04_PHYTON (Collection Types)

Lists

Introduction

There are various collection types in Python. While types such as `int` and `str` hold a single value, collection types hold multiple values.

In your programs, you usually need to group several items to render as a single object. We use collection types of data to do this job.

One of the most useful collections in Python is a `list`. In Python, a `list` is only an ordered collection of valid Python values.

The `list` type is probably the most commonly used collection type in Python. In spite of its name, a `list` is more like an array in some other languages (e.g. JavaScript).

Creating a List

A `list` can be created by enclosing values, separated by commas, in square brackets 👉`[]`.

Let's create a simple `list` that includes some country names.

```
country = ['USA', 'Brasil', 'UK', 'Germany', 'Turkey', '
New Zealand']
```

That is our first `list` in this course. Now let's print the `list`.

---

input :

```
country = ['USA', 'Brasil', 'UK', 'Germany', 'Turkey', '
New Zealand']

print(country)
```

output :

```
['USA', 'Brasil', 'UK', 'Germany', 'Turkey', 'New Zealan
d']
```

> 💡 Tips:
> - All the country names are printed in the same order as they were stored in the list because lists are ordered.

Another way to create a `list` is to call the '`list()`' function.

You do this when you want to create a `list` from an iterable object: that is, type of object whose elements you can import individually. **The lists are iterable like other collections and string types.** Let's create another `list` using `list()` function and compare with 👉'`[]`'.

---

input :

```
string_1 = 'I quit smoking'

new_list_1 = list(string_1)  # we created multi element
list
print(new_list_1)

new_list_2 = [string_1]  # this is a single element list
print(new_list_2)
```

output :

```
['I', ' ', 'q', 'u', 'i', 't', ' ', 's', 'm', 'o', 'k',
'i', 'n', 'g']
['I quit smoking']
```

> 💡 Tips:

- Note that, using `list()` function, all characters of string_1 including spaces was moved into a new_list_1.
- If you noticed, lists can contain more than one of the same value.

As it appears, the `list()` function creates a `list` that contains each component of a specific iterable object, such as a string. You can use square brackets or `list()` functions, depending on what you are going to do.

The components of a `list` are not limited to a single data type, given that Python is a dynamic language: e.g.

```
mixed_list = [11, 'Joseph', False, 3.14, None, [1, 2, 3]]
```

💡 Tips:

- As you see above, one or more of the list elements can even be a `list`.

Basic Operations with Lists

In Python, there are many methods and functions for dealing with the `list` structures. You'll learn some of them which are basic and the most common. Let's begin:

In most cases, we'll have to make an empty `list` to fill it later with the data you want.

```
empty_list_1= []

empty_list_2 = list()
```

We can add an element into a `list` using `.append()` or `.insert()` methods.

- `.append()` : Append an object to end of a `list`. Using only `list.append(element)` syntax, returns none. If you want to see the new appended `list`, you have to call or print it. See the example :

---

input :

```
empty_list_1 = []
empty_list_1.append('114')
empty_list_1.append('plastic-free sea')

print(empty_list_1)
```

output :

```
['114', 'plastic-free sea']
```

---

input :

```
city = ['New York', 'London', 'Istanbul', 'Seoul', 'Sydn
ey']
city.append('Addis Ababa')

print(city)
```

output :

```
['New York', 'London', 'Istanbul', 'Seoul', 'Sydney', 'A
ddis Ababa']
```

- `.insert()` : Add a new object to `list` at a specific index. The syntax looks like `list.insert(index, object)`. See the example :

---

input :

```
city = ['New York', 'London', 'Istanbul', 'Seoul', 'Sydn
ey', 'Addis Ababa']
city.insert(2, 'Stockholm')

print(city)
```

output :

```
['New York', 'London', 'Stockholm', 'Istanbul', 'Seoul',
 'Sydney', 'Addis Ababa']
```

We can remove the elements in `lists` using `list.remove()` method or sort the elements using `list.sort()` method. Examine the example :

---

input :

```
city = ['New York', 'London', 'Stockholm', 'Istanbul', '
Seoul', 'Sydney', 'Addis Ababa']
city.remove('London')
print(city)  # we have deleted 'London'
```

output :

```
['New York', 'Stockholm', 'Istanbul', 'Seoul', 'Sydney',
 'Addis Ababa']
```

---

input :

```
city = ['New York', 'Stockholm', 'Istanbul', 'Seoul', 'S
ydney', 'Addis Ababa']
city.sort()  # lists the items in alphabetical order
print(city)
```

output :

```
['Addis Ababa', 'Istanbul', 'New York', 'Seoul', 'Stockh
olm', 'Sydney']
```

💡 Tips:

- Remember! Elements of a list are counted from left to right and start with zero as in string types.

Likewise, the length of the `list` elements can be calculated with the `len()` function also. Let's calculate the length of 'city' variable we have.

---

input :

```
city = ['Addis Ababa', 'Istanbul', 'New York', 'Seoul',
'Stockholm', 'Sydney']
print(len(city))
```

output :

```
6
```

One of the important operations of the `list`s is assigning an element to the specific index number.

---

input :

```
city = ['New York', 'Stockholm', 'Istanbul', 'Seoul', 'S
ydney', 'Addis Ababa']
city[1] = 'Melbourne'  # we assign 'Melbourne' to index
1
print(city)
```

output :

```
['New York', 'Melbourne', 'Istanbul', 'Seoul', 'Sydney',
 'Addis Ababa']
```

✏️Homework:

- Examine the use of `index()`, `del()` and `pop()` functions.

> There are many other 'list operations' (mutable sequence types operations) methods here that you can examine in detail.

Tuples

Definitions

Up to this section of our lesson, we saw the most used collection types of Python : `list`. A `tuple` is another collection type that can hold multiple data very similar to the `list`.

The most important difference from the `list` is that the `tuple` is immutable. Therefore, methods like `append()` or `remove()` do not exist in the operations of this type.

Tuples are **commonly used for small collections of values** that will not need to change, such as an IP address and port. **If we have unchanged data, we should choose tuples because it is much faster than lists.**

We used square brackets 👉'`[]`' to define the lists. In the tuple, normal parentheses 👉'`()`' are used.

The same indexing rules for lists also apply to tuples. Tuples can also be nested and the values can be any valid Python valid.

> Q: What is the difference between list and tuple?
> A:
> LISTs :
>
> - Lists are mutable i.e they can be edited.
> - Lists are slower than tuples.
> - `Syntax: list_1 = [True, 'Space', 20]`
>
> TUPLEs :
> - Tuples are immutable (tuples are lists which can't be edited).
> - Tuples are faster than list.
> - `Syntax: tup_1 = (True, 'Space' , 20)`
>
> - Interview Q&A

Creating a Tuple

A tuple also can be created by enclosing values, separated by commas, in parentheses.

You can compare `tuple` to a case. When you put the data that you want it to not change and close the lid, you can no longer change this data, modify its size and edit it.

Let's create a simple empty `tuple` :

```
empty_tuple = ()
```

This is our first `tuple` in this course. Now let's print its type.

---

input :

```
empty_tuple = ()
print(type(empty_tuple))
```

output :

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

If you want to create a single element `tuple`, you should use a comma.

---

input :

```
try_tuple = ('love')

print(try_tuple)

print(type(try_tuple)) # it's not tuple type.
```

output :

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

It occurs in only single element `tuple`s and we can fix the problem using comma at the end of the element.

> 💡 Tips:
> - Remember to always use a comma when defining a singleton tuple.

---

input :

```
try_tuple = ('love',)

print(try_tuple)

print(type(try_tuple)) # it's a tuple type.
```

output :

```
('love',)
<class 'tuple'>
```

Actually, if your `tuple` contains more than one element, separating elements with commas will be enough.

Another way to create a `tuple` is to call the `tuple()` function. You do this when you want to create a `tuple` from an iterable object: that is, a type of object whose elements you can import individually.

The `tuple` is also iterable like other collections and `str`ing types. Let's create another `tuple` using `tuple()` function. With this function, you can create an empty `tuple` as well.

Let's examine some examples of creating `tuple`s :

---

input :

```
planets = 'mercury', 'jupiter', 'saturn'

print(planets)
print(type(planets))
```

output :

```
('mercury', 'jupiter', 'saturn')
<class 'tuple'>
```

---

input :

```
empty_tuple_1 = tuple()

print(empty_tuple_1)
print(type(empty_tuple_1))
```

output :

```
()
<class 'tuple'>
```

It is easy to convert between `list` and `tuple` as in the examples below :

---

input :

```
my_tuple=(1, 4, 3, 4, 5, 6, 7, 4)

my_list = list(my_tuple)

print(type(my_list), my_list)
```

output :

```
<class 'list'> [1, 4, 3, 4, 5, 6, 7, 4]
```

---

input :

```
my_list = [1, 4, 3, 4, 5, 6, 7, 4]

my_tuple = tuple(my_list)

print(type(my_tuple), my_tuple)
```

output :

```
<class 'tuple'> (1, 4, 3, 4, 5, 6, 7, 4)
```

An iterable `string` can be converted to a `tuple` :

input :

```
mountain = tuple('Alps')
print(mountain)
```

output :

```
('A', 'l', 'p', 's')
```

How can We Use a Tuple ?

If you want, let's take a look at the common features of the `list` and `tuple`. So you can have an idea of what to do with `tuple`s.

Both `list`s and `tuple`s are ordered. It means that when storing elements to these containers, you can sure that their order will remain the same. You can also duplicate values or mix different data types in `tuple`s.

input :

```
mix_value_tuple = (0, 'bird', 3.14, True)
```

```
print(len(mix_value_tuple))
```

output :

```
4
```

As we stated at the beginning, just like `lists`, `tuples` support indexing :

---

input :

```
even_no = (0, 2, 4)
print(even_no[0])
print(even_no[1])
print(even_no[2])
print(even_no[3])
```

output :

```
0
2
4
------------------------------------------------------------
-------------------
print(even_no[3]) : IndexError: tuple index out of range
```

And one of the most important differences of `tuples` from `lists` is that 'tuple' object does not support item assignment. Yes, because `tuple` is immutable. See the example :

---

input :

```
city_list = ['Tokyo', 'Istanbul', 'Moskow', 'Dublin']

city_list[0] = 'Athens'
city_list[1] = 'Cairo'
print(city_list)
```

output :

```
['Athens', 'Cairo', 'Moskow', 'Dublin']
```

---

input :

```
city_list = ['Tokyo', 'Istanbul', 'Moskow', 'Dublin']

city_tuple = tuple(city_list)

city_tuple[0] = 'New York'  # you can't assign a value
```

output :

```
---------------------------------------------------------------------------
TypeError: 'tuple' object does not support item assignment
```

Benefits of Immutability

Let's take a look at the basic advantages of `tuple`s :

- **Tuples are faster and more powerful in-memory than `list`s.** You should give it a thought whenever you need to deal with large amounts of data. If you don't want to change your data you may have to choose `tuple`s.
- **Because of its immutability, the data stored in a `tuple` can not be altered by mistake.**
- **A `tuple` can be used as a `dictionary`** (we will see in the next lesson) **`key`**, while 'TypeError' can result in `list`s as keys. And this is the usefulness of `tuple`s in the data processing.

Dictionaries

In this topic, we will examine the collection types which store item pairs. What does it mean?

Think of a real dictionary. It contains words and their meanings. **In Python, you can accept the words as `key` and the meaning of the words as `value`.**

A `dictionary` in Python is a collection of `key`-`value` pairs called items of a dictionary.

The dictionary is enclosed by curly braces 👉`{}`. Each pair (item) is separated by a comma and the `key` and `value` are separated by a colon.

Creating a Dictionary

A `dic`tionary also can be **created by enclosing pairs, separated by commas, in curly-braces.** Looks like `list` or `tuple`, right?

And of course, **we can use a function to create a `dic`tionary : '`dict()`' function.** Let's create a simple empty `dict` :

```
empty_dict_1 = {}

empty_dict_2 = dict()
```

This is our first `dict` in this lesson. Now let's print its type.

---

input :

```
empty_dict_1 = {}

print(type(empty_dict_1))
```

output :

```
<class 'dict'>
```

The basic form of `dict` looks like :

```
my_dict = {'key1': 'value1',
           'key2': 'value2',
           'key3': 'value3'
           }
```

The syntax for accessing an item is very simple. We write a `key` that we want to access in square brackets. This method works both for adding items to a `dict` and for reading them from there.

In the following examples, you'll see several methods that allow us to create a `dict` and add a `key-value` pair to it.

---

input :

```
state_capitals = {'Arkansas': 'Little Rock',
                  'Colorado': 'Denver',
                  'California': 'Sacramento',
                  'Georgia': 'Atlanta'
                 }

print(state_capitals['Colorado']) # accessing method
```

output :

```
Denver
```

---

input :

```
state_capitals = {'Arkansas': 'Little Rock',
                  'Colorado': 'Denver',
                  'California': 'Sacramento',
                  'Georgia': 'Atlanta'
                 }

state_capitals['Virginia'] = 'Richmond' # adding a new i
tem

print(state_capitals)
```

output :

```
{'Arkansas': 'Little Rock',
'Colorado': 'Denver',
'California': 'Sacramento',
'Georgia': 'Atlanta',
'Virginia': 'Richmond'}
```

💡 Tips:

- **Note that keys and values can be of different types.**

```
mix_values = {'animal': ('dog', 'cat'),  # tuple type
              'planet': ['Neptun', 'Saturn', 'Jupiter'],
              'number': 40,  # int type
              'pi': 3.14,  # float type
              'is_good': True}  # bool type

mix_keys = {22 : "integer",
            1.2 : "float",
            True : "boolean",
            "key" : "string"}
```

And now, let's use `dict()` function to create a dictionary :

---

input :

```
dict_by_dict = dict(animal='dog', planet='neptun', numbe
r=40, pi=3.14, is_good=True)

print(dict_by_dict)
```

output :

```
{'animal': 'dog',
'planet': 'neptun',
'number': 40,
'pi': 3.14,
'is_good': True}
```

⚠️ Avoid ! :

- Do not use quotes for `keys` when using the `dict()` function to create a dictionary.
- You cannot use iterables as `keys` to create a dictionary.

> Q: What is a dictionary in Python?
> A: Python dictionary is one of the supported data types in Python. It is an unordered collection of elements. The elements in dictionaries are stored as `key`–`value` pairs. Dictionaries are indexed by `keys`. For example, below we have a `dict` named `my_dict`. It contains two `keys`, fruit and vegatable, along with their corresponding `values`, banana and onion.
>
> `my_dict = {'fruit':'banana', 'vegatable':'onion'}`
>
> - Interview Q&A

Main Operations with Dictionaries

There are several methods that allow us to access items, keys, and values. You can access all items using the `.items()` method, all keys using the `.keys()` method, and all values using the `.values()` method:

---

input :

```
dict_by_dict = {'animal': 'dog',
                'planet': 'neptun',
                'number': 40,
                'pi': 3.14,
                'is_good': True}

print(dict_by_dict.items(), '\n')
print(dict_by_dict.keys(), '\n')
print(dict_by_dict.values())
```

output :

```
dict_items([('animal', 'dog'), ('planet', 'neptun'),
            ('number', 40), ('pi', 3.14), ('is_good', True

dict_keys(['animal', 'planet', 'number', 'pi', 'is_good'
])

dict_values(['dog', 'neptun', 40, 3.14, True])
```

You have learned that you can add a new item by assigning value to a `key` that is not in the `dict`ionary. Likewise, you can add new items using the `.update()` method. Let's see :

---

input :

```
dict_by_dict = {'animal': 'dog',
                'planet': 'neptun',
                'number': 40,
                'pi': 3.14,
                'is_good': True}

dict_by_dict.update({'is_bad': False})

print(dict_by_dict)
```

output :

```
{'animal': 'dog',
'planet': 'neptun',
'number': 40,
'pi': 3.14,
'is_good': True,
'is_bad': False}
```

You can also remove an item using the `del` function:

> The formula syntax is : `del dictionary_name['key']`.

See the example.

input :

```
dict_by_dict = {'animal': 'dog',
                'planet': 'neptun',
                'number': 40,
                'pi': 3.14,
                'is_good': True,
                'is_bad': False}

del dict_by_dict['animal']

print(dict_by_dict)
```

output :

```
{'planet': 'neptun',
'number': 40,
'pi': 3.14,
'is_good': True,
'is_bad': False}
```

Using the `in` and the `not in` operator, you can check if the `key` is in the `dict`ionary.

> - When we use the `in` operator; if the `key` is in the dictionary, the result will be `True` otherwise `False`.
> - When we use the `not in`; if the `key` is not in the dictionary, the result will be `True` otherwise `False`.

Look at the example :

---

input :

```
dict_by_dict = {'planet': 'neptun',
                'number': 40,
                'pi': 3.14,
                'is_good': True,
                'is_bad': False}
```

```
print('pi' in dict_by_dict)
print('animal' not in dict_by_dict)  # remember, we have
 deleted 'animal'
```

output :

```
True
True
```

Nested Dictionaries

In some cases, you need to work with the nested `dict`. When you decide to specialize in data science, we will work very often with dictionaries in the future.

```
school_records={
    "personal_info":
        {"kid":{"tom": {"class": "intermediate", "age": 10
                "sue": {"class": "elementary", "age": 8}
               },
         "teen":{"joseph":{"class": "college", "age": 19},
                 "marry":{"class": "high school", "age": 1
               },
        },

    "grades_info":
        {"kid":{"tom": {"math": 88, "speech": 69},
                "sue": {"math": 90, "speech": 81}
               },
         "teen":{"joseph":{"coding": 80, "math": 89},
                 "marry":{"coding": 70, "math": 96}
               },
        },
}
```

We can use square brackets to access internal `dict`s :

---

input :

```
school_records={
    "personal_info":
        {"kid":{"tom": {"class":"intermediate", "age":10},
                "sue": {"class":"elementary", "age":8}
               },
         "teen":{"joseph":{"class":"college", "age":19},
                 "marry":{"class":"high school", "age":16}
               },
        },
}

print(school_records['personal_info']['teen']['marry']['
age'])
```

output :

```
16
```

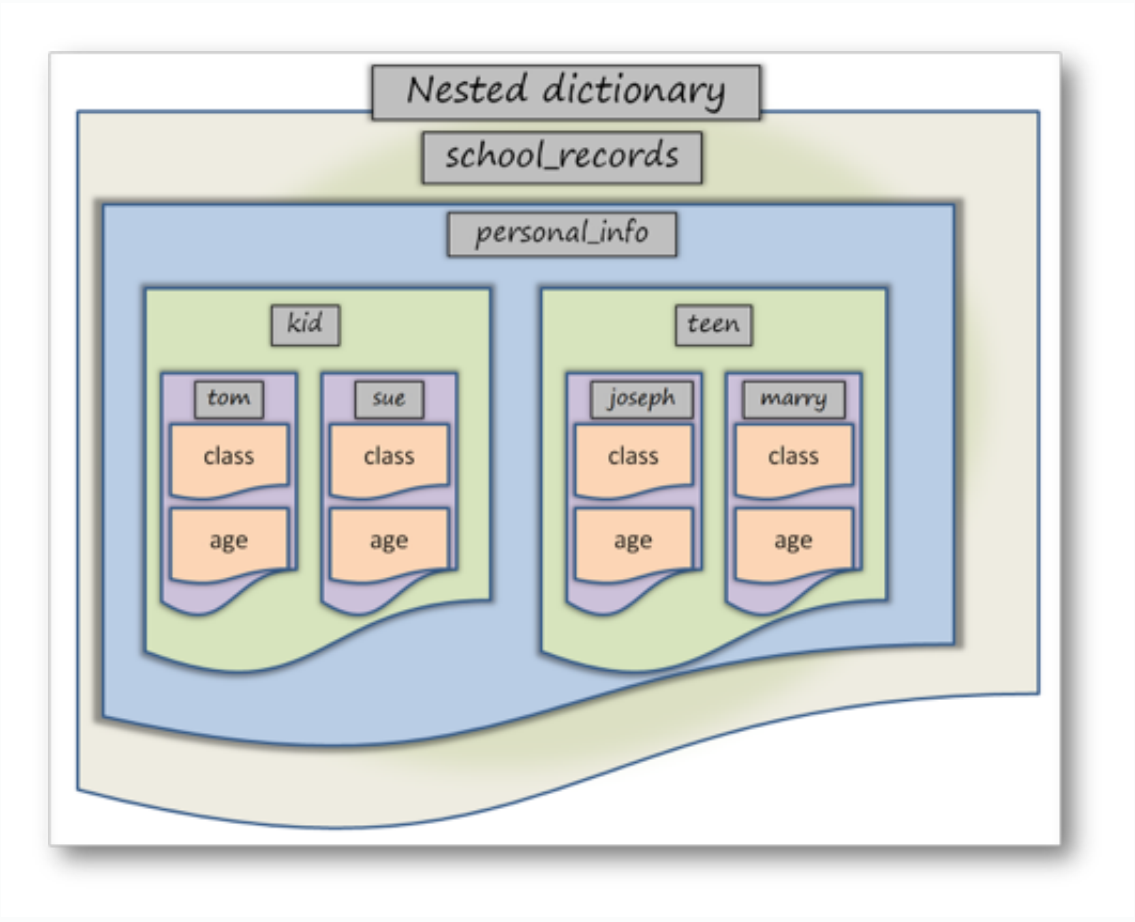

Diagram of Nested Dictionary

💡 Tips:

- Dictionaries strongly resemble JSON syntax. The native json module in the Python standard library can be used to convert between JSON and dictionaries.

✏️ Homework:

- What is 'JSON' and what is it used for?

> If you want to go deep into `dicts`, here you will find what you want.

Sets

A set is a collection of elements with no repeats and without insertion order but sorted order.

Basic uses include membership testing and eliminating duplicate entries. **Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.**

They can hold multiple data in them, but only one of value. They are used in situations where it is only important that some things are grouped together, and not what order they were included.

Creating a Set

Curly braces '`{}`' or the `set()` function can be used to create `set`s. But the only way to create an empty `set` is: use the `set()` function.

⚠️ Avoid ! :

- Note that, **to create an empty `set` you have to use `set()` function. Do not use `{}` to create an empty `set`. Otherwise, you will create an empty `dict`ionary.**

Let's create a simple empty `set` :

```
empty_set = set()
```

This is our first `set` in this lesson. Now let's print its type.

---

input :

```
empty_set = set()

print(type(empty_set))
```

output :

```
<class 'set'>
```

We will now see how `set`s have unordered and unique objects.

---

input :

```
colorset = {'purple', 'orange', 'red', 'darkblue', 'yell
ow', 'red'}

print(colorset)
print(colorset)
```

output :

```
{'darkblue', 'orange', 'purple', 'red', 'yellow'}
{'darkblue', 'purple', 'orange', 'yellow', 'red'}
```

As you can see in the output, the two 'red' values we have defined in the `set` have fallen to one. And every time you print the `set`, the order of the objects in the `set` changes.

Let's look at another example :

---

input :

```
s = set('unselfishness')

print(s)
```

output :

```
{'f', 'l', 'i', 'u', 'e', 'n', 'h', 's'}
```

As you can see, the letters of the `str`ing type data are only written once in the `set`. Within this scope, using `set`s can help you avoid repetitions. Let's convert a `list` into a `set` and look at the repetitions of its elements:

---

input :

```
flower_list = ['rose', 'violet', 'carnation', 'rose', 'o
rchid', 'rose', 'orchid']
flowerset = set(flower_list)
flowerlist = list(flowerset)

print(flowerset)
print(flowerlist)
```

output :

```
{'orchid', 'carnation', 'violet', 'rose'}
['orchid', 'carnation', 'violet', 'rose']
```

✏️Homework:

- `{'carnation', 'orchid', 'rose', 'violet'}` 👈👉 `{'rose', 'orchid', 'rose', 'violet', 'carnation'}` Do these two `sets` give the same output and why? (Note: Try to figure out the answer before run on the Playground)

> Q: Which one of the following is not the correct syntax for creating a set in Python?
> A:
> **a. `set([[1,2],[3,4],[4,5]])`**
> b. `set([1,2,2,3,4,5])`
> c. `{1,2,3,4}`
> d. `set((1,2,3,4))`
>
> Explanation: The iterable argument given for the set must be used in a correct way.
>
> - Interview Q&A

Main Operations with Sets

There are several methods that allow us to add and remove items to/from sets. Moreover, we have the methods of intersection, unification, and differentiation of sets :

These methods are :

- `.add()` : Adds a new item to the set.
- `.remove()` : Allows us to delete an item.
- `.intersection()` : Returns the intersection of two sets.
- `.union()` : Returns the unification of two sets.
- `.difference()` : Gets the difference of two sets.

Now, let's do some examples of these methods :

---

input :

```
a = set('abracadabra')

print(a)
```

output :

```
{'a', 'b', 'c', 'd', 'r'}
```

---

input :

```
a = set('abracadabra')
b = set('alacazam')

print(a - b)  # same as '.difference()' method
print(a.difference(b)) # a difference from b
```

output :

```
{'b', 'd', 'r'}
{'b', 'd', 'r'}
```

---

input :

```
a = set('abracadabra')
b = set('alacazam')

print(a | b)  # same as '.union()' method
print(a.union(b)) # unification of a with b
```

output :

```
{'a', 'b', 'c', 'd', 'l', 'm', 'r', 'z'}
{'a', 'b', 'c', 'd', 'l', 'm', 'r', 'z'}
```

---

input :

```
a = set('abracadabra')
b = set('alacazam')

print(a & b)  # same as '.intersection()' method
print(a.intersection(b)) # intersection of a and b
```

output :

```
{'a', 'c'}
{'a', 'c'}
```

input :

```
a = set('abracadabra')

a.remove('c') # we delete 'c' from the set
print(a)
```

output :

```
{'a', 'b', 'd', 'r'}
```

input :

```
a = set('abracadabra')

a.add('c') # we add 'c' again into the set
print(a)
```

output :

```
{'a', 'b', 'c', 'd', 'r'}
```

Additionally, you can:

- Get the number of set's elements using `len()` function,
- Check if an element belongs to a specific set(`in` / `not in`operators), you get the boolean value.

Thus, we have completed this topic which is the most important one in Python.