Week 2

#Data Science/4 - Python for Data Science, AI & Development#

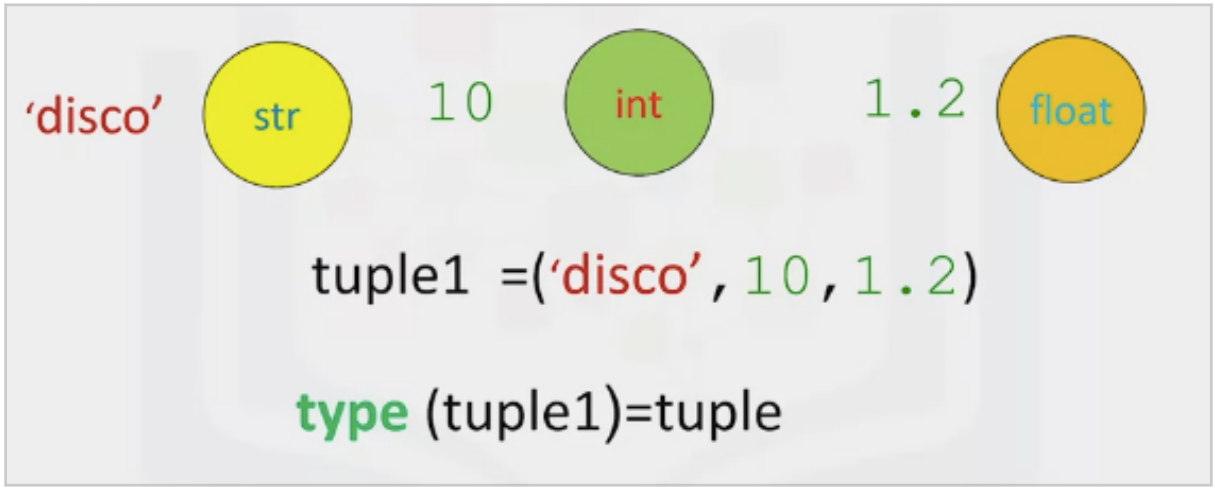
Lists and Tuples

In this video we will cover lists and tuples. These are called compound data types and are one of the key types of data structures in Python.

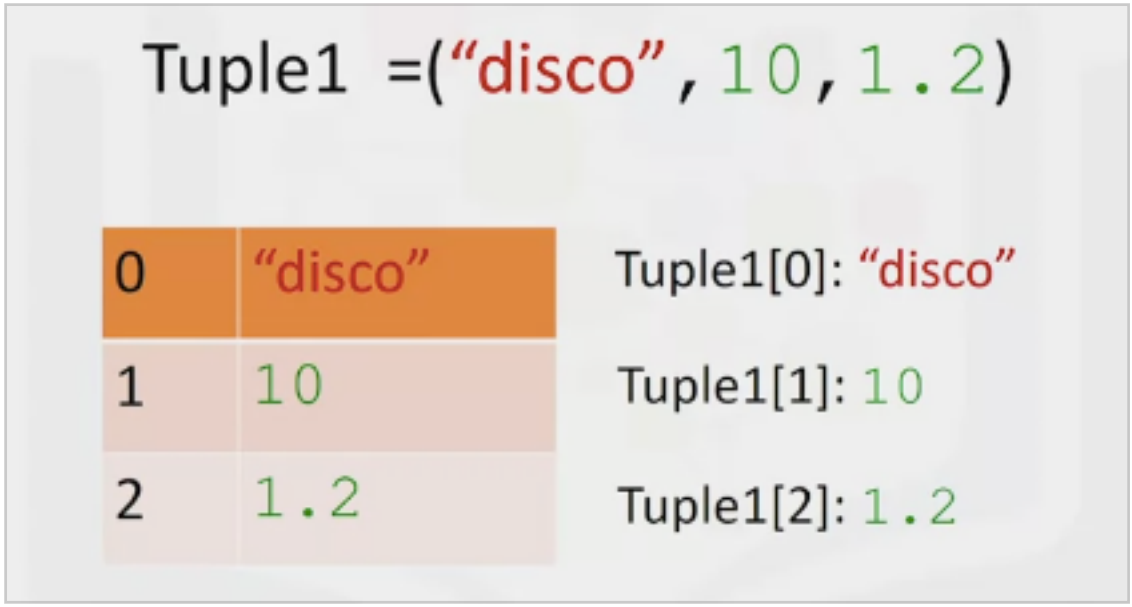
Tuples.

Tuples are an ordered sequence. Here is a tuple ratings. Tuples are expressed as comma separated elements within parentheses. These are values inside the parentheses.

In Python, there are different types: **strings**, **integer**, **float**. **They can all be contained in a tuple** but the type of the variable is tuple.

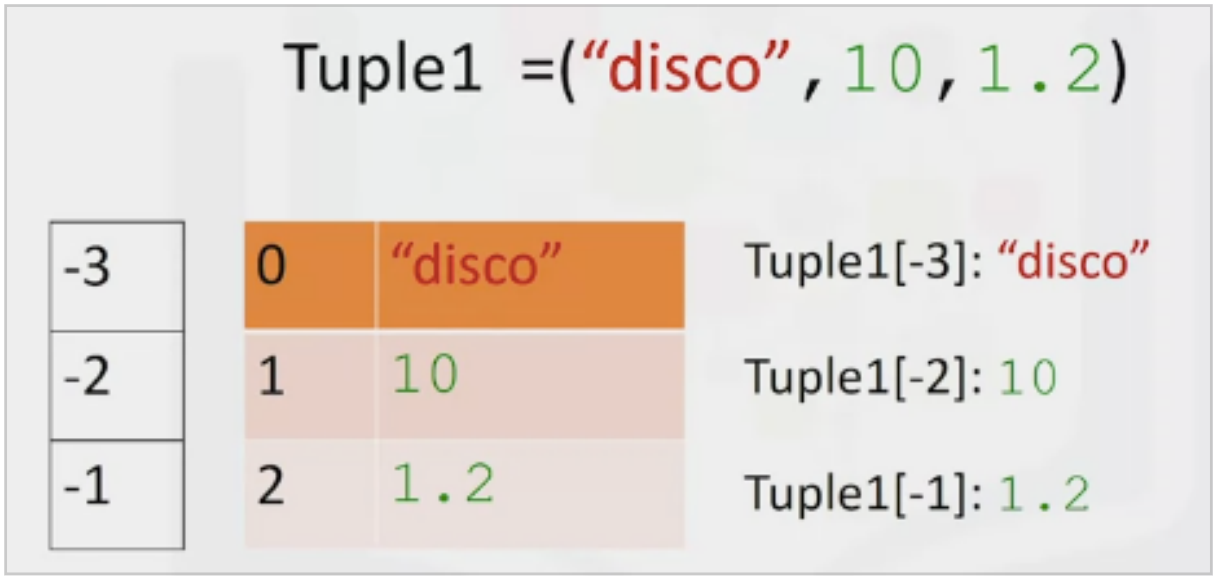


Each element of a tuple can be accessed via an index. The following table represents the relationship between the index and the elements in the tuple.

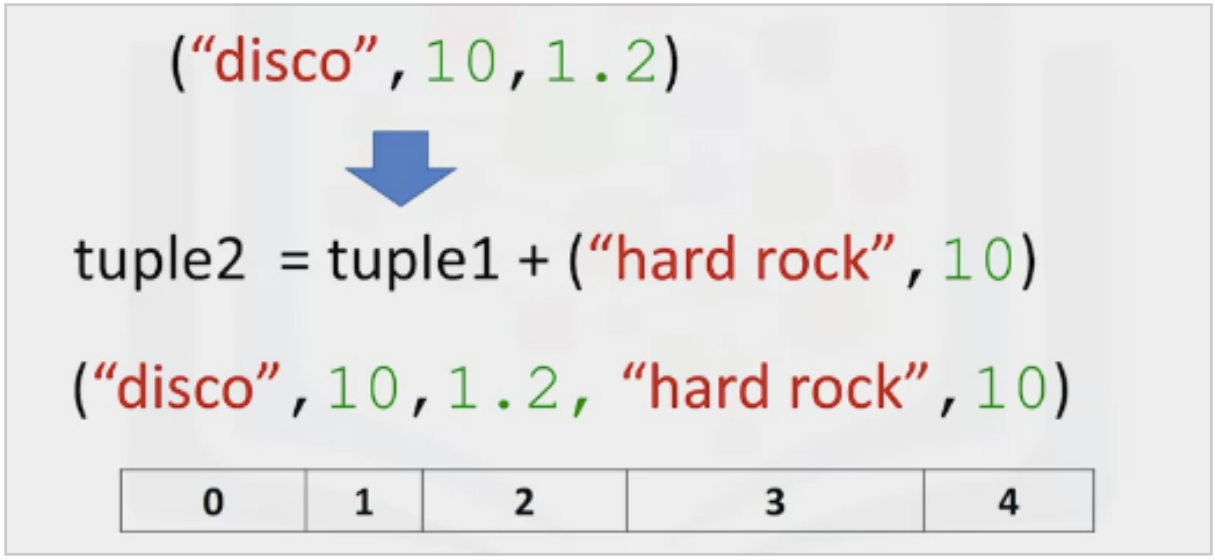


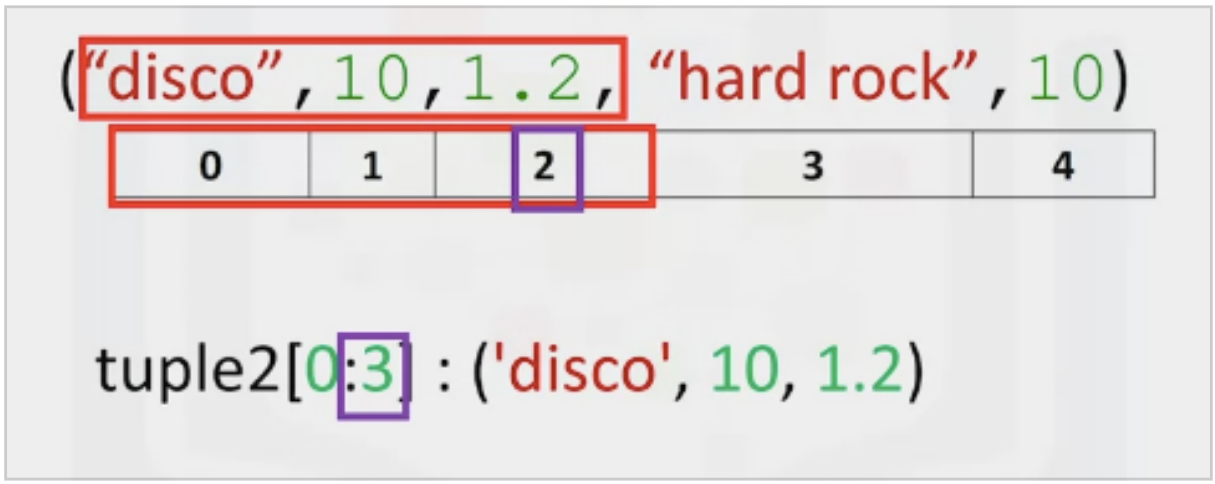
The first element can be accessed by the name of the tuple followed by a square bracket with the index number, in this case zero. We can access the second element as follows. We can also access the last element.

In Python, we can use negative index. The relationship is as follows. The corresponding values are shown here.

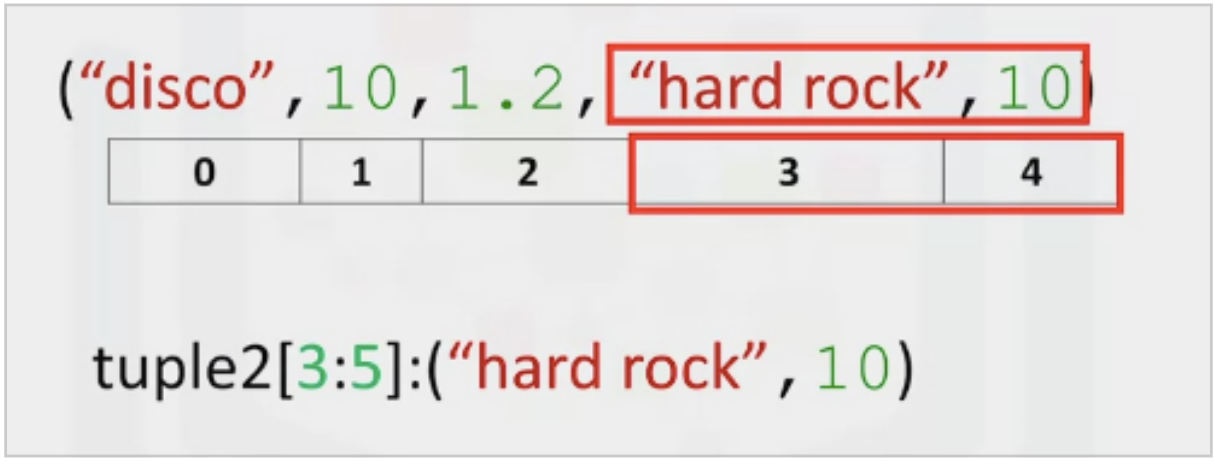


We can concatenate or combine tuples by adding them. The result is the following with the following index.

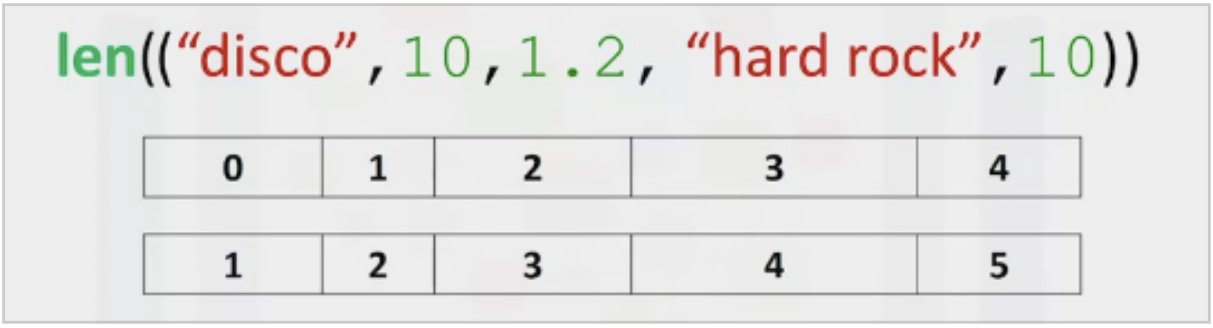




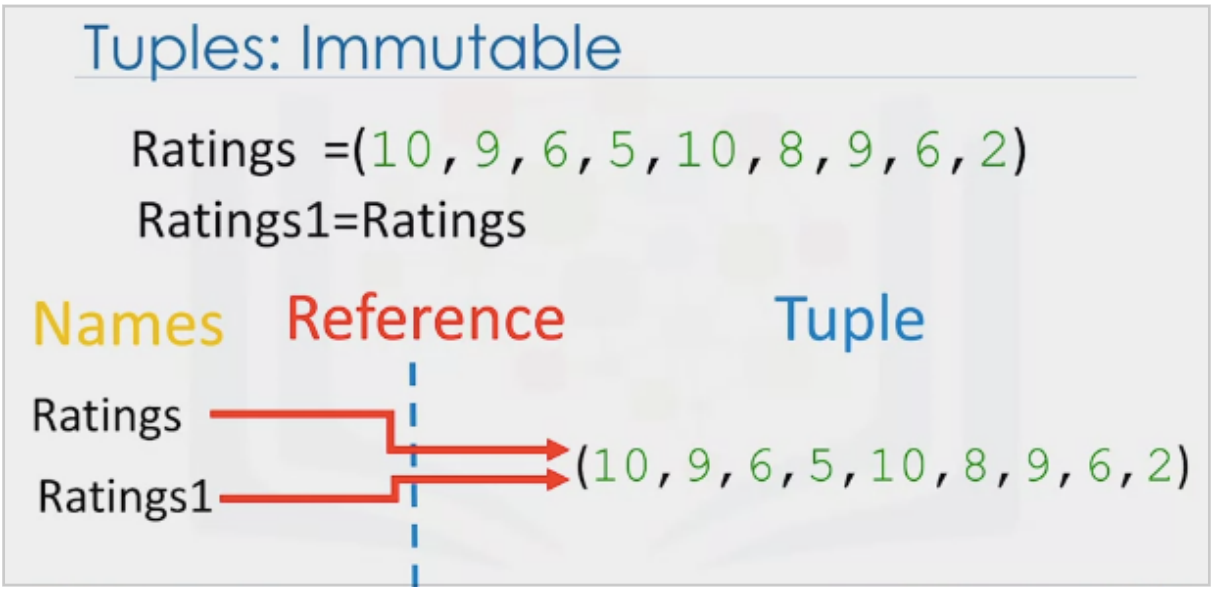
If we would like multiple elements from a tuple, we could also slice tuples. For example, if we want the first three elements we use the following command. **The last index is one larger than the index you want**;



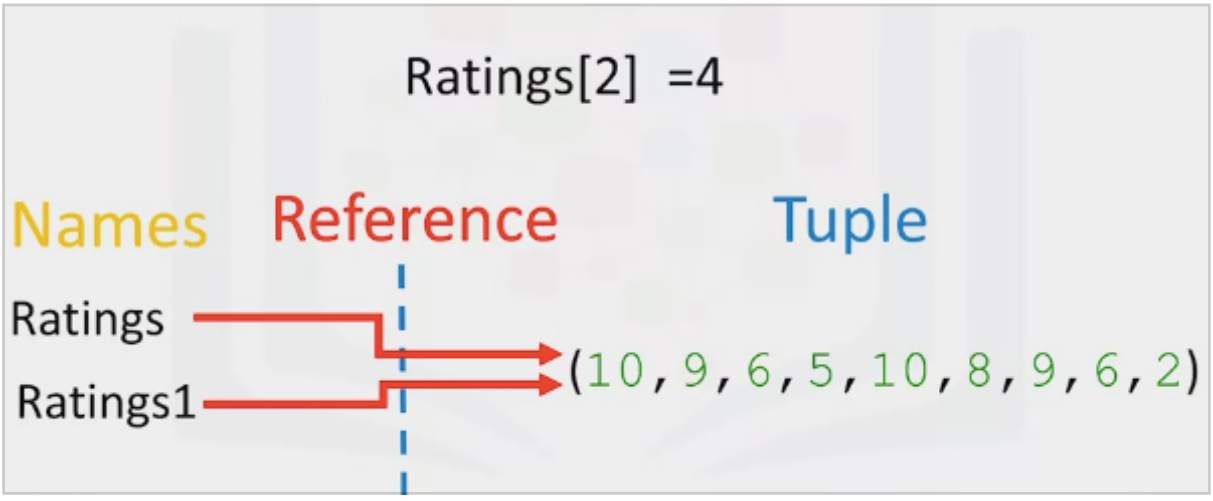
similarly if we want the last two elements, we use the previous command. Notice, how the last index is one larger than the length of the tuple.



We can use the len command to obtain the length of a tuple. As there are five elements, the result is 5.

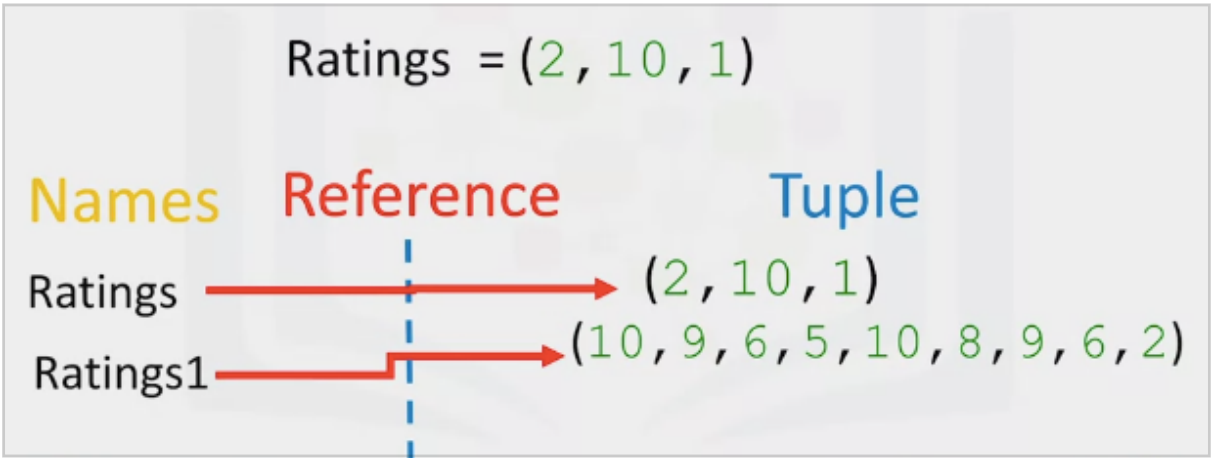


Tuples are immutable which means we can't change them. To see why this is important, let's see what happens when we set the variable ratings 1 to ratings. Let's use the image to provide a simplified explanation of what's going on. Each variable does not contain a tuple, but references the same immutable tuple object. See the objects and classes module for more about objects.

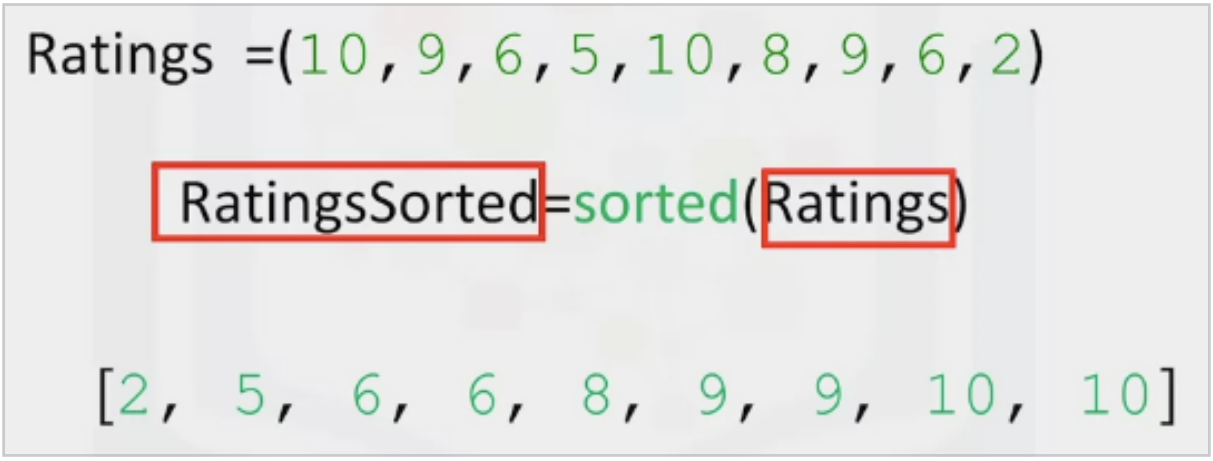


Let's say, we want to change the element at index 2. Because tuples are immutable we can't, therefore ratings 1 will not be affected by a change in rating because the tuple is

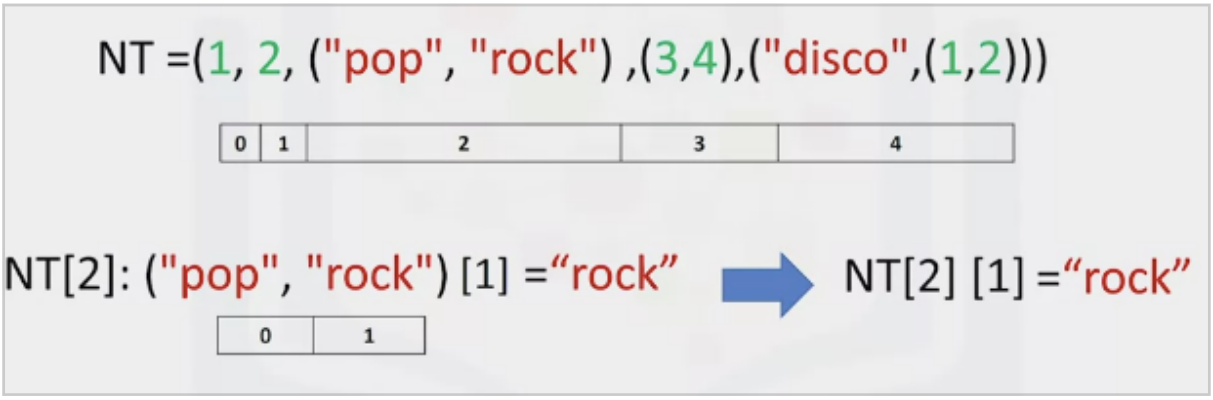
immutable, i.e we can't change it.



We can assign a different tuple to the ratings variable. The variable ratings now references another tuple. **As a consequence of immutability, if we would like to manipulate a tuple we must create a new tuple instead**.

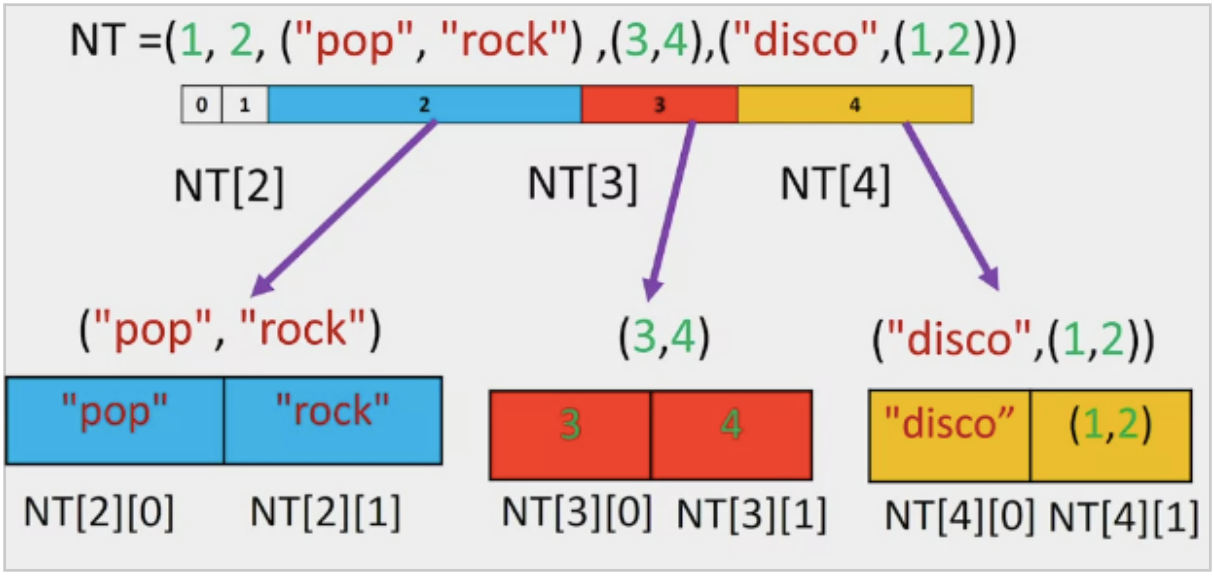


For example, if we would like to sort a tuple we use the function sorted. The input is the original tuple, the output is a new sorted list. For more on functions, see our video on functions.

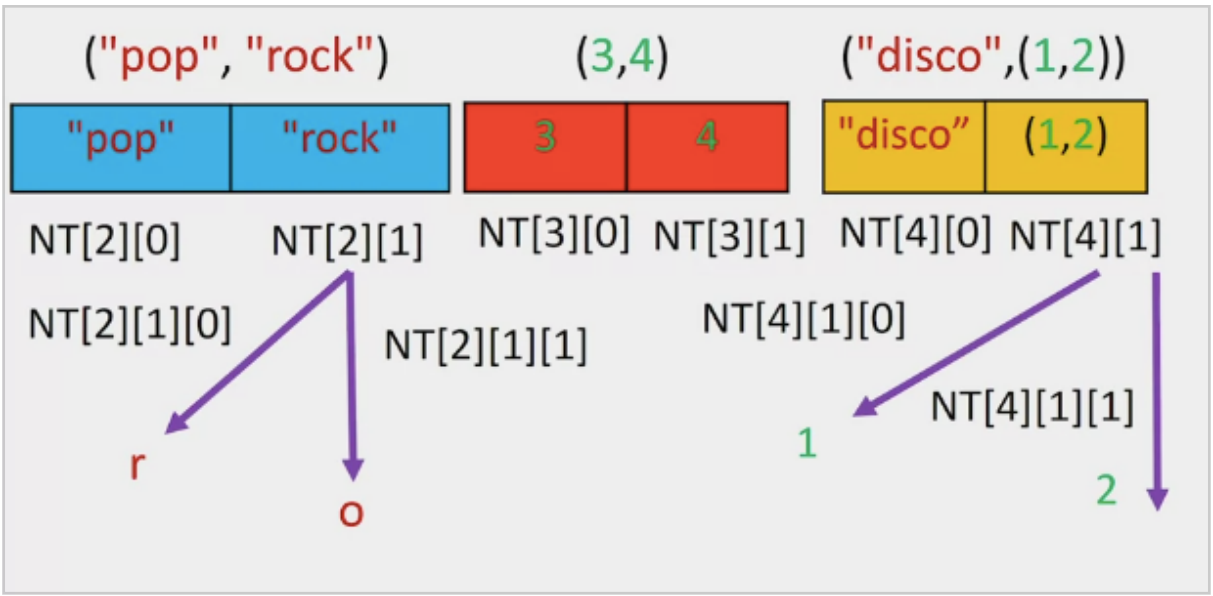


A tuple can contain other tuples as well as other complex data types. This is called **nesting**. We can access these elements using the standard indexing methods. If we select an index with a tuple, the same index convention applies. As such, we can then access

values in the tuple. For example, we could access the second element. We can apply this indexing directly to the tuple variable NT.

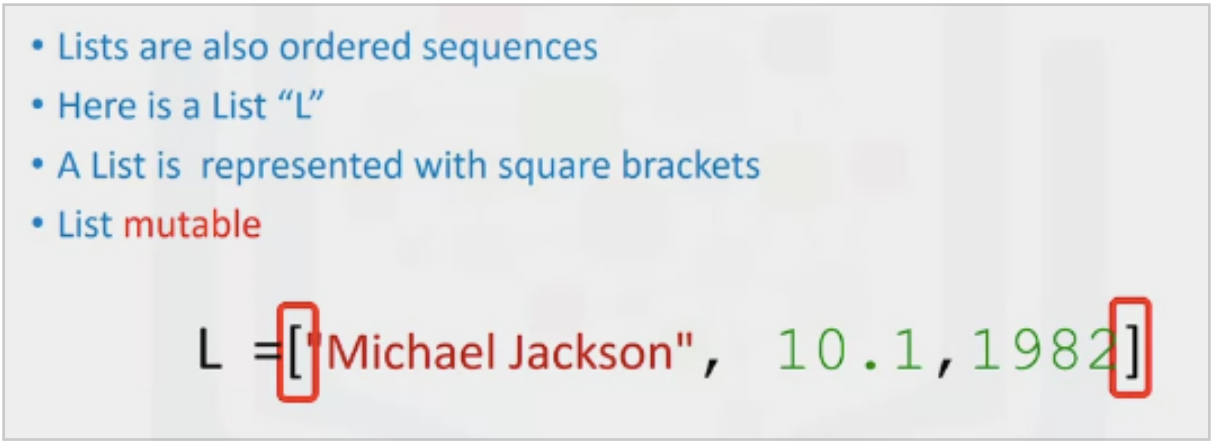


It is helpful to visualize this as a tree. We can visualize this nesting as a tree. The tuple has the following indexes. If we consider indexes with other tuples, we see the tuple at index 2 contains a tuple with two elements. We can access those two indexes. The same convention applies to index 3. We can access the elements in those tuples as well. We can continue the process.



We can even access deeper levels of the tree by adding another square bracket. We can access different characters in the string or various elements in the second tuple contained in the first. Lists are also a popular data structure in Python.

Lists



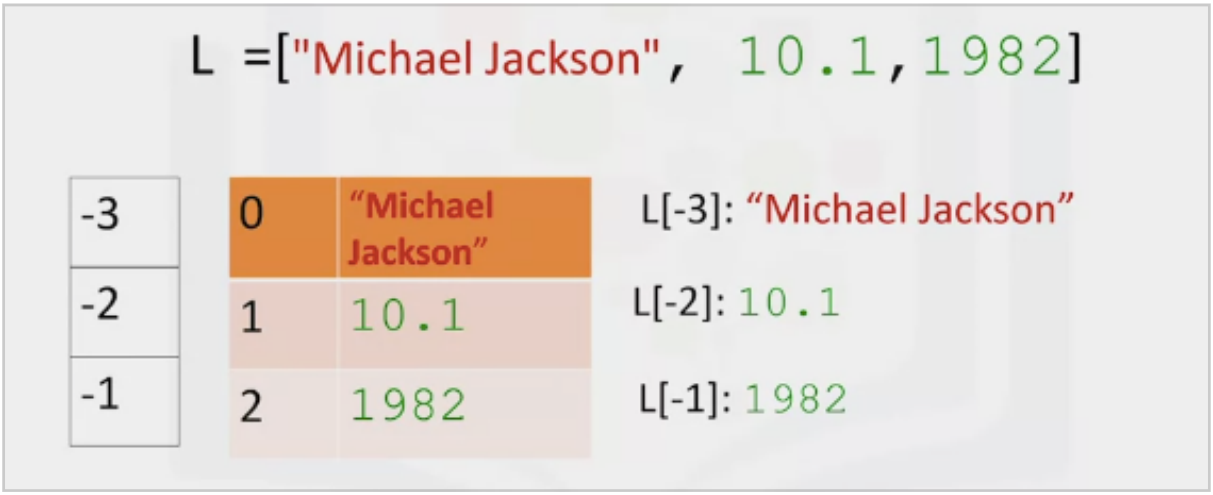
Lists are also an ordered sequence. Here is a list, "L." A list is represented with square brackets. In many respect, lists are like tuples. One key difference is they are **mutable**. Lists can contain strings, floats, integers.

We can nest other lists.

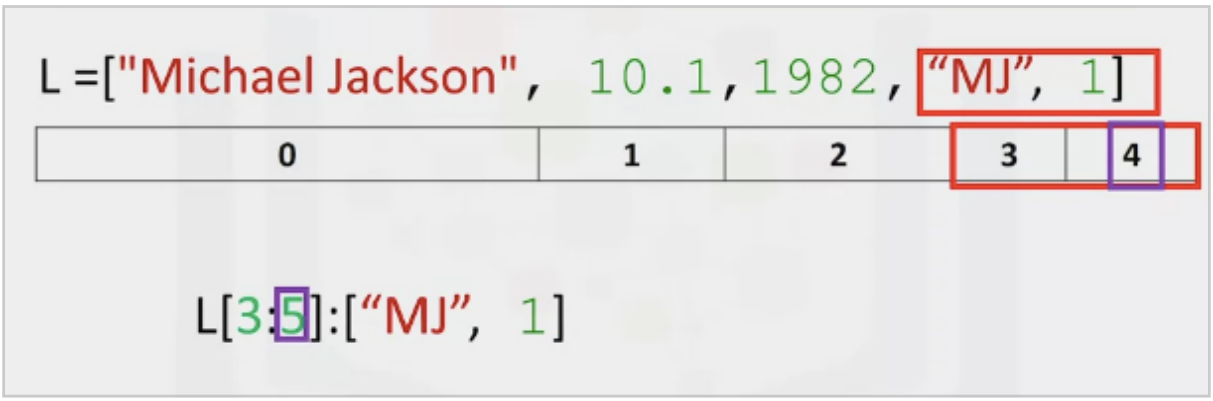
We also nest tuples and other data structures.

The same indexing conventions apply for nesting Like tuples, each element of a list can be accessed via an index. The following table represents the relationship between the index and the elements in the list. The first element can be accessed by the name of the list followed by a square bracket with the index number, in this case zero. We can access the second element as follows. We can also access the last element.

In Python, we can use a negative index; the relationship is as follows. The corresponding indexes are as follows.

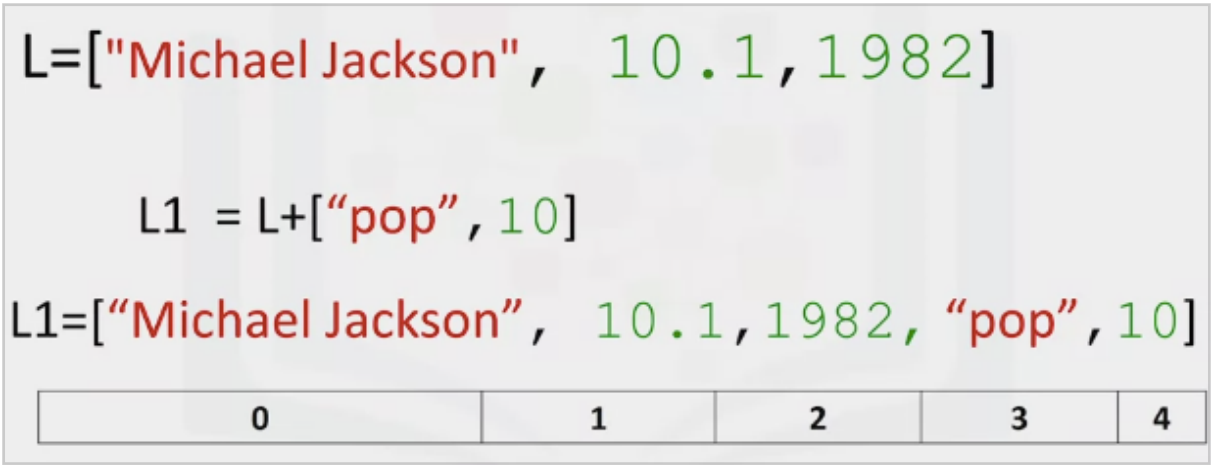


We can also perform slicing in lists. For example, if we want the last two elements in this list we use the following command.



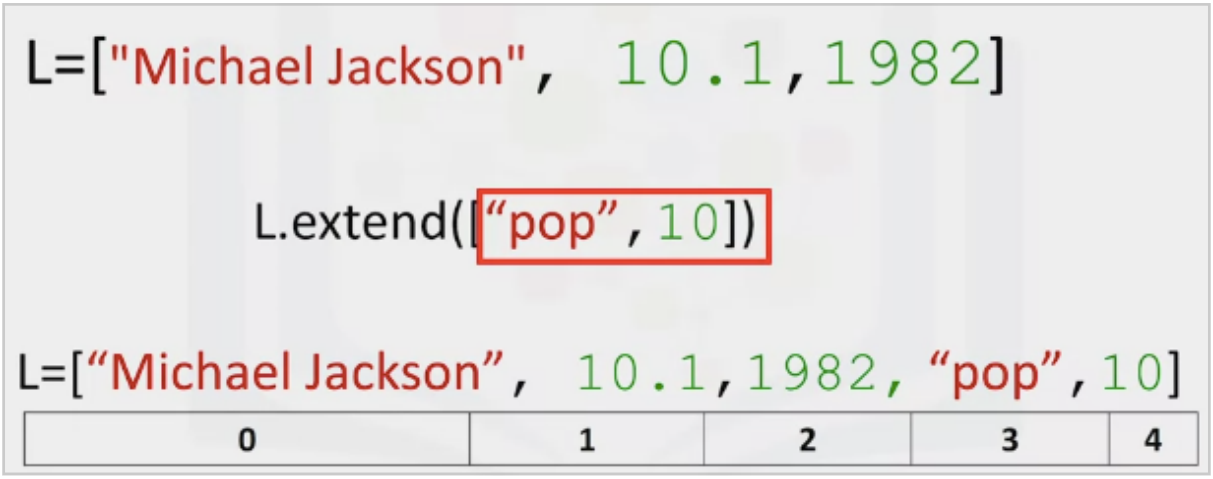
Notice how the last index is one larger than the length of the list. The index conventions for lists and tuples are identical. Check the labs for more examples.

We can **concatenate** or combine lists by adding them. The result is the following.



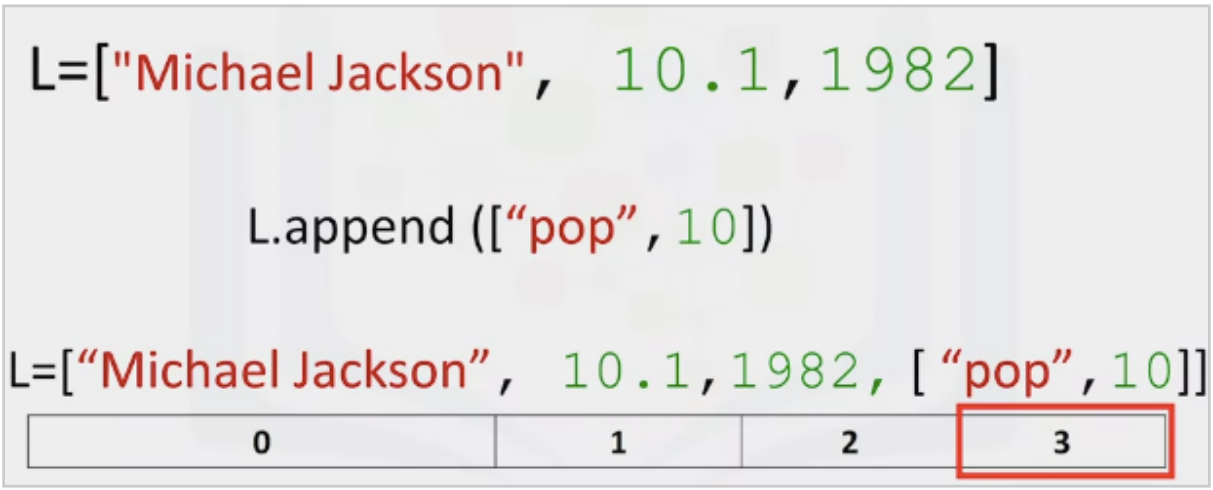
The new list has the following indices. **Lists are mutable**, therefore we can change them. For example, we apply the method `extend` by adding a dot followed by the name of the

method then parentheses.

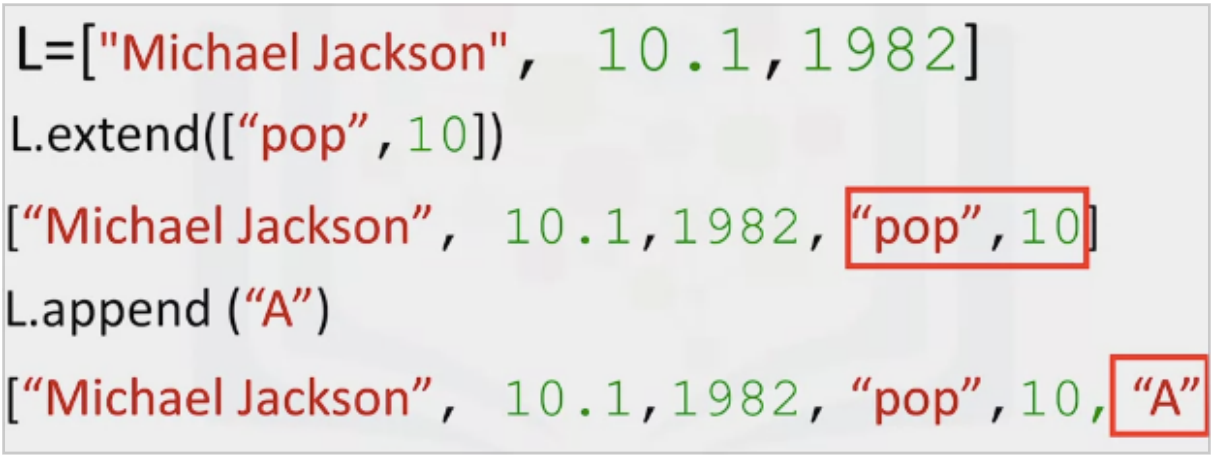


The argument inside the parentheses is a new list that we are going to concatenate to the original list. In this case, instead of creating a new list, "L1," the original list, "L," is modified by adding two new elements. To learn more about methods check out our video on objects and classes.

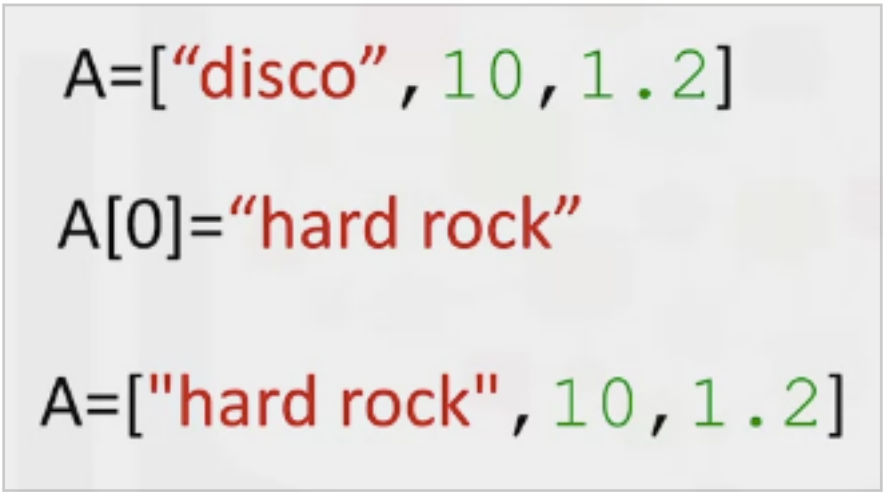
Another similar method is `append`. If we apply append instead of extended, **we add one element to the list**. If we look at the index there is only one more element.



Index 3 contains the list we appended. Every time we apply a method, the list changes. If we apply extend, we add two new elements to the list. The list L is modified by adding two new elements. If we append the string A, we further change the list, adding the string A. As lists are mutable we can change them.

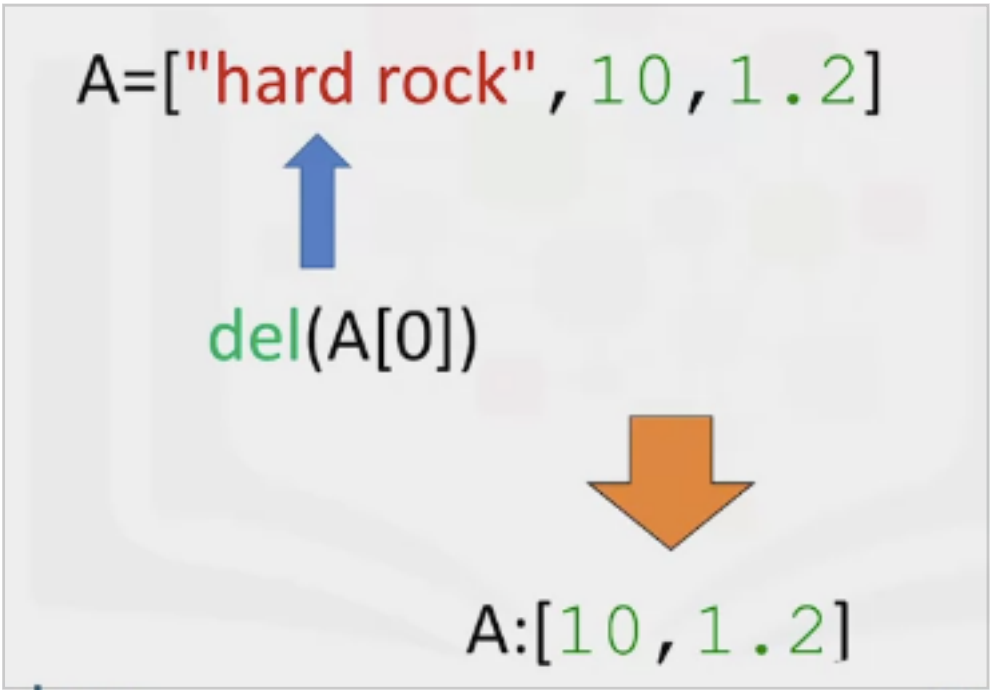


For example, we can change the first element as follows.

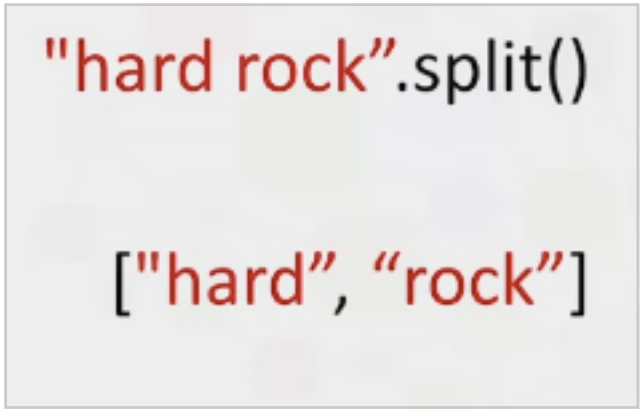


The list now becomes hard rock 10 1.2.

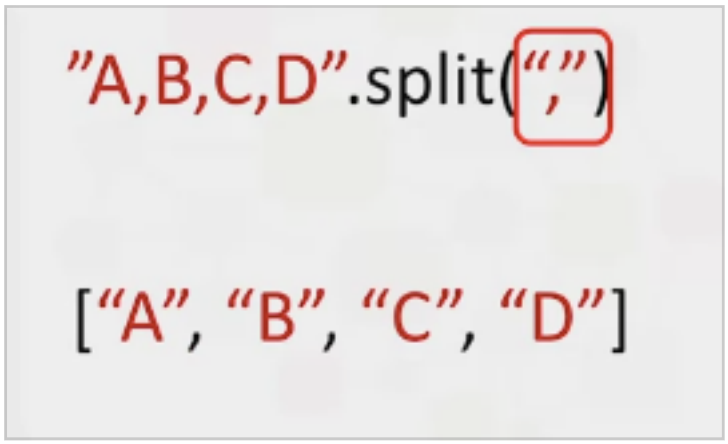
We can `delete` an element of a list using the del command. We simply indicate the list item we would like to remove as an argument. For example, if we would like to remove the first element the result becomes 10 1.2.



We can delete the second element. This operation removes the second element off the list. We can **convert** a string to a list using `split`.



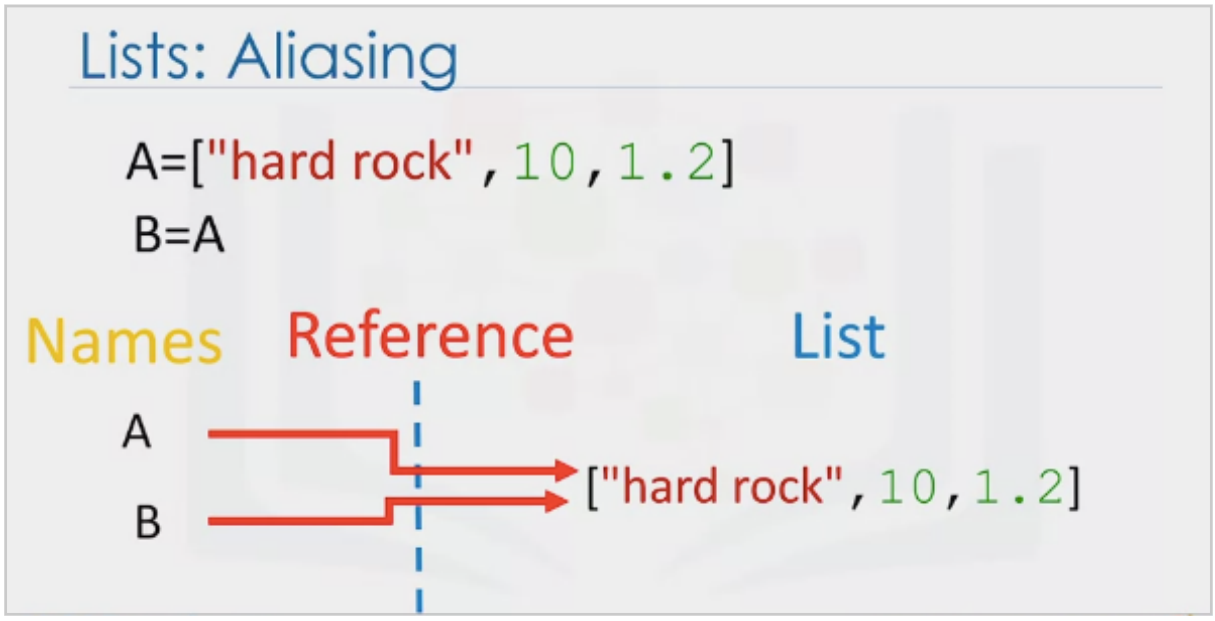
For example, the method split converts every group of characters separated by a space into an element of a list. We can use the split function to separate strings on a specific character known, as a delimiter.



We simply pass the delimiter we would like to split on as an argument, in this case a comma. The result is a list. Each element corresponds to a set of characters that have been separated by a comma.

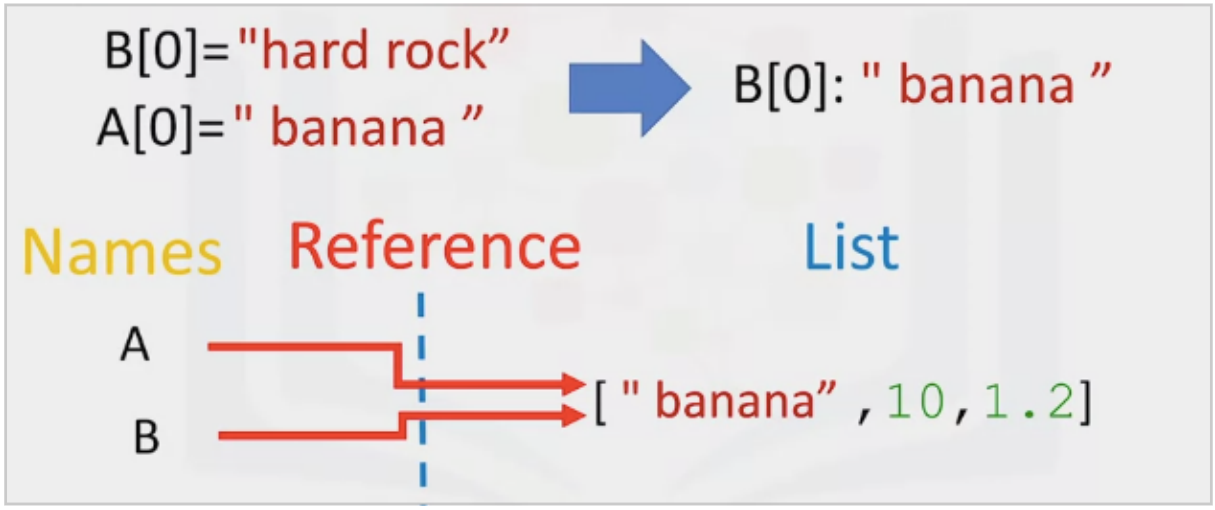
When we set one variable B equal to A, both A and B are referencing the same list.

Multiple names referring to the same object is known as **aliasing**.

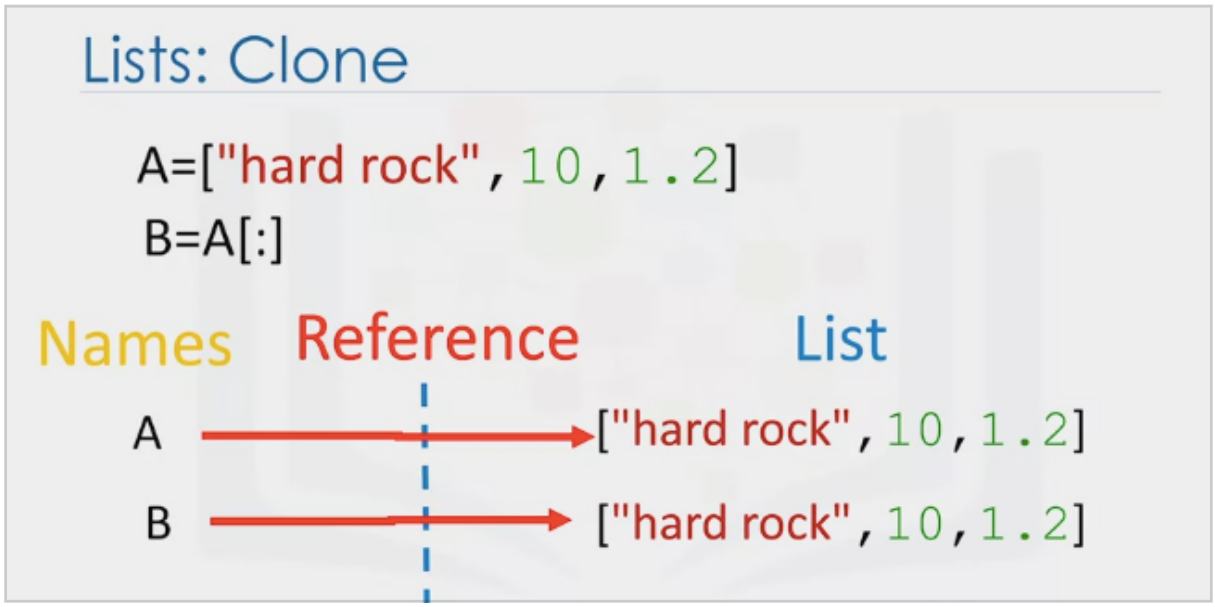


We know from the list slide that the first element in B is set as hard rock. If we change the

first element in A to banana, we get a side effect, the value of B will change as a consequence.

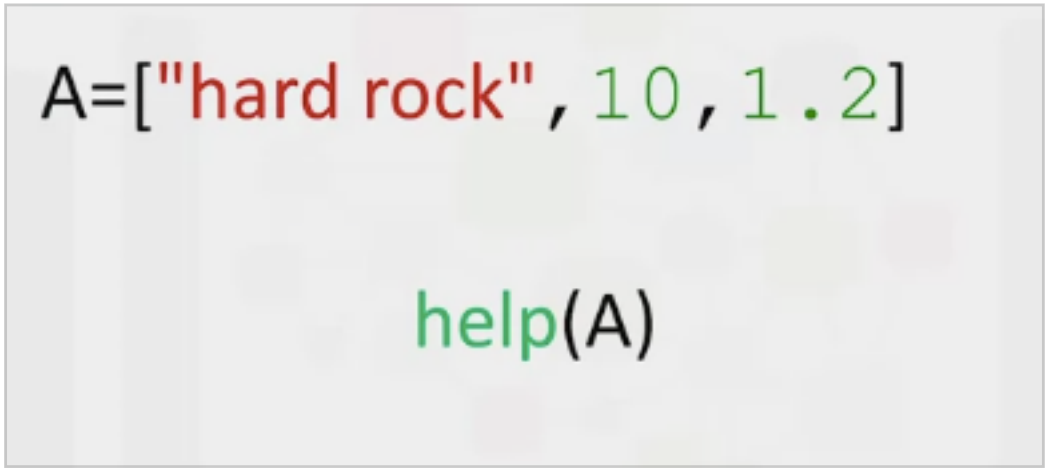


A and B are referencing the same list, therefore if we change A, list B also changes. If we check the first element of B after changing list A, we get banana instead of hard rock.
You can `clone` list A by using the following syntax.
Variable A references one list. Variable B references a new copy or clone of the original list.



Now if you change A, B will not change.

We can get more info on lists, tuples, and many other objects in Python using the `help` command.



Simply pass in the list, tuple, or any other Python object. See the labs for more things, you can do with lists.

Lab 1

Lists in Python

Objectives

After completing this lab you will be able to:

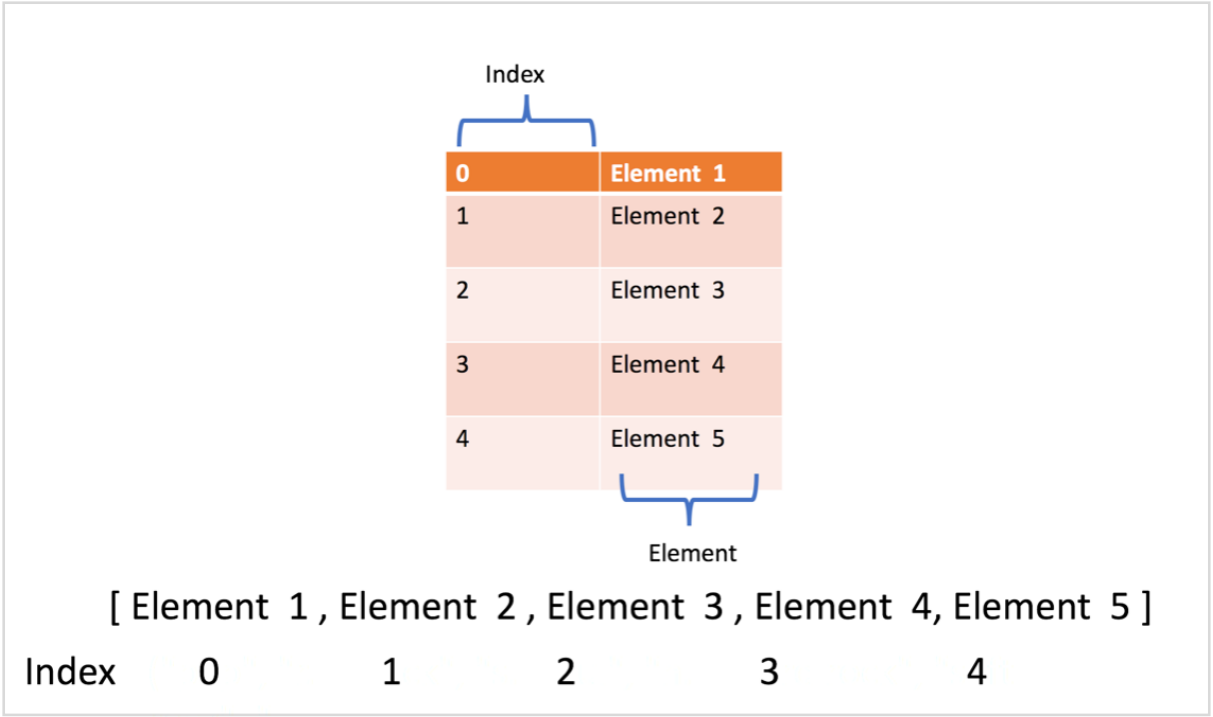
- Perform list operations in Python, including indexing, list manipulation, and copy/clone list.

Table of Contents

Lists

Indexing - List

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 even 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.

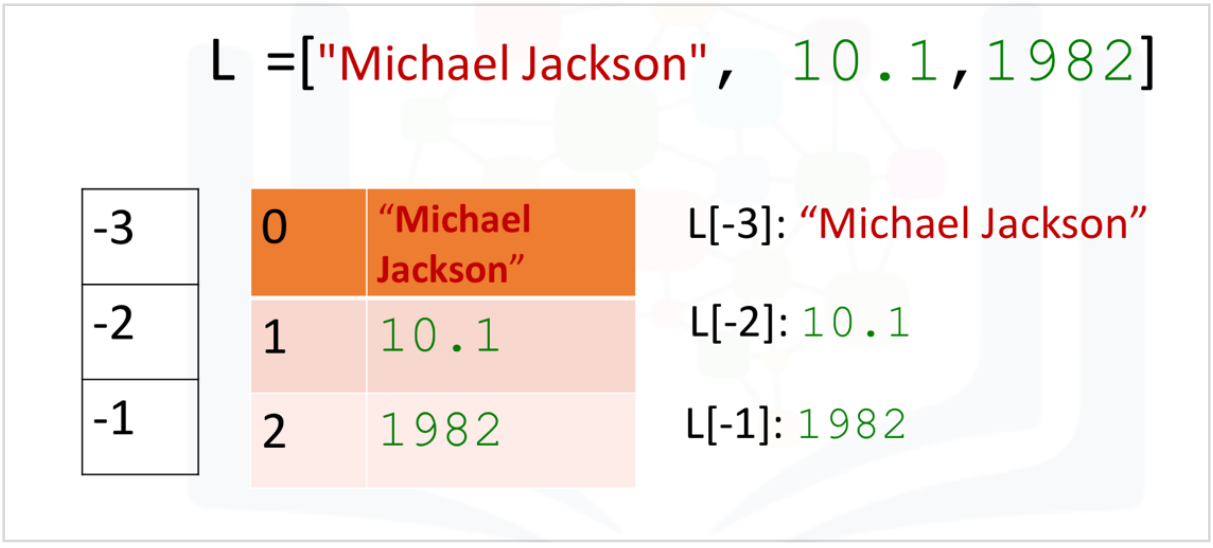


To create a list, type the list within square brackets \[ ], with your content inside the parenthesis and separated by commas. Let's try it!

```
# Create a list
L = ["Michael Jackson", 10.1, 1982]
L
```

['Michael Jackson', 10.1, 1982]

We can use negative and regular indexing with a list:



```
# 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]  )
```

the same element using negative and positive indexing:
Postive: Michael Jackson
Negative: Michael Jackson
the same element using negative and positive indexing:
Postive: 10.1
Negative: 10.1
the same element using negative and positive indexing:
Postive: 1982
Negative: 1982

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:

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

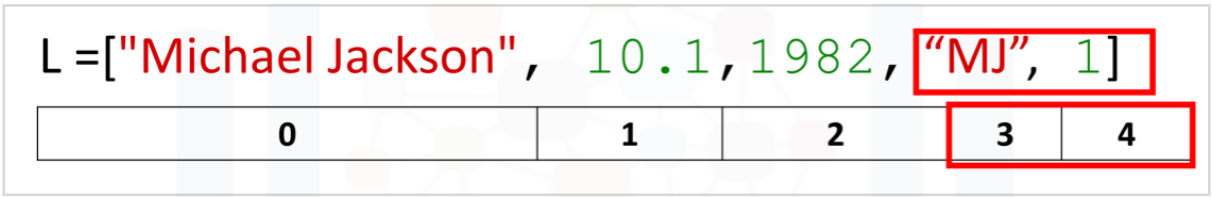
['Michael Jackson', 10.1, 1982, [1, 2], ('A', 1)]

List Operations

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

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

['Michael Jackson', 10.1, 1982, 'MJ', 1]




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

['MJ', 1]

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

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

['Michael Jackson', 10.2, 'pop', 10]

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

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

['Michael Jackson', 10.2, ['pop', 10]]

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

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

['Michael Jackson', 10.2, 'pop', 10]

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

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

['Michael Jackson', 10.2, 'pop', 10, ['a', 'b']]

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

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

Before change: ['disco', 10, 1.2]
After change: ['hard rock', 10, 1.2]

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

```

```

```
# Delete the element based on the index
print('Before change:', A)
del(A[0])
print('After change:', A)
```

Before change: ['hard rock', 10, 1.2]
After change: [10, 1.2]

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:

```
# Split the string, default is by space
'hard rock'.split()
```

['hard', 'rock']

We can use the split function to separate strings on a specific character which we call a **delimiter**. 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:

```
# Split the string by comma
'A,B,C,D'.split(',')
```

['A', 'B', 'C', 'D']

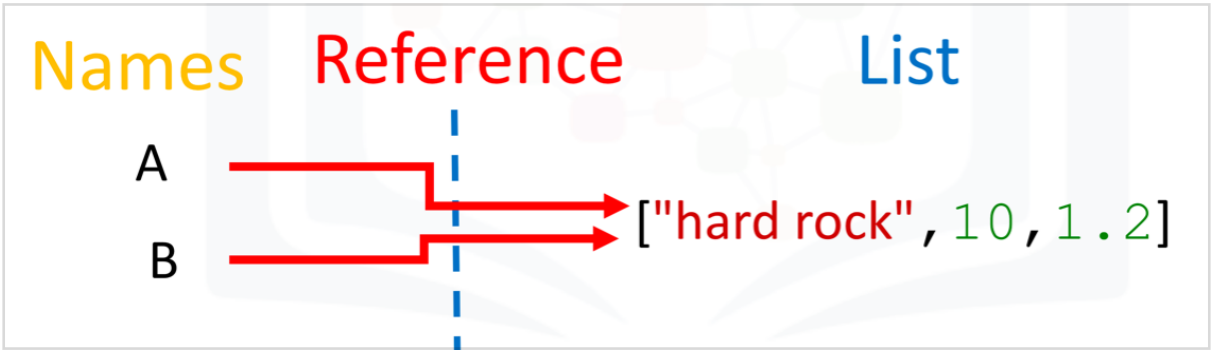
Copy and Clone List

When we set one variable `B` equal to `A`, both `A` and `B` are referencing the same list in memory:

```
# Copy (copy by reference) the list A
A = ["hard rock", 10, 1.2]
B = A
```

```
print('A:', A)
print('B:', B)
```

A: ['hard rock', 10, 1.2]
B: ['hard rock', 10, 1.2]



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

```
# Examine the copy by reference
print('B[0]:', B[0])
A[0] = "banana"
print('B[0]:', B[0])
```

B[0]: hard rock
B[0]: banana

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

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

['banana', 10, 1.2]

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:

```
print('B[0]:', B[0])
A[0] = "hard rock"
print('B[0]:', B[0])
```

B[0]: banana
B[0]: banana

Quiz on List

Create a list `a_list`, with the following elements `1`, `hello`, \`[1,2,3]` and `True`.

```
# Write your code below and press Shift+Enter to execute
a_list=[1,'hello',[1,2,3],'True']
a_list
```

[1, 'hello', [1, 2, 3], 'True']

Find the value stored at index 1 of `a_list`.

```
# Write your code below and press Shift+Enter to execute
a_list[1]
```

'hello'

Retrieve the elements stored at index 1, 2 and 3 of `a_list`.

```

```

```
# Write your code below and press Shift+Enter to execute
a_list[1:4]
```

['hello', [1, 2, 3], 'True']

Concatenate the following lists `A = \[1, 'a']` and `B = \[2, 1, 'd']` :

```
# Write your code below and press Shift+Enter to execute
A = [1, 'a']
B = [2, 1, 'd']
A + B
```

[1, 'a', 2, 1, 'd']

Lab 2

Tuples in Python

Objectives

After completing this lab you will be able to:

- Perform the basics tuple operations in Python, including indexing, slicing and sorting

Table of Contents

Tuples

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.

```
# Create your first tuple
tuple1 = ("disco",10,1.2 )
tuple1
```

('disco', 10, 1.2)

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

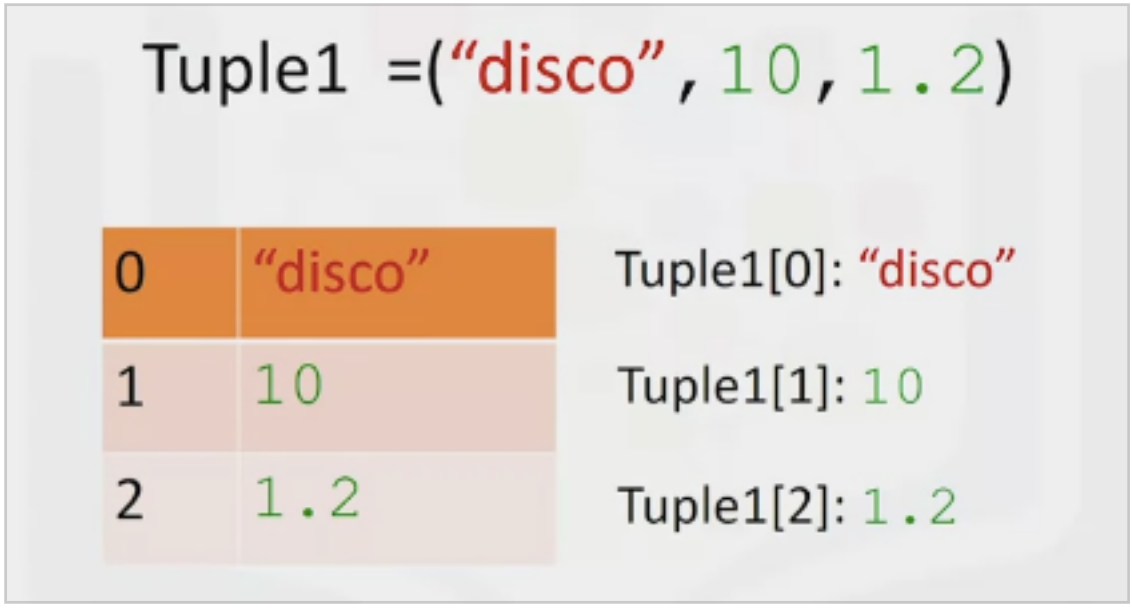
```
# Print the type of the tuple you created

type(tuple1)
```

tuple

Indexing Tuple

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:

```
# Print the variable on each index
print(tuple1[0])
print(tuple1[1])
print(tuple1[2])
```

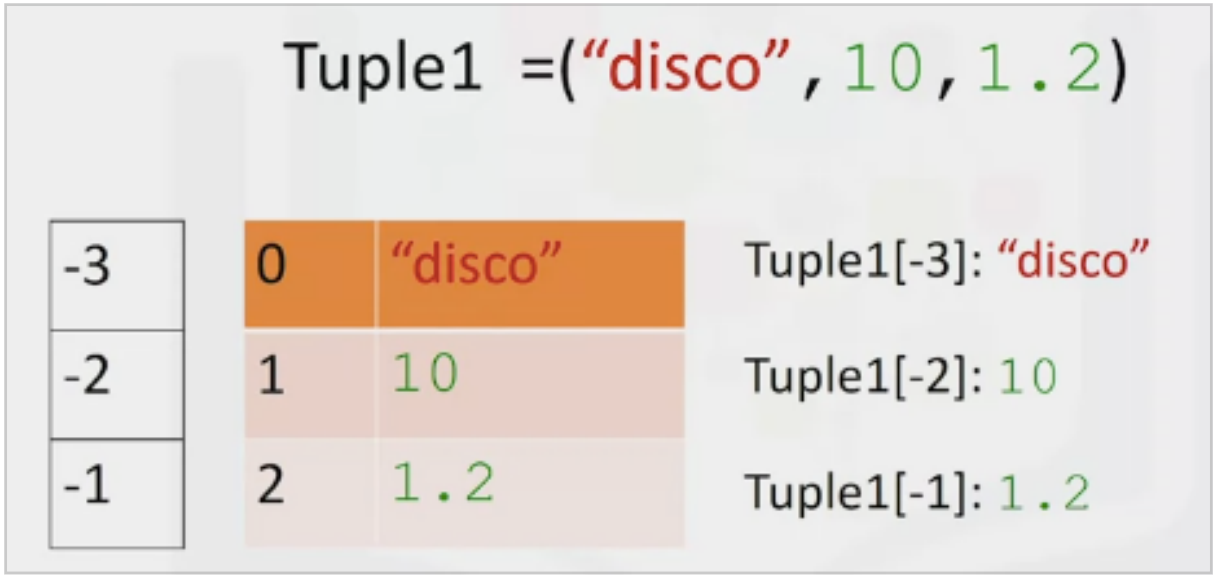
```
disco
10
1.2
```

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

```
# Print the type of value on each index
print(type(tuple1[0]))
print(type(tuple1[1]))
print(type(tuple1[2]))
```

```
<class 'str'>
<class 'int'>
<class 'float'>
```

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):

```
# Use negative index to get the value of the last element
tuple1[-1]
```

1.2

We can display the next two elements as follows:

```
# Use negative index to get the value of the second last element
tuple1[-2]
```

10

```
# Use negative index to get the value of the third last element
tuple1[-3]
```

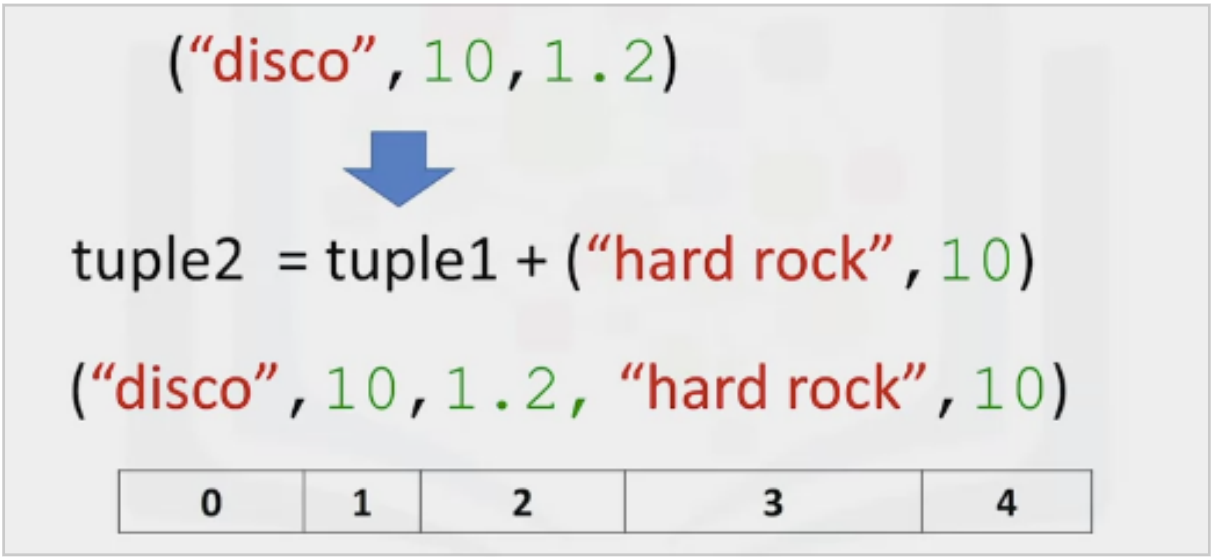
'disco'

Concatenate Tuples

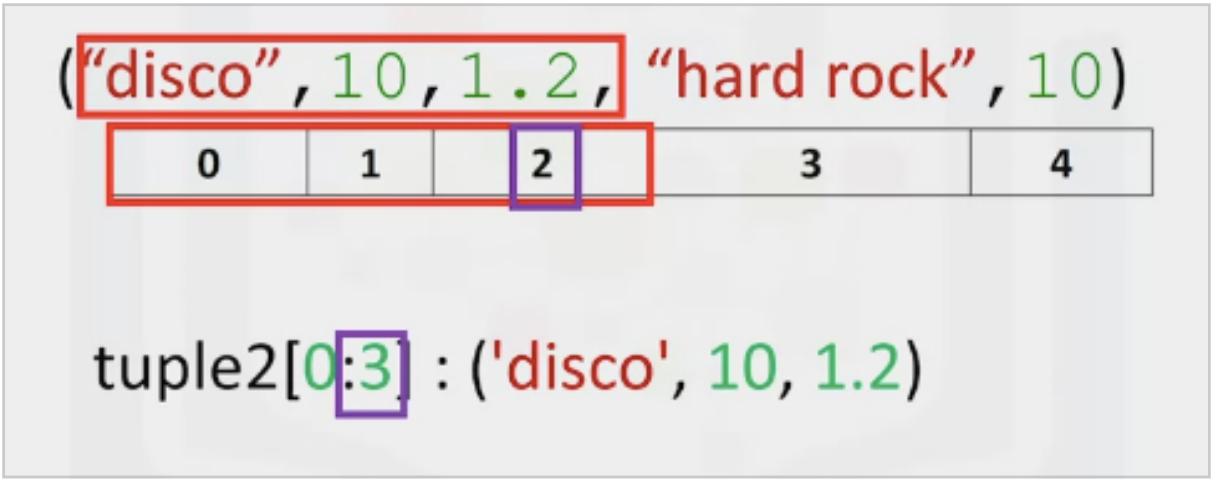
We can concatenate or combine tuples by using the **+** sign:

```
# Concatenate two tuples
tuple2 = tuple1 + ("hard rock", 10)
tuple2
```

('disco', 10, 1.2, 'hard rock', 10)



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



Slicing

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

```
# Slice from index 0 to index 2
tuple2[0:3]
```

('disco', 10, 1.2)

We can obtain the last two elements of the tuple:

```
# Slice from index 3 to index 4
tuple2[3:5]
```

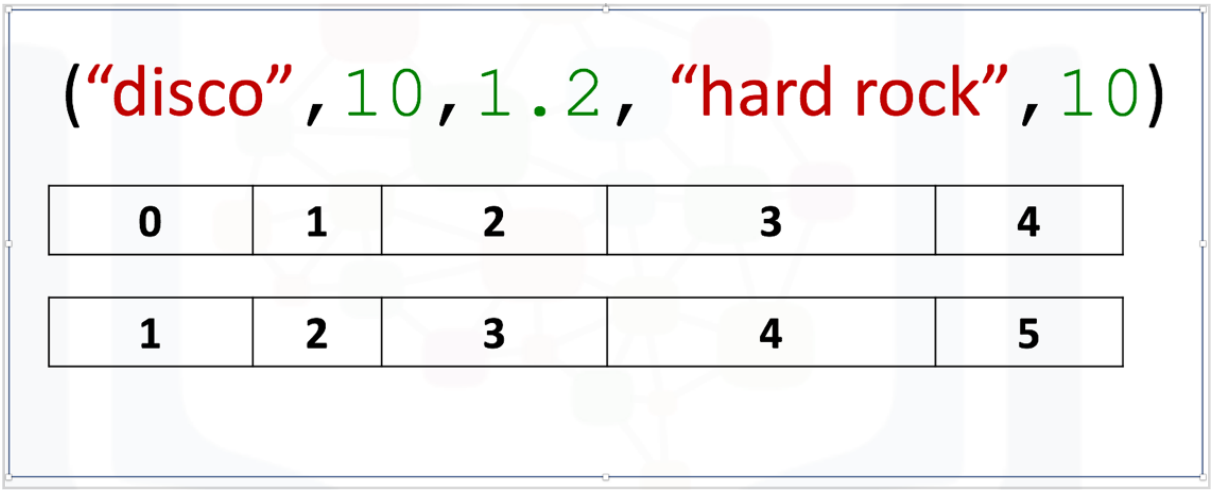
('hard rock', 10)

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

```
# Get the length of tuple
len(tuple2)
```

5

This figure shows the number of elements:



Sorting

Consider the following tuple:

```
# 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:

```
# Sort the tuple
RatingsSorted = sorted(Ratings)
RatingsSorted
```

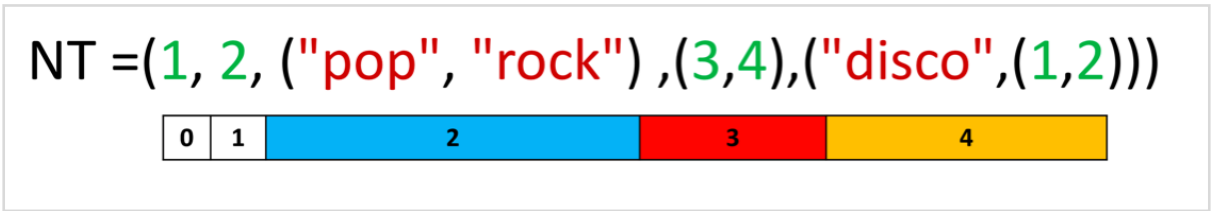
[0, 2, 5, 6, 6, 8, 9, 9, 10]

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:

```
# 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:

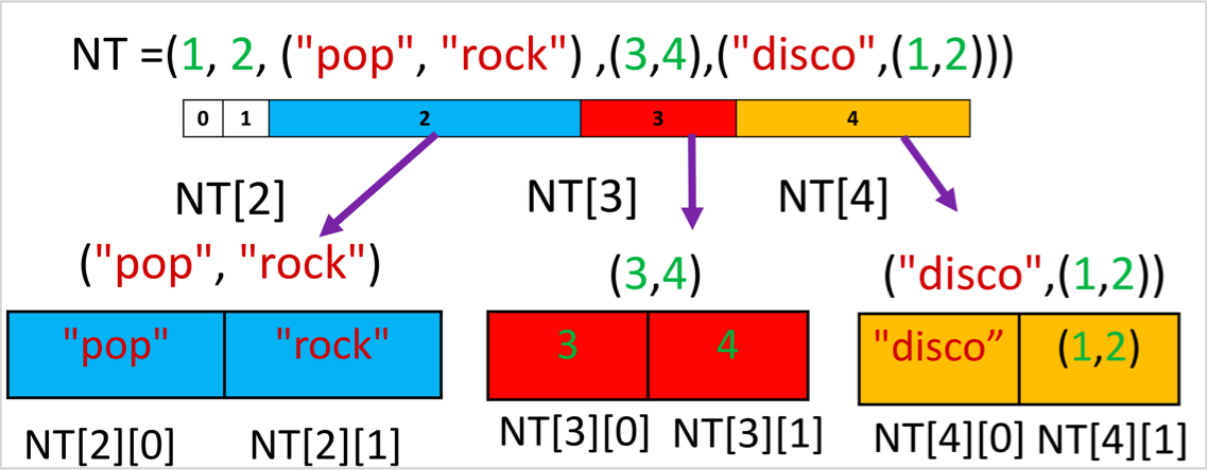



```
# Print element on each index
print("Element 0 of Tuple: ", NestedT[0])
print("Element 1 of Tuple: ", NestedT[1])
```

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

Element 0 of Tuple: 1
Element 1 of Tuple: 2
Element 2 of Tuple: ('pop', 'rock')
Element 3 of Tuple: (3, 4)
Element 4 of Tuple: ('disco', (1, 2))

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



We can access the nested tuples:

```
# 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])
```

Element 2, 0 of Tuple: pop
Element 2, 1 of Tuple: rock
Element 3, 0 of Tuple: 3

Element 3, 1 of Tuple: 4

Element 4, 0 of Tuple: disco

Element 4, 1 of Tuple: (1, 2)

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

```
# Print the first element in the second nested tuples
NestedT[2][1][0]
```

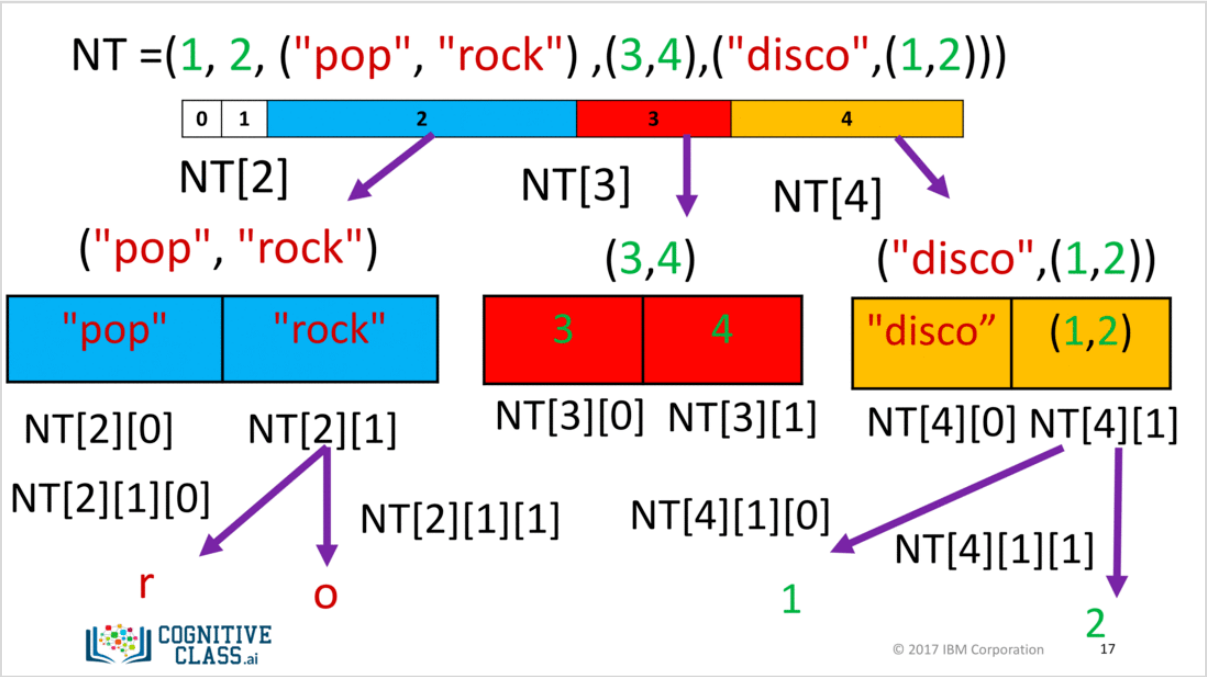
'r'

```
# Print the second element in the second nested tuples

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

'o'

We can use a tree to visualize 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 third index:

```
# Print the first element in the second nested tuples
NestedT[4][1][0]
```

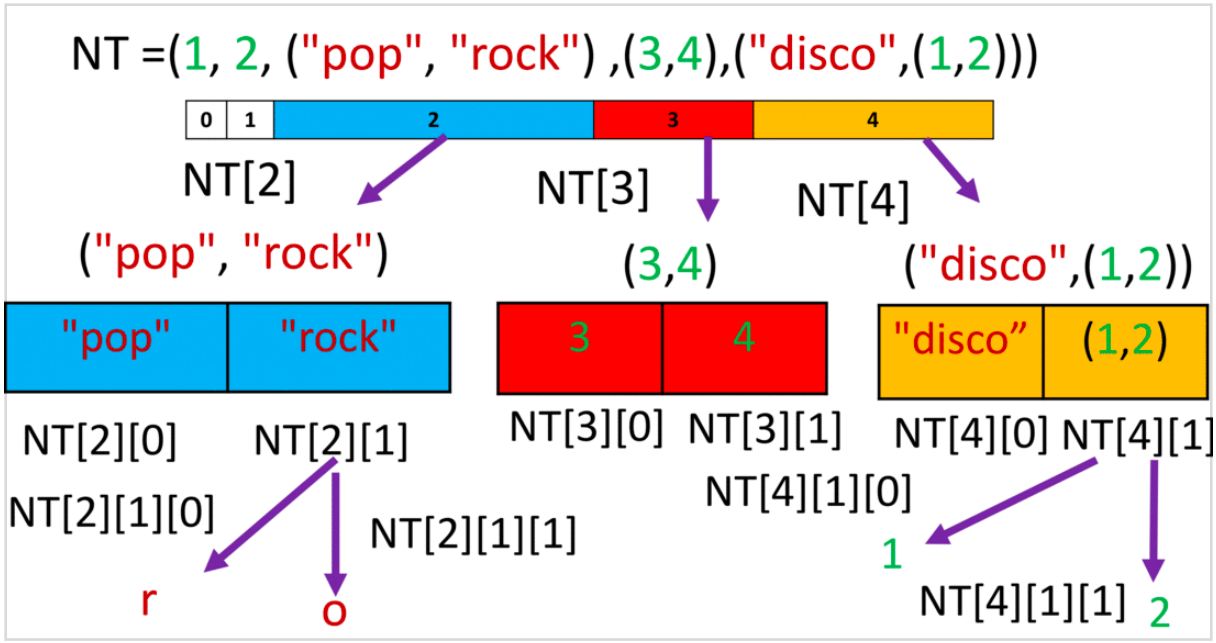
1

```
# Print the second element in the second nested tuples

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

2

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



Quiz on Tuples

Consider the following tuple:

```
# sample tuple
genres_tuple = ("pop", "rock", "soul", "hard rock", "soft rock", \
```

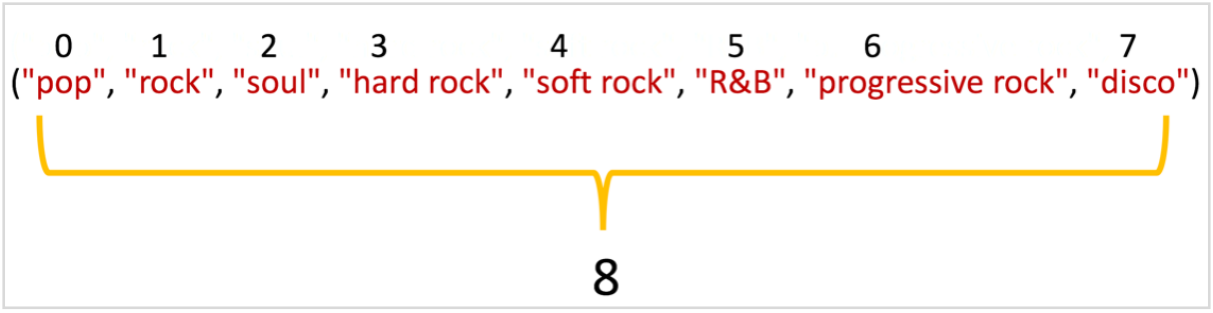
```
                "R&B", "progressive rock", "disco")
genres_tuple
```

```
('pop',
 'rock',
 'soul',
 'hard rock',
 'soft rock',
 'R&B',
 'progressive rock',
 'disco')
```

Find the length of the tuple, `genres_tuple` :

```
# Write your code below and press Shift+Enter to execute
len(genres_tuple)
```

```
8
```



Access the element, with respect to index 3:

```
# Write your code below and press Shift+Enter to execute
genres_tuple[3]
```

```
'hard rock'
```

Use slicing to obtain indexes 3, 4 and 5:

```
# Write your code below and press Shift+Enter to execute
genres_tuple[3:6]
```

('hard rock', 'soft rock', 'R&B')

Find the first two elements of the tuple `genres_tuple`:

```
# Write your code below and press Shift+Enter to execute
genres_tuple[0:2]
```

('pop', 'rock')

Find the first index of `"disco"`:

```
# Write your code below and press Shift+Enter to execute
genres_tuple.index("disco")
```

7

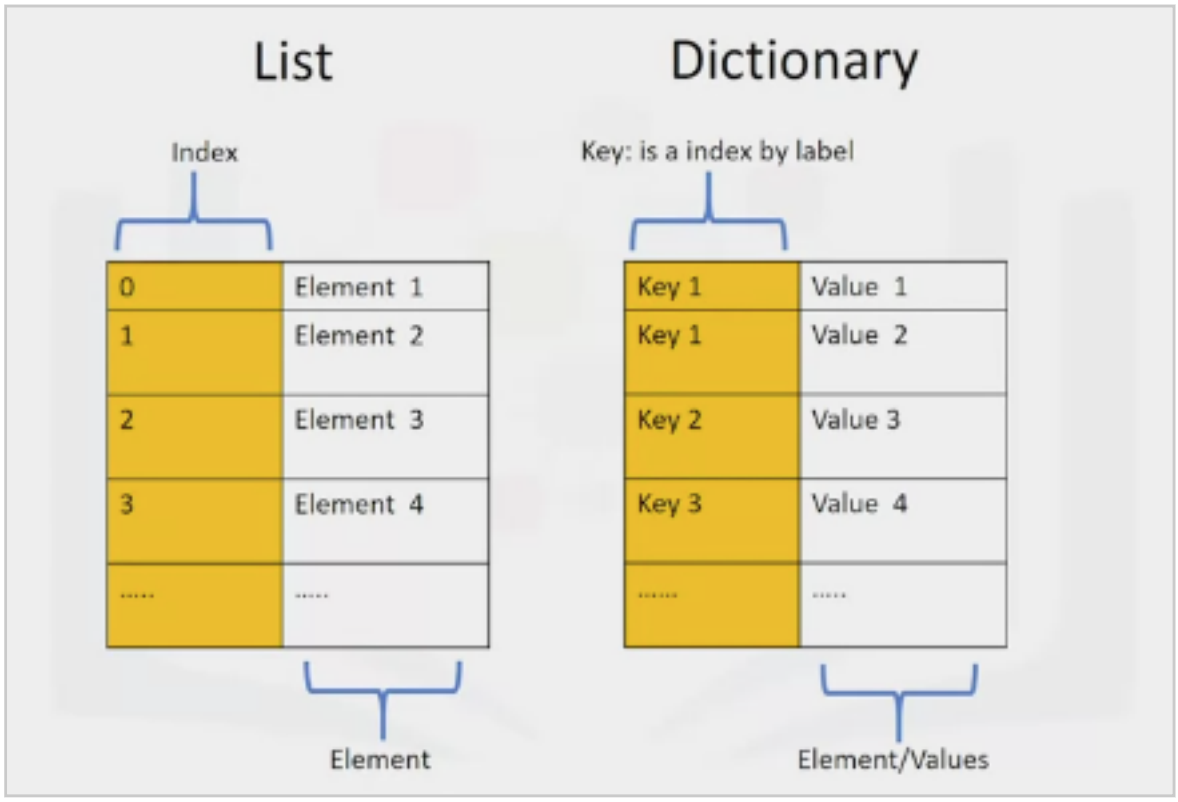
Generate a sorted List from the Tuple `C_tuple=(-5, 1, -3)`:

```
# Write your code below and press Shift+Enter to execute
C_tuple = (-5,1,-3)
sorted_C_tuple = sorted(C_tuple)
sorted_C_tuple
```

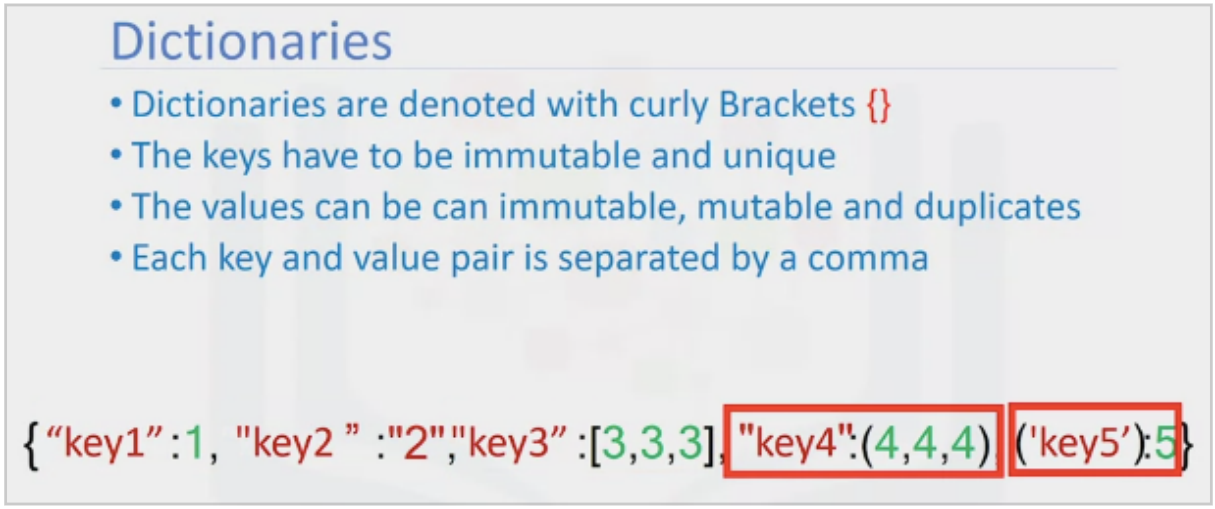
[-5, -3, 1]

Dictionaries theory

Let's cover Dictionaries in Python. **Dictionaries are a type of collection in Python**. If you recall, a list is integer indexes. These are like addresses. A list also has elements. A **dictionary has keys and values**. The key is analogous to the index. They are like addresses, but they don't have to be integers. They are usually characters. The values are similar to the element in a list and contain information.



To create a dictionary, we use curly brackets.



The keys are the first elements. They must be immutable and unique. Each key is followed by a value separated by a colon. The values can be immutable, mutable, and duplicates. Each key and value pair is separated by a comma.

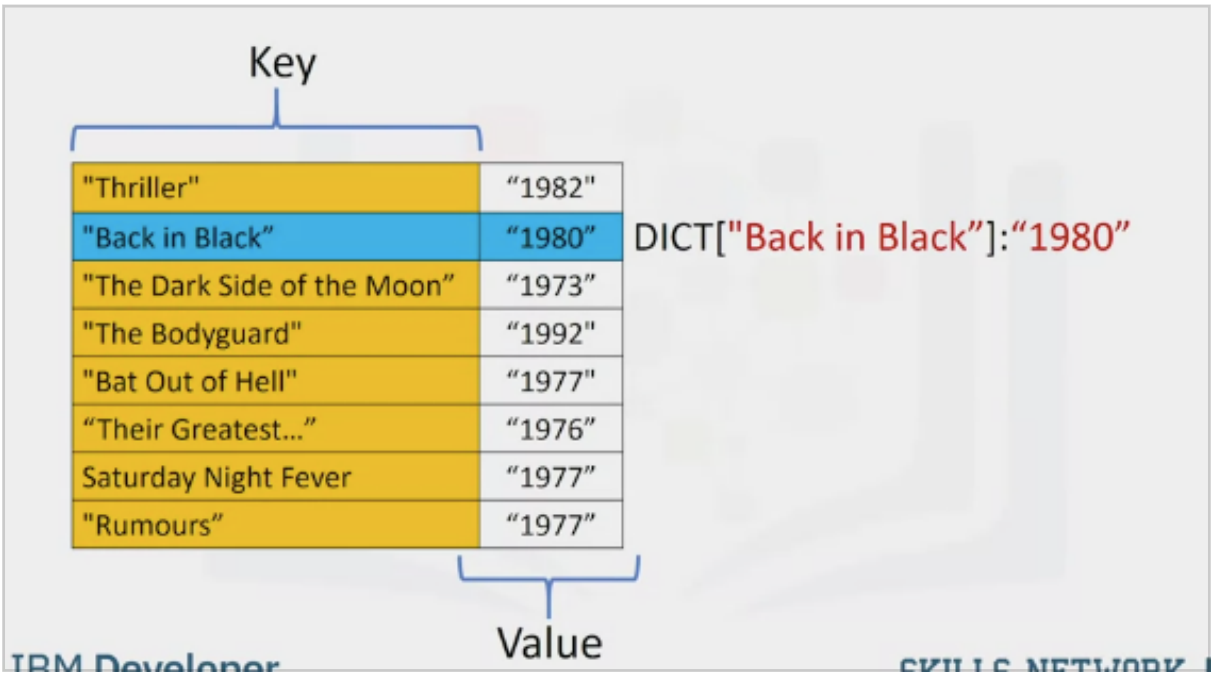
Consider the following example of a dictionary.

The album title is the key, and the value is the released data. We can use yellow to highlight the keys and leave the values in white. It is helpful to use the table to visualize a dictionary where the first column represents the keys, and the second column represents the values. We can add a few more examples to the dictionary.

We can also assign the dictionary to a variable.

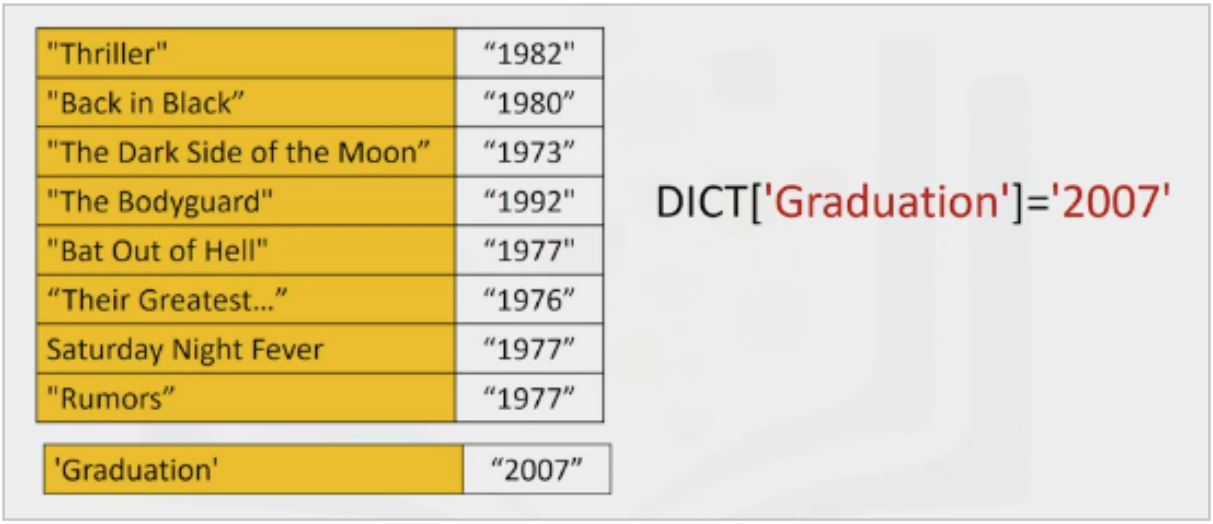
```
Dict={"A":1,"B":"2","C":[3,3,3],"D":(4,4,4),'E':5,'F':6}
```

The **key is used to look at the value**.



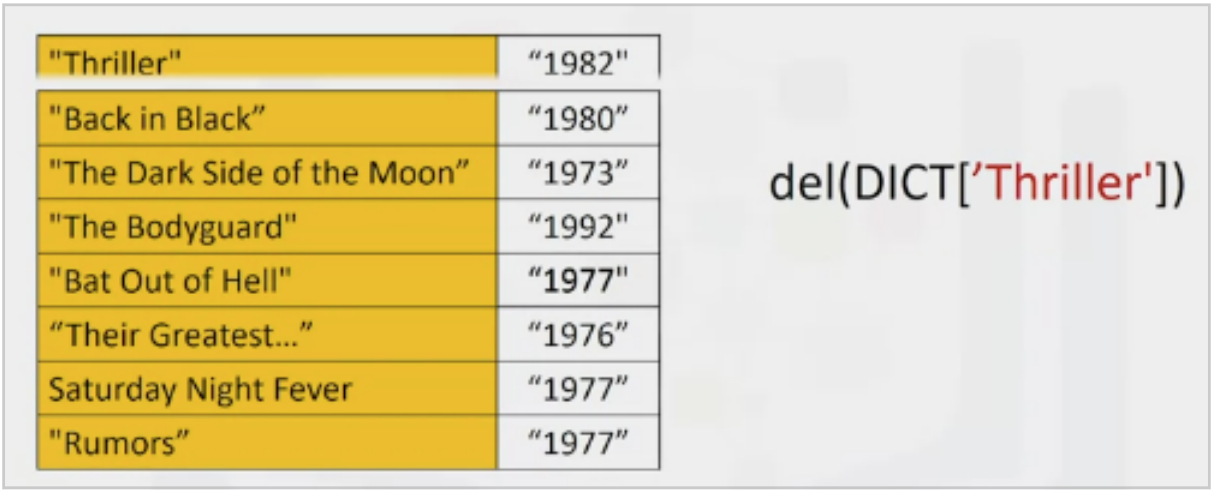
We use square brackets. The argument is the key. This outputs the value. Using the key of "Back in Black," this returns the value of 1980. The key, "The Dark Side Of The Moon," gives us the value of 1973. Using the key,"The bodyguard," gives us the value 1992 and so on.

We can add a new entry to the dictionary as follows.



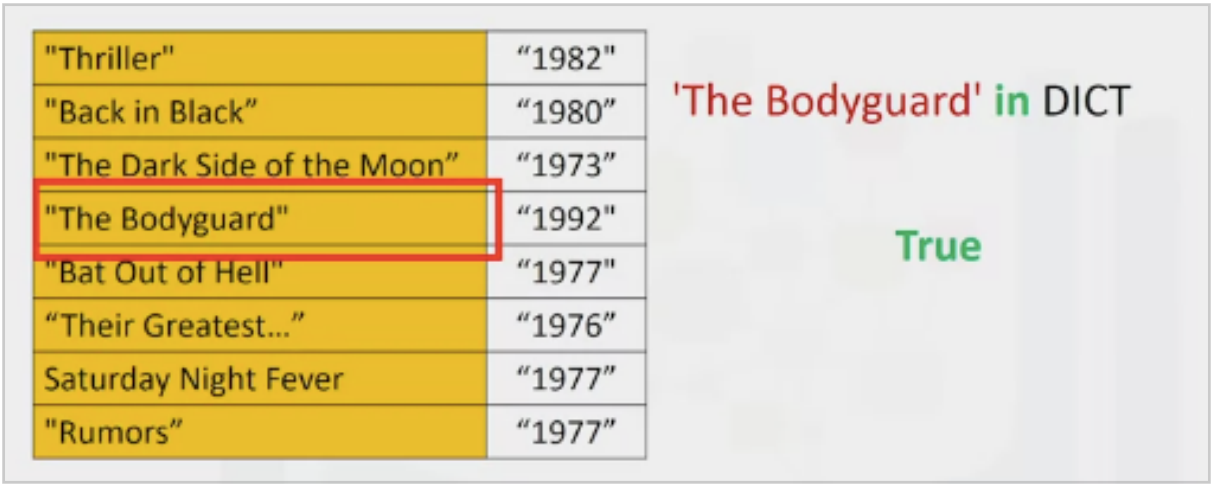
This will add the value 2007 with a new key called "Graduation."

We can delete an entry as follows.



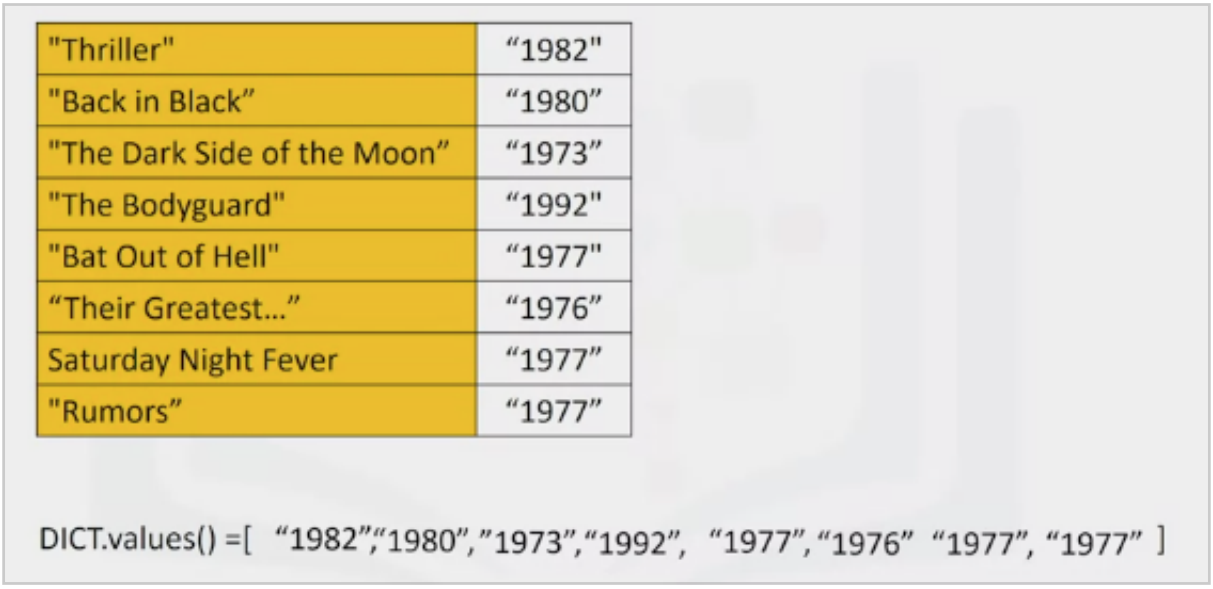
This gets rid of the key "Thriller" and it's value.

We can verify if an element is in the dictionary using the "in" command as follows: The command checks the keys.



If they are in the dictionary, they return a true. If we try the same command with a key that is not in the dictionary, we get a false. In order to see all the keys in the dictionary, we can use the method keys to get the keys.

The output is a list-like object with all the keys. In the same way, we can obtain the values using the method values



Lab -Dictionaries

Objectives

After completing this lab you will be able to:

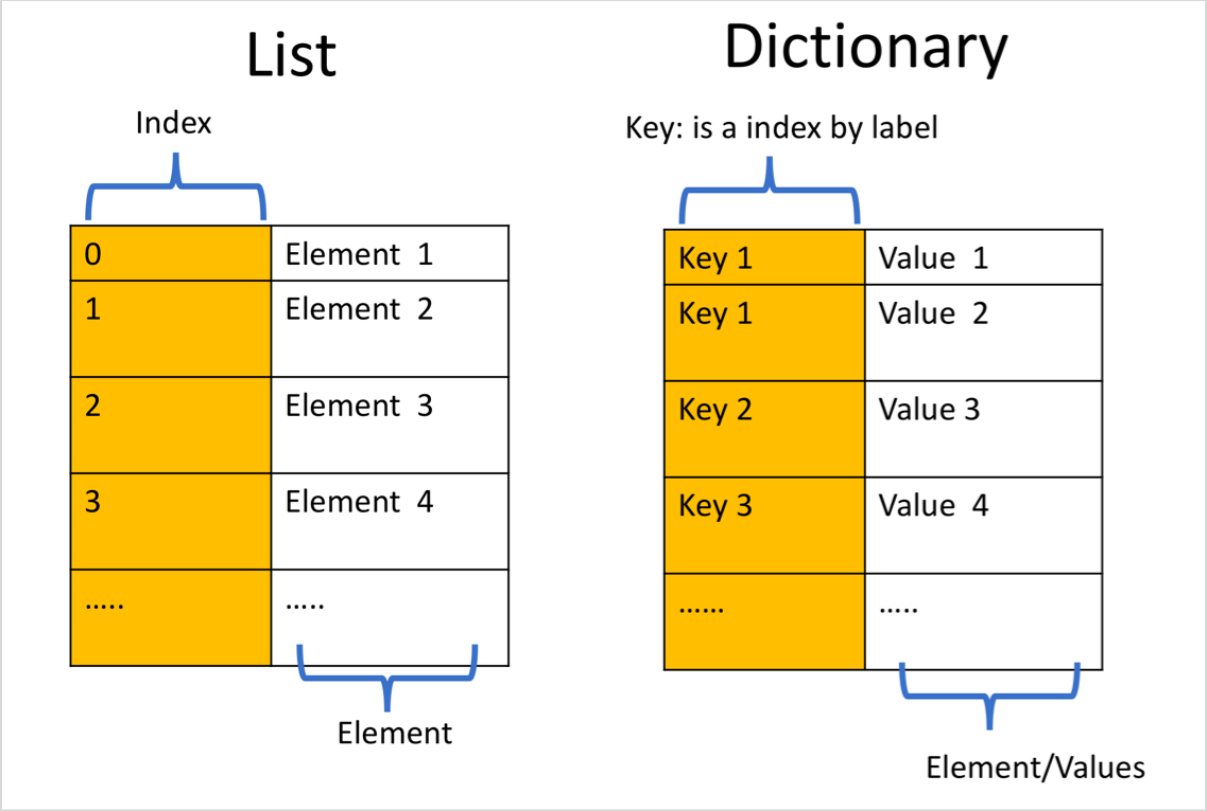
- Work with and perform operations on dictionaries in Python

Table of Contents

Dictionaries

What are Dictionaries?

A dictionary consists of keys and values. It is helpful to compare a dictionary to a list. Instead of being indexed numerically like a list, dictionaries have keys. These keys are the keys that are used to access values within a dictionary.



An example of a Dictionary <code>Dict</code>:

```
# Create the dictionary
Dict = {"key1": 1, "key2": "2", "key3": [3, 3, 3], "key4": (4, 4, 4), ('key5'):
5, (0, 1): 6}
Dict
```

```
{'key1': 1,
 'key2': '2',
 'key3': [3, 3, 3],
 'key4': (4, 4, 4),
 'key5': 5,
 (0, 1): 6}
```

The keys can be strings:

```
# Access to the value by the key
Dict["key1"]
```

1

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

```
# Access to the value by the key
Dict[(0, 1)]
```

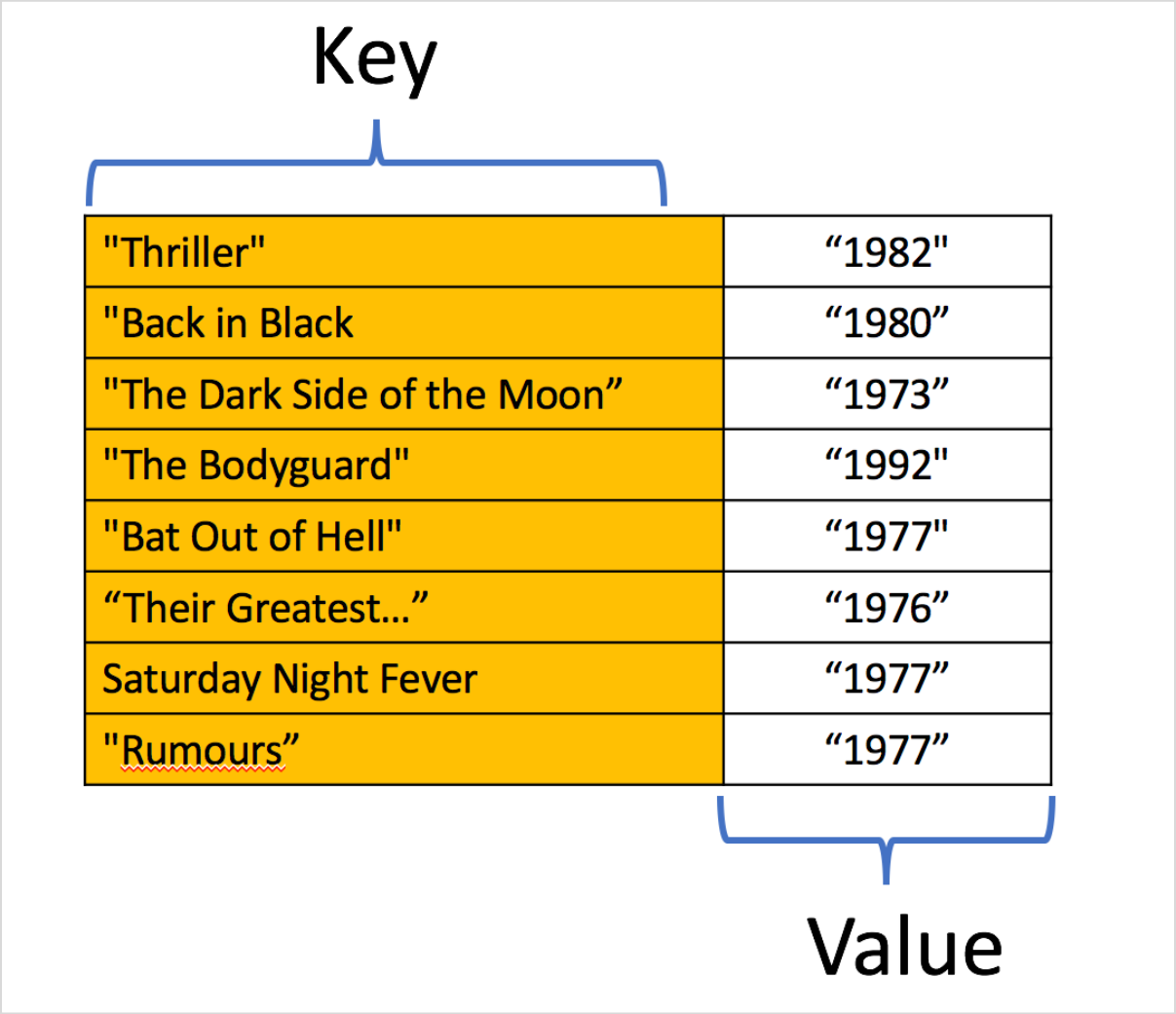
6

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 `{}`.

```
# 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
```

{'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'}

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:

```
# Get value by keys
release_year_dict['Thriller']
```

'1982'

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

```
# Get value by key
release_year_dict['The Bodyguard']
```

'1992'

"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 us retrieve the keys of the dictionary using the method `keys()`:

```
# Get all the keys in dictionary
release_year_dict.keys()
```

```
dict_keys(['Thriller', 'Back in Black', 'The Dark Side of the Moon', 'The Bodyguard', 'Bat Out of Hell', 'Their Greatest Hits (1971–1975)', 'Saturday Night Fever', 'Rumours'])
```

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

```
# Get all the values in dictionary
release_year_dict.values()
```

dict_values(['1982', '1980', '1973', '1992', '1977', '1976', '1977', '1977'])

We can add an entry:

```
# Append value with key into dictionary
release_year_dict['Graduation'] = '2007'
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',
 'Graduation': '2007'}
```

We can delete an entry:

```
# Delete entries by key
del(release_year_dict['Thriller'])
del(release_year_dict['Graduation'])
release_year_dict
```

```
{'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'}
```

We can verify if an element is in the dictionary:

```
# Verify the key is in the dictionary
'The Bodyguard' in release_year_dict
```

True

Quiz on Dictionaries

You will need this dictionary for the next two questions:

```
# Question sample dictionary
soundtrack_dic = {"The Bodyguard":"1992", "Saturday Night Fever":"1977"}
soundtrack_dic
```

{'The Bodyguard': '1992', 'Saturday Night Fever': '1977'}

a) In the dictionary `soundtrack_dic` what are the keys ?

```
# Write your code below and press Shift+Enter to execute
soundtrack_dic.keys()
```

dict_keys(['The Bodyguard', 'Saturday Night Fever'])

b) In the dictionary `soundtrack_dic` what are the values ?

```
# Write your code below and press Shift+Enter to execute
soundtrack_dic.values()
```

dict_values(['1992', '1977'])

You will need this dictionary for the following questions:

The Albums Back in Black, The Bodyguard and Thriller have the following music recording

sales in millions 50, 50 and 65 respectively:

a) Create a dictionary `album_sales_dict` where the keys are the album name and the sales in millions are the values.

```
# Write your code below and press Shift+Enter to execute
album_sales_dict={'Back in Black':50,'The Bodyguard':50, 'Thriller':60}
album_sales_dict
```

{'Back in Black': 50, 'The Bodyguard': 50, 'Thriller': 60}

b) Use the dictionary to find the total sales of Thriller:

```
# Write your code below and press Shift+Enter to execute
album_sales_dict['Thriller']
```

60

c) Find the names of the albums from the dictionary using the method `keys()`:

```
# Write your code below and press Shift+Enter to execute
album_sales_dict.keys()
```

dict_keys(['Back in Black', 'The Bodyguard', 'Thriller'])

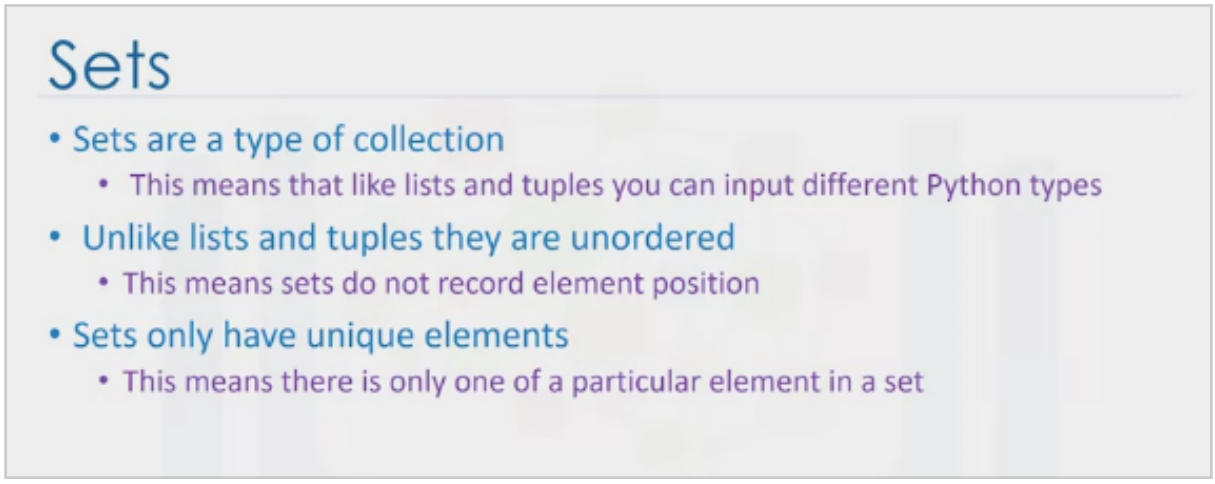
d) Find the values of the recording sales from the dictionary using the method `values`:

```
# Write your code below and press Shift+Enter to execute
album_sales_dict.values()
```

dict_values([50, 50, 60])

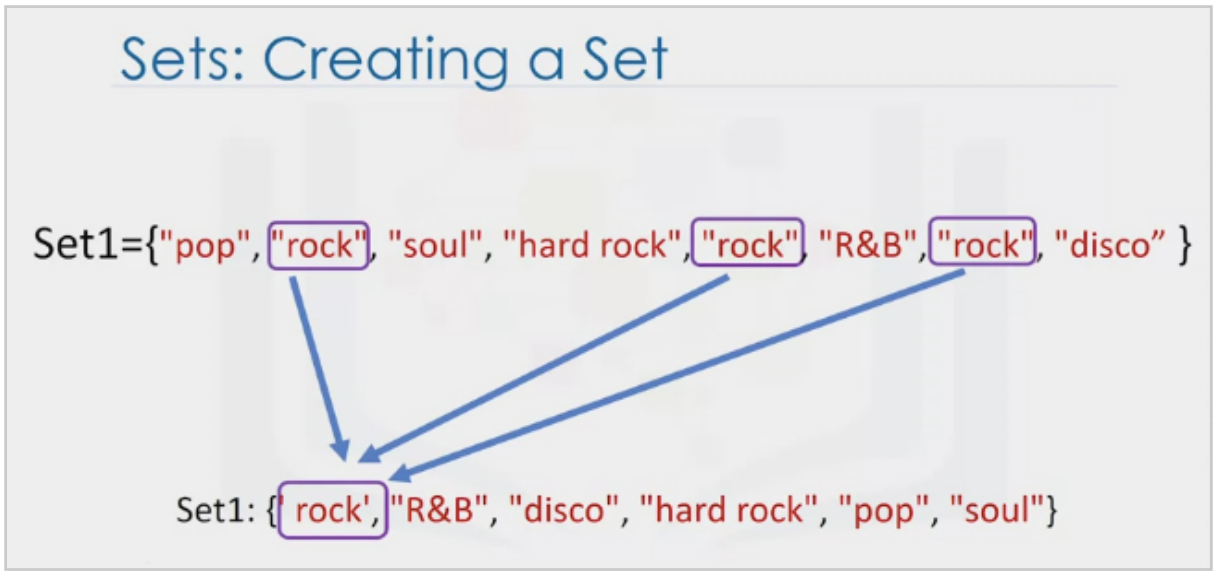
Sets Theory

Sets are a type of collection. This means that like lists and tuples, you can input different Python types. Unlike lists and tuples, **they are unordered**. This means sets do not record element position. **Sets only have unique elements**.



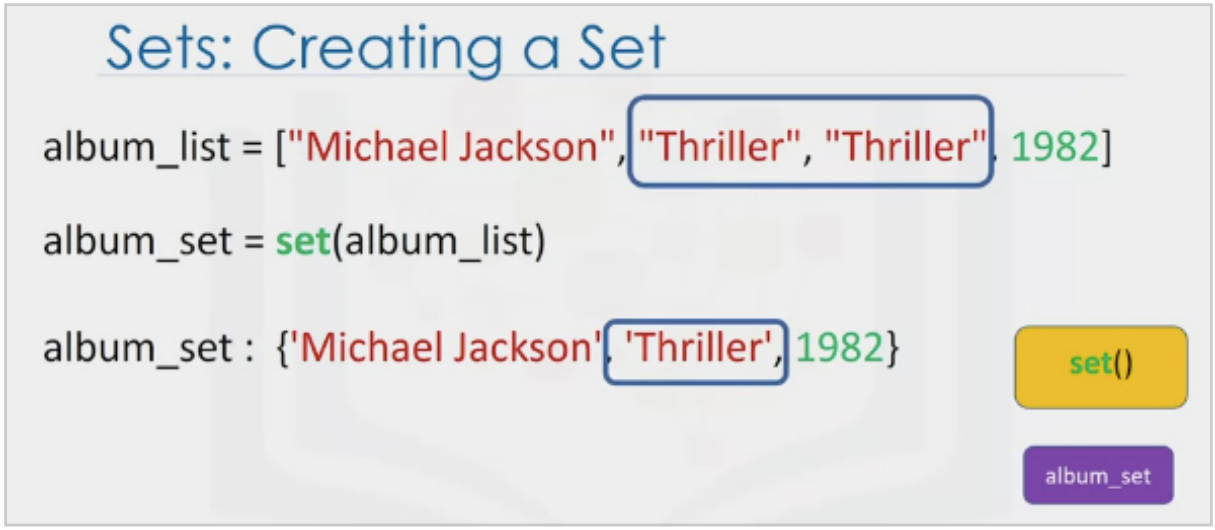
This means there is only one of a particular element in a set.

To define a set, you use curly brackets. You place the elements of a set within the curly brackets.



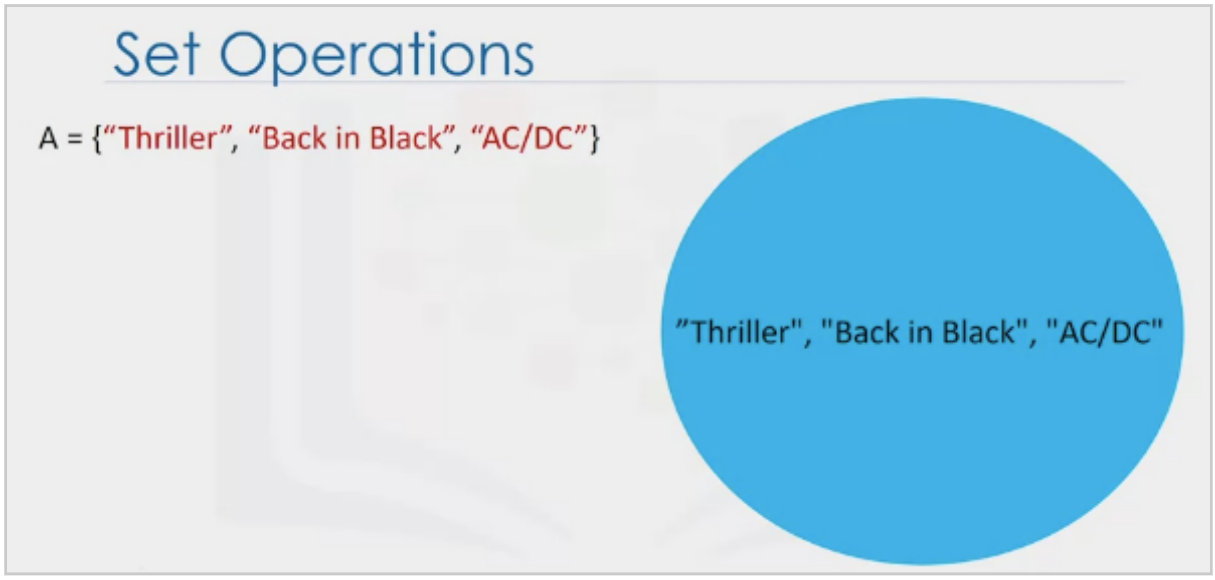
You notice there are duplicate items. **When the actual set is created, duplicate items will not be present**.

You can convert a list to a set by using the function set, this is called type casting. You simply use the list as the input to the function set. The result will be a list converted to a set. Let's go over an example.

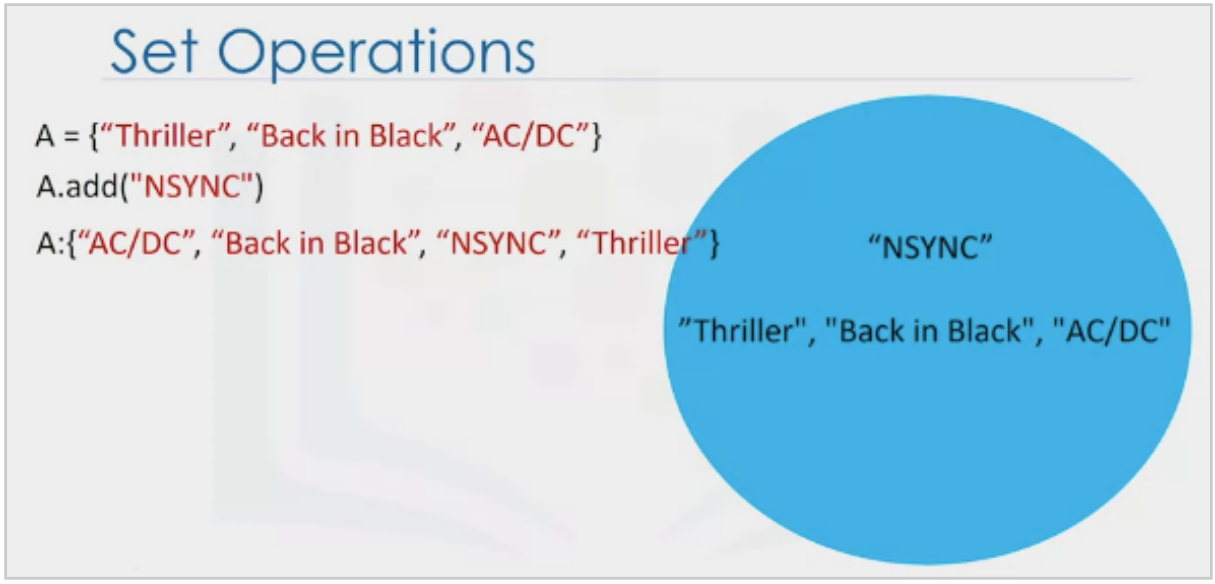


We start off with a list. We input the list to the function set. The function set returns a set. **Notice how there are no duplicate elements**.

Let's go over set operations. These could be used to change the set. Consider the set A. Let's represent this set with a circle.

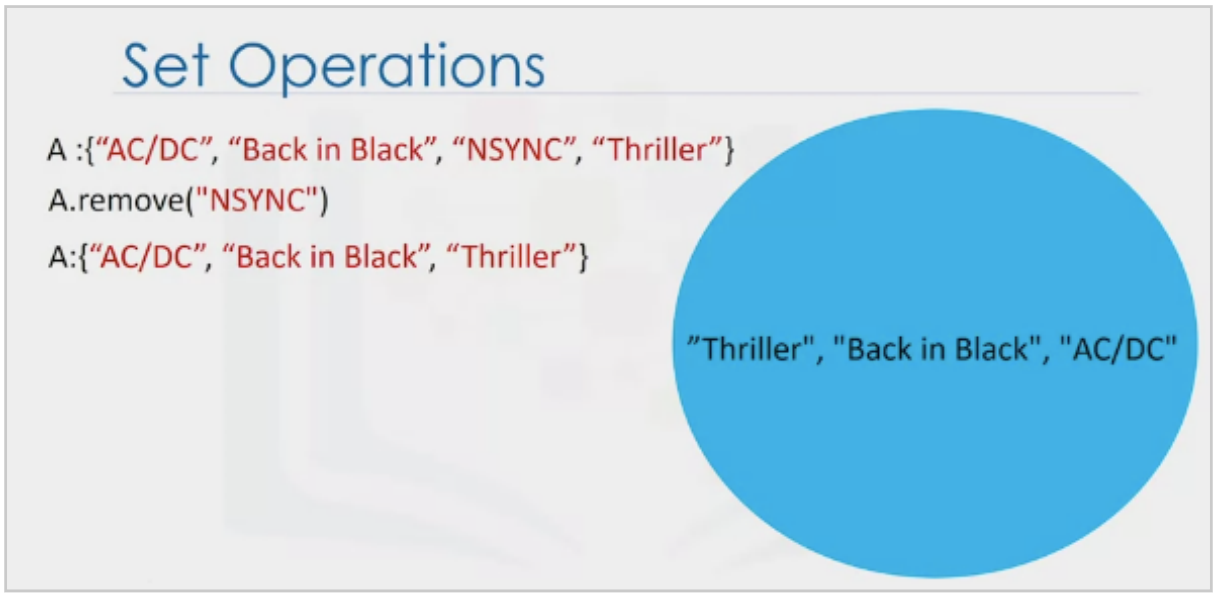


If you are familiar with sets, this could be part of a **Venn Diagram**. A venn diagram is a tool that uses shapes usually to represent sets. We can add an item to a set using the add-method. We just put the set name followed by a dot, then the add-method. The argument is the new element of the set we would like to add, in this case, NSYNC.



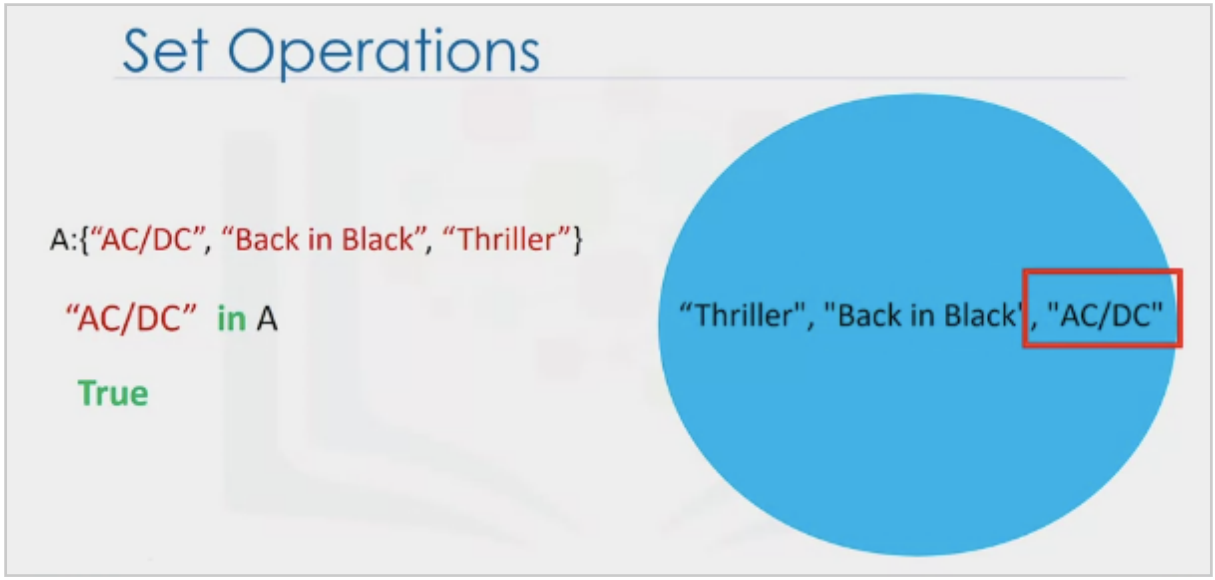
The set A now has in NSYNC as an item. If we add the same item twice, nothing will happen as there can be no duplicates in a set. Let’s say we would like to remove NSYNC from set A.

We can also remove an item from a set using the remove-method. We just put the set name followed by a dot, then the remove-method.

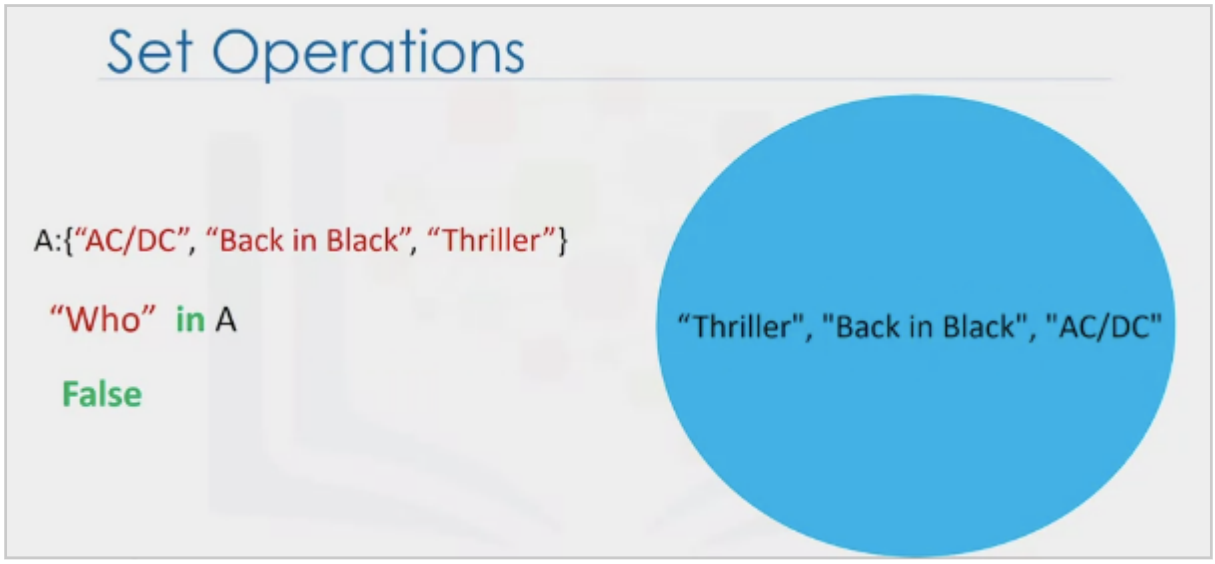


The argument is the element of the set we would like to remove, in this case, NSYNC. After the remove-method is applied to the set, set A does not contain the item NSYNC. You can use this method for any item in the set.

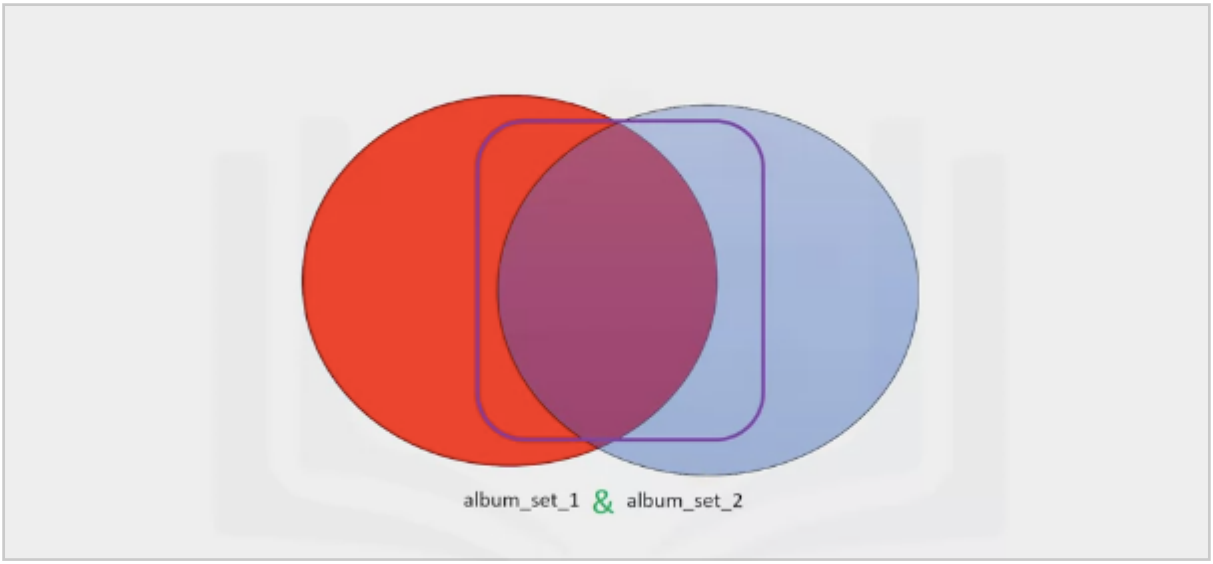
We can verify if an element is in the set using the `in` command as follows.



The command checks that the item, in this case AC/DC, is in the set. If the item is in the set, it returns true. If we look for an item that is not in the set, in this case for the item Who, adds the item is not in the set, we will get a false.



There are lots of useful mathematical operations we can do between sets. Let's define the set album set one. We can represent it using a red circle or venn diagram. Similarly, we can define the set album set two. We can also represent it using a blue circle or venn diagram. The **intersection** of two sets is a new set containing elements which are in both of those sets.

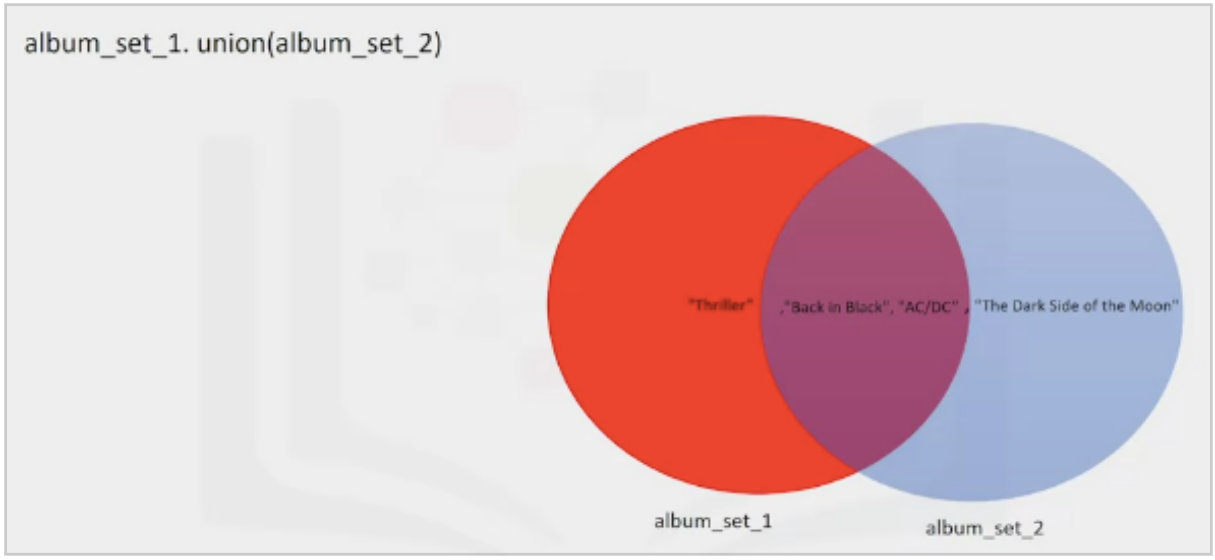


It's helpful to use venn diagrams. The two circles that represent the sets combine, the overlap, represents the new set. As the overlap is comprised with the red circle and blue circle, we define the intersection in terms of and. In Python, we use an `&` to find the intersection of the two sets. If we overlay the values of the set over the circle placing the common elements in the overlapping area, we see the correspondence.

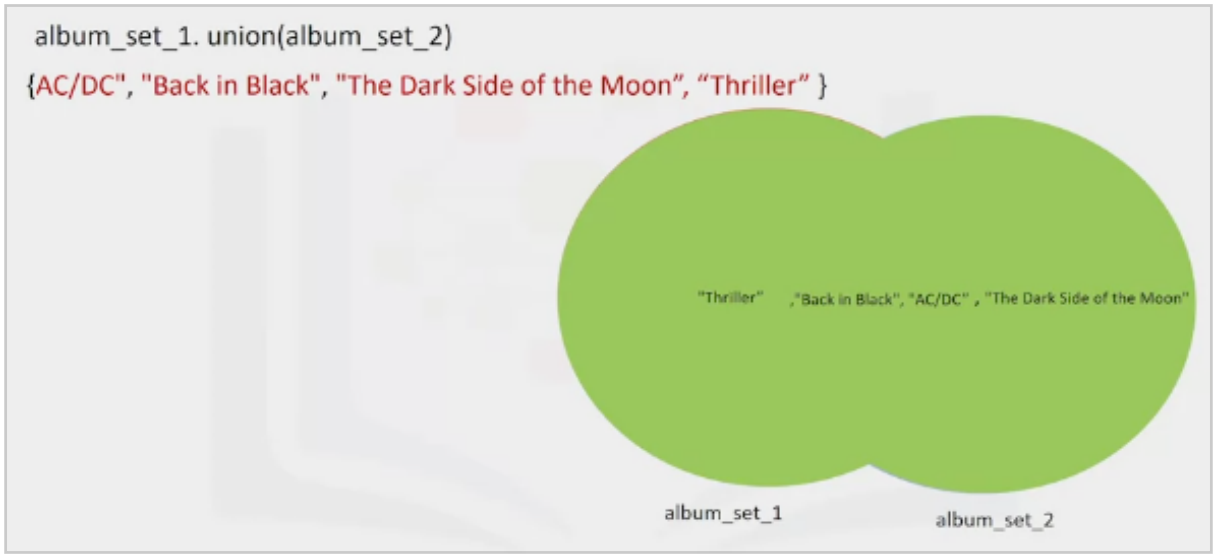
```
album_set_1 ={"AC/DC"," Back in Black ","Thriller }
album_set_2={"AC/DC", "Back in Black","The Dark Side of the Moon"}
album_set_3=album_set_1 & album_set_2
album_set_3: {"AC/DC"," Back in Black "}
```

After applying the intersection operation, all the items that are not in both sets disappear. In Python, we simply just place the ampersand between the two sets. We see that both AC/DC and Back in Black are in both sets. The result is a new set album: **set three containing all the elements in both albums set one and album set two**.

The **union** of two sets is the new set of elements which contain all the items in both sets.

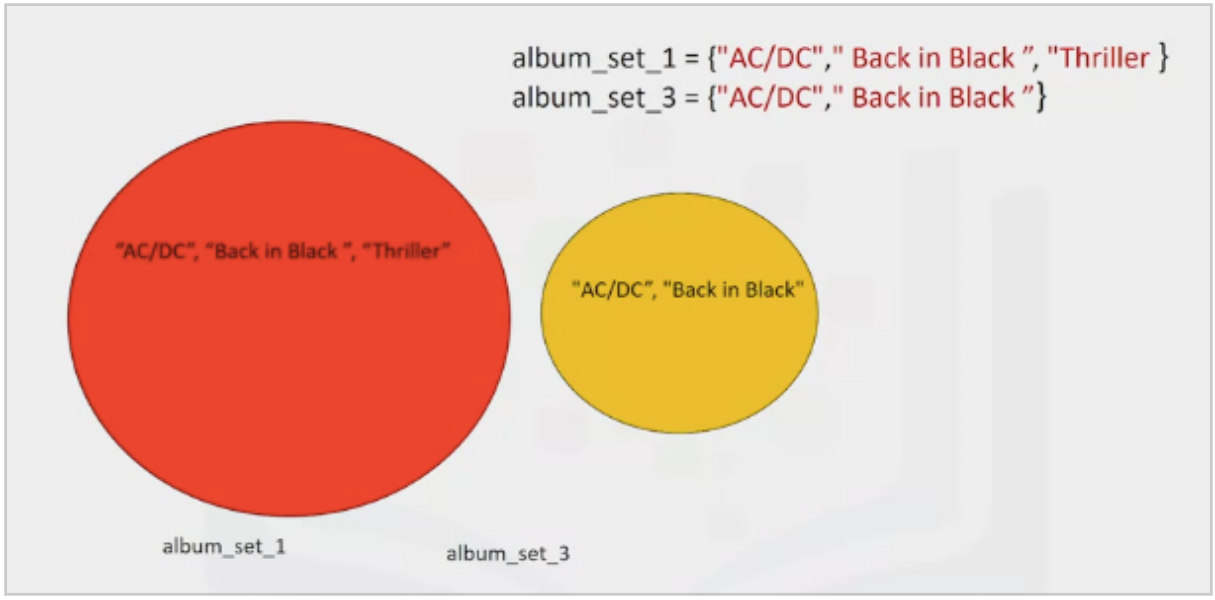


We can find the `union` of the sets album set one and album set two as follows. The result is a new set that has all the elements of album set one and album set two.



This new set is represented in green.

Consider the new album set-album set three. The set contains the elements AC/DC and Back in Black. We can represent this with a Venn diagram, as all the elements and album set three are in album set one.



The circle representing album set one encapsulates the circle representing album set three. We can check if a set is a subset using the `issubset` method.

As album set three is a subset of the album set one, the result is true.

There is a lot more you can do with sets.

Lab - Sets

Objectives

After completing this lab you will be able to:

- Work with sets in Python, including operations and logic operations.

Table of Contents

- Quiz on Sets

Sets

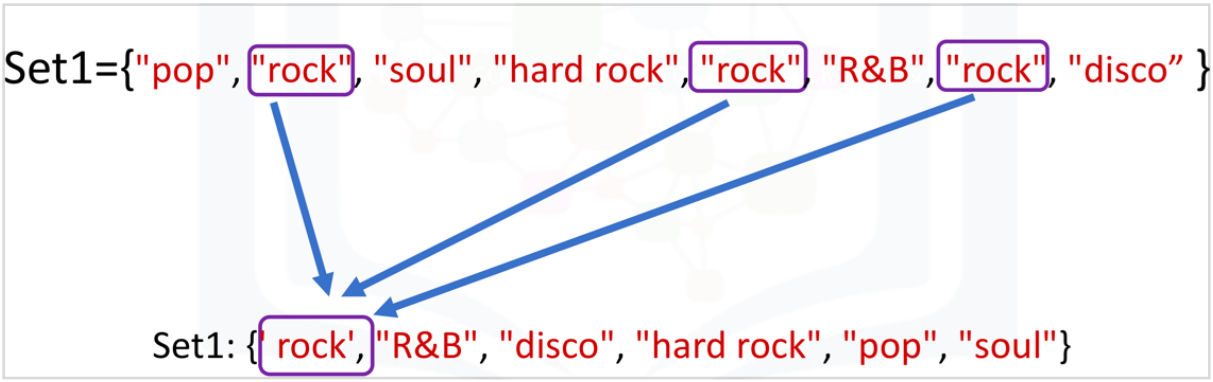
Set Content

A set is a unique collection of objects in Python. You can denote a set with a pair of curly brackets `{}`. Python will automatically remove duplicate items:

```
# Create a set
set1 = {"pop", "rock", "soul", "hard rock", "rock", "R&B", "rock", "disco"}
set1
```

```
{'R&B', 'disco', 'hard rock', 'pop', 'rock', 'soul'}
```

The process of mapping is illustrated in the figure:



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

```
# 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
```

```
    {'00:42:19',
     10.0,
     1982,
     '30-Nov-82',
```

```
     46.0,
     65,
     'Michael Jackson',
     None,
     'Pop, Rock, R&B',
     'Thriller'}
```

Now let us create a set of genres:

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

```
    {'R&B',
     'disco',
     'folk rock',
     'hard rock',
     'pop',
     'progressive rock',
     'rock',
     'soft rock',
     'soul'}
```

Set Operations

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

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

{'AC/DC', 'Back in Black', 'Thriller'}

We can add an element to a set using the <code>add()</code> method:

```
# Add element to set
A.add("NSYNC")
A
```

{'AC/DC', 'Back in Black', 'NSYNC', 'Thriller'}

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

```
# Try to add duplicate element to the set
A.add("NSYNC")
A
```

{'AC/DC', 'Back in Black', 'NSYNC', 'Thriller'}

We can remove an item from a set using the <code>remove</code> method:

```
# Remove the element from set
A.remove("NSYNC")
A
```

{'AC/DC', 'Back in Black', 'Thriller'}

We can verify if an element is in the set using the <code>in</code> command:

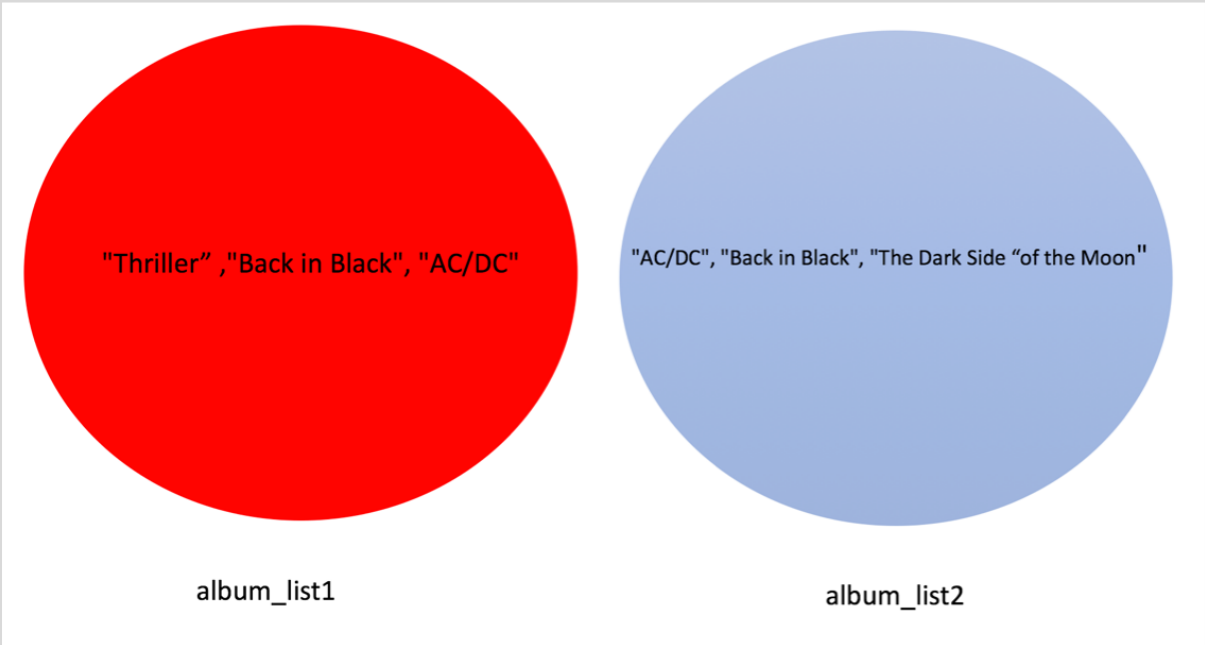
```
# Verify if the element is in the set
"AC/DC" in A
```

True

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:

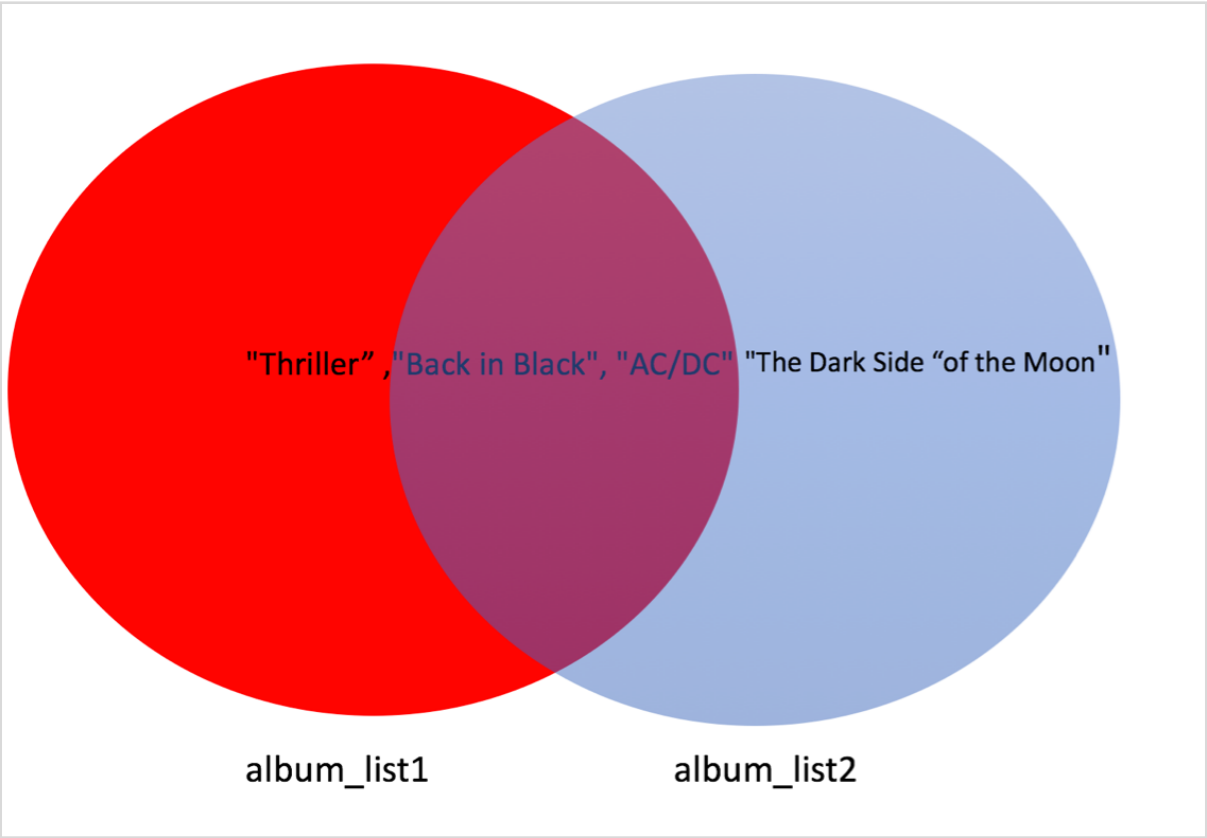
```
# Sample Sets
album_set1 = set(["Thriller", 'AC/DC', 'Back in Black'])
album_set2 = set([ "AC/DC", "Back in Black", "The Dark Side of the Moon"])
```




```
# Print two sets
album_set1, album_set2
```

```
    ({'AC/DC', 'Back in Black', 'Thriller'},
     {'AC/DC', 'Back in Black', 'The Dark Side of the Moon'})
```

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 `&`:

```
# Find the intersections
intersection = album_set1 & album_set2
intersection
```

{'AC/DC', 'Back in Black'}

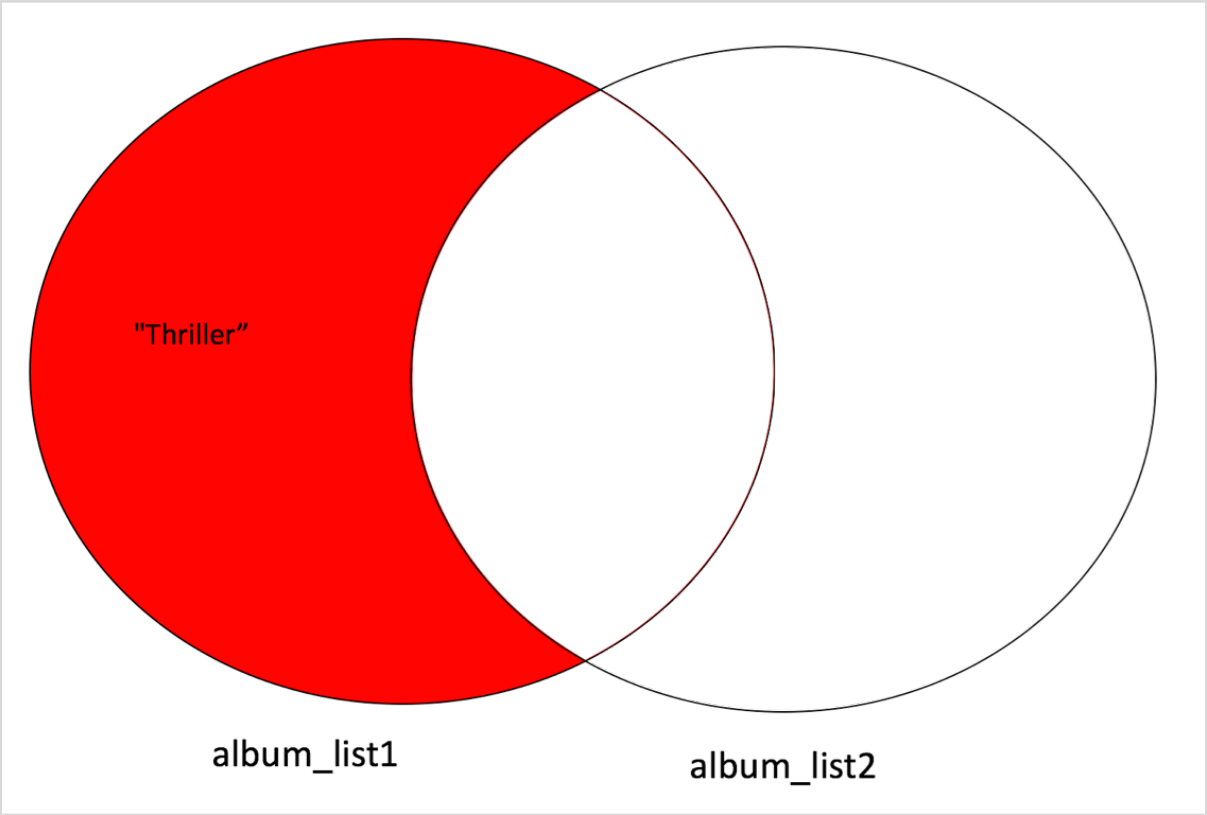
You can find all the elements that are only contained in `album_set1` using the `difference` method:

```
# Find the difference in set1 but not set2
album_set1.difference(album_set2)
```

{'Thriller'}

You only need to consider elements in `album_set1`; all the elements in `album_set2`,

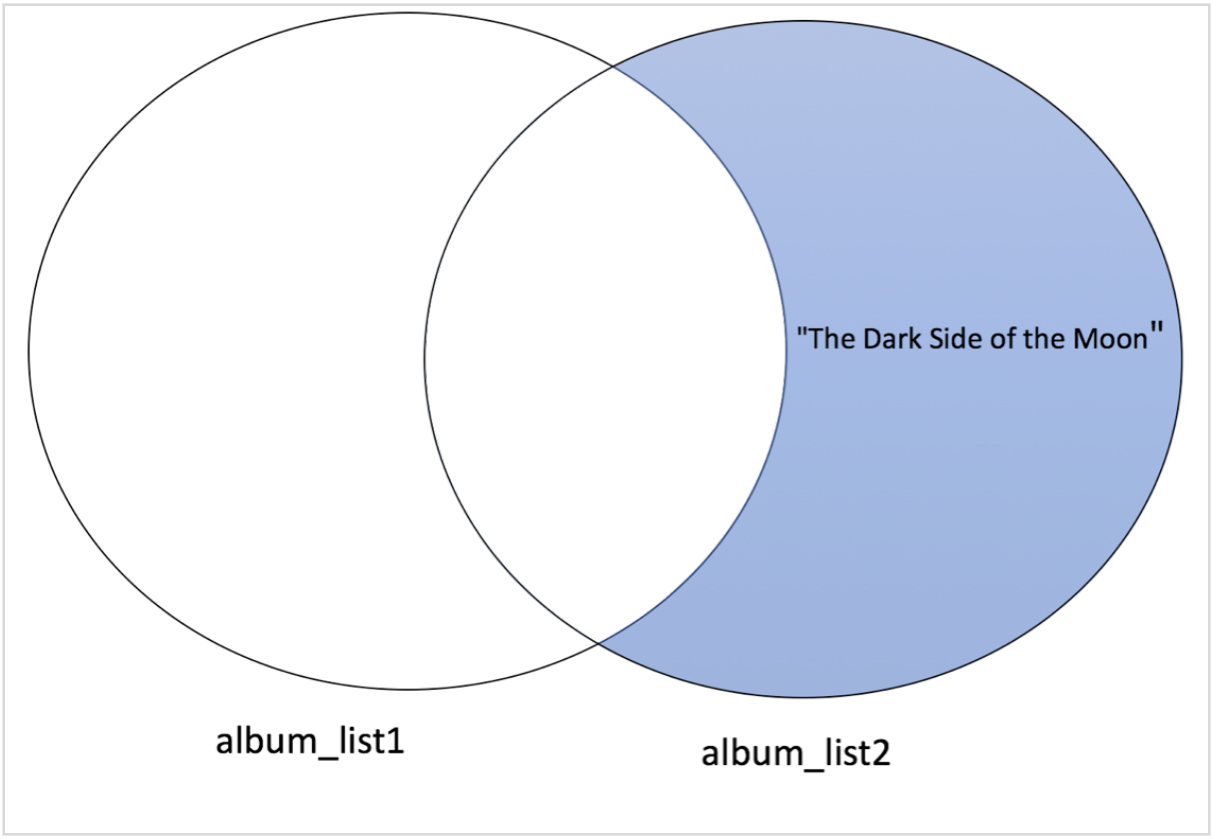
including the intersection, are not included.



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

```
album_set2.difference(album_set1)
```

{'The Dark Side of the Moon'}

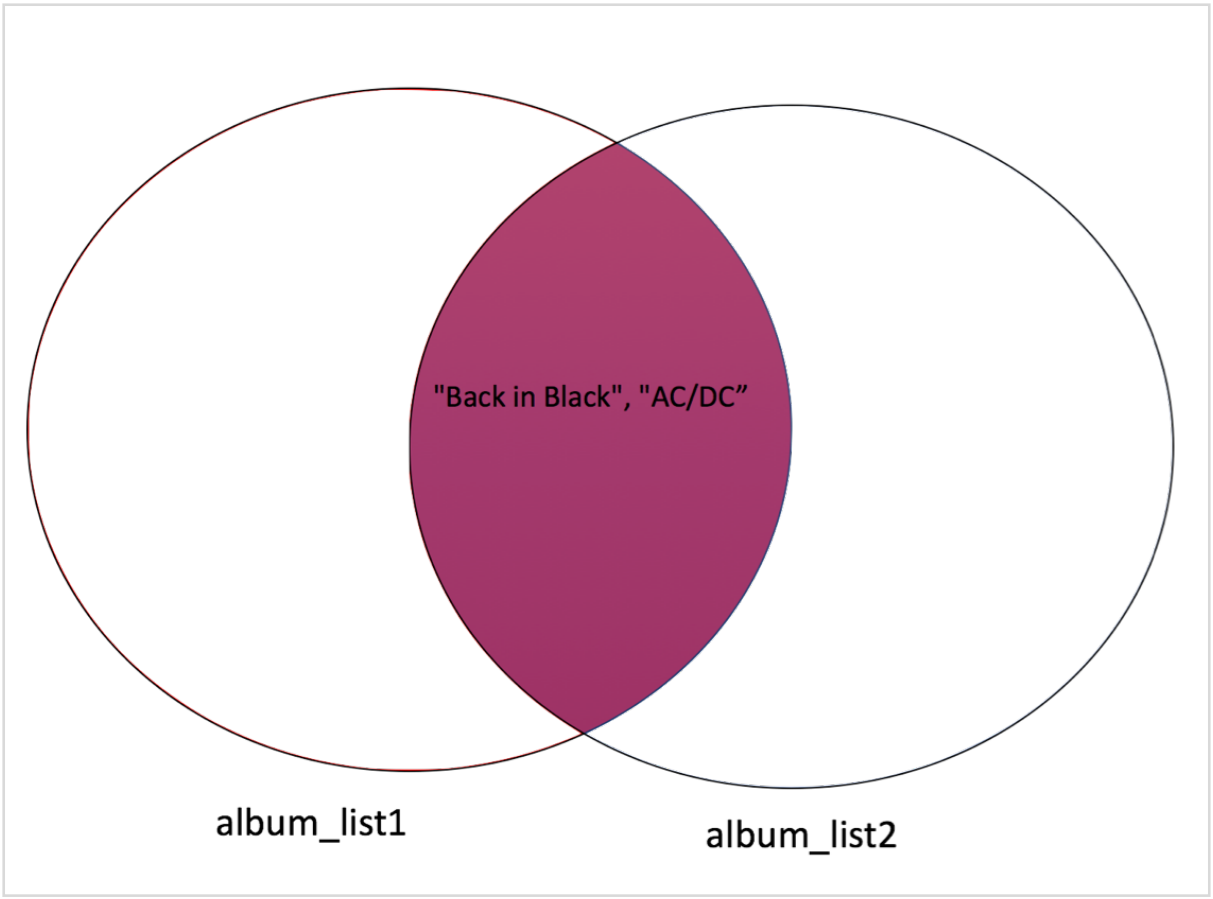


You can also find the intersection of `album_list1` and `album_list2`, using the `intersection` method:

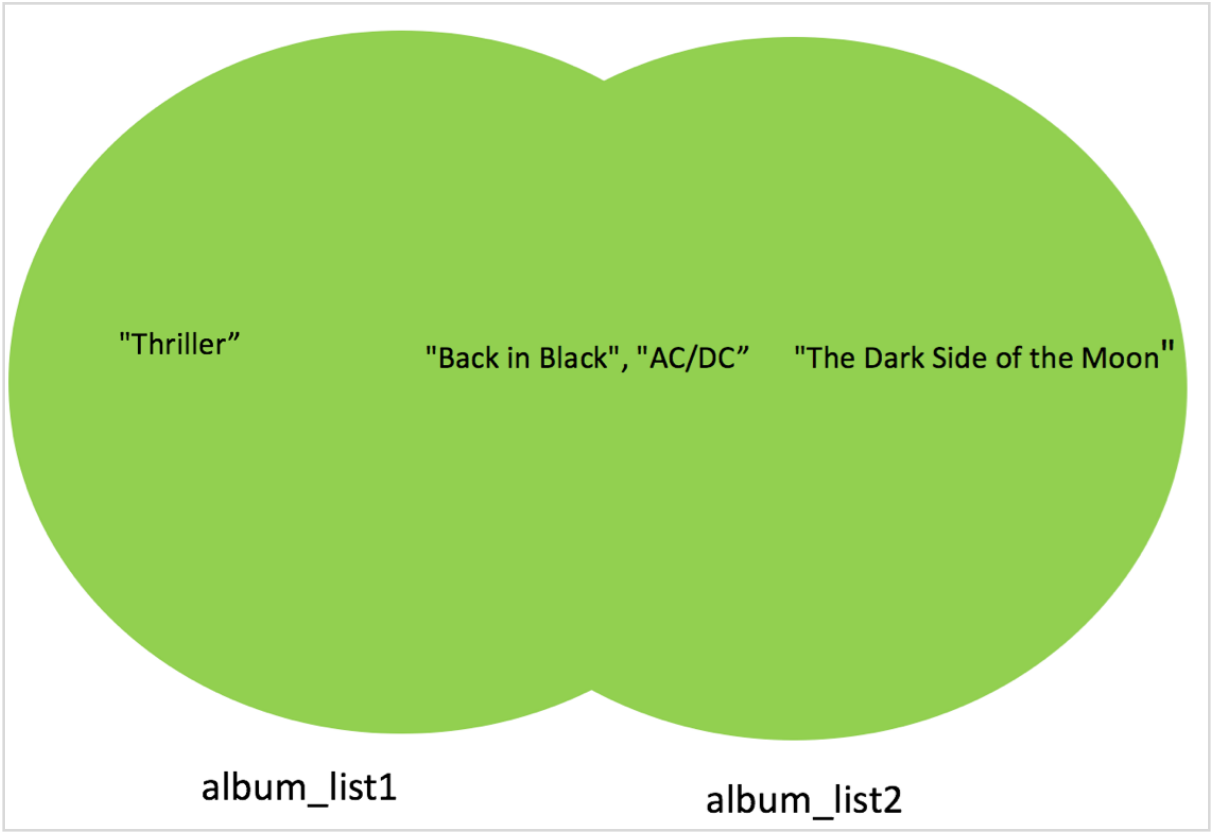
```
# Use intersection method to find the intersection of album_list1 and
album_list2
album_set1.intersection(album_set2)
```

{'AC/DC', 'Back in Black'}

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:

```
# Find the union of two sets
album_set1.union(album_set2)
```

{'AC/DC', 'Back in Black', 'The Dark Side of the Moon', 'Thriller'}

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

```
# Check if superset
set(album_set1).issuperset(album_set2)
```

False

```
# Check if subset
set(album_set2).issubset(album_set1)
```

False

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

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

True

```
# Check if superset

album_set1.issuperset({"Back in Black", "AC/DC"})
```

True

Quiz on Sets

Convert the list `['rap','house','electronic music', 'rap']` to a set:

```
# Write your code below and press Shift+Enter to execute
list=['rap','house','electronic music', 'rap']
test=set(list)
test
```

{'electronic music', 'house', 'rap'}

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

```
# Write your code below and press Shift+Enter to execute
A = [1, 2, 2, 1]
B = set([1, 2, 2, 1])
print("Sum of A is", sum(A))
print("Sum of B is", sum(B))
# Oppure
sum(A) == sum(B)
```

```
Sum of A is 6
Sum of B is 3
False
```

Create a new set `album_set3` that is the union of <code>album_set1</code> and `album_set2`:

```
# 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"])
album_set3 = album_set1.union(album_set2)
```

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

```
# Write your code below and press Shift+Enter to execute
album_set1.issubset(album_set3)
```

True