1.5 Collection Types

We have many collection types in Python, `str`, `int` objects hold only value, but coming to collection types, we can have various objects stored in the collections.

The Collection Types we have in Python are:

- Tuple
- List
- Set
- Dictionary

Tuple

A Tuple is a ordered collection of objects and it is of fixed length and immutable, so the values in the tuple can not be changed nor added or removed.

Tuples are generally used for small collections which we are sure about them from right before such as IP addresses and port numbers. Tuples are represented with paranthesis `()`

Example:

In [1]:
```
ip_address_port = ("127.0.0.1", 8080)
```

A tuple with a single member needs to have a trailing comma, else the type of the variable would be the datatype of the member itself.

In [2]:
```
# Proper way to create a single member tuple.
single_member_tuple = ("one",)
print(type(single_member_tuple))
single_member_tuple = ("one",)
print(type(single_member_tuple))
```

```
<class 'tuple'>
<class 'tuple'>
```

In [3]:
```
# Improper way trying to create a single member tuple.
single_member_tuple = "one"
print(type(single_member_tuple))
```

```
<class 'str'>
```

List

List collection types are similar to tuples, the only difference would be that new objects can be created, removed or object's data can be modified 😮.

In [4]:
```
int_list = [1, 2, 3]
string_list = ["abc", "defghi"]
```

In [5]:
```
# A list can be empty:
empty_list = []
```

objects in the list are not restricted to be of a particular datatype. let's see an example 👇.

In [6]:
```
mixed_list = [1, "abc", True, 3.14, None]
```

list can contain lists as objects too. These are called nested lists.

In [7]:
```
nested_list = [[1, 2, 3], ["a", "b", "c"]]
```

The objects present in the list can be accessed by the index it is placed. The index starts from 0 👻.

In [8]:
```
my_list = ["Iron man", "Thor", "Wonder Woman", "Wolverine", "Naruto"]
```

In [9]:
```
print(my_list[0])
print(my_list[1])
```

```
Iron man
Thor
```

In the `my_list`, we have 5 strings in the list, but in the below example, let's give a try to get the 100th index element which is not present in the `my_list` 🤭.

As there is no 100th element, we would be seeing an `IndexError` exception.

In [10]:
```
try:
    print(my_list[100])
except IndexError as exc:
    print(f"👻 Ouch! we got into IndexError exception: {exc}")
```

```
👻 Ouch! we got into IndexError exception: list index out of range
```

The question I have is, how do I get the 2nd element from the last 🤔? Should I find the length of the list and access the <length - 2>? Yup, it works 😖.

But we have one good way to do it by negative index, example: `-2`

In [11]:
```
# Access the 2nd element from the last.
print(my_list[-2])
```

```
Wolverine
```

We have a few methods of list that we can give it a try now 😎

`append`

In [12]:
```
# Append a new item to the list.
# We use append method of the list.
my_list.append("Zoro")
print(my_list)
```

```
['Iron man', 'Thor', 'Wonder Woman', 'Wolverine', 'Naruto', 'Zoro']
```

`remove`

In [13]:
```
# Remove the item present in the list.
# We use remove method of the list.
# If there's no object that we are trying to remove in the list, then ValueError would be raised.
try:
    my_list.remove("Zoro")
    print(my_list)
except ValueError as exc:
    print(f"Caught ValueError: {exc}")
```

```
['Iron man', 'Thor', 'Wonder Woman', 'Wolverine', 'Naruto']
```

`insert`

In [14]:
```
# Insert a object at a particular index.
# We use insert method of the list.
my_list.insert(1, "Super Man")
print(my_list)
```

```
['Iron man', 'Super Man', 'Thor', 'Wonder Woman', 'Wolverine', 'Naruto']
```

`reverse`

In [15]:
```
# Reverse the objects in the list.
# we use reverse method of the list.
my_list.reverse()
print(my_list)

# revert to the actual order
my_list.reverse()

# We have one more method too for this 😉
# The indexing of the list would be in the form of list[start: end: step]
# We will use step as -1 to get the elements in reverse order 😅
print(my_list[::-1])
```

```
['Naruto', 'Wolverine', 'Wonder Woman', 'Thor', 'Super Man', 'Iron man']
['Naruto', 'Wolverine', 'Wonder Woman', 'Thor', 'Super Man', 'Iron man']
```

`index`

In [16]:
```
# Index of an object in the list.
# we use index method of the list.
# raises a ValueError, if no given object is found in the list.
try:
    print(my_list.index("Naruto"))
except ValueError as exc:
    print(f"Caught ValueError: {exc}")
```

```
5
```

`pop`

In [17]:
```
# Pop is used to remove and return the element present at the last in the list(index=-1) by default.
# when index argument is passed, it would remove and return the element at that index.
# raises IndexError when no object is present at the given index.
try:
    last_element = (
        my_list.pop()
    )  # can be passed index argument value, if required to pop at a specific index.
    print(last_element)
except IndexError as exc:
    print(f"Caught IndexError: {exc}")
```

```
Naruto
```

set

A set is collection of unique items, the items does not follow insertion order.

Defining an set is pretty similar to a list or tuple, it is enclosed in `{}`

PS 🔔: If we need to have a empty set, `{}` won't create a set, it creates a empty dictionary instead. So we need to create a empty set by using `set()`

In [18]:
```
anime = {"Dragon ball", "One Piece", "Death Note", "Full Metal Alchemist", "Naruto"}
print(anime)
```

```
{"Death Note', 'Dragon ball', 'One Piece', 'Full Metal Alchemist', 'Naruto'}
```

`add`

In [19]:
```
anime.add("Tokyo Ghoul")
print(anime)
```

```
{'Death Note', 'Dragon ball', 'One Piece', 'Full Metal Alchemist', 'Naruto', 'Tokyo Ghoul'}
```

`remove`

remove method of set can be used to remove a particular object from the set, if the object is not present, KeyError would be raised.

In [20]:
```
try:
    anime.remove("Tokyo Ghoul")
    print(anime)
except KeyError as exc:
    print(
        f"Caught KeyError as there's given anime series present in the anime set: {exc}"
    )
```

```
{"Death Note', 'Dragon ball', 'One Piece', 'Full Metal Alchemist', 'Naruto'}
```

Dictionary

As in few other languages, we have hashmaps, Dictionaries in python are similar. It has unique Key - Value pairs.

The Key and Value can be of any object. Each Key-Value pair is separated by a `,`

In [21]:
```
anime_protagonist = {
    "Dragon Ball": "Goku",
    "One Piece": "Luffy",
    "Death Note": "Yagami Light",
    "Full Metal Alchemist": "Edward Elric",
    "Naruto": "Naruto",
}
print(anime_protagonist)
```

```
{"Dragon Ball': 'Goku', 'One Piece': 'Luffy', 'Death Note': 'Yagami Light', 'Full Metal Alchemist': 'Edward Elric', 'Naruto': 'Naruto'}
```

We can access the values of the dictionary by `<dictionary>[<key>]`. If there's no `<key>` in the dictionary, we would be seeing an KeyError 🔑❌

In [22]:
```
try:
    print(anime_protagonist["Dragon Ball"])
except KeyError as exc:
    print(
        f"👻 Ouch, Keyerror has been raised as no given key is found in the dictionary: {exc}"
    )
```

```
Goku
```

Iterate over keys, values and both in the dictionary 🐇.

In [23]:
```
# Keys
print("===Keys===")
for my_key in anime_protagonist.keys():
    print(my_key)

# Values
print("===Values===")
for my_value in anime_protagonist.values():
    print(my_value)

# Key-Values
print("===Key-Values===")
for my_key, my_value in anime_protagonist.items():
    print(f"{my_key} : {my_value}")
```

```
===Keys===
Dragon Ball
One Piece
Death Note
Full Metal Alchemist
Naruto
===Values===
Goku
Luffy
Yagami Light
Edward Elric
Naruto
===Key-Values===
Dragon Ball : Goku
One Piece : Luffy
Death Note : Yagami Light
Full Metal Alchemist : Edward Elric
Naruto : Naruto
```

PS 🔔: Are dictionaries ordered collection🤷?

From Python 3.7 dictionaries follow insertion order 😎

In python versions older than 3.7, the insertion of items is not ordered🤓. No problem 😌, we still have OrderedDict(present in collections module) `from collections import OrderedDict` which does the same 😉