic structures. They are the backbone of learning python.

Assignment Link

Now we would try out some practical examples with what we have learnt so far ! Let us try out this notebook

About this Instructor:

ChisomLoius is very passionate about Data Analysis and Machine Learning and does lot of free lance teaching and learning. Holding a B.Eng. in Petroleum Engineering, my focused is leveraging the knowledge of Data Science and Machine Learning to help build solutions in Education and High Tech Security. I currently work as a Petrochemist.

#
M
In

Visit our website, or further enquire more informatio via our email.

notebook and its source code are released under the terms of the MIT License.