
LEARN
PYTHON
- DATA TYPES -

Sets in Python

1.1 Objectives

After completing this lab you will be able to:

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

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:

[1]:
```
# Create a set

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

[1]:
```
{'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:

[2]:
```
# 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
```

[2]:
```
{'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:

[3]:
```
# Convert list to set

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

[3]:
```
{'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:

[4]:
```
# Sample set

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

[4]:
```
{'AC/DC', 'Back in Black', 'Thriller'}
```

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

[5]:
```
# Add element to set

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

[5]:
```
{'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:

[6]:
```
# Try to add duplicate element to the set

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

[6]:
```
{'AC/DC', 'Back in Black', 'NSYNC', 'Thriller'}
```

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

[7]:
```
# Remove the element from set

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

[7]:
```
{'AC/DC', 'Back in Black', 'Thriller'}
```

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

[8]:
```
# Verify if the element is in the set

"AC/DC" in A
```

[8]:
```
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:

[9]:
```
# 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"])
```

[10]:
```
# Print two sets

album_set1, album_set2
```

[10]:
```
({'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 &:

[11]:
```
# Find the intersections

intersection = album_set1 & album_set2
intersection
```

[11]:
```
{'AC/DC', 'Back in Black'}
```

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

[12]:
```
# Find the difference in set1 but not set2

album_set1.difference(album_set2)
```

[12]:
```
{'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:

[13]:
```
album_set2.difference(album_set1)
```

[13]:
```
{'The Dark Side of the Moon'}
```

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

[14]:
```
# Use intersection method to find the intersection of album_list1 and␣
 ↪album_list2

album_set1.intersection(album_set2)
```

[14]:
```
{'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:

[15]:
```
# Find the union of two sets

union = album_set1.union(album_set2)
union
```

[15]:
```
{'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:

[16]:
```
# Check if superset

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

[16]:
```
False
```

[17]:
```
# Check if subset

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

[17]:
```
False
```

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

[18]:
```
# Check if subset

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

[18]:
```
True
```

[19]:
```
# Check if superset

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

[19]:
```
True
```

Quiz on Sets

Convert the list [‘rap’,‘house’,‘electronic music’, ‘rap’] to a set:

[20]:
```
# Write your code below and press Shift+Enter to execute
list = ['rap','house','electronic music', 'rap']
set_1 = set(list)
set_1
```

[20]:
```
{'electronic music', 'house', 'rap'}
```

```
set(['rap','house','electronic music','rap'])
```

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

[21]:
```
# Write your code below and press Shift+Enter to execute
A = [1, 2, 2, 1]
B = set([1, 2, 2, 1])
```

```
sum_A = sum(A)
sum_B = sum(A)
sum_A == sum_B
```

[21]: True

```
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:

[22]:
```
# 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)
album_set3
```

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

```
album_set3 = album_set1.union(album_set2)
album_set3
```

Find out if album_set1 is a subset of album_set3:

[23]:
```
# Write your code below and press Shift+Enter to execute
album_set1.issubset(album_set3)
```

[23]: True

```
album_set1.issubset(album_set3)
```

Cast a List to a Set

Cast the following list to a set:

[24]:
```
A_List = ['A','B','C','A','B','C']
A_set = set(A_List)
A_set
```

[24]: {'A', 'B', 'C'}

```
set(['A','B','C','A','B','C'])
```

Add an Element to the Set

Add the string 'D' to the set S.

[25]:
```
S={'A','B','C'}
S.add('D')
S
```

[25]:
```
{'A', 'B', 'C', 'D'}
```

```
S.add('D')
S
```

Intersection of Sets

Find the intersection of set A and B.

[26]:
```
A={1,2,3,4,5}
B={1,3,9, 12}
A&B
```

[26]:
```
{1, 3}
```

Lists in Python

1.1 Objectives

After completing this lab you will be able to:

- Perform list operations in Python, including indexing, list manipulation, 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

[1]:
```
# Create a list

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

[1]:
```
['Michael Jackson', 10.1, 1982]
```

We can use negative and regular indexing with a list:

[2]:
```
# 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:

[3]:
```
# Sample List

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

[3]:
```
['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:

[4]:
```
# Sample List

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

[4]:
```
['Michael Jackson', 10.1, 1982, 'MJ', 1]
```

[5]:
```
# List slicing

L[3:5]
```

[5]:
```
['MJ', 1]
```

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

[6]:
```
# Use extend to add elements to list

L = [ "Michael Jackson", 10.2]
L.extend(['pop', 10])
L
```

[6]:
```
['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:

[7]:
```
# Use append to add elements to list

L = [ "Michael Jackson", 10.2]
L.append(['pop', 10])
L
```

[7]:
```
['Michael Jackson', 10.2, ['pop', 10]]
```

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:

[ ]:
```
# Use extend to add elements to list

L = [ "Michael Jackson", 10.2]
L.extend(['pop', 10])
L
```

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

[8]:
```
# Use append to add elements to list

L.append(['a','b'])
L
```

[8]:
```
['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:

[9]:
```
# 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:

[10]:
```
# 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:

[11]:
```
# Split the string, default is by space

'hard rock'.split()
```

[11]:
```
['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:

[12]:
```
# Split the string by comma

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

[12]:
```
['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:

[13]:
```
# 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”:

[14]:
```
# 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
```

This is demonstrated in the following figure:

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

[15]:
```
# Clone (clone by value) the list A

B = A[:]
B
```

[15]:
```
['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:

[16]:
```
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.

[18]:
```
# Write your code below and press Shift+Enter to execute
a_list = [1, 'hello', [1,2,3], True]
a_list
```

[18]:
```
[1, 'hello', [1, 2, 3], True]
```

```
a_list = [1, 'hello', [1, 2, 3] , True]
a_list
```

Find the value stored at index 1 of a_list.

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

```
a_list[1]
```

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]
```

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
```

The last exercise!

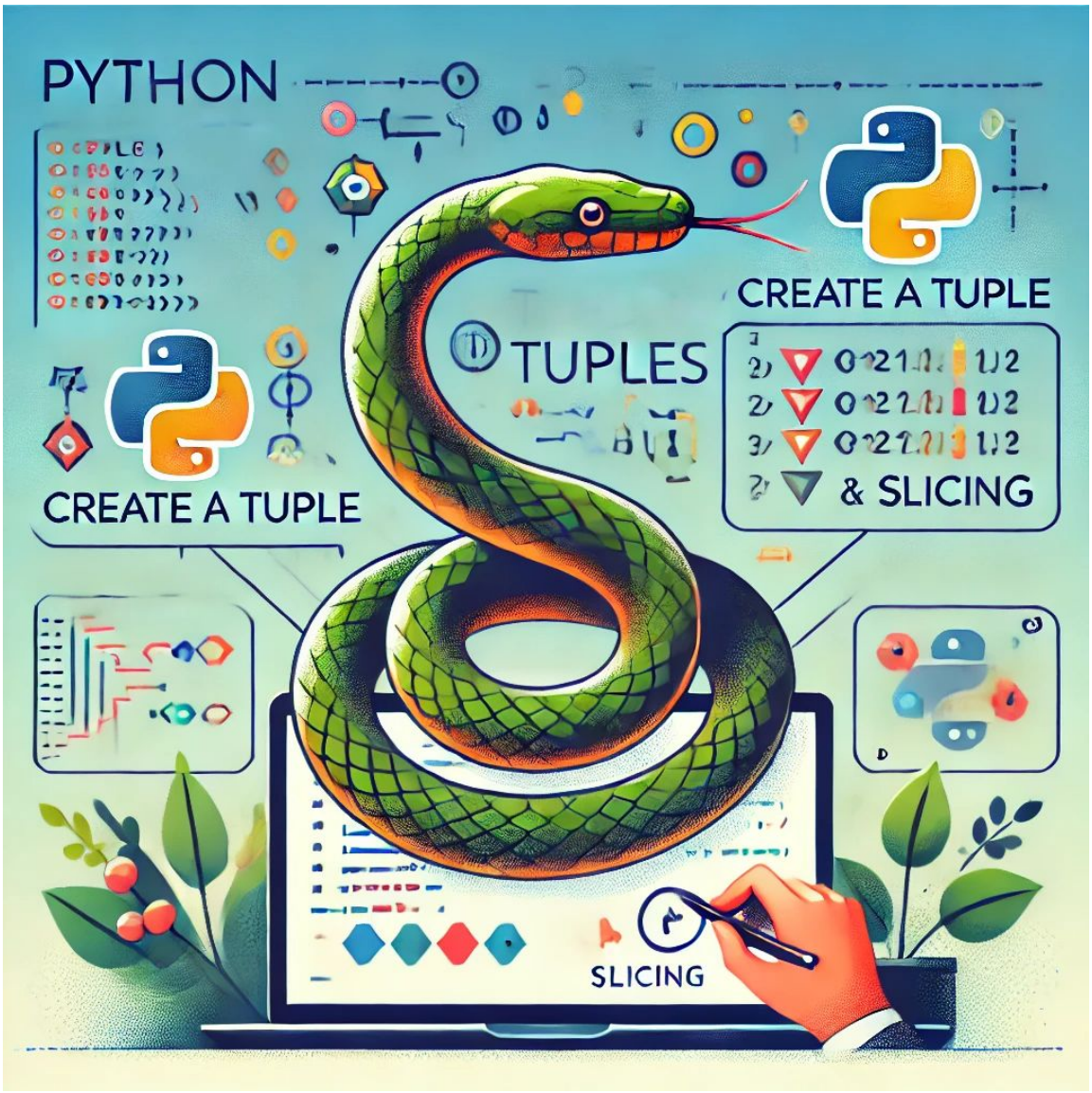
Congratulations, you have completed your first lesson and hands-on lab in Python.

Tuples in Python

1.1 Objectives

After completing this lab you will be able to:

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



About the Dataset

Imagine you received album recommendations from your friends and compiled all of the recommendations 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

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

[1]:
```
('disco', 10, 1.2)
```

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

[2]:
```
# Print the type of the tuple you created

type(tuple1)
```

[2]:
```
tuple
```

Indexing

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:

[3]:
```
# 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:

[4]:
```
# 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):

[5]:
```
# Use negative index to get the value of the last element

```

```
tuple1[-1]
```

[5]: 1.2

We can display the next two elements as follows:

[6]:
```
# Use negative index to get the value of the second last element

tuple1[-2]
```

[6]: 10

[7]:
```
# Use negative index to get the value of the third last element

tuple1[-3]
```

[7]: 'disco'

Concatenate Tuples

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

[8]:
```
# Concatenate two tuples

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

[8]: ('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:

[9]:
```
# Slice from index 0 to index 2

tuple2[0:3]
```

[9]: ('disco', 10, 1.2)

We can obtain the last two elements of the tuple:

[10]:
```
# Slice from index 3 to index 4

tuple2[3:5]
```

[10]: ('hard rock', 10)

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

[11]:
```
# Get the length of tuple

len(tuple2)
```

[11]:
```
5
```

This figure shows the number of elements:

Sorting

Consider the following tuple:

[13]:
```
# 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:

[14]:
```
# Sort the tuple

RatingsSorted = sorted(Ratings)
RatingsSorted
```

[14]:
```
[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:

[15]:
```
# 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:

[16]:
```
# 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:

[17]:
```
# 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:

[18]:
```
# Print the first element in the second nested tuples

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

[18]:
```
'r'
```

[19]:
```
# Print the second element in the second nested tuples

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

[19]:
```
'o'
```

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 third index:

[20]:
```
# Print the first element in the second nested tuples

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

[20]:
```
1
```

[21]:
```
# Print the second element in the second nested tuples

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

[21]:
```
2
```

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

Quiz on Tuples

Consider the following tuple:

[2]:
```
# sample tuple

genres_tuple = ("pop", "rock", "soul", "hard rock", "soft rock", \
                "R&B", "progressive rock", "disco")
genres_tuple
```

[2]:
```
('pop',
 'rock',
 'soul',
 'hard rock',
 'soft rock',
 'R&B',
 'progressive rock',
 'disco')
```

Find the length of the tuple, genres_tuple:

[23]:
```
# Write your code below and press Shift+Enter to execute
len(genres_tuple)
```

[23]:
```
8
```

```
len(genres_tuple)
```

Access the element, with respect to index 3:

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

[3]:
```
'hard rock'
```

```
genres_tuple[3]
```

Use slicing to obtain indexes 3, 4 and 5:

[4]:
```
# Write your code below and press Shift+Enter to execute
genres_tuple[3:6]
```

[4]:
```
('hard rock', 'soft rock', 'R&B')
```

```
genres_tuple[3:6]
```

Find the first two elements of the tuple genres_tuple:

[6]:
```
# Write your code below and press Shift+Enter to execute
genres_tuple[0:2]
```

[6]:
```
('pop', 'rock')
```

```
genres_tuple[0:2]
```

Find the first index of “disco”:

[7]:
```
# Write your code below and press Shift+Enter to execute
genres_tuple[7][0]
```

[7]:
```
'd'
```

```
genres_tuple.index("disco")
```

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

[11]:
```
# Write your code below and press Shift+Enter to execute
C_tuple = (-5, 1, -3)
C_tuple = C_tuple.sorted(-5, 1, -3)
C_tuple
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
/tmp/ipykernel_69/606060434.py in <module>
      1 # Write your code below and press Shift+Enter to execute
      2 C_tuple = (-5, 1, -3)
----> 3 C_tuple = C_tuple.sorted(-5, 1, -3)
      4 C_tuple

AttributeError: 'tuple' object has no attribute 'sorted'
```

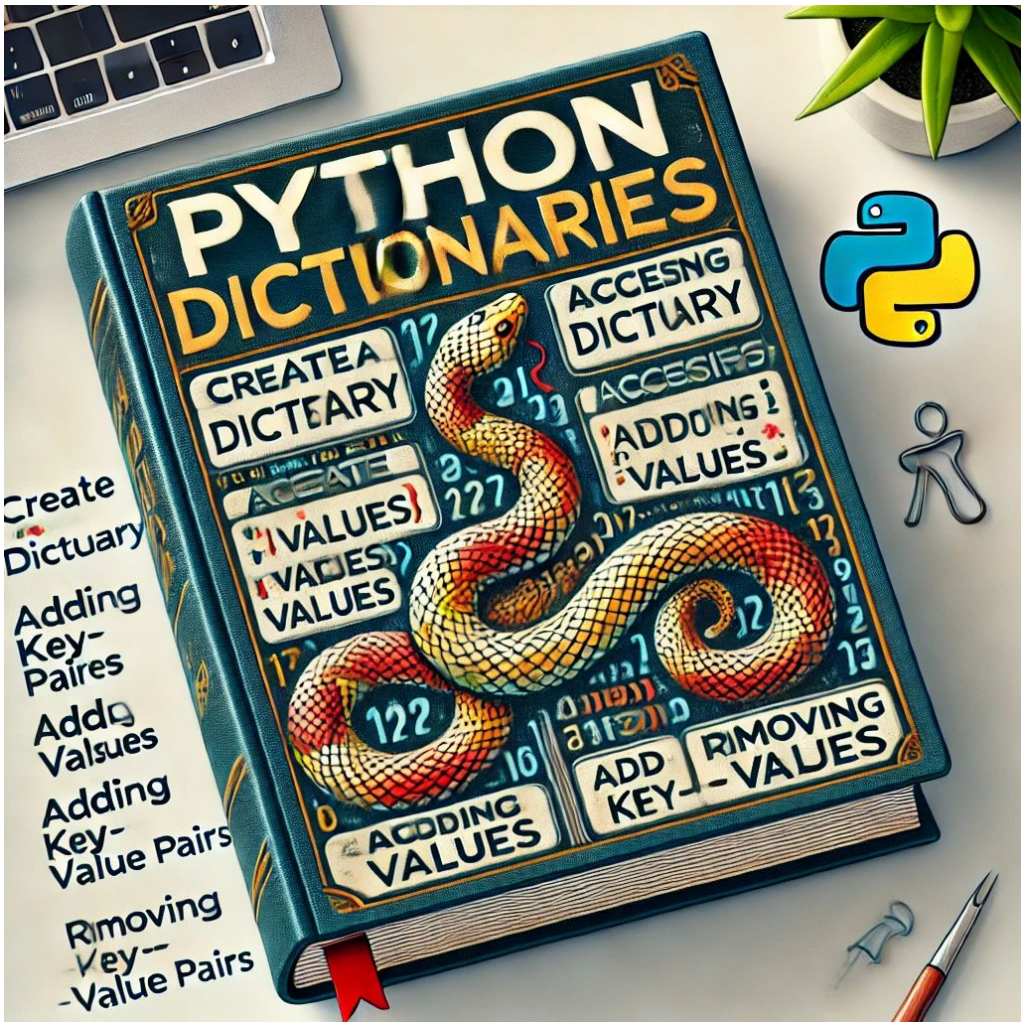
```
C_tuple = (-5, 1, -3)
C_list = sorted(C_tuple)
C_list
```

Dictionaries in Python

1.1 Objectives

After completing this lab you will be able to:

- Work with and perform operations on dictionaries in Python



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 Dict:

[1]:
```
# Create the dictionary

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

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

- “key1”: 1- The key is a string “key1”, and its corresponding value is the integer 1.
- “key2”: “2” - The key is a string “key2”, and its corresponding value is the string “2”.
- “key3”: [3, 3, 3] - The key is a string “key3”, and its corresponding value is a list [3, 3, 3]. The list contains three elements, all of which are integer 3.
- “key4”: (4, 4, 4) - The key is a string “key4”, and its corresponding value is a tuple (4, 4, 4). The tuple contains three elements, all of which are the integer 4.
- (‘key5’): 5 - The key is a tuple (‘key5’,), and its corresponding value is the integer 5. Note that the key is enclosed in parentheses to indicate it as a tuple. In this case, the tuple contains a single element: the string “key5”.
- (0, 1): 6 - The key is a tuple (0, 1), and its corresponding value is the integer 6. This tuple contains two elements: 0 and 1.

The keys can be strings:

[2]:
```
# Access to the value by the key

Dict["key1"]
```

[2]:
```
1
```

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

[3]:
```
# Access to the value by the key

Dict[(0, 1)]
```

[3]:
```
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 “{}”.

[4]:
```
# 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
```

[4]:
```
{'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:

[5]:
```
# Get value by keys

release_year_dict['Thriller']
```

[5]:
```
'1982'
```

This corresponds to:

Similarly for The Bodyguard

[6]:
```
# Get value by key

release_year_dict['The Bodyguard']
```

[6]:
```
'1992'
```

Now let us retrieve the keys of the dictionary using the method keys():

[7]:
```
# Get all the keys in dictionary

release_year_dict.keys()
```

[7]:
```
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():

[8]:
```
# Get all the values in dictionary

release_year_dict.values()
```

[8]:
```
dict_values(['1982', '1980', '1973', '1992', '1977', '1976', '1977', '1977'])
```

We can add an entry:

[9]:
```
# Append value with key into dictionary

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

[9]:
```
{'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:

[10]:
```
# Delete entries by key

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

[10]:
```
{'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:

[11]:
```
# Verify the key is in the dictionary

'The Bodyguard' in release_year_dict
```

[11]:
```
True
```

Quiz on Dictionaries

You will need this dictionary for the next two questions:

[12]:
```
# Question sample dictionary

soundtrack_dic = {"The Bodyguard":"1992", "Saturday Night Fever":"1977"}
soundtrack_dic
```

[12]:
```
{'The Bodyguard': '1992', 'Saturday Night Fever': '1977'}
```

a) In the dictionary soundtrack_dic what are the keys ?

[13]:
```
# Write your code below and press Shift+Enter to execute
soundtrack_dic.keys()
```

[13]:
```
dict_keys(['The Bodyguard', 'Saturday Night Fever'])
```

```
soundtrack_dic.keys() # The Keys "The Bodyguard" and "Saturday Night Fever"
```

b) In the dictionary soundtrack_dic what are the values ?

[14]:
```
# Write your code below and press Shift+Enter to execute
soundtrack_dic.values()
```

[14]:
```
dict_values(['1992', '1977'])
```

```
soundtrack_dic.values() # The values are "1992" and "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.

[15]:
```
# Write your code below and press Shift+Enter to execute
album_sales_dict = {"Thriller": 1.2, "Back in Black": 2, \
                    "The Dark Side of the Moon": 2.2, "The Bodyguard": 3, \
                    "Bat Out of Hell": 4, "Their Greatest Hits (1971-1975)": 5,␣
 ↪\
                    "Saturday Night Fever": 6, "Rumours": 7}
album_sales_dict
```

[15]:
```
{'Thriller': 1.2,
 'Back in Black': 2,
 'The Dark Side of the Moon': 2.2,
```

```
 'The Bodyguard': 3,
 'Bat Out of Hell': 4,
 'Their Greatest Hits (1971-1975)': 5,
 'Saturday Night Fever': 6,
 'Rumours': 7}
```

```
album_sales_dict = {"The Bodyguard":50, "Back in Black":50, "Thriller":65}
```

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

[16]:
```
# Write your code below and press Shift+Enter to execute
album_sales_dict["Thriller"]
```

[16]:
```
1.2
```

```
album_sales_dict["Thriller"]
```

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

[17]:
```
# Write your code below and press Shift+Enter to execute
album_sales_dict.keys()
```

[17]:
```
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'])
```

```
album_sales_dict.keys()
```

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

[18]:
```
# Write your code below and press Shift+Enter to execute
album_sales_dict.values()
```

[18]:
```
dict_values([1.2, 2, 2.2, 3, 4, 5, 6, 7])
```

```
album_sales_dict.values()
```

You will need the dictionary D:

[19]:
```
D={'a':0,'b':1,'c':2}
D
```

[19]:
```
{'a': 0, 'b': 1, 'c': 2}
```

[20]:
```
# Find the value for the key 'a':
```

[21]:
```
D_value = D['a']
D_value
```

[21]:
```
0
```

[22]:
```
# Find the keys of the dictionary D:
```

[23]:
```
D_keys = D.keys()
D_keys
```

[23]:
```
dict_keys(['a', 'b', 'c'])
```

Congratulations, you have completed your first lesson and hands-on lab in Python.